MPI 和 OpenMP 中的并行编程 TheArt of HPC, volume2

Victor Eijkhout

第 2 版 2022 年，格式化于 2024 年 1 月 14 日图书和幻灯片下载：
https://tinyurl.com/vle335course 公共代码仓库：
https://github.com/VictorEijkhout/TheArtOfHPC_vol2_parallelprogrammingHTML 版本：
https://theartofhpc.com/pcse/

“并行计算”一词在不同的应用领域有不同的含义。在本书中，我们专注于并行计算 —— 更具体地说是并行 *programming*；我们不会讨论大量理论 —— 而是在科学计算的背景下。

科学计算中最常用的两种并行编程软件系统是 MPI 和 OpenMP。它们针对不同类型的并行性，使用非常不同的构造。因此，通过在一本书中涵盖它们两者，我们可以提供涵盖广泛应用的并行性处理。

最后，我们还讨论了 PETSc（科学计算便携工具包）库，它提供了比 MPI 或 OpenMP 更高层次的抽象，专门针对并行线性代数，特别是源自偏微分方程建模的线性代数计算。

T科学计算中的主要语言是 C/C++ 和 Fortran。我们将讨论 MPI 和 OpenMP w这两种语言中的许多示例。对于 MPI 和 PETSc 库，我们还将讨论 Python i接口。

Comments 本书处于不断修订和完善的状态。请将任何形式的评论发送至 `eijkhout@tacc.utexas.edu`。

目录

目录

第一部分

MPI

本书本节介绍 MPI （“Message Passing Interface”），这是科学与工程中分布式内存编程的主导模型。它将培养以下能力。

基础水平：

- 学生将理解 SPMD 模型及其在 MPI 中的实现（第 2 章）。
- 学生将了解基本的集合调用，包括有根进程和无根进程的调用，并能在应用中使用它们（第 3 章）。
- 学生了解基本的阻塞和非阻塞点对点调用，以及如何使用它们（第 4 章）。

Intermediate level:

- 学生了解派生数据类型，并能在通信例程中使用它们（第 6 章）。
- 学生了解 intra-communicators，以及创建子通信器的一些基本调用 cators(chapter 7); 还包括 Cartesian 进程拓扑 (section 11.1).
- 学生理解 MPI I/O 调用的基本设计，并能在基本应用中使用它们 (chapter 10).
- 学生了解图进程拓扑和邻域集合操作 (sec - 11.2 ）。

高级水平：

- 学生理解单边通信例程，包括窗口创建例程和同步机制（第 9 章）。
- 学生理解 MPI 共享内存、其优势及其基于窗口的原理（第 12 章）。
- 学生理解 MPI 进程生成机制和进程间通信器（第 8 章）。

第 1 章

MPI 入门

在本章中，您将学习分布式内存编程的主要工具：消息传递接口（ MPI ）库的使用。MPI 库大约有 250 个例程，其中许多您可能永远不会用到。由于这是一本教科书，而非参考手册，我们将重点关注重要的概念，并针对每个概念给出重要的例程。您在这里学到的内容应足以满足大多数常见需求。建议您随时备有参考文档，以防有专门的例程，或查阅您使用的例程的细节。

1.1 分布式内存和消息传递

在最简单的形式中，分布式内存机器是一组通过网络线缆连接的单台计算机。事实上，这种结构有一个名称：*Beowulf* 集群。正如您从该结构中认识到的，每个处理器可以运行独立的程序，并且拥有自己的内存，无法直接访问其他处理器的内存。MPI 是使同一可执行文件的多个实例能够相互了解并通过网络交换数据的魔法。

MPI 作为集群高性能计算工具之所以如此成功的原因之一是它非常明确：程序员可以控制处理器之间数据传输的许多细节。因此，有能力的程序员可以用 MPI 编写非常高效的代码。不幸的是，程序员必须详细说明这些细节。正因为如此，人们有时称 MPI 为 “ 并行编程的汇编语言 ”。如果这听起来很吓人，请放心，情况并没有那么糟。你可以很快用 MPI 入门，只需掌握基础知识，只有在必要时才使用更复杂的工具。

MPI 受到程序员欢迎的另一个原因是它不要求你学习一门新语言：它是一个可以与 C/C++ 或 Fortran 接口的库；甚至还有 Python 和 *Java* 的绑定（本课程未涉及）。然而，这并不意味着 “ 为现有的串行程序添加并行性 ” 很简单。串行程序的 MPI 版本需要大量重写；肯定比本书后面讨论的通过 OpenMP 实现的共享内存并行更复杂。

MPI 也很容易安装：有免费的实现版本，你可以下载并安装在任何具有类 Unix 操作系统的计算机上，即使那不是一台并行机器。然而，如果你在超级计算机集群上工作，那里很可能已经安装了 MPI，并且针对该机器的网络进行了调优。

1.2 历史

在 1993-94 年 MPI 标准制定之前，存在许多用于分布式内存计算的库，通常是某个厂商平台的专有库。MPI 标准化了进程间通信机制。其他功能，如 *PVM* 中的进程管理或并行 I/O 被省略。标准的后续版本包含了许多这些功能。

由于 MPI 是由大量学术和商业参与者设计的，它迅速成为了一个标准。少数来自 MPI 之前时代的包，如 *Charmpp* [18]，仍在使用，因为它们支持 MPI 中不存在的机制。

1.3 基本模型

H这里我们简要介绍使用 MPI 的两种最常见场景。在第一种场景中，用户正在使用一个i交互式机器，具有对多个主机的网络访问权限，通常是工作站网络；s见图 1.1。用户输入命令 `mpiexec`[1] 并提供

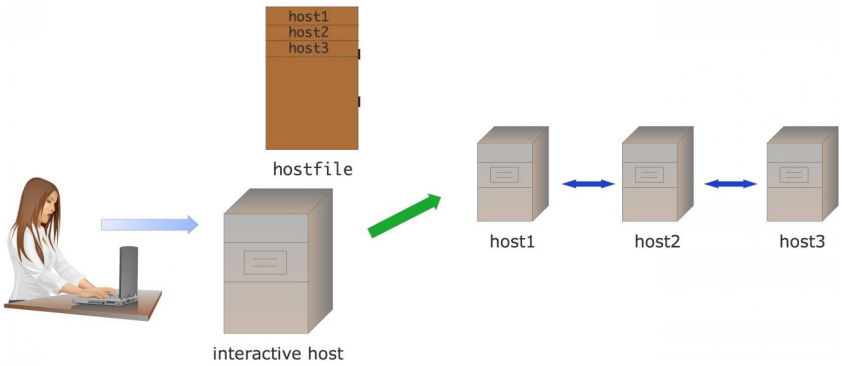


图 1.1：交互式 MPI 设置

- 涉及的主机数量，
- 它们的名称，可能在主机文件中，
- 以及其他参数，例如是否包含交互主机；随后是

程序的名称及其参数。

`mpiexec` 程序随后与每个主机建立 `ssh` 连接，向它们提供足够的信息以便相互找到。所有处理器的输出都通过 `mpiexec` 程序传输，并显示在交互式控制台上。

在第二种场景（图 1.2 ）中，用户准备了一个包含命令的批处理脚本，这些命令将在批处理调度器为作业分配多个主机时运行。现在批处理脚本包含 `mpiexec` 命令，主机文件在作业开始时动态生成。由于作业此时运行时用户可能未登录，任何屏幕输出都会写入输出文件。

1. 一个命令变体是 *mpirun*；您本地的集群可能有不同的机制。

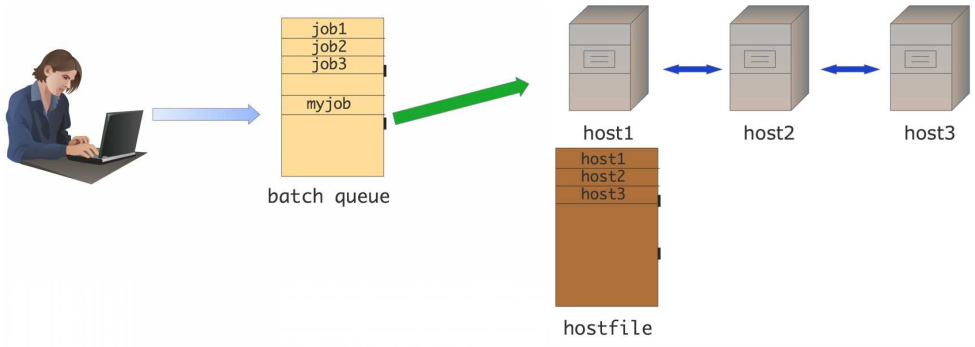


图 1.2：批处理 MPI 设置

你会看到在这两种场景中，使用 `mpiexec` 命令以单程序多数据（SPMD）执行模式启动并行程序：所有主机执行相同的程序。不同主机执行不同程序是可能的，但本书不考虑这种情况。

某些 MPI 安装或网络可能有特定的选项和环境变量。

- *mpich* 及其衍生版本如 *Intel MPI* 或 *Cray MPI* 具有 *mpiexec* 选项：https://www.mpich.org/static/docs/v3.1/www1/mpiexec.html

1.4 制作和运行 MPI 程序

MPI 是一个库，从普通编程语言如 C/C++ 或 Fortran 的程序中调用。要编译这样的程序，你使用常规的编译器：

```
gcc -c my_mpi_prog.c -I/path/to/mpi.h
gcc -o my_mpi_prog my_mpi_prog.o -L/path/to/mpi -lmpich
```

然而，不同架构之间 MPI 库的名称可能不同，这使得编写可移植的 makefile 变得困难。因此，MPI 通常在你的编译器调用周围有 shell 脚本，分别称为 `mpicc`、`mpicxx`、`mpif90`，对应 C/C++/Fortran。

```
mpicc -c my_mpi_prog.c
mpicc -o my_mpi_prog my_mpi_prog.o
```

如果你想知道 `mpicc` 做了什么，通常有一个选项可以打印出它的定义。在使用 *clang* 编译器的 Mac 上：

```
$$ mpicc -show
clang -fPIC -fstack-protector -fno-stack-check -Qunused-arguments -g3 -O0 -Wno-implicit-functi
```

Remark1 在 OpenMPI 中，这些命令默认是二进制可执行文件，但你可以通过在配置时传递 `--enable-script-wrapper-compilers` 选项将其变成一个脚本。

MPI 程序可以在许多不同的架构上运行。显然，你的目标（或至少是你的梦想）是在拥有十万处理器和高速网络的集群上运行你的代码。但也许你只有一个带有普通以太网的小型集群。或者你正坐在飞机上，只有你的笔记本电脑。MPI 程序可以在所有这些情况下运行 —— 当然是在你可用内存的限制范围内。

其工作方式是你不直接启动可执行文件，而是使用一个程序，通常称为 `mpiexec` 或类似名称，它会连接到所有可用的处理器并在那里启动你的可执行文件运行。因此，如果你的集群中有一千个节点，`mpiexec` 可以在每个节点上启动你的程序一次；如果你只有笔记本电脑，它可以在那里启动几个实例。在后一种情况下，你当然不会获得很好的性能，但至少你可以测试代码的正确性。

Python 注释 *1*：运行 *mpi4py* 程序。加载 TACC 提供的 python：

```
module load python
```

并运行它为：

```
ibrun python-mpi yourprogram.py
```

1.5 语言绑定

1.5.1 C

MPI 库是用 C 编写的。然而，标准小心地区分了 MPI 例程与它们的 *C* 绑定。事实上，从 MPI-4 开始，对于许多例程有两种绑定，取决于你是想要 4 字节整数，还是更大的整数。参见第 6.4 节，特别是 6.4.1。

1.5.2 C++，包括 MPL

C++ bindings 曾在标准中定义过，但它们被声明为已弃用，并且在 MPI-3 标准中已被正式移除。因此，可以通过包含 C++ 来使用 MPI

```
#include <mpi.h>
```

and using the C API.

The *boost* library 有自己的 MPI 版本，但似乎不再进行进一步开发。最近在惯用 C++ 支持方面的努力是 *Message Passing Layer(MPL)* https://github.com/rabauke/mpl。本书有 MPL 注释和命令的索引：第 57.4 节。

MPL note 1: Notes format. MPL 是一个仅包含头文件的 C++ 库。来自 MPL 的 MPI 使用注释将以这种方式标示。

1.5.3 Fortran

Fortran note 1: Formatting of Fortran notes. Fortran 特定的注释将以这样的注释形式标示。

传统上，*Fortran* 绑定 对于 MPI 看起来非常像 C 绑定，除了每个例程都有一个最终的 错误返回 参数。你会发现很多 Fortran 中的 MPI 代码都符合这一点。

然而，在 *MPI 3* 标准中，建议提供 Fortran 接口的 MPI 实现提供一个名为 `mpi_f08` 的模块，该模块可以在 Fortran 程序中使用。这包含以下改进：

- 这定义了 MPI 例程具有一个可选的最终错误参数。
- 使用 `mpi_f08` 模块有一些明显的影响，主要与某些 “MPI 数据类型” 有关，例如 `MPI_Comm`，这些类型之前声明为 `Integer`，现在是 Fortran `Type` 类型。详情见以下章节：Communicator 7.1，Datatype 6.1，Info 15.1.1，Op 3.10.2，Request 4.2.1，Status 4.3，Window9.1。
- 该 `mpi_f08` 模块解决了之前 *Fortran90* 绑定的问题：Fortran90 是强类型语言，因此无法像 C/C++ 那样通过引用传递参数地址，`void*` 用于发送和接收缓冲区的类型。这通过为每种数据类型提供单独的例程并在 MPI 模块中提供一个 `Interface` 块来解决。如果你请求了不存在的版本，编译器会显示类似的消息
  `There is no matching specific subroutine for this generic subroutine call [MPI_Send]`。详情见 http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node409.htm。

1.5.4 Python

Python 注释 *2*：*Python* 注释。Python 特定的注释将以这样的注释形式标出。

Python 绑定的包不是由 MPI 标准委员会定义的。相反，它是个人 Lisandro Dalcin 的工作成果。

在某种程度上，Python 接口是最优雅的。它使用面向对象（OO）技术，如对象上的方法，以及许多默认参数。

Python 绑定的一个显著特点是许多通信例程存在两种变体：

- 一种版本可以发送任意 Python 对象。这些例程的名称为小写，如 `bcast`；以及
- 一个发送 *numpy* 对象的版本；这些例程的名称如 `Bcast`。它们的语法可能略有不同。

第一个版本看起来更 “pythonic”，更容易编写，并且可以执行诸如发送 python 对象之类的操作，但由于数据通过 `pickle` 进行打包和解包，它的效率明显较低。作为常识性指导，在性能关键的代码部分使用 `numpy` 接口，而仅在设置阶段的复杂操作中使用 pythonic 接口。

带有 `mpi4py` 的代码可以通过 Swig 或转换例程与其他语言接口。

`numpy` 中的数据可以指定为简单对象，或 `[data, (count,displ), datatype]`。

1.5.5 如何阅读例程签名

在本书的 MPI 部分，我们将给出例程的参考语法。通常包括：

- 语义：例程名称和参数列表及其含义。
- C 语法：例程定义在 `mpi.h` 文件中的形式。

- Fortran 语法：带参数的例程定义，给出输入 / 输出说明。
- Python 语法：例程名称，指明其适用的类，以及参数，指明哪些是可选的。

这些 “ 例程签名 ” 看起来像代码，但它们不是！下面是如何翻译它们。

1.5.5.1 C

MPI 中典型的 C 例程规范如下所示：

```
int MPI_Comm_size(MPI_Comm comm,int *nprocs)
```

这意味着

- 该例程返回一个 `int` 参数。严格来说，你应该针对 `MPI_SUCCESS` (有关所有错误代码，请参见第 15.2.1 节)：

  ```
  MPI_Comm comm = MPI_COMM_WORLD;
  int nprocs;
  int errorcode;
  errorcode = MPI_Comm_size( MPI_COMM_WORLD,&nprocs);
  if (errorcode!=MPI_SUCCESS) {
    printf("Routine MPI_Comm_size failed! code=%d\n",
           errorcode);
    return 1;
  }
  ```

  然而，错误代码几乎没有用处，而且你的程序几乎无法从错误中恢复。大多数人调用该例程的方式是

  ```
  MPI_Comm_size( /* parameter ... */ );
  ```

  有关错误处理的更多内容，请参见第 15.2 节。
- 第一个参数的类型是 `MPI_Comm`。这不是 C 语言的内置数据类型，但它的行为类似于内置类型。MPI 中有许多这样的 `MPI_something` 数据类型。因此你可以写：

  ```
  MPI_Comm my_comm =
      MPI_COMM_WORLD; // using a predefined value
  MPI_Comm_size( comm, /* remaining parameters */ );
  ```
- 最后，有一个 “ 星号 ” 参数。这意味着该例程需要一个地址，而不是一个值。你通常会写成：

  ```
  MPI_Comm my_comm = MPI_COMM_WORLD; // using a predefined value
  int nprocs;
  MPI_Comm_size( comm, &nprocs );
  ```

  看到 “ 星号 ” 参数通常意味着：该例程有一个数组参数，或者：该例程内部设置了一个变量的值。这里是后一种情况。

1.5.5.2 Fortran

Fortran 规范如下：

```
MPI_Comm_size(comm, size, ierror)
Type(MPI_Comm), Intent(In) :: comm
Integer, Intent(Out) :: size
Integer, Optional, Intent(Out) :: ierror
```

或者用于 Fortran90 传统模式：

```
MPI_Comm_size(comm, size, ierror)INTEGER,
 INTENT(IN) :: commINTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

使用此例程的语法接近于此规范：你写

```
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
! legacy: Integer :: comm = MPI_COMM_WORLD
Integer :: comm = MPI_COMM_WORLDInteger :: size,ierr
CALL MPI_Comm_size( comm, size ) ! without the optional ierr
```

- 大多数 Fortran 例程的参数与相应的 C 例程相同，只是它们都将错误代码作为最后一个参数，而不是作为函数结果。与 C 一样，你可以忽略该参数的值。只是不要忘记它。
- 参数的类型在规范中给出。
- 当 C 例程有 `MPI_Comm` 和 `MPI_Request` 等参数时，Fortran 有 `INTEGER` 参数，或者有时是整数数组。

1.5.5.3 Python

The Pyt[hon interf] ace to MPI 使用类和对象。因此，a specificat ion like:

```
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

should be parsed as follows.

- First of all, you need the `MPI` class:

  ```
  from mpi4py import MPI
  ```

- Next, you need a `Comm` object. Often you will use the predefined communicator

  ```
  comm = MPI.COMM_WORLD
  ```

- 关键字 `self` 表示实际例程 `Send` 是 `Comm` 对象的一个方法，因此你调用：

  ```
  comm.Send( .... )
  ```

- 单独列出的参数，如 `buf`，是位置参数。带有类型列出的参数，如 `int dest` 是关键字参数。指定了值的关键字参数，如 `int tag=0` 是可选的，默认值已指明。因此，该例程的典型调用是：

  ```
  comm.Send(sendbuf,dest=other)
  ```

将发送缓冲区指定为位置参数，目标指定为关键字参数，并对可选的 tag 使用默认值。

一些 python 例程是 ‘类方法’，它们的定义缺少 `self` 关键字。例如：

```
MPI.Request.Waitall(type cls, requests, statuses=None)
```

将被用作

```
MPI.Request.Waitall(requests)
```

1.6 复习

复习 1.1. 是什么决定了 MPI 任务的并行度?

1. 你运行的集群规模。
2. 每个集群节点的核心数。3. MPI 启动器的参数（`mpiexec`, `ibrun`,... ）

复习 1.2. 判断题：你笔记本电脑的核心数限制了你能启动多少个 MPI 进程。

复习 1.3. 以下语言是否有面向对象的 MPI 接口？以何种意义?

1. C 2. C++3. Fortran2008
4. Python

第 2 章

MPI 主题：功能并行

2.1 TheSPMD 模型

MPI 程序大体上遵循单程序多数据（ Single Program Multiple Data，SPMD ）模型，其中每个处理器运行相同的可执行文件。我们称这个正在运行的可执行文件为进程。

当 MPI 在 20 年前首次编写时，处理器的定义非常明确：它是某人桌面上的计算机或机架中的计算机。如果这台计算机是网络集群的一部分，你称它为节点。因此，如果你运行一个 MPI 程序，每个节点将有一个 MPI 进程；见图 2.1。当然你也可以运行

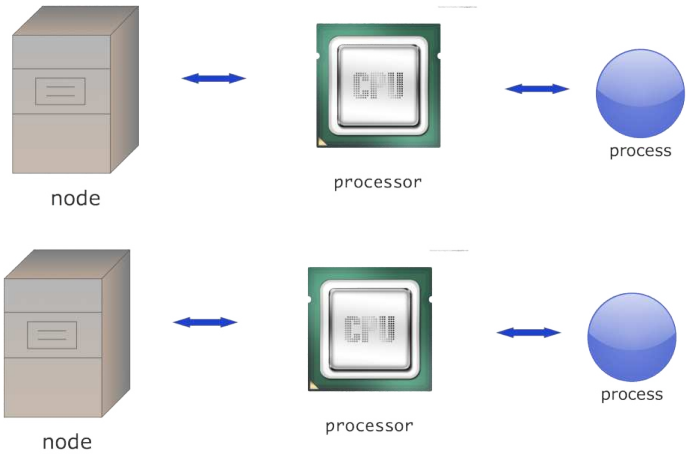


图 2.1：1990 年代中期的集群结构

多个进程，使用操作系统（ OS ）的时间片切换，但这不会带来额外的性能提升。

如今情况更加复杂。你仍然可以谈论集群中的一个节点，但现在一个节点可以包含多个处理器芯片（有时称为 *socket* ），而且每个处理器芯片可能有多个核心。图 2.2 展示了如何通过节点间的 MPI 和节点上的共享内存编程系统的混合来探索这一点。

然而，由于每个核心都可以像独立的处理器一样工作，你也可以在每个节点上运行多个 MPI 进程。对于 MPI 来说，核心看起来就像以前完全独立的处理器。这就是 “ 纯 MPI” 模型。

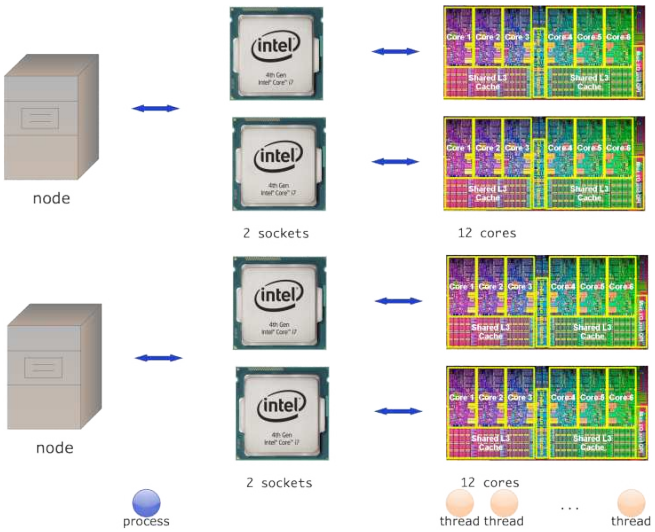


图 2.2：混合集群结构

图 2.3，我们将在本书的大部分内容中使用。（混合计算将在第 45 章讨论。）

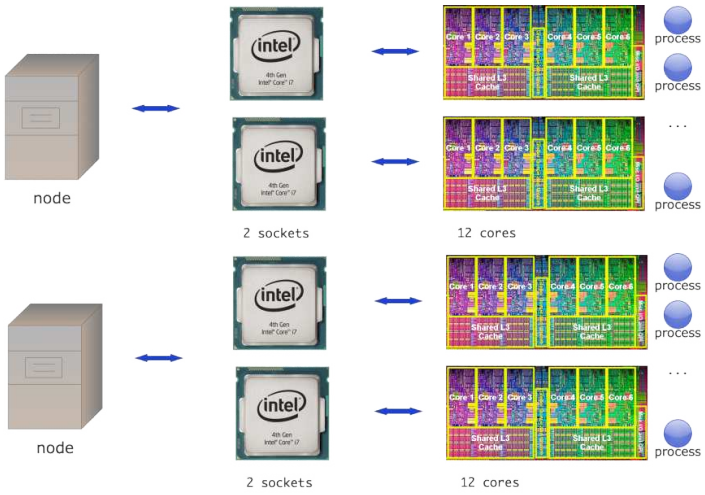


图 2.3：仅 MPI 集群结构

这有点令人困惑：旧处理器需要 MPI 编程，因为它们在物理上是分离的。另一方面，现代处理器上的核心共享相同的内存，甚至共享一些缓存。在其基本模式下，MPI 似乎忽略了所有这些：每个核心接收一个 MPI 进程，程序员无论另一个进程位于何处，都编写相同的发送 / 接收调用。实际上，你无法立即看出两个核心是在同一节点上还是不同节点上。当然，在实现层面，MPI 会根据核心是否

在同一插槽上或不同节点上，因此您无需担心效率不足的问题。

备注 2 在某些罕见情况下，您可能希望以 多程序多数据（ MPMD ） 模式运行，而不是 *SPMD*。这可以通过操作系统级别实现（参见第 *15.9.4* 节），使用 `mpiexec` 机制的选项，或者您可以使用 *MPI* 内置的进程管理；详见第 *8* 章。正如我所说，这仅涉及罕见情况。

2.2 启动和运行 MPI 进程

SPMD 模型起初可能令人困惑。尽管只有一个源代码，编译成一个可执行文件，但并行运行包含多个独立启动的 MPI 进程（参见第 1.3 节，了解机制）。

以下练习旨在帮助您直观理解这种一源多进程的设置。在第一个练习中，您将看到启动 MPI 程序的机制会启动独立的副本。源代码中没有任何内容表示 “ 现在你变成并行了 ”。

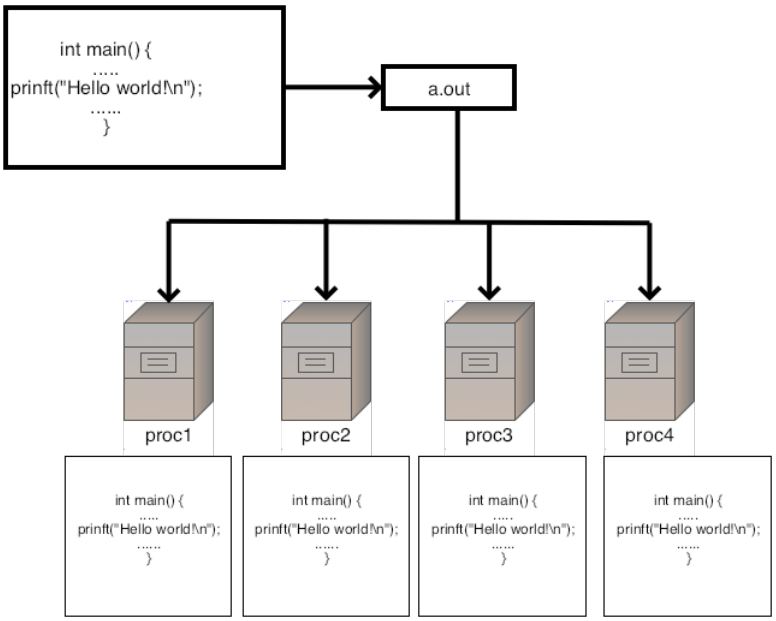


图 2.4：并行运行一个 hello world 程序

The following exercise demonstrates this point.

练习 2.1. 编写一个不包含任何 MPI 的 “hello world” 程序，并使用 `mpiexec` 或您的本地等效工具并行运行它。解释输出结果。（本练习的骨架代码名为 `hello`。）

本练习在图 2.4 中有说明。

图 2.1 `MPI_Init`

Name	Param name	Explanation	C type	F type	inout

```
MPI_Init (
       )
```

2.2.1 头文件

如果在程序文件中使用 MPI 命令，务必包含适当的头文件， `mpi.h` 针对 C/C++。

```
#include "mpi.h" // for C
```

这些文件的内部结构在不同的 MPI 安装之间可能不同，因此不能用一个 *mpi.h* 文件编译一个文件，而用另一个 MPI 在同一台机器上用相同的编译器编译另一个文件。

Fortran 注释 2：MPI 模块。对于 Fortran 中的 MPI 使用，请使用 MPI 模块。.

```
use mpi     ! pre 3.0
use mpi_f08 ! 3.0 standard
```

新的语言发展，例如大量计数；章节 6.4.2 将仅包含在 `mpi_f08` 模块，而不是早期机制中的。

头文件 `mpif.h` 自 MPI-4.1 起已弃用：安装可能支持它，但强烈不建议这样做。

Python 注释 *3*：导入 *mpi* 模块。 这最简单

```
from mpi4py import MPI
```

MPL note 2: Header file. 要编译 MPL 程序，添加一行

```
#include <mpl/mpl.hpp>
```

to your file. You need to add a path to your compile line:

```
mpicxx -o mpiprog -I${MPL_LOCATION}/include mympiprog.cpp
```

其中 *MPL_LOCATION* 是系统相关的。

2.2.2 初始化 / 终结

每个（有用的） MPI 程序都必须以 *MPI* 初始化 开始，通过调用 `MPI_Init` (图 2.1)，并且必须有 `MPI_Finalize` (图 2.2) 来结束程序中对 MPI 的使用。不同语言中的初始化调用有所不同。

In C, you can pass `argc` and `argv`, the arguments of a C language main program:

```
int main(int argc,char **argv) {....
return 0;}
```

图 2.2 MPI_Finalize

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Finalize	( )				

(允许为这些参数传递 *NULL* 。)

Fortran(before2008) lacks this commandline argument handling, so `MPI_Init` 缺少这些参数。

After `MPI_Finalize` 不能调用任何 MPI 例程（有少数例外，如 `MPI_Finalized` ）。特别是，不允许再次调用 `MPI_Init` 。如果你想这样做，请使用 会话模型；详见第 8.3 节。

Python 注释 *4*：初始化 / 终结。 在许多情况下，不需要调用初始化和终结：语句

```
## mpi.py
from mpi4py import MPI
```

执行初始化。同样，finalize 在程序结束时发生。

但是，对于特殊情况，有一个 *mpi4py.rc* 对象可以设置在 importing*mpi4py* 和 importing *mpi4py.MPI* 之间：

```
import mpi4py
mpi4py.rc.initialize = False
mpi4py.rc.finalize = False
from mpi4py import MPI
MPI.Init()
# stuff
MPI.Finalize()
```

MPL note 3: Init, finalize. 没有初始化或 finalize 调用。

Implementation note: 初始化在第一次 `mpl::environment` 方法调用时完成，例如 `comm_world`。

这看起来有点像声明 “ 这是程序的并行部分 ”，但事实并非如此：同样，整个代码是多次并行执行的。

练习 2.2. 将命令 `MPI_Init` 和 `MPI_Finalize` 添加到你的代码中。在代码中放置三个不同的打印语句：一个在 init 之前，一个在 init 和 finalize 之间，一个在 finalize 之后。再次解释输出。

备注 3 对于混合 *MPI* 加线程编程，还有一个调用 *MPI_Init_thread*。有关内容，请参见第 *13.1* 节。

2.2.2.1 中止 *MPIrun*

除了 `MPI_Finalize`，它表示 MPI 运行成功结束外，可以通过 `MPI_Abort` (图 2.3) 强制异常结束运行。这会停止与通信器相关的所有进程的执行，但许多实现会终止所有进程。该 *value* 参数会返回给环境。

图 2.3 MPI_Abort

名称	参数名	说明	C 类型	F 类型	i nout
MPI_Abort (					
	comm	communicator of MPI processes to abort	MPI_Comm	TYPE (MPI_Comm)	IN
	errorcode	error code to return to invoking environment	int	INTEGER	IN
)					

MPL:

```
void mpl::communicator::abort ( int ) const
```

Python:

```
MPI.Comm.Abort(self, int errorcode=0)
```

图 2.4 MPI_Initialized

Name	Param name	说明	C 类型	F 类型	输入输 出
MPI_Initialized (					
	flag	Flag is true if MPI_INIT has been called and false otherwise	int*	LOGICAL	OUT
)					

代码：

```
// return.c
MPI_Abort(MPI_COMM_WORLD,17);
```

Output:

```
mpicc -o return return.o
mpirun -n 1 ./return ; \
        echo "MPI program
    ↪return code $?"
application called
    ↪MPI_Abort(MPI_COMM_WORLD,
    ↪17) - process 0
MPI program return code 17
```

2.2.2.2 测试初始化 / 终止状态

命令行参数 *argc* 和 *argv* 只保证传递给进程零，因此传递命令行信息的最佳方式是通过广播（章节 3.3.3 ）。

有一些命令，比如 MPI_Get_processor_name，允许在 MPI_Init 之前调用。

如果 MPI 在库中使用，MPI 可能已经在主程序中初始化。因此，可以通过 MPI_Initialized 测试是否调用了 MPI_Init （图 2.4 ）。

你可以通过 MPI_Finalized 测试是否调用了 MPI_Finalize （图 2.5 ）。

图 2.5 MPI_Finalized

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Finalized (					
	flag	true if MPI was finalized	int*	LOGICAL	OUT
)					

2.2.2.3 运行信息

一旦 MPI 被初始化，MPI_INFO_ENV 对象包含许多描述运行特定信息的键 / 值对；参见章节 15.1.1.1。

2.2.2.4 命令行参数

The MPI_Init 例程接受对 argc 和 argv 的引用，原因如下：MPI_Init 调用会过滤掉传递给 *mpirun* 或 mpiexec 的参数，从而降低 *argc* 的值并消除部分 *argv* 参数。

另一方面，传递给 mpiexec 的命令行参数最终会作为一组键 / 值对存储在 MPI_INFO_ENV 对象中；参见第 15.1.1 节。

2.3 处理器识别

由于 MPI 作业中的所有进程都是同一可执行文件的实例，你可能会认为它们都执行完全相同的指令，这样其实用性不大。现在你将学习如何区分不同的进程，以便它们能够协同开始执行有用的工作。

2.3.1 处理器名称

在以下练习中，你将使用 MPI_Get_processor_name<style id='4'} 打印出每个 MPI 进程的主机名（见图 2.6），作为区分进程的第一种方法。该例程有一个字符缓冲区参数，需要你分配。缓冲区的长度也会被传入，返回时该参数包含实际使用的长度。最大所需长度为 MPI_MAX_PROCESSOR_NAME。

图 2.6 MPI_Get_processor_name

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Get_processor_name (					
	name	A unique specifier for the actual (as opposed to virtual) node.	char*	CHARACTER	OUT
	resultlen	Length (in printable characters) of the result returned in name	int*	INTEGER	OUT
)					

Python:

```
MPI.Get_processor_name()
```

Code:

```
// procname.cint name_length = MPI_MAX_PROCESSOR_NAME;
char proc_name[name_length];
MPI_Get_processor_name(proc_name,&name_length);
printf("Process %d/%d is running on node <<%s>>\n",
procid,nprocs,proc_name);
```

输出：

```
make[3]: `procname' is up to
    ↪date.
TACC:  Starting up job
    ↪4328411
TACC:  Starting parallel
    ↪tasks...
This process is running on
    ↪node
    ↪<<c205-036.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-036.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-036.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-036.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-036.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-035.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-035.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-035.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-035.frontera.tacc.utexas.edu>>
This process is running on
    ↪node
    ↪<<c205-035.frontera.tacc.utexas.edu>>
TACC:  Shutdown complete.
    ↪Exiting.
```

（缓冲区分配不足不会导致运行时错误。）

Fortrannote 3: 处理器名称。 分配一个 `Character` 变量，长度合适。返回的长度参数值可以帮助打印结果：

代码：

```
!! procname.F90
  Character(len=MPI_MAX_PROCESSOR_NAME) :: proc_name
  Integer :: len
  len = MPI_MAX_PROCESSOR_NAME
  call MPI_Get_processor_name(proc_name,len)
  print *,"Proc",procid,"runs on ",proc_name(1:len),"."
```

输出：

```
Proc           1 runs on
   ↪c202-010.frontera.tacc.utexas.edu.
Proc           3 runs on
   ↪c202-011.frontera.tacc.utexas.edu.
Proc           0 runs on
   ↪c202-010.frontera.tacc.utexas.edu.
Proc           2 runs on
   ↪c202-011.frontera.tacc.utexas.edu.
```

练习 2.3. 使用命令 `MPI_Get_processor_name`。确认你能够运行一个使用两个不同节点的程序。

MPL 注释 *4*：处理器名称。 调用 `processor_name` 是一个环境方法，返回一个 `std::string`：

```
std::string mpl::environment::processor_name ();
```

2.3.2 通信器

首先我们需要介绍 *communicator* 的概念，它是对一组进程的抽象描述。目前你只需要知道 `MPI_Comm` 数据类型的存在，以及有一个预定义的通信器 `MPI_COMM_WORLD`，它描述了你并行运行中涉及的所有进程。

在过程式语言 C 中，*communicator* 是一个传递给大多数例程的变量：

```
#include <mpi.h>
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Send( /* stuff */ comm );
```

Fortran 注释 *4*：通信器类型。在 Fortran 中，2008 年之前通信器是一个不透明句柄，存储在 `Integer` 中。随着 *Fortran 2008* 的引入，通信器成为派生类型：

```
use mpi_f098
Type(MPI_Comm} :: comm = MPI_COMM_WORLD
call MPI_Send( ... comm )
```

Pythonnote 5: Communicator objects. 在面向对象语言中，communicator 是一个对象，并且通常不是将其传递给例程，而是将例程作为 communicator 对象的方法：

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
comm.Send( buffer, target )
```

MPL note 5: Worldcommunicator. 声明 communicator 的简单方法是：

```
// commrank.cxx
mpl::communicator comm_world =
  mpl::environment::comm_world();
```

调用预定义的环境方法 `comm_world`.

然而，如果变量总是对应于 worldcommunicator，最好将其做成 `const` 并声明为引用：

```
const mpl::communicator &comm_world =
  mpl::environment::comm_world();
```

MPL 注释 *6*：*Communicator* 复制。communicator 类删除了其复制操作符；然而，存在复制初始化：

```
// commcompare.cxx
const mpl::communicator &comm =
  mpl::environment::comm_world();
cout << "same: " << boolalpha << (comm==comm) << endl;

mpl::communicator copy =
  mpl::environment::comm_world();
cout << "copy: " << boolalpha << (comm==copy) << endl;

mpl::communicator init = comm;
cout << "init: " << boolalpha << (init==comm) << endl;
```

(This outputs true/false/false respectively.)

Implementation note: The copy initializer performs an `MPI_Comm_dup`.

MPLnote7: Communicator passing. 通过引用传递通信器以避免通信器重复：

```
// commpass.cxx
// BAD! this does a MPI_Comm_dup.
void comm_val( const mpl::communicator comm );

// correct!
void comm_ref( const mpl::communicator &comm );
```

你将在章节中学到更多关于通信器的内容 pter 7.

2.3.3 进程和通信器属性：秩和大小

为了区分通信器中的进程，MPI 提供了两种调用 s

1. `MPI_Comm_size` (图 2.7) 显示了所有进程的数量；2. `MPI_Comm_rank` (图 2.8) 表示调用此例程的进程编号。

如果每个进程执行 `MPI_Comm_size` 调用，它们都会得到相同的结果，即运行中的进程总数。另一方面，如果每个进程执行 `MPI_Comm_rank`，它们都会得到不同的结果，即它们自己唯一的编号，一个介于零到进程总数减一之间的整数。见图 2.5。换句话说，每个进程都可以知道 “ 我是 20 个进程中的第 5 个进程 ”。

练习 2.4. 编写一个程序，使每个进程打印一条消息，报告其编号以及进程总数：

图 2.7 MPI_Comm_size

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_size (					
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	size	number of processes in the group of comm	int*	INTEGER	OUT
)					

MPL:

```
int mpl::communicator::size ( ) const
```

Python:

```
MPI.Comm.Get_size(self)
```

图 2.8 MPI_Comm_rank

名称	参数名	说明	C 类型	F type	inout
MPI_Comm_rank (					
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	rank	rank of the calling process in group of comm	int*	INTEGER	OUT
)					

MPL:

```
int mpl::communicator::rank ( ) const
```

Python:

```
MPI.Comm.Get_rank(self)
```

> `Hello from process 2 out of 5!` 编写该程序的第二个版本，其中每个进程打开一个唯一的文件并写入内容。在某些集群上，如果处理器数量较多，这可能不建议使用，因为它可能会使文件系统过载。（该练习的骨架代码以 `commrank` 命名。）

练习 2.5. 编写一个程序，只有编号为零的进程报告总共有多少个进程。

在面向对象的 MPI 方法中，即 `mpi4py` 和 *MPL*，`MPI_Comm_rank` 和 `MPI_Comm_size` 例程是 communicator 类的方法：

Python note 6: Communicator rank and size. Rank 和 size 是 communicator 对象的方法。注意它们的名称与 MPI 标准名称略有不同。

```
comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
```

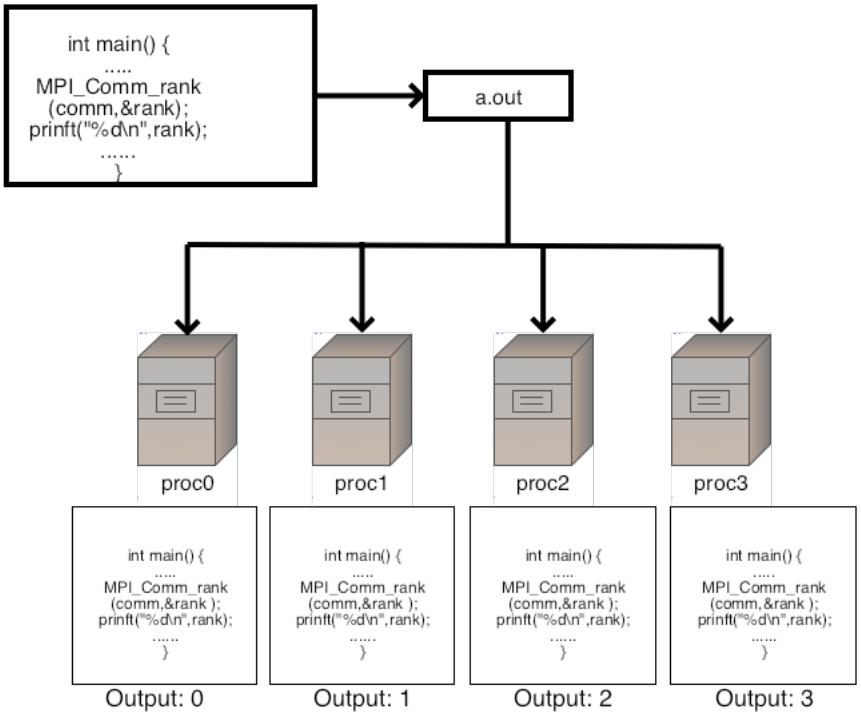


图 2.5：打印进程秩的并行程序

MPL 注释 *8*：秩和大小。进程的秩（通过 `mpl::communicator::rank` ）和通信器的大小（通过 `mpl::communicator::size` ）都是 `communicator` 类的方法：

```
const mpl::communicator &comm_world =
    mpl::environment::comm_world();
int procid = comm_world.rank();
int nprocs = comm_world.size();
```

2.4 功能性并行

现在进程可以相互区分，它们可以决定从事不同的活动。在极端情况下，你可能会有如下代码

```
// climate simulation:
if (procid==0)
  earth_model();
else if (procid=1)
  sea_model();
else
  air_model();
```

实际情况比这稍微复杂一些。但我们将开始探索基于进程编号决定其活动的进程这一概念。

能够区分进程已经足以编写一些应用程序，而无需了解任何其他 MPI 知识。我们将看一个简单的并行搜索算法：基于其 rank，处理器可以找到其搜索空间的部分。例如，在 *Monte Carlo codes* 中，会生成大量随机样本，并对每个样本执行一些计算。（这个特定的例子要求每个 MPI 进程运行一个独立的随机数生成器，这并非完全简单。）

Exercise 2.6. 数字 $N = 2,000,000,111$ 是质数吗？让每个进程测试一组不相交的整数，并打印出它们找到的任何因子。你不必测试所有整数 $< N$：任何因子最大为 $\sqrt{N} \approx 45,200$。（提示：`i%0` 可能会导致运行时错误。）你能找到多个解吗？（这个练习的骨架代码名为 `prime`。）

Remark4 通常，我们期望并行算法比顺序算法更快。现在考虑上述练习。假设我们测试的数字可以被某个小质数整除，但每个进程都有一大块数字需要测试。在这种情况下，顺序算法会比并行算法更快。值得深思。

作为另一个例子，在布尔可满足性 问题中，需要评估搜索空间中的多个点。知道一个进程的 rank 就足以让它生成自己负责的搜索空间部分。*Mandelbrot* 集 的计算也可以看作是功能性并行的一个案例。然而，构建的图像是需要保存在一个处理器上的数据，这打破了并行运行的对称性。

当然，在功能性并行运行结束时，你需要汇总结果，例如打印出某个总数。你将在接下来学习到实现这一点的机制。

2.5 分布式计算和分布式数据

使用 MPI 的一个原因是，有时你需要处理一个单一对象，比如一个向量或矩阵，其数据大小超过单个处理器内存所能容纳的范围。在分布式内存系统中，每个处理器获得整个数据结构的一部分，并且只处理那部分数据。

假设我们有一个大数组，并且我们想要将数据分布到各个处理器上。这意味着，对于 `p` 个进程和每个处理器 `n` 个元素，我们总共有 $\texttt{n}\cdot\texttt{p}$ 个元素。

在图 2.6 中，我们说 `data` 是一个 分布式数组 的局部部分，该数组的总大小为 $\texttt{n}\cdot\texttt{p}$ 个元素。然而，这个数组只是概念上的：每个处理器都有一个最低索引为零的数组，你必须自己将其转换为全局数组中的索引。换句话说，你必须以一种方式编写代码，使其表现得像是在处理一个分布在各个处理器上的大数组，而实际上只操作处理器上的局部数组。

你的典型代码看起来像这样

```
int myfirst = .....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
```

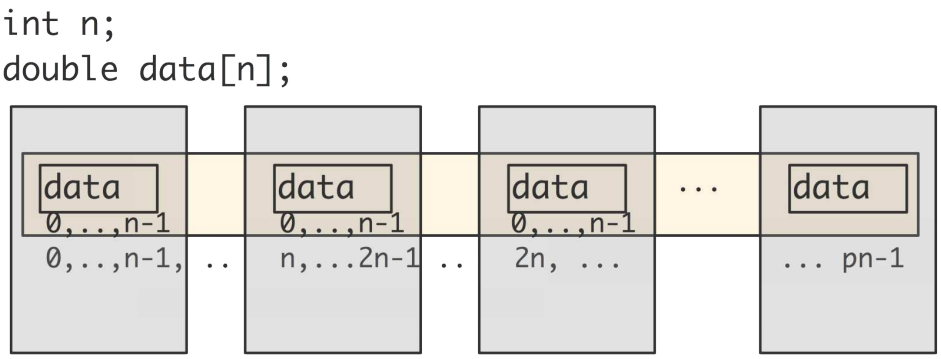


图 2.6：分布式数组的本地部分

```
    int iglobal = myfirst+ilocal;
    array[ilocal] = f(iglobal);
}
```

练习 2.7. 在每个进程上分配一个数组：`int my_ints[10];`，并填充它，使得进程 0 拥有整数 $0 \cdots 9$，进程 1 拥有 $10 \cdots 19$，依此类推。以不混乱的方式打印输出可能会比较困难。

如果数组大小不能被处理器数量整除，我们必须设计一个不均匀但不过分的划分。例如，你可以写成

```
int Nglobal, // is something large
    Nlocal = Nglobal/ntids,
    excess = Nglobal%ntids;
if (mytid==ntids-1)
  Nlocal += excess;
```

练习 2.8. 论证该策略不是最优的。你能想出更好的分配方案吗？负载均衡在 HPC 书籍第 2.10 节中有进一步讨论。

2.6 复习问题

对于所有判断题，如果你认为某个陈述是错误的，请给出一句简短的解释。

练习 2.9. 判断正误：`mpicc` 是一个编译器。

练习 2.10. 判断正误？1. 在 C 语言中，`MPI_Comm_rank` 是从零到 number-of-processes-minus-one，包含在内。2. 在 Fortran 中，`MPI_Comm_rank` 的结果是从一到 number-of-processes 的数字，包含在内。

Exercise2.11. hostfile 的作用是什么？

第 3 章

MPI 主题：集合操作

某一类 MPI 例程被称为 “ 集合操作 ”，或者更准确地说是 “ 在通信器上的集合操作 ”。这意味着如果该通信器中的进程 1 调用该例程，那么所有进程都需要调用该例程。本章将讨论关于合并该通信器中所有进程数据的集合例程，但也有一些操作如打开共享文件是集合操作，这将在后续章节中讨论。

3.1 使用全局信息

如果所有进程都有各自的数据，例如局部计算的结果，你可能想将这些信息汇总起来，比如找到最大计算值或所有值的总和。相反，有时一个处理器拥有需要与所有进程共享的信息。针对这类操作，MPI 提供了集合操作。

有多种情况，如图 3.1 所示，你可以通过考虑一些课堂活动来（某种程度上）理解这些情况：

- 老师告诉班级考试的时间。这是一个广播：相同的信息项传递给每个人。
- 考试结束后，老师执行一个收集操作来收集各个学生的试卷。
- 另一方面，当老师计算平均成绩时，每个学生有一个单独的分数，但这些分数现在被合并以计算一个单一的数值。这是一个归约。
- 现在老师有了一份成绩单，并将各自的成绩分发给每个学生。这是一个分散操作，其中一个进程拥有多个数据项，并将不同的数据项分发给所有其他进程。

这个故事与 MPI 进程中的情况略有不同，因为 MPI 进程更为对称；执行归约和广播的进程与其他进程没有区别。任何进程都可以作为此类集合操作中的根进程。

练习 3.1. 你将如何使用 MPI 集合操作实现以下场景？1. 让每个进程计算一个随机数。你想将这些数中的最大值打印到屏幕上。2. 每个进程再次计算一个随机数。现在你想用它们的最大值来缩放这些数。

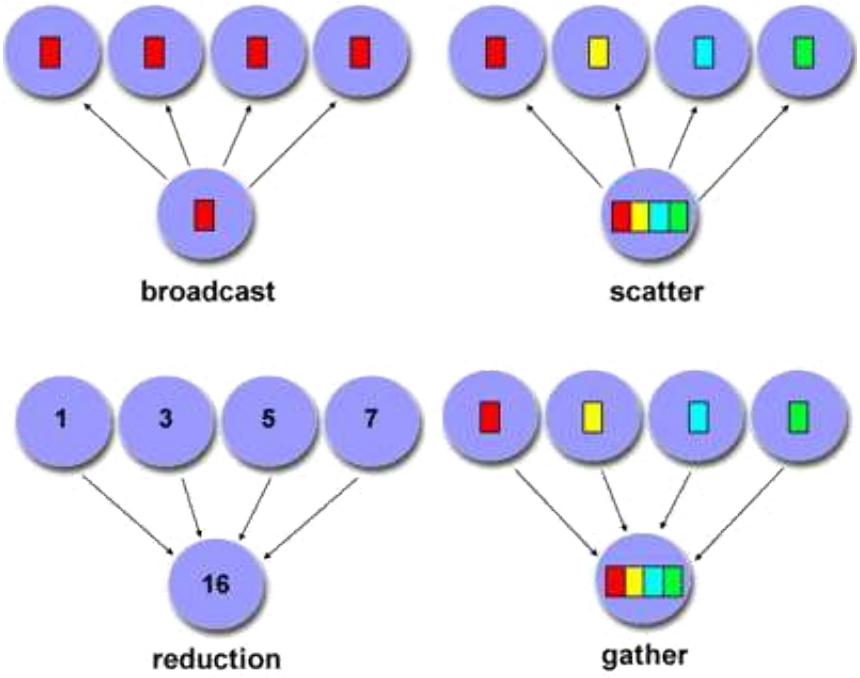


图 3.1：四种最常见的集合操作

3. 让每个进程计算一个随机数。你想打印出最大值是在哪个处理器上计算的。考虑你建议的时间和空间复杂度。

3.1.1 Practicaluse of collective s

集合操作在科学应用中非常常见。例如，如果一个进程从磁盘或命令行读取数据，它可以使用 broadcast 或 gather 将信息传递给其他进程。同样，在程序运行结束时，可以使用 gather 或 reduction 来收集关于程序运行的汇总信息。

然而，更常见的情况是集合操作的结果需要在所有进程上使用。

考虑计算标准差：

$$\sigma = \sqrt{\frac{1}{N-1}\sum_i^N (x_i-\mu)^2} \qquad \text{where} \qquad \mu = \frac{\sum_i^N x_i}{N}$$

并假设每个进程只存储一个 x_i 值。

1. 平均值 μ 的计算是一个归约操作，因为所有分布的值都需要相加。2. 现在每个进程需要计算其值 x_i 的 $x_i - \mu$，因此 μ 在所有地方都是必需的。你可以通过先做归约再广播来计算，但更好的是使用所谓的 *allreduce* 操作，它执行归约并将结果保留在所有处理器上。

3. $\sum_i (x_i - \mu)$ 的计算是另一个分布式数据的求和，因此我们需要另一个归约操作。根据每个进程是否需要知道 σ，我们可以再次使用 allreduce。

3.1.2 Synchronization

Collectives 是涉及通信子中所有进程的操作。Collective 是一次单独的调用，并且在所有处理器上都会阻塞，这意味着调用 collective 的进程不能继续执行，直到其他进程也同样调用了该 collective。

这并不意味着所有处理器会同时退出调用：由于实现细节和网络延迟，它们在执行时不必同步。然而，从语义上讲，我们可以说一个进程不能完成 collective，直到每个其他进程至少开始了该 collective。

除了这些 collective 操作之外，还有一些被称为 “ 在其通信子上是 collective” 的操作，但它们不涉及数据移动。Collective 的含义是所有处理器必须调用此例程；不这样做会导致错误，表现为代码 “ 挂起 ”。其中一个例子是 `MPI_File_open`。

3.1.3 Collectivesin MPI

我们将按以下顺序解释 MPI 集合操作。
Allreduce 我们使用 allreduce 作为介绍集合操作背后概念的切入点；章节 3.2。如上所述，该例程适用于许多实际场景。Broadcast 和 reduce 然后我们介绍 reduce （章节 3.3.1 ）和 broadcast （章节 3.3.3 ）集合操作中 root 的概念。

- 有时你需要带有部分结果的归约操作，其中每个处理器计算编号较低的处理器值的和（或其他操作）。为此，你可以使用 *scan* 集合操作（章节 3.4 ）。

Gather 和 scatter gather/scatter 集合操作处理多个数据项；章节 3.5。
还有更多集合操作或上述操作的变体。

- 如果每个处理器都需要向每个其他处理器广播，则使用 *all-to-all* 操作（第 3.6 节）。
- reduce-scatter 是一个较少为人知的集合操作组合；见第 3.7 节。
- barrier 是一种操作，使所有进程等待直到每个进程都到达屏障（第 3.8 节）。

- 如果你想收集或分散信息，但每个处理器的贡献大小不同，则有 “variable” 集合操作；它们的名字中带有 `v` （第 3.9 节）。

最后，还有一些集合操作中的高级主题。

- 用户定义的归约操作符；见第 3.10.2 节。
- 非阻塞集合操作；章节 3.11.
- 我们将在章节 3.12 中简要讨论集合操作的性能方面。
- 我们将在章节 3.13 中讨论同步方面。

图 3.1 MPI_Allreduce

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Allreduce (				
MPI_Allreduce_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in send buffer	int MPI_Count	INTEGER	IN
datatype	datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation	MPI_Op	TYPE(MPI_Op)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
template<typename T , typename F >
void mpl::communicator::allreduce
( F,const T &, T & ) const;( F,const T *, T *,
const contiguous_layout< T > & ) const;
( F,T & ) const;
( F,T *, const contiguous_layout< T > & ) const;
F : reduction functionT : typePython:


MPI.Comm.Allreduce(self, sendbuf, recvbuf, Op op=SUM)
```

3.2 归约

3.2.1 全部归约

上面我们看到了一些场景，其中一个量被归约，所有进程都获得结果。对应的 MPI 调用是 `MPI_Allreduce` (图 3.1)。

示例：我们给每个进程一个随机数，并将这些数字相加。结果应大约是进程数的 $1/2$ 倍。

```
// allreduce.c
float myrandom,sumrandom;
myrandom = (float) rand()/(float)RAND_MAX;
// add the random variables together
MPI_Allreduce(&myrandom,&sumrandom,
              1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx nprocs/2:
if (procno==nprocs-1)
  printf("Result %6.9f compared to .5\n",sumrandom/nprocs);
```

Or:

```
MPI_Count buffersize = 1000;double *indata,*outdata;
indata = (double*) malloc( buffersize*sizeof(double) );
outdata = (double*) malloc( buffersize*sizeof(double) )
                                                                    ;
MPI_Allreduce_c(indata,outdata,buffersize,MPI_DOUBLE,
MPI_SUM,MPI_COMM_WORLD);
```

3.2.1.1 缓冲区描述

这是本课程中第一个涉及 MPI 数据缓冲区的示例： `MPI_Allreduce` 调用包含两个缓冲区参数。在大多数 MPI 调用中（单边通信是一个很大的例外），缓冲区由三个参数描述：

1. 指向数据的指针，2. 缓冲区中的元素数量，3. 缓冲区中元素的数据类型。

每个都需要一些说明。

1. 缓冲区规范取决于编程语言。默认值见第 3.2.4 节。2. 在 MPI 标准中，count 是一个 4 字节整数，直到并包括 MPI-3。在 MPI-4 标准中， `MPI_Count` 数据类型变为允许。详见第 6.4 节。3. 数据类型可以是预定义的，如上例所示，也可以是用户定义的。详见第 6 章。

备注 5 同时具有发送和接收缓冲区的例程不应使它们别名。相反，参见 *MPI_IN_PLACE* 的讨论；第 *3.3.2* 节。

3.2.1.2 Examples and exercises

练习 3.2. 让每个进程计算一个随机数，并使用 `MPI_Allreduce` 例程计算这些数字的和。

$$\xi = \sum_i x_i$$

然后每个进程将其值按该和进行缩放。

$$x_i' \leftarrow x_i/\xi$$

计算缩放后数字的和

$$\xi' = \sum_i x_i'$$

and check that it is 1.
（本练习的骨架文件名为 `randommax`。）

Exercise 3.3. 实现一个（非常简单的）傅里叶变换：如果 f 是区间 $[0,1]$，上的一个函数，那么第 n 个傅里叶系数是

$$f_n \hat{=} \int_0^1 f(t)e^{-2\pi x}\,dx$$

我们用以下方式近似

$$f_n \hat{=} \sum_{i=0}^{N-1} f(ih)e^{-in\pi/N}$$

- 为 e^{-inh} 系数创建一个分布式数组，
- 为 $f(ih)$ 值创建一个分布式数组
- 计算几个系数

练习 3.4. 在前面的练习中，你处理了一个分布式数组，计算了一个局部量并将其合并成一个全局量。为什么将整个分布式数组集中到单个处理器上并在本地完成所有计算不是一个好主意?

MPL 注释 *9*：*Allreduce* 操作符。 常用的归约操作符以模板操作符的形式给出：

```
float
    xrank = static_cast<float>( comm_world.rank() ),
xreduce;// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
// in place
comm_world.allreduce(mpl::plus<float>(), xrank);
```

注意操作符后面的括号。还要注意操作符是放在前面，而不是最后。

可用的有： `max`, `min`, `plus`, `multiplies`, `logical_and`, `logical_or`, `logical_xor`, `bit_and`, `bit_or`, `bit_xor`。

实现说明：归约操作符必须与 `T(T,T)>` 兼容。

有关操作符的更多内容，请参见第 3.10 节。

3.2.2 作为 allreduce 的 Innerproduct

归约操作的一个更常见的应用是内积计算。通常，你有两个向量 x,y，它们具有相同的分布，也就是说，所有进程存储 x 和 y的相等部分。计算过程为

```
local_inprod = 0;
for (i=0; i<localsize; i++)
  local_inprod += x[i]*y[i];
MPI_Allreduce( &local_inprod, &global_inprod, 1,MPI_DOUBLE ... )
```

练习 3.5. *Gram-Schmidt* 方法是正交化两个向量的一种简单方法：

$$u \leftarrow u - (u^t v)/(u^t u)$$

实现此方法，并检查结果确实是正交的。建议：用 $\sin 2nh\pi$ 填充 v，其中 $n = 2\pi/N$，用 $\sin 2nh\pi + \sin 4nh\pi$ 填充 u。正交化后 u 变成什么？

3.2.3 Reduction operations

几个 `MPI_Op` 值是预定义的。有关列表，请参见章节 3.10.1。

对于归约和扫描操作，可以定义您自己的操作符。

```
MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
```

更多细节，请参见章节 3.10.2。

3.2.4 数据缓冲区

集合操作是您看到的第一个涉及用户数据传输的 MPI 例程。在这里，以及其他所有情况下，您会看到数据描述包括：

- 一个缓冲区。它可以是标量或数组。
- 一种数据类型。这描述了缓冲区中是整数、单精度 / 双精度浮点数，还是稍后讨论的更复杂类型。
- 一个计数。这说明发送缓冲区中给定数据类型的元素数量，或者将接收到接收缓冲区中的元素数量。

这三者共同描述了 MPI 需要通过网络发送的内容。

在各种语言中，这样的缓冲区 / 计数 / 数据类型三元组以不同方式指定。

首先，在 *C* 语言中，缓冲区 总是一个 不透明句柄，即一个 `void*` 参数，你需要为其提供一个地址。这意味着 MPI 调用可以有两种形式。

对于标量，我们需要使用取地址符号 (&) 来获取地址：

```
float x,y;
MPI_Allreduce( &x,&y,1,MPI_FLOAT, ... );
```

但对于数组，我们利用数组和地址在以下方面或多或少是等价的事实：

```
float xx[2],yy[2];
MPI_Allreduce( xx,yy,2,MPI_FLOAT, ... );
```

你可以强制转换缓冲区并写成：

```
MPI_Allreduce( (void*)&x,(void*)&y,1,MPI_FLOAT, ... );
MPI_Allreduce( (void*)xx,(void*)yy,2,MPI_FLOAT, ... );
```

但这不是必须的。如果省略强制类型转换，编译器也不会报错。

C++ 注释 *1*：缓冲区处理。在 C++ 中标量的处理方式与 C 相同。然而，对于数组，你可以选择 C 风格数组，或者 `std::vector` 或 `std::array`。对于后者，有两种处理缓冲区的方法：

```
vector<float> xx(25);
MPI_Send( xx.data(),25,MPI_FLOAT, .... );
MPI_Send( &xx[0],25,MPI_FLOAT, .... );
```

Fortran 注释 *5*：*MPI* 发送 / 接收缓冲区。在 *Fortran* 中，参数总是通过引用传递，因此 缓冲区的处理方式相同：

```
Real*4 :: xReal*4,dimension(2) :: xx
call MPI_Allreduce( x,1,MPI_REAL4, ... )
call MPI_Allreduce( xx,2,MPI_REAL4, ... )
```

在讨论面向对象语言时，我们首先注意到官方的 C++ 应用程序编程接口（ API ）已从标准中移除。

在面向对象语言中，不需要显式指定缓冲区 / 计数 / 数据类型三元组。

Python 注释 *7*：来自 *numpy* 的缓冲区。Python 中的大多数 MPI 例程都有两种变体，一种可以发送任意 Python 数据，另一种基于 *numpy* 数组。前者看起来最 “Pythonic”，且非常灵活，但通常效率明显较低。

```
## allreduce.py
random_number = random.randint(1,random_bound)
# native mode send
max_random = comm.allreduce(random_number,op=MPI.MAX)
```

> 在 numpy 变体中，所有缓冲区都是 *numpy* 对象，携带有关其类型和大小的信息。对于标量归约，这意味着我们必须为接收缓冲区创建一个数组，尽管只使用了一个元素。

```
myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number
allrandom = np.empty(nprocs,dtype=int)
# numpy mode send
comm.Allreduce(myrandom,allrandom[:1],op=MPI.MAX)
```

Python note 8: Buffers fromsubarrays. 在许多示例中，您将传递整个 Numpy 数组作为发送 / 接收缓冲区。如果想使用对应于数组子集的缓冲区，可以使用以下表示法：

```
MPI_Whatever( buffer[...,5] # more stuff
```

> 用于传递从数组位置 5 开始的缓冲区。
>
> 对于更复杂的效果，使用 `numpy.frombuffer`：

Code:

```
## bcastcolumn.pydatatype = np.intc
elementsize = datatype().itemsize
typechar = datatype().dtype.char
buffer = np.zeros( [nprocs,nprocs], dtype=datatype)
buffer[:,:] = -1for proc in range(nprocs):
if procid==proc:buffer[proc,:] = proccomm.Bcast\
( [ np.frombuffer\( buffer.data,dtype=datatype,
offset=(proc*nprocs+proc)*elementsize ), nprocs-proc,
 typechar ], root=proc )
```

Output:

```
int size: 4
i
[[ 0  0  0  0  0  0]
 [-1  1  1  1  1  1]
 [-1 -1  2  2  2  2]
 [-1 -1 -1  3  3  3]
 [-1 -1 -1 -1  4  4]
 [ 5  5  5  5  5  5]]
```

MPL note 10: 标量缓冲区。缓冲区类型处理是通过多态（模板和 ADL ）完成的：无需显式指明类型。

标量按如下方式处理：

```
float x,y;comm.bcast( 0,x ); // note: root first
comm.allreduce( mpl::plus<float>(), x,y ); // op first
```

其中归约函数需要与缓冲区的类型兼容。

MPL 注释 *11*：向量缓冲区。如果你的缓冲区是一个 `std::vector`，你需要取它的 `.data()` 组件：

```
vector<float> xx(2),yy(2);comm.allreduce( mpl::plus<float>(),
xx.data(), yy.data(), mpl::contiguous_layout<float>(2) );
```

The `contiguous_layout` 是一个 “ 派生类型 ” ；这将在其他地方更详细地讨论（参见注释 44 及后续内容）。目前，将其解释为指示缓冲区规范中的计数 / 类型部分的一种方式。

MPL 注释 *12*：数组缓冲区。你可以传递一个 C 风格的数组作为缓冲区，这需要一个布局：

```
// collectarray.cxx
float rank2p2p1[2] = { 2*xrank,2*xrank+1 };
mpl::contiguous_layout<float> p2layout(2);
comm_world.allreduce(mpl::plus<float>(), rank2p2p1, p2layout);
```

MPL 注释 *13*：迭代器缓冲区。MPL 点对点例程有一种通过 `begin` 和 `end` 迭代器指定缓冲区的方法。

```
// sendrange.cxx
vector<double> v(15);
  comm_world.send(v.begin(), v.end(), 1);  // send to rank 1
  comm_world.recv(v.begin(), v.end(), 0);  // receive from rank 0
```

不适用于 collectives。

MPL note 14: Send vs recv buffer. 根 collectives 的非根使用有一个单独的变体：

```
// scangather.cxxif (procno==0) {comm_world.reduce
( mpl::plus<int>(),0,my_number_of_elements,
total_number_of_elements );} else {comm_world.reduce
( mpl::plus<int>(),0,my_number_of_elements );}
```

3.3 有根集合通信：广播，归约

In some scenarios there is a certain process that has a privileged status.

- 一个进程可以生成或读取程序运行的初始数据。然后需要将其 broadcasted to all other processes.
- 在程序运行结束时，通常需要一个进程输出一些汇总信息。

这个进程称为 *root* 进程，我们现在将考虑具有 root 的例程。

3.3.1 Reduce to a root

在广播操作中，单个数据项被传达给所有进程。带有 `MPI_Reduce` (图 3.2) 的归约操作则相反：每个进程都有一个数据项，这些数据项被汇聚成一个单一的数据项。

以下是归约操作的基本要素：

```
MPI_Reduce( senddata, recvdata..., operator,root,
 comm );
```

- 有原始数据，也有归约后的数据。MPI 的设计决定是默认不会覆盖原始数据。发送数据和接收数据的大小和类型相同：如果每个处理器有一个实数，归约结果仍然是一个实数。
- 可以显式地指明使用单个缓冲区，从而覆盖原始数据；详见第 3.3.2 节关于这种 “ 就地 ” 模式的说明。
- 有归约操作符。常用的选择有 `MPI_SUM`, `MPI_PROD` 和 `MPI_MAX`，但也存在复杂的操作符，比如寻找最大值的位置。（完整列表见第 3.10.1 节。）你也可以自定义操作符；详见第 3.10.2 节。
- 有一个根进程接收归约结果。由于非根进程不接收归约数据，它们实际上可以不定义接收缓冲区。

图 3.2 MPI_Reduce

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Reduce (				
MPI_Reduce_c (				
sendbuf	address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in send buffer	int / MPI_Count	INTEGER	IN
datatype	datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	reduce operation	MPI_Op	TYPE(MPI_Op)	IN
root	rank of root process	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
void mpl::communicator::reduce// root, in place
( F  f,int  root_rank,T &  sendrecvdata ) const
( F  f,int  root_rank,T *  sendrecvdata,const contiguous_layout< T > &  l ) const
// non-root( F  f,int  root_rank,const T &  senddata ) const( F  f,int  root_rank,
const T * senddata,const contiguous_layout< T > &  l ) const// general
( F  f,int  root_rank,const T &  senddata,T &  recvdata ) const( F  f,int  root_rank,
const T *  senddata,T *  recvdata,const contiguous_layout< T > &  l ) const
```

Python:

```
comm.Reduce(self, sendbuf, recvbuf, Op op=SUM, int root=0)
native:
comm.reduce(self, sendobj=None, recvobj=None, op=SUM, int root=0)
```

```
// reduce.c
float myrandom = (float) rand()/(float)RAND_MAX,
  result;
int target_proc = nprocs-1;
// add all the random variables together
MPI_Reduce(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,
           target_proc,comm);
// the result should be approx nprocs/2:
if (procno==target_proc)
  printf("Result %6.3f compared to nprocs/2=%5.2f\n",
         result,nprocs/2.);
```

练习 3.6. 编写一个程序，使每个进程计算一个随机数，且

进程 0 找到并打印生成的最大值。让每个进程打印它的值，以检查你的程序的正确性。

Collective operations 也可以接受数组参数，而不仅仅是标量。在这种情况下，操作会逐点应用于数组中的每个位置。

Exercise 3.7. 在每个进程上创建一个长度为 2 的整数数组，并在每个进程上放入值 1,2。对该数组执行求和归约。你能预测结果应该是什么吗？编写代码。你的预测正确吗？

3.3.2 Reduce in place

默认情况下，MPI 不会用归约结果覆盖原始数据，但你可以使用 `MPI_IN_PLACE` 指定符来告诉它这样做：

```
// allreduceinplace.c
for (int irand=0; irand<nrandoms; irand++)
    myrandoms[irand] = (float) rand()/(
  float)RAND_MAX;// add all the random variables together
MPI_Allreduce(MPI_IN_PLACE,myrandoms,nrandoms,MPI_FLOAT,
MPI_SUM,comm);
```

现在每个进程只有一个接收缓冲区，因此这有节省一半内存的优点。每个进程将其输入值放入接收缓冲区，这些值随后被归约结果覆盖。

上述示例使用了 `MPI_IN_PLACE` 在 `MPI_Allreduce` 中；在 `MPI_Reduce` 中则稍微复杂一些。其推理如下：

- 在 `MPI_Reduce` 中，每个进程都有一个用于贡献的缓冲区，但只有根进程需要接收缓冲区。因此，`MPI_IN_PLACE` 在除根进程外的任何处理器上都代替了接收缓冲区的位置 ……
- … 而根进程需要一个接收缓冲区，`MPI_IN_PLACE` 取代了发送缓冲区的位置。为了贡献它的值，根进程需要将其放入接收缓冲区。

这里是一种编写就地版本的方式 `MPI_Reduce`：

```
if (procno==root)MPI_Reduce(MPI_IN_PLACE,myrandoms,
nrandoms,MPI_FLOAT,MPI_SUM,root,comm);else
MPI_Reduce(myrandoms,MPI_IN_PLACE,nrandoms,MPI_FLOAT,
MPI_SUM,root,comm);
```

然而，作为一种风格问题，在条件的不同分支中拥有同一集合操作的不同版本是不合适的。以下写法可能更好：

```
float *sendbuf,*recvbuf;
if (procno==root) {
  sendbuf = MPI_IN_PLACE; recvbuf = myrandoms;
} else {
  sendbuf = myrandoms; recvbuf = MPI_IN_PLACE;
}
MPI_Reduce(sendbuf,recvbuf,
        nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

在 Fortran 中你不能进行这些地址计算。你可以使用带条件的解决方案：

```
!! reduceinplace.F90call random_number(mynumber)
target_proc = ntids-1;
! add all the random variables together
if (mytid.eq.target_proc) thenresult = mytid
call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
target_proc, comm)elsemynumber = mytid
call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
target_proc, comm)end if
```

但你也可以用指针来解决这个问题：

```
!! reduceinplaceptr.F90in_place_val = MPI_IN_PLACE
if (mytid.eq.target_proc) then! set pointers
result_ptr => resultmynumber_ptr => in_place_val
! target sets value in receive bufferresult_ptr = mytidelse
! set pointersmynumber_ptr => mynumber
result_ptr => in_place_val
! non-targets set value in send buffermynumber_ptr = mytid
end if
call MPI_Reduce(mynumber_ptr,result_ptr,1,MPI_REAL,MPI_SUM,&
target_proc, comm, err)
```

Python note 9: In-place collectives. 值 `MPI.IN_PLACE` 可用于发送缓冲区：

```
## allreduceinplace.py
myrandom = np.empty(1,dtype=int)
myrandom[0] = random_number

comm.Allreduce(MPI.IN_PLACE,myrandom,op=MPI.MAX)
```

MPL note 15: Reduce in place. 通过只指定一个而非两个缓冲区参数来激活就地变体。

```
floatxrank = static_cast<float>( comm_world.rank() ),
xreduce;// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
// in place
comm_world.allreduce(mpl::plus<float>(), xrank);
```

图 3.3 MPI_Bcast

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Bcast (				
MPI_Bcast_c (				
buffer	starting address of buffer	void*	TYPE(*), DIMENSION(..)	INOUT
count	number of entries in buffer	int MPI_Count	INTEGER	IN
datatype	datatype of buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
root	rank of broadcast root	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
template<typename T >
void mpl::communicator::bcast
   ( int  root, T &  data ) const
   ( int  root, T *  data, const layout< T > &  l ) const
```

Python:

```
MPI.Comm.Bcast(self, buf, int root=0)
```

减少缓冲区需要指定一个 `contiguous_layout`:

```
// collectbuffer.cxx
float
  xrank = static_cast<float>( comm_world.rank() );
vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
comm_world.allreduce
  (mpl::plus<float>(), rank2p2p1.data(),reduce2p2p1.data(),two_floats);
if ( iprint )
  cout << "Got: " << reduce2p2p1.at(0) << ","
       << reduce2p2p1.at(1) << endl;
```

注意缓冲区的类型是 `T *`，因此有必要获取任何 `std::vector` 等的 `data()`。

3.3.3 Broadcast

广播模拟了这样一种场景：一个进程，即 “ 根 ” 进程，拥有一些数据，并将其传达给所有其他进程。

广播例程 `MPI_Bcast` (图 3.3) 具有以下结构：

```
MPI_Bcast( data..., root , comm);
```

这里：

- 只有一个缓冲区，即发送缓冲区。调用之前，根进程在该缓冲区中有数据；其他进程分配一个同样大小的缓冲区，但对它们来说内容无关紧要。
- 根是发送其数据的进程。通常，它将是广播树的根。

示例：通常我们不能假设所有进程都能获得命令行参数，因此我们从进程 0 广播它们。

```
// init.c
if (procno==0) {
  if ( argc==1 || // the program is called without parameter
       ( argc>1 && !strcmp(argv[1],"-h") ) // user asked for help
       ) {
    printf("\nUsage: init [0-9]+\n");
    MPI_Abort(comm,1);
  }
  input_argument = atoi(argv[1]);
}
MPI_Bcast(&input_argument,1,MPI_INT,0,comm);
```

Python 注释 *10*：发送对象。在 python 中既可以发送对象，也可以发送更类似 C 的缓冲区。这两种可能性对应（见第 1.5.4 节）不同的例程名称；缓冲区必须被创建为 `numpy` 对象。

我们展示了通用的 Python 和 numpy 两种变体。在前者变体中，结果作为函数返回；在 numpy 变体中，发送缓冲区被重复使用。

```
## bcast.py
# first native
if procid==root:
    buffer = [ 5.0 ] * dsize
else:
    buffer = [ 0.0 ] * dsize
buffer = comm.bcast(obj=buffer,root=root)
if not reduce( lambda x,y:x and y,
               [ buffer[i]==5.0 for i in  range(len(buffer)) ] ):
    print( "Something wrong on proc %d: native buffer <<%s>>" \
           % (procid,str(buffer)) )

# then with NumPy
buffer = np.arange(dsize, dtype=np.float64)
if procid==root:
    for i in range(dsize):
        buffer[i] = 5.0
comm.Bcast( buffer,root=root )
if not all( buffer==5.0 ):
    print( "Something wrong on proc %d: numpy buffer <<%s>>" \
           % (procid,str(buffer)) )
else:
    if procid==root:
        print("Success.")
```

MPL note 16: Broadcast. 广播调用有两种变体，带标量参数和通用布局：

```
template<typename T >
void mpl::communicator::bcast
   ( int root_rank, T &data ) const;
void mpl::communicator::bcast
   ( int root_rank, T *data, const layout< T > &l ) const;
```

注意 root 参数放在第一位。

对于以下练习，请研究图 3.2。

练习 3.8. 用于求解线性系统的 *Gauss-Jordan* 算法（或计算其逆矩阵）运行如下：对于主元 $k = 1, \ldots, n$，设缩放向量为 $\ell_i^{(k)} = A_{ik}/A_{kk}$，针对行 $r \neq k$ 和列 $c = 1, \ldots, n$ $A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{kc}$，其中忽略了右侧向量的更新或逆矩阵的形成。设矩阵按列分布，每个进程存储一列。将 Gauss-Jordan 算法实现为一系列广播：在第 k 次迭代中，进程 k 计算并广播缩放向量 $\{\ell_i^{(k)}\}_i$。在所有处理器上复制右侧向量。（该练习的骨架代码名为 `jordan`。）

练习 3.9. 在你的 Gauss-Jordan 消元实现中添加部分主元选择。修改你的实现，使每个处理器存储多列，但每列仍只进行一次广播。有没有办法让每个处理器只进行一次广播?

3.4 扫描操作

T该 `MPI_Scan` 操作也执行归约，但它保留部分结果。也就是说，如果处理器 i c包含一个数字 x_i，且 $\oplus$ 是一个操作符，那么扫描操作会将 $x_0 \oplus \cdots \oplus x_i$ 保留在处理器 i 上。这 t种操作通常称为 前缀操作；参见 HPC 书籍，第 -28 节。

该 `MPI_Scan`（图 3.4）例程是一个 包含扫描操作，意味着它包括进程自身的数据； `MPI_Exscan`（参见第 3.4.1 节）是排他扫描，不包括调用进程上的数据。

process :	0	1	2	$\cdots$	$p-1$
data :	x_0	x_1	x_2	$\cdots$	x_{p-1}
inclusive :	x_0	$x_0 \oplus x_1$	$x_0 \oplus x_1 \oplus x_2$	$\cdots$	$\oplus_{i=0}^{p-1} x_i$
exclusive :	unchanged	x_0	$x_0 \oplus x_1$	$\cdots$	$\oplus_{i=0}^{p-2} x_i$

Initial:

	matrix		sol	rhs	action
2	2	13	1	17	
4	5	32	1	41	
-2	-3	-16	1	-21	

Step 1:

	matrix		sol	rhs	action
2	2	13	1	17	take this row
4	5	32	1	41	
-2	-3	-16	1	-21	

Step 2:

	matrix		sol	rhs	action
2	2	13	1	17	take this row
↓	↓	↓			
4	5	32	1	41	minus $4 \times (1/2)$
-2	-3	-16	1	-21	

Step 3:

	matrix		sol	rhs	action
2	2	13	1	17	take this row
0	1	6	1	7	
-2	-3	-16	1	-21	

Step 4:

	matrix		sol	rhs	action
2	2	13	1	17	take this row
⇊	⇊	⇊			
0	1	6	1	7	
-2	-3	-16	1	-21	minus $(-2) \times (1/2)$

Step 5:

	matrix		sol	rhs	action
2	2	13	1	17	take this row
0	1	6	1	7	
0	-1	-3	1	-4	

Step 6:

	matrix		sol	rhs	action
2	2	13	1	17	first column done
0	1	6	1	7	
0	-1	-3	1	-4	

Step 7:

scaling		matrix		sol	rhs	action
	2	2	13	1	17	
	0	1	6	1	7	take this row
	0	-1	-3	1	-4	

Step 8:

scaling		matrix		sol	rhs	action
1/2	2	**2**	13	1	17	minus 2×1
	↑	↑	↑			
	0	**1**	6	1	7	take this row
	0	-1	-3	1	-4	

Step 9:

scaling		matrix		sol	rhs	action
1/2	2	0	1	1	3	
	0	1	6	1	7	take this row
	0	-1	-3	1	-4	

Step 10:

scaling		matrix		sol	rhs	action
1/2	2	0	1	1	3	
−2	0	**1**	6	1	7	take this row
	↓	↓	↓			
	0	**-1**	-3	1	-4	minus $(-1) \times 1$

Step 11:

scaling		matrix		sol	rhs	action
1/2	2	0	1	1	3	
−2	0	1	6	1	7	take this row
	0	0	3	1	3	

Step 12:

scaling		matrix		sol	rhs	action
1/2	2	0	1	1	3	
−2	0	1	6	1	7	second column done
+1	0	0	3	1	3	

Step 13:

scaling		matrix		sol	rhs	action
1/2	2	0	1	1	3	
−2	0	1	6	1	7	
+1	0	0	3	1	3	take this row

Step 14:

scaling		matrix		sol	rhs	action	*scaling*
	2	0	**1**	1	3	minus $1 \times 1/3$	
1	0	1	6	1	7		
	⇈	⇈	⇈				
	0	0	**3**	1	3	take this row	1/3

Step 15:

scaling		matrix		sol	rhs	action	*scaling*
	2	0	0	1	2		−1/3
1	0	1	6	1	7		
	0	0	3	1	3	take this row	1/3

Step 16:

scaling		matrix		sol	rhs	action	*scaling*
	2	0	0	1	2		−1/3
	0	1	**6**	1	7	minus $6 \times 1/3$	
	↑	↑	↑				
1	0	0	**3**	1	3	take this row	1/3

Step 17:

scaling		matrix		sol	rhs	action	*scaling*
	2	0	0	1	2		
	0	1	0	1	1		
−2	0	0	3	1	3	take this row	
	↑	↑	↑				
1	0	0	**3**	1	3	take this row	1/3

Step 18:

scaling		matrix		sol	rhs	action	*scaling*
	2	0	0	1	2		−1/3
−2	0	1	0	1	1		−2/3
1	0	0	3	1	3	third column done	1/3
+1							

Finished:

scaling		matrix		sol	rhs	action	*scaling*
	2	0	0	1	2		
	0	1	0	1	1		
1/3	0	0	3	1	3		

图 3.2：高斯 - 乔丹消元示例

图 3.4 MPI_Scan

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Scan (				
MPI_Scan_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in input buffer	int MPI_Count	INTEGER	IN
datatype	datatype of elements of input buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation	MPI_Op	TYPE(MPI_Op)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
template<typename T , typename F >
void mpl::communicator::scan
( F,const T &, T & ) const;( F,const T *, T *,
const contiguous_layout< T > & ) const;
( F,T & ) const;
( F,T *, const contiguous_layout< T > & ) const;
F : reduction functionT : typePython:


res = Intracomm.scan( sendobj=None,recvobj=None,op=MPI.SUM)
```

```
// scan.c// add all the random variables together
MPI_Scan(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approaching nprocs/2:
if (procno==nprocs-1)
printf("Result %6.3f compared to nprocs/2=%5.2f\n",
result,nprocs/2.);
```

在 python 模式下，结果是函数的返回值；而在 numpy 中，结果作为第二个参数传递。

```
## scan.py
mycontrib = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.scan(mycontrib)
sbuf = np.empty(1,dtype=int)
rbuf = np.empty(1,dtype=int)
sbuf[0] = mycontrib
comm.Scan(sbuf,rbuf)
```

图 3.5 MPI_Exscan

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Exscan (				
MPI_Exscan_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in input buffer	int MPI_Count	INTEGER	IN
datatype	datatype of elements of input buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation	MPI_Op	TYPE(MPI_Op)	IN
comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
template<typename T , typename F >
void mpl::communicator::exscan( F,const T &, T & ) const;
( F,const T *, T *,const contiguous_layout< T > & ) const;
( F,T & ) const;
( F,T *, const contiguous_layout< T > & ) const;
F : reduction functionT : typePython:
res = Intracomm.exscan( sendobj=None,recvobj=None,op=MPI.SUM)
```

您可以使用任何给定的归约操作符，（有关列表，请参见第 3.10.1 节），或者使用用户自定义的操作符。在后一种情况下，`MPI_Op` 操作不会返回错误代码。

MPL note 17: Scan operations. 与 C/F 接口一样，MPL 对扫描例程的接口具有与 “Allreduce” 例程相同的调用顺序。

3.4.1 Exclusive scan

通常，更有用的变体是 *exclusivescan* `MPI_Exscan` (图 3.5) 具有相同的签名。

处理器 0 上的 exclusive scan 结果是未定义的 (`None` 在 python 中)，而处理器 1 上的结果是处理器 1 发送值的副本。特别地，`MPI_Op` 不需要在这两个处理器上调用。

练习 3.10. exclusive 定义，计算 $x_0 \oplus x_{i-1}$ 在处理器 i 上，可以从 inclusive 操作推导出来，对于诸如 `MPI_SUM` 或 `MPI_PROD` 的操作。是否存在不适用这种情况的操作符?

3.4.2 扫描操作的使用

该 MPI_Scan 操作在索引数据处理中常常非常有用。假设每个处理器 p 都有一个本地向量，其中元素数量 n_p 是动态确定的。为了将本地编号 $0 \ldots n_p - 1$ 转换为全局编号，可以对本地元素数量进行扫描。输出结果即为第一个本地变量的全局编号。

A举例来说，设置快速傅里叶变换（*FFT*）系数就需要这种转换。如果本地大小
都相等，那么确定第一个元素的全局索引就是一个简单的计算。对于不规则情况，我们首先进行一次扫描：

```
// fft.c
MPI_Allreduce( &localsize,&globalsize,1,MPI_INT,MPI_SUM, comm );
globalsize += 1;
int myfirst=0;
MPI_Exscan( &localsize,&myfirst,1,MPI_INT,MPI_SUM, comm );

for (int i=0; i<localsize; i++)
  vector[i] = sin( pi*freq* (i+1+myfirst) / globalsize );
```

图 3.3：局部数组共同形成一个连续区间

Exercise 3.11.

- 让每个进程计算一个随机值 $n$local，并分配一个该长度的数组。定义

$$N = \sum n_{\text{local}}$$

- 用连续整数填充数组，使得所有局部数组首尾相接后包含数字 $0 \cdots N - 1$。

（见图 3.3。）（本练习的骨架代码名为 `scangather`。）

Exercise 3.12. 你在前一个练习中使用了 MPI_Scan 还是 MPI_Exscan？给定相同输入，你如何描述另一种 scan 操作的结果?

可以执行分段扫描。设 x_i 为一系列我们想要求和到 X_i 的数字，定义如下 s.
设 y_i 是一系列布尔值，使得

$$\begin{cases} X_i = x_i & \text{if } y_i = 0 \\ X_i = X_{i-1} + x_i & \text{if } y_i = 1 \end{cases}$$

(这是将 稀疏矩阵向量乘积 作为前缀操作实现的基础；参见 HPC book, section-28.2.) 这意味着 X_i 对位于 $y_i = 0$ 和第一个之间的段进行求和 s后续位置满足 $y_i = 1$。要实现这一点，您需要一个用户定义的操作符

$$\begin{pmatrix} X \\ x \\ y \end{pmatrix} = \begin{pmatrix} X_1 \\ x_1 \\ y_1 \end{pmatrix} \oplus \begin{pmatrix} X_2 \\ x_2 \\ y_2 \end{pmatrix} : \begin{cases} X = x_1 + x_2 & \text{if } y_2 == 1 \\ X = x_2 & \text{if } y_2 == 0 \end{cases}$$

该操作符不是交换律的，需要用 `MPI_Op_create` 声明；参见第 3.10.2 节

3.5 有根集合操作：gather 和 scatter

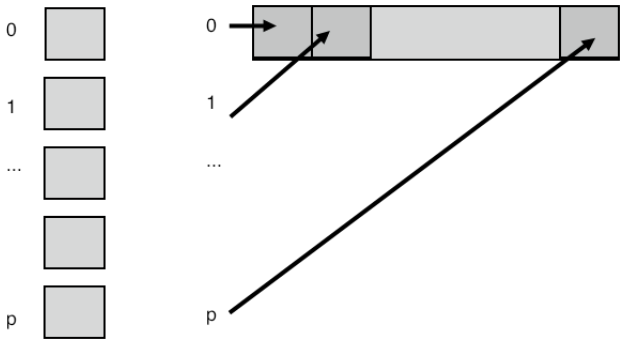


图 3.4：Gather 将所有数据收集到根节点

在 `MPI_Scatter` 操作中，根节点将信息传播给所有其他进程。与广播的区别在于，它涉及每个进程的单独信息传输。因此，gather 操作通常有一个元素数组，每个发送进程贡献一个元素，而 scatter 有一个数组，包含一个

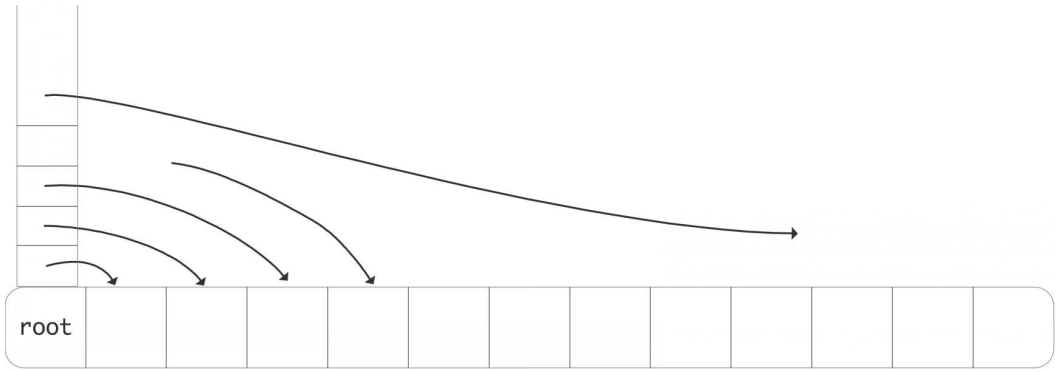


图 3.5：Scatter 操作

每个接收进程的单独项；见图 3.5.

这些 gather 和 scatter 集合操作的参数列表与 broadcast/reduce 不同。broadcast/reduce 涉及每个进程相同数量的数据，因此只需一个数据类型 / 大小的参数说明；broadcast 中是一个缓冲区，reduce 调用中是两个缓冲区。在 gather/scatter 调用中，你有

- 根进程上的一个大缓冲区，带有数据类型和大小的参数说明，和
- 每个进程上的一个较小缓冲区，带有其自身的数据类型和大小参数说明。

在 gather 和 scatter 调用中，每个处理器有 n 个单独的数据元素。还有一个根处理器，它有一个长度为 np 的数组，其中 p 是处理器的数量。gather 调用将所有这些数据从各个处理器收集到根处理器；scatter 调用假设信息最初在根处理器上，并将其分发到各个处理器。

这里有一个小例子：

```
// gather.c
// we assume that each process has a value "localsize"
// the root process collects these values

  if (procno==root)
    localsizes = (int*) malloc( nprocs*sizeof(int) );

// everyone contributes their info
  MPI_Gather(&localsize,1,MPI_INT,
             localsizes,1,MPI_INT,root,comm);
```

这也将成为第 3.9 节中更详细示例的基础。

练习 3.13。让每个进程计算一个随机数。你想打印最大值以及它是在哪个处理器上计算的。你会使用什么 collective？写一个简短的程序。

该 `MPI_Scatter` 操作在某种意义上是 gather 的逆操作：根进程有一个长度为 np 的数组，其中 p 是处理器 f 的数量，n 是每个处理器将接收的元素数量。 .

```
int MPI_Scatter
  (void* sendbuf, int sendcount, MPI_Datatype sendtype,
   void* recvbuf, int recvcount, MPI_Datatype recvtype,
   int root, MPI_Comm comm)
```

关于这些例程，有两点需要注意：

- 函数签名 `MPI_Gather`（图 3.6）有两个“count”参数，一个用于单个发送缓冲区的长度，另一个用于接收缓冲区。然而，令人困惑的是，第二个参数（仅在根进程相关）并不表示接收的总信息量，而是表示每个贡献的大小。因此，这两个 count 参数通常是相同的（至少在根进程上）；如果发送和接收处理器使用不同的 `MPI_Datatype` 值，它们可以不同。
- 虽然每个进程在 gather 调用中都有一个发送缓冲区，在 scatter 调用中都有一个接收缓冲区，但只有根进程需要用于收集或分发的长数组。然而，由于在 SPMD 模式下所有进程都需要使用相同的例程，因此总是存在一个用于该长数组的参数。非根进程可以在这里使用空指针。
- 更优雅地，`MPI_IN_PLACE` 选项可用于不适用的缓冲区，例如发送进程上的接收缓冲区。参见第 3.3.2 节。

MPL 注释 18：*Gather/scatter*。聚集（由 `communicator::gather`）或分散（由 `communicator::scatter`）单个标量时，接受一个标量参数和一个原始数组：

图 3.6 MPI_Gather

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Gather (					
MPI_Gather_c (					
	sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
	sendcount	number of elements in send buffer	int MPI_Count	INTEGER	IN
	sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
	recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
	recvcount	number of elements for any single receive	int MPI_Count	INTEGER	IN
	recvtype	datatype of recv buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
	root	rank of receiving process	int	INTEGER	IN
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	)				

MPL:

```
void mpl::communicator::gather
   ( int  root_rank, const T & senddata ) const
   ( int  root_rank, const T & senddata, T *  recvdata ) const
   ( int  root_rank, const T * senddata, const layout< T > &  sendl ) const
   ( int  root_rank, const T * senddata, const layout< T > &  sendl,T * recvdata,
    const layout< T > &  recvl ) const
```

Python:

```
MPI.Comm.Gather
   (self, sendbuf, recvbuf, int root=0)
```

```
vector<float> v;
float x;
comm_world.scatter(0, v.data(), x);
```

如果收集或分散的不止一个标量，则需要指定布局：

```
vector<float> vrecv(2),vsend(2*nprocs);
mpl::contiguous_layout<float> twonums(2);comm_world.scatter
(0, vsend.data(),twonums, vrecv.data(),twonums );
```

MPLnote 19: Gather on nonroot. 从逻辑上讲，在每个非根进程上，gather 调用只有发送缓冲区。MPL 通过提供两种仅指定发送数据的变体来支持这一点。

```
if (procno==0) {
  vector<int> size_buffer(nprocs);
```

```
  comm_world.gather
    (
     0,my_number_of_elements,size_buffer.data()
     );
} else {
  /*
   * If you are not the root, do versions with only send buffers
   */
  comm_world.gather
    ( 0,my_number_of_elements );
```

3.5.1 示例

在某些应用中，每个进程计算矩阵的一行或一列，但对于某些计算（例如行列式），将整个矩阵放在一个进程上会更高效。当然，只有当该矩阵本质上比整个问题小得多时，比如接口系统或多重网格中的最后粗化层，才应这样做。

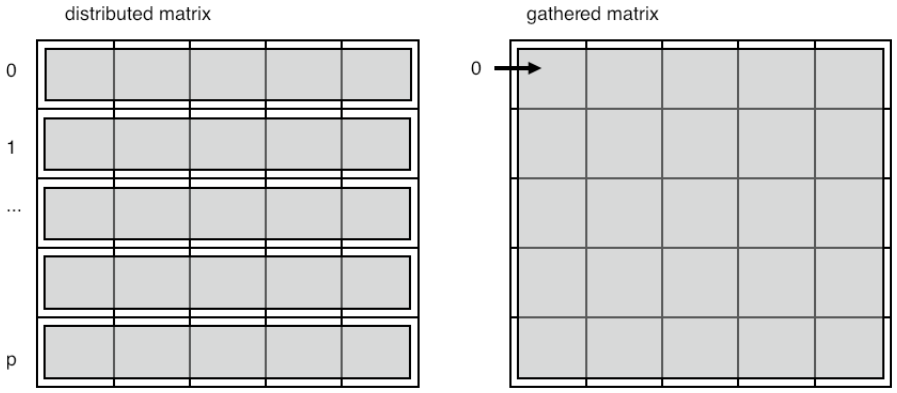


图 3.6：将分布式矩阵收集到一个进程上

Figure3.6 展示了这一点。请注意，概念上我们是在收集一个二维对象，但缓冲区当然是一维的。稍后你将看到如何使用 “subarray” 数据类型更优雅地完成此操作；详见第 6.3.4 节。

你还可以对分布式矩阵进行转置。

```
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
MPI_Scatter( regular,1,MPI_DOUBLE,
&(transpose[iproc]),1,MPI_DOUBLE,iproc,comm);}
```

在这个例子中，每个进程散布它的列。这只需要做一次，但散布操作发生在一个循环中。这里的技巧是，只有当进程是根进程时才发起散布操作，而这只发生一次。为什么需要循环？这是因为进程行的每个元素都来源于不同的散布操作。

图 3.7 MPI_Allgather

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Allgather (				
MPI_Allgather_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcount	number of elements in send buffer	int MPI_Count	INTEGER	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcount	number of elements received from any process	int MPI_Count	INTEGER	IN
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

Exercise 3.14. 你能重写这段代码，使其使用 gather 而不是 scatter 吗？这会改变代码结构的本质吗？

练习 3.15. 取练习 3.11 中的代码并扩展它，将所有本地缓冲区收集到 rank 0。由于本地数组长度不同，这需要 `MPI_Gatherv`。你如何构造 lengths 和 displacements 数组？（本练习的骨架代码名为 `scangather`。）

3.5.2 Allgather

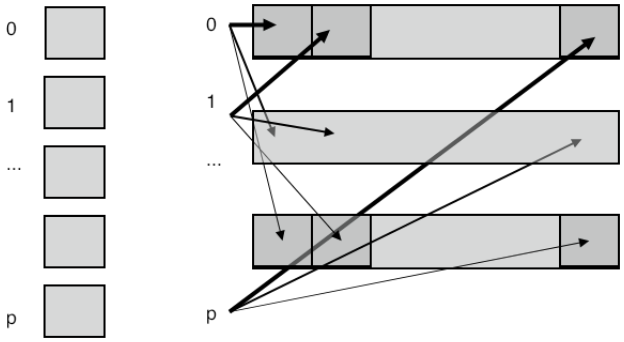


图 3.7：All gather 将所有数据收集到每个进程上

该 `MPI_Allgather` (图 3.7) 例程执行相同的收集操作到每个进程：每个进程最终拥有所有数据的全集；见图 3.7。

图 3.8 MPI_Alltoall

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Alltoall (				
MPI_Alltoall_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcount	number of elements sent to each process	int MPI_Count	INTEGER	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcount	number of elements received from any process	int MPI_Count	INTEGER	IN
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

此例程可用于最简单实现的 密集矩阵 - 向量乘积 中，为每个处理器提供完整输入；参见 HPCbook，第 7.2.3 节。

可以与 all-gather 一起使用 `MPI_IN_PLACE` 关键字：

1. 使用 `MPI_IN_PLACE` 作为发送缓冲区；2. 发送计数和数据类型被 MPI 忽略；3. 每个进程需要将其 “ 发送内容 ” 放入聚集缓冲区的正确位置。

有些情况看起来像 all-gather，但可以更高效地实现。以两个分布式向量的集合差为例。也就是说，你有两个分布式向量，想要创建一个新的向量，同样是分布式的，包含第一个向量中不出现在第二个向量中的元素。你可以通过在每个处理器上聚集第二个向量来实现，但这可能在内存使用上是不可行的。

练习 3.16. 你能想到另一种算法来求两个分布式向量的集合差吗？提示：查阅 *bucketbrigade* 算法；第 4.1.5 节。该算法的时间和空间复杂度是多少？除了减少工作空间之外，你还能想到其他优点吗？

3.6 全对全

全对全操作 `MPI_Alltoall` (图 3.8) 可以看作是一组同时进行的广播或同时进行的收集。参数规格与 allgather 类似，具有单独的发送和接收缓冲区，且未指定根节点。与 gather 调用一样，receivecount 对应于单个接收，而不是总量。

与 gather 调用不同，发送缓冲区现在遵循相同的原则：当发送计数为 1 时，缓冲区的长度等于进程数。

3.6.1 全对全作为数据转置

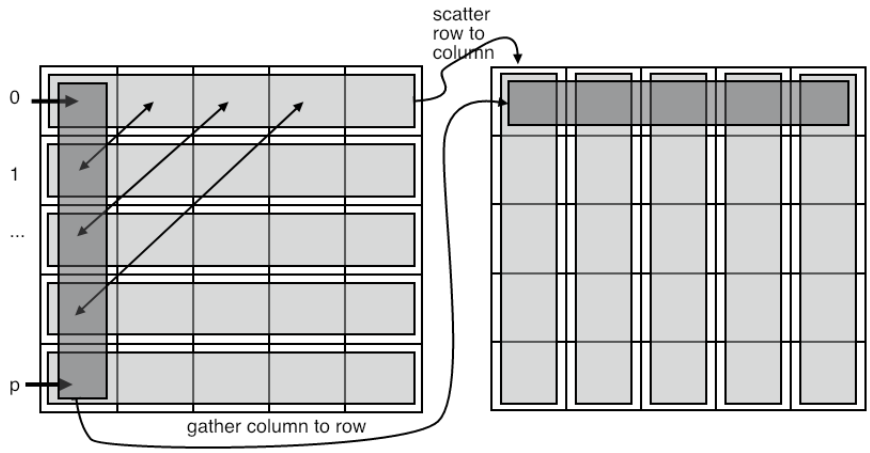


图 3.8：全对全转置数据

all-to-all 操作可以被视为数据转置。例如，假设每个进程都知道要发送多少数据给每个其他进程。如果你绘制一个大小为 $P \times P$ 的连接矩阵，表示谁发送给谁，那么发送信息可以放在行中：

$$\forall_i : C[i,j] > 0 \quad \text{if process } i \text{ sends to process } j.$$

相反，列则表示接收信息：

$$\forall_j : C[i,j] > 0 \quad \text{if process } j \text{ receives from process } i.$$

这种数据转置的典型应用是在 FFT 算法中，在大型集群上它可能占用运行时间的百分之几十。

我们将考虑数据转置的另一个应用，即基数排序，但我们将分几步进行。首先：

练习 3.17. 在基数排序的初始阶段，每个进程考虑向其他每个进程发送多少元素。使用 `MPI_Alltoall` 推导出它们将从其他每个进程接收多少元素。

3.6.2 All-to-all-v

基数排序算法的主要部分包括每个进程向其他每个进程发送部分元素。该例程 `MPI_Alltoallv` (图 3.9) 用于此模式：

- 每个进程将其数据分散到所有其他进程，

图 3.9 MPI_Alltoallv

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Alltoallv (				
MPI_Alltoallv_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	[const int[] MPI_Count[]	INTEGER(*)	IN
sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	[const int[] MPI_Aint[]	INTEGER(*)	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	[const int[] MPI_Count[]	INTEGER(*)	IN
rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	[const int[] MPI_Aint[]	INTEGER(*)	IN
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

- 但每个进程的数据量不同。

Exercise 3.18. 基数排序的实际数据重排 可以用 `MPI_Alltoallv` 来完成。完成练习 3.17 的代码。

3.7 Reduce-scatter

T这里有几个 MPI 集合操作在功能上等同于其他操作的组合。你已经见过 `MPI_Allreduce`，它等同于先进行归约再进行广播。通常这种组合调用比单独使用各个调用更高效；参见 HPC 书籍，第 7.1 节。

这里是另一个例子： `MPI_Reduce_scatter` 等价于对一个数据数组的归约（意味着对每个数组位置的逐点归约），随后将该数组散播到各个进程。

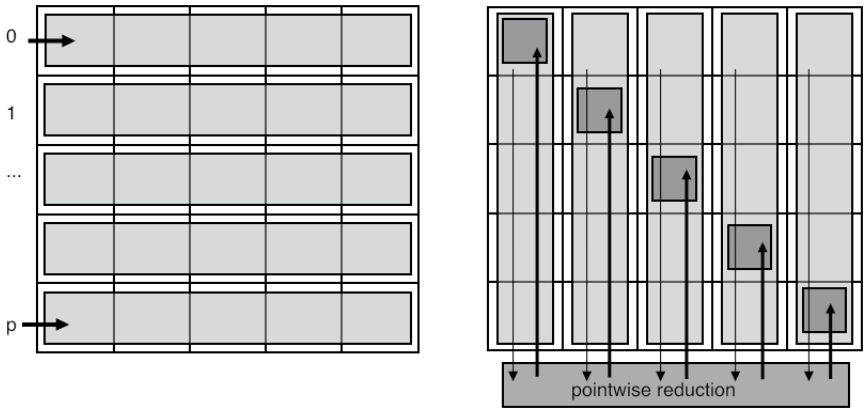


图 3.9：Reduce scatter

我们将讨论这个例程，或者更确切地说是它的变体 `MPI_Reduce_scatter_block`（图 3.10），使用一个重要的例子： 稀疏矩阵 - 向量乘积（参见 HPC 书，章节 -7.5.1 了解背景信息）。每个进程包含一个或多个矩阵行，因此通过查看索引，进程可以决定它需要从哪些其他进程接收数据，也就是说，每个进程知道它将接收多少消息，以及来自哪些进程。问题是一个进程如何找出它需要发送数据到哪些其他进程。

让我们设置数据：

```
// reducescatter.c
int
// data that we know:
  *i_recv_from_proc = (int*) malloc(nprocs*sizeof(int)),
  *procs_to_recv_from, nprocs_to_recv_from=0,
// data we are going to determin:
  *procs_to_send_to,nprocs_to_send_to;
```

每个进程创建一个由 1 和 0 组成的数组，描述它需要从谁那里获取数据。理想情况下，我们只需要数组 *procs_to_recv_from*，但最初我们需要（可能大得多的）数组 *i_recv_from_proc*。

接下来， `MPI_Reduce_scatter_block` 调用在每个进程上计算它需要发送多少条消息。

```
MPI_Reduce_scatter_block
(i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,MPI_SUM,
comm);
```

我们还没有发送到哪些进程的信息。为此，每个进程向它的每个发送者发送一个零大小消息。相反，它随后执行一个接收，使用 `MPI_ANY_SOURCE` 来发现谁在请求它的数据。该 `MPI_Reduce_scatter_block` 调用的关键点是，没有它，进程将不知道要期待多少个这样的零大小消息。

```
/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
  int proc=procs_to_recv_from[iproc];
  double send_buffer=0.;
  MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
              &(send_requests[iproc]));
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
  double recv_buffer;
  MPI_Status status;
  MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
           &status);
  procs_to_send_to[iproc] = status.MPI_SOURCE;
}
MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);
```

该 `MPI_Reduce_scatter`（图 3.10）调用更为通用：通过拥有单独的接收计数，不仅仅是指示两个进程之间消息的存在，还可以例如指示消息的大小。

我们可以将 reduce-scatter 看作是上述全对全数据转置（章节 3.6.1）的有限形式。假设矩阵 C 仅包含 0/1，表示是否发送消息，而非实际数量。如果接收进程只需要知道要接收多少条消息，而不关心它们来自哪里，那么只需知道列和，而非整列（见图 3.9）。

reduce-scatter 机制的另一个应用是在密集矩阵 - 向量乘积中，如果使用二维数据分布。

图 3.10 MPI_Reduce_scatter

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Reduce_scatter (				
MPI_Reduce_scatter_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcounts	non-negative integer array (of length group size) specifying the number of elements of the result distributed to each process.	[const int[] MPI_Count[]	INTEGER(*)	IN
datatype	datatype of elements of send and receive buffers	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation	MPI_Op	TYPE(MPI_Op)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

Exercise 3.19. Implement the dense matrix-vector product, where the matrix is distributed by columns: each process stores A_{*j} for a disjoint set of j-values. At the start and end of the algorithm each process should store a distinct part of the input and output vectors. Argue that this can be done naively with an `MPI_Reduce` operation, but more efficiently using `MPI_Reduce_scatter`.
For discussion, see HPC book, section-7.2.2.

3.7.1 Examples

这方面的一个重要应用是建立不规则的通信模式。假设每个进程都知道它想与哪些其他进程通信；问题是让其他进程知道这一点。解决方案是使用 `MPI_Reduce_scatter` 来找出有多少进程想与你通信

```
MPI_Reduce_scatter_block
  (i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
  MPI_SUM,comm);
```

然后等待正好那么多源值为 `MPI_ANY_SOURCE` 的消息。

```
/*
* Send a zero-size msg to everyone that you receive from,
* just to let them know that they need to send to you.*/
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
int proc=procs_to_recv_from[iproc];
```

图 3.11 MPI_Barrier

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Barrier (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

```
  double send_buffer=0.;
  MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
            &(send_requests[iproc]));
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
  double recv_buffer;
  MPI_Status status;
  MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
           &status);
  procs_to_send_to[iproc] = status.MPI_SOURCE;
}
MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);
```

Use of `MPI_Reduce_scatter` 来实现二维矩阵 - 向量乘积。使用 `MPI_Comm_split` 设置独立的行和列通信器，使用 `MPI_Reduce_scatter` 来合并局部乘积。

```
MPI_Allgather(&my_x,1,MPI_DOUBLE,local_x,1,MPI_DOUBLE,
environ.col_comm);
MPI_Reduce_scatter(local_y,&my_y,&ione,MPI_DOUBLE,MPI_SUM,
environ.row_comm);
```

3.8 Barrier

一个 barrier 调用，`MPI_Barrier`（图 3.11）是一个阻塞所有进程直到它们都到达该 barrier 调用的例程。因此它实现了进程的时间同步。

该调用的简单性与其用途形成对比，后者非常有限。几乎不需要通过 barrier 来同步进程：在大多数情况下，处理器不同步并不重要。相反，集合操作（除了新的非阻塞集合；章节 3.11）本身引入了一种类似 barrier 的机制。

3.9 可变大小输入的集合操作

在上述的 gather 和 scatter 调用中，每个处理器接收或发送的项目数量是相同的。在许多情况下这是合适的，但有时每个处理器希望或贡献的项目数量是不同的。

以 gather 调用为例。假设每个处理器执行本地计算，产生一定数量的数据元素，而这个数量对每个处理器来说是不同的（或者至少不是全部相同）。在常规的 `MPI_Gather` 调用中，根处理器有一个大小为 nP 的缓冲区，其中 n 是每个处理器产生的元素数量，P 是处理器的数量。处理器 p 的贡献将放入位置 $pn, \ldots, (p+1)n-1$。

对于可变情况，我们首先需要计算所需的总缓冲区大小。这可以通过一个简单的 `MPI_Reduce`，使用 `MPI_SUM` 作为归约操作符来完成：缓冲区大小为 $\sum_p n_p$，其中 n_p 是处理器 p 上的元素数量。但你也可以稍后再进行这个计算。

下一个问题是处理器的贡献将放入该缓冲区的何处。对于处理器 p 的贡献，即 $\sum_{q<p} n_p$, $\ldots \sum_{q\leq p} n_p - 1$。为了计算这个，根处理器需要拥有所有 n_p 数字，并且它可以通过 `MPI_Gather` 调用来收集它们。

我们现在拥有了所有要素。所有处理器都指定了一个发送缓冲区，就像使用 `MPI_Gather` 一样。然而，根处理器上的接收缓冲区规格更为复杂。它现在包括：

```
outbuffer, array-of-outcounts, array-of-displacements, outtype
```

你刚刚已经看到了如何构造这些信息。

例如，在一个 `MPI_Gatherv`（图 3.12）调用中，每个进程有一个单独的贡献项数。为了收集这些，根进程需要通过 `MPI_Gather` 调用找到这些单独的数量，并在本地构造偏移数组。注意偏移数组的大小为 `ntids+1`：最终的偏移值自动是所有传入数据的总大小。见下面的示例。

有各种调用，其中处理器可以拥有不同大小的缓冲区。

- 在 `MPI_Scatterv`（图 3.13）根进程为每个接收者拥有不同数量的数据。
- 相反，`MPI_Gatherv` 每个进程为接收结果贡献不同大小的发送缓冲区；`MPI_Allgatherv`（图 3.14）也做同样的事情，但将结果保留在所有进程上；`MPI_Alltoallv` 在每个进程上执行不同变量大小的聚集。

3.9.1 Gatherv 示例

我们使用 `MPI_Gatherv` 来对根进程进行不规则聚集。我们首先需要一个 `MPI_Gather` 来确定偏移量。

图 3.12 MPI_Gatherv

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Gatherv (MPI_Gatherv_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcount	number of elements in send buffer	int MPI_Count	INTEGER	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	const int[] MPI_Count[]	INTEGER(*)	IN
displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i	const int[] MPI_Aint[]	INTEGER(*)	IN
recvtype	datatype of recv buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
root	rank of receiving process	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
template<typename T>void gatherv
(int root_rank, const T *senddata, const layout<T> &sendl,T *recvdata,
 const layouts<T> &recvls, const displacements &recvdispls) const
(int root_rank, const T *senddata, const layout<T> &sendl,T *recvdata,
 const layouts<T> &recvls) const
(int root_rank, const T *senddata, const layout<T> &sendl ) constPython:

Gatherv(self, sendbuf, [recvbuf,counts], int root=0)
```

图 3.13 MPI_Scatterv

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Scatterv (				
MPI_Scatterv_c (				
sendbuf	address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	[const int[] MPI_Count[]	INTEGER(*)	IN
displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i	[const int[] MPI_Aint[]	INTEGER(*)	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcount	number of elements in receive buffer	[int MPI_Count	INTEGER	IN
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
root	rank of sending process	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

Code:

```
// gatherv.c
// we assume that each process has an array "localdata"
// of size "localsize"

// the root process decides how much data will be coming:
// allocate arrays to contain size and offset information
  if (procno==root) {
    localsizes = (int*) malloc( nprocs*sizeof(int) );
    offsets = (int*) malloc( nprocs*sizeof(int) );
  }
// everyone contributes their local size info
  MPI_Gather(&localsize,1,MPI_INT,
             localsizes,1,MPI_INT,root,comm);
// the root constructs the offsets array
  if (procno==root) {
    int total_data = 0;
    for (int i=0; i<nprocs; i++) {
      offsets[i] = total_data;
      total_data += localsizes[i];
    }
    alldata = (int*) malloc( total_data*sizeof(int) );
  }
// everyone contributes their data
  MPI_Gatherv(localdata,localsize,MPI_INT,

    ↪alldata,localsizes,offsets,MPI_INT,root,comm);
```

Output:

```
make[3]: `gatherv' is up to
    ↪date.
TACC:  Starting up job
    ↪4328411
TACC:  Starting parallel
    ↪tasks...
Local sizes: 13, 12, 13, 14,
    ↪11, 12, 14, 6, 12, 8,
Collected:

    ↪0:1,1,1,1,1,1,1,1,1,1,1,1;
  1:2,2,2,2,2,2,2,2,2,2,2,2;

    ↪2:3,3,3,3,3,3,3,3,3,3,3,3;

    ↪3:4,4,4,4,4,4,4,4,4,4,4,4;
  4:5,5,5,5,5,5,5,5,5,5,5;
  5:6,6,6,6,6,6,6,6,6,6,6,6;

    ↪6:7,7,7,7,7,7,7,7,7,7,7,7;
  7:8,8,8,8,8,8;
  8:9,9,9,9,9,9,9,9,9,9,9,9;
  9:10,10,10,10,10,10,10,10;
TACC:  Shutdown complete.
    ↪Exiting.
```

图 3.14 MPI_Allgatherv

名称 参数名	说明	C 类型	F 类型	inout
MPI_Allgatherv (				
MPI_Allgatherv_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcount	number of elements in send buffer	[int MPI_Count	INTEGER	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	[const int[] MPI_Count[]	INTEGER(*)	IN
displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	[const int[] MPI_Aint[]	INTEGER(*)	IN
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

Python:

```
MPI.Comm.Allgatherv(self, sendbuf, recvbuf)
    where recvbuf = "[ array, counts, displs, type]"
```

```
## gatherv.py
# implicitly using root=0
globalsize = comm.reduce(localsize)
if procid==0:
    print("Global size=%d" % globalsize)
collecteddata = np.empty(globalsize,dtype=int)
counts = comm.gather(localsize)
comm.Gatherv(localdata, [collecteddata, counts])
```

3.9.2 Allgatherv 示例

在实际调用 gatherv 之前，我们需要构造 count 和 displacement 数组。最简单的方法是使用归约。

```
// allgatherv.cMPI_Allgather( &my_count,1,MPI_INT,recv_counts,
1,MPI_INT, comm );int accumulate = 0;
for (int i=0; i<nprocs; i++) {
recv_displs[i] = accumulate; accumulate += recv_counts[i]; }
int *global_array = (int*) malloc(accumulate*sizeof(int));
MPI_Allgatherv( my_array,procno+1,MPI_INT,global_array,
recv_counts,recv_displs,MPI_INT, comm );
```

在 python 中，接收缓冲区必须包含 counts 和 displacements 数组。

```
## allgatherv.py
mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)
```

```
my_count = np.empty(1,dtype=int)
my_count[0] = mycount
comm.Allgather( my_count,recv_counts )

accumulate = 0
for p in range(nprocs):
    recv_displs[p] = accumulate; accumulate += recv_counts[p]
global_array = np.empty(accumulate,dtype=np.float64)
comm.Allgatherv( my_array, [global_array,recv_counts,recv_displs,MPI.DOUBLE] )
```

3.9.3 变量 all-to-all

变量 all-to-all 例程 `MPI_Alltoallv` 在第 3.6.2 节中讨论。

3.10 MPI 操作符

MPI 操作符，即类型为 `MPI_Op` 的对象，用于归约操作符。大多数常见操作符，如求和或最大值，已内置于 MPI 库中；参见第 3.10.1 节。也可以定义新的操作符；参见第 3.10.2 节。

3.10.1 预定义操作符

以下是 预定义操作符 `MPI_Op` 的列表。

MPI 类型	含义	适用范围
MPI_MAX	最大值	整数，浮点数
MPI_MIN	最小值	
MPI_SUM	sum	整数，浮点数，复数，多语言类型
MPI_REPLACE	overwrite	
MPI_NO_OP	no change	
MPI_PROD	product	
MPI_LAND	逻辑与	C 整数，逻辑
MPI_LOR	逻辑或	
MPI_LXOR	逻辑异或	
MPI_BAND	按位与	整数，字节，多语言类型
MPI_BOR	按位或	
MPI_BXOR	按位异或	
MPI_MAXLOC	最大值及位置	MPI_DOUBLE_INT 和类似的
MPI_MINLOC	最小值及位置	

3.10.1.1 Minloc 和 maxloc

The MPI_MAXLOC and MPI_MINLOC 操作同时返回最大值及其所在的秩。它们的结果是一个 struct，表示进行归约的数据，以及一个 int。

在 C 语言中，归约调用中使用的类型是： MPI_FLOAT_INT, MPI_LONG_INT, MPI_DOUBLE_INT, MPI_SHORT_INT, MPI_2INT, MPI_LONG_DOUBLE_INT。同样，输入需要由这样的结构组成：输入应为 an 个此类结构类型的数组，其中 int 是数字的秩。

这些类型可能具有一些不寻常的大小属性 ：

Code:

```
// longint.c
MPI_Type_size( MPI_LONG_INT,&s );
printf("MPI_LONG_INT size=%d\n",s);
MPI_Aint ss;
MPI_Type_extent( MPI_LONG_INT,&ss );
printf("MPI_LONG_INT extent=%ld\n",ss);
```

Output:

```
MPI_LONG_INT size=12
MPI_LONG_INT extent=16
```

Fortrannote6: Min/maxloctypes. MPI 的原始 Fortran 接口是围绕 Fortran77 特性设计的，因此它不使用 Fortran 派生类型（ Type 关键字）。相反，所有整数索引都存储在被归约的类型中。可用的结果类型为 MPI_2REAL, MPI_2DOUBLE_PRECISION, MPI_2INTEGER。

同样，输入也需要是此类类型的数组。考虑以下示例：

```
Real*8,dimension(2,N) :: input,output
call MPI_Reduce( input,output, N, MPI_2DOUBLE_PRECISION, &
    MPI_MAXLOC, root, comm )
```

MPL 注释 20：运算符。算术： *plus*， *multiplies*， *max*， *min*。

图 3.15 MPI_Op_create

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Op_create (				
MPI_Op_create_c (				
user_fn	user defined function	MPI_User_function* MPI_User_function_c*	PROCEDURE (MPI_User_function)	IN
commute	true if commutative; false otherwise.	int	LOGICAL	IN
op	operation	MPI_Op*	TYPE(MPI_Op)	OUT
)				

Python:

```
MPI.Op.create(cls,function,bool commute=False)
```

逻辑：*logical_and*, *logical_or*, *logical_xor*。

按位：*bit_and*, *bit_or*, *bit_xor*。

3.10.2 用户自定义操作符

除了预定义操作符外，MPI 还可以使用用户自定义操作符进行归约或扫描操作。

该例程是 `MPI_Op_create`（图 3.15），它接受一个用户函数并将其转换为类型为 `MPI_Op` 的对象，然后可以在任何归约中使用：

```
MPI_Op rwz;
MPI_Op_create(reduce_without_zero,1,&rwz);
MPI_Allreduce(data+procno,&positive_minimum,1,MPI_INT,rwz,comm);
```

Python 注释 *11*：定义归约操作符。在 python 中，`Op.Create` 是 `MPI` 类的一个方法。

```
rwz = MPI.Op.Create(reduceWithoutZero)
positive_minimum = np.zeros(1,dtype=intc)
comm.Allreduce(data[procid],positive_minimum,rwz);
```

用户函数需要具有以下签名：

```
typedef void MPI_User_function
    ( void *invec, void *inoutvec, int *len,
      MPI_Datatype *datatype);
```

```
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE
```

例如，下面是一个用于在非负整数数组中查找最小非零数的操作符：

```
*(int*)inout = m;
}
```

Python 注释 *12*：归约函数。此类函数的 Python 等价物接收裸缓冲区作为参数。因此，最好先使用 `np.frombuffer` 将它们转换为 NumPy 数组：

```
## reductpositive.py
def reduceWithoutZero(in_buf, inout_buf, datatype):
    typecode = MPI._typecode(datatype)
    assert typecode is not None ## check MPI datatype is built-in
    dtype = np.dtype(typecode)

    in_array = np.frombuffer(in_buf, dtype)
    inout_array = np.frombuffer(inout_buf, dtype)

    n = in_array[0]; r = inout_array[0]
    if n==0:
        m = r
    elif r==0:
        m = n
    elif n<r:
        m = n
    else:
        m = r
    inout_array[0] = m
```

`assert` 语句考虑到这种 MPI 数据类型到 NumPy dtype 的映射仅适用于内置 MPI 数据类型。

MPL 注释 *21*：用户定义的操作符。用户定义的操作符可以是带有 `operator()` 的模板类。示例：

```
// reduceuser.cxx
template<typename T>
class lcm {
public:
  T operator()(T a, T b) {
    T zero=T();
    T t((a/gcd(a, b))*b);
    if (t<zero)
      return -t;
    return t;
  }
```

```
comm_world.reduce(lcm<int>(), 0, v, result);
```

（模板类可以是一个 lambda 表达式）

MPL 注释 *22*：*Lambda* 操作符。你也可以通过 lambda 来做归约：

```
comm_world.reduce
( [] (int i,int j) -> int { return i+j; },0,
data );
```

该函数有一个数组长度参数 `len`，允许一次对整个数组进行逐点归约。`inoutvec` 数组包含部分归约的结果，通常会被该函数覆盖。

用户函数有一些限制：

图 3.16 MPI_Op_commutative

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Op_commutative (				
op	operation	MPI_Op	TYPE(MPI_Op)	IN
commute	true if op is commutative, false otherwise	int*	LOGICAL	OUT
)				

图 3.17 MPI_Reduce_local

Name Param name	Explanation	C 类型	F 类型	输入输出
MPI_Reduce_local (				
MPI_Reduce_local_c (				
inbuf	input buffer	const void*	TYPE(*), DIMENSION(..)	IN
inoutbuf	combined input and output buffer	void*	TYPE(*), DIMENSION(..)	INOUT
count	number of elements in inbuf and inoutbuf buffers	[int MPI_Count	INTEGER	IN
datatype	datatype of elements of inbuf and inoutbuf buffers	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation	MPI_Op	TYPE(MPI_Op)	IN
)				

- 它可能不会调用 MPI 函数，除了 MPI_Abort。
- 它必须是结合的；它可以是可选的交换的，这一事实会传递给 MPI_Op_create 调用。

练习 3.20. 编写归约函数以实现向量的一范数：

$$\|x\|_1 \equiv \sum_i |x_i|.$$

（此练习的骨架代码名为 onenorm。）

该操作符可以通过相应的 MPI_Op_free 销毁。

```
int MPI_Op_free(MPI_Op *op)
```

这将操作符设置为 MPI_OP_NULL。在面向对象语言中这不是必需的，因为析构函数会处理它。

你可以使用 MPI_Op_commutative（图 3.16）来查询操作符的交换性。

3.10.3 局部归约

一个 MPI_Op 的应用可以通过例程 MPI_Reduce_local（图 3.17）来执行。使用此例程和一些发送 / 接收方案，你可以构建自己的全局归约。注意此例程不需要通信器，因为它是纯粹的局部操作。

3.11 非阻塞集合操作

上面你已经看到 ‘Isend’ 和 ‘Irecv’ 例程如何实现通信与计算的重叠。到目前为止你所见的集合操作不支持这种方式：它们表现得像阻塞的发送或接收。然而，MPI-3 引入了非阻塞集合操作。

此类操作可用于提高效率。例如，计算

$$y \leftarrow Ax + (x^t x)y$$

涉及矩阵 - 向量乘积，在稀疏矩阵情况下主要由计算主导，以及一个内积，内积通常主要由通信成本主导。你可以这样编写代码

```
MPI_Iallreduce( .... x ..., &request);
// compute the matrix vector product
MPI_Wait(request);
// do the addition
```

这也可以用于 3D FFT 操作 [15]。有时，非阻塞集合操作可以用于非显而易见的目的，例如 `MPI_Ibarrier` 中 [16]。

这些调用序列大致与其阻塞对应函数相同，除了它们输出一个 `MPI_Request`。然后你可以使用一个 `MPI_Wait` 调用来确保集合操作已完成。

Nonblocking collectives offer a number of performance advantages:

- 同时对同一通信器执行两个不同操作符的归约：

  $$\alpha \leftarrow x^t y$$
  $$\beta \leftarrow \|z\|_\infty$$

  这转换为：

  ```
  MPI_Allreduce( &local_xy,  &global_xy, 1,MPI_DOUBLE,MPI_SUM,comm);
  MPI_Allreduce( &local_xinf,&global_xin,1,MPI_DOUBLE,MPI_MAX,comm);
  ```

- 在重叠的通信器上同时执行集合操作；
- 重叠一个非阻塞集合操作和一个阻塞集合操作。

练习 3.21. 重温练习 7.1。仅让第一行和第一列拥有某些数据，它们分别通过列和行进行广播。每个进程现在参与两个同时进行的集合操作。用非阻塞广播实现此功能，并计时阻塞和非阻塞解决方案之间的差异。（此练习的骨架代码名为 `procgridnonblock`。）

备注 6 阻塞和非阻塞不匹配：要么所有进程调用非阻塞，要么所有进程调用阻塞。因此，以下代码是不正确的：

```
if (rank==root)
  MPI_Reduce( &x /* ... */ root,comm );
else
  MPI_Ireduce( &x /* ... */ );
```

图 3.18 MPI_Iallgather

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Iallgather (				
MPI_Iallgather_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcount	number of elements in send buffer	int / MPI_Count	INTEGER	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcount	number of elements received from any process	int / MPI_Count	INTEGER	IN
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

这不同于非阻塞调用的点对点行为：你可以用 *MPI_Irecv* 捕获一条用 *MPI_Send* 发送的消息。

备注 7 与发送和接收不同，集合操作没有标识标签。对于阻塞集合操作，这不会导致歧义问题。对于非阻塞集合操作，这意味着所有进程需要以相同的顺序发出它们。

非阻塞集合操作列表：

- MPI_Igather,MPI_Igatherv, MPI_Iallgather(图 3.18),MPI_Iallgatherv,
- MPI_Iscatter, MPI_Iscatterv,
- MPI_Ireduce, MPI_Iallreduce (图 3.19), MPI_Ireduce_scatter, MPI_Ireduce_scatter_block.
- MPI_Ialltoall,MPI_Ialltoallv, MPI_Ialltoallw,

3.11.2,

- MPI_Ibcast,
- MPI_Iexscan, MPI_Iscan,

MPL 注释 *23*：非阻塞集合操作。非阻塞集合操作与对应的阻塞变体具有相同的参数列表，区别在于它们不是返回一个 `void` 结果，而是返回一个 `irequest`。（参见 30 ）

Wait 调用是 `irequest` 对象的方法。

```
// ireducescalar.cxx
float x{1.},sum;
auto reduce_request =
```

图 3.19 MPI_Iallreduce

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Iallreduce (				
MPI_Iallreduce_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in send buffer	int MPI_Count	INTEGER	IN
datatype	datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation	MPI_Op	TYPE(MPI_Op)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

```
  comm_world.ireduce(mpl::plus<float>(), 0, x, sum);
reduce_request.wait();
if (comm_world.rank()==0) {
  std::cout << "sum = " << sum << '\n';
}
```

3.11.1 示例

3.11.1.1 数组转置

为了说明多个非阻塞集合操作的重叠，考虑转置一个数据矩阵。最初，每个进程拥有矩阵的一行；转置后，每个进程拥有一列。由于每一行都需要分发给所有进程，从算法角度看，这对应于一系列 scatter 调用，每个调用都起始于一个进程。

```
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
MPI_Scatter( regular,1,MPI_DOUBLE,
&(transpose[iproc]),1,MPI_DOUBLE,iproc,comm);}
```

Introducing the nonblocking MPI_Iscatter call, this becomes:

```
MPI_Request scatter_requests[nprocs];
for (int iproc=0; iproc<nprocs; iproc++) {
MPI_Iscatter( regular,1,MPI_DOUBLE,&(transpose[iproc]),
1,MPI_DOUBLE,iproc,comm,scatter_requests+iproc);}
MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE);
```

图 3.20 MPI_Ibarrier

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Ibarrier (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

练习 3.22. 你能用 `MPI_Igather` 实现相同的算法吗?

3.11.1.2 Stencils

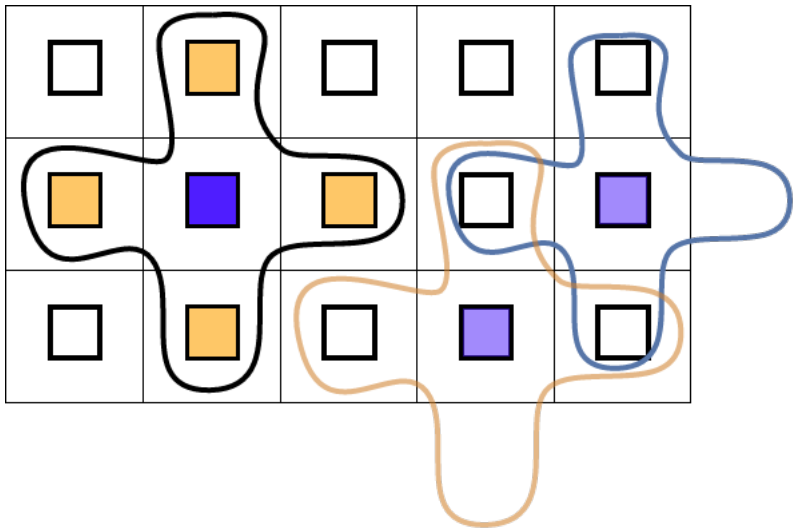


Figure 3.10: Illustration of five-point stencil gather

广受欢迎的 五点模板 计算看起来不像是一个集合操作，实际上，它通常是通过（非阻塞）send/recv 操作来计算的。然而，如果我们在每个子域上创建一个子通信器，该通信器恰好包含该域及其邻居，（见图 3.10 ）我们可以将通信模式表述为在每个子通信器上的 gather。使用普通的集合操作，这无法以 无死锁的方式来表述，但非阻塞集合操作使这成为可能。

我们将在第 11.2 节看到这一操作更优雅的表达方式。

3.11.2 Nonblocking barrier

可能最令人惊讶的非阻塞集合操作是 *nonblocking barrier* `MPI_Ibarrier` (图 3.20)。理解它的方法是不要将 barrier 看作时间上的同步，而是状态上的一致：达到 barrier 表示一个进程已经达到某个状态，而离开 barrier

意味着所有进程处于相同状态。普通屏障是一个阻塞等待达成一致的操作，而非阻塞屏障则是：

- 发布屏障意味着一个进程已经达到某个状态；并且
- 请求被满足意味着所有进程都已达到屏障。

一种场景是局部细化，其中一些进程决定细化它们的子域，这一事实需要它们与邻居通信。这里的问题是大多数进程不在这些邻居之中，因此它们不应发布任何类型的接收。相反，任何细化进程向其邻居发送消息，每个进程都发布一个屏障。

```
// ibarrierprobe.cif (i_do_send) {/*
* Pick a random process to send to,
* not yourself.*/int receiver = rand()%nprocs;
MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);}
/** Everyone posts the non-blocking barrier
* and gets a request to test/wait for*/
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);
```

现在每个进程交替地探测消息和测试屏障的完成情况。探测是通过非阻塞的 `MPI_Iprobe` 调用完成的，而测试屏障完成则是通过 `MPI_Test` 完成的。

```
for ( ; ; step++) {int barrier_done_flag=0;
MPI_Test(&barrier_request,&barrier_done_flag,
MPI_STATUS_IGNORE);//stop if you're done!
if (barrier_done_flag) {break;} else {
// if you're not done with the barrier:
int flag; MPI_Status status;MPI_Iprobe
( MPI_ANY_SOURCE,MPI_ANY_TAG,comm,  &flag,
 &status );if (flag) {// absorb message!
```

我们可以有效地使用非阻塞屏障，利用阻塞屏障所导致的空闲时间。在下面的代码片段中，进程测试屏障的完成情况，如果未检测到完成，则执行一些本地工作。

```
// findbarrier.c
MPI_Request final_barrier;
MPI_Ibarrier(comm,&final_barrier);
```

```
int global_finish=mysleep;do {int all_done_flag=0;
MPI_Test(&final_barrier,&all_done_flag,MPI_STATUS_IGNORE);
if (all_done_flag) {break;} else {
int flag; MPI_Status status;// force progressMPI_Iprobe
( MPI_ANY_SOURCE,MPI_ANY_TAG,comm,  &flag,
 MPI_STATUS_IGNORE );
printf("[%d] going to work for another second\n",procid);
sleep(1);global_finish++;}} while (1);
```

3.12 集体操作的性能

很容易将广播可视化，如图 3.11 所示：见图 3.11。根进程将其所有数据直接发送给

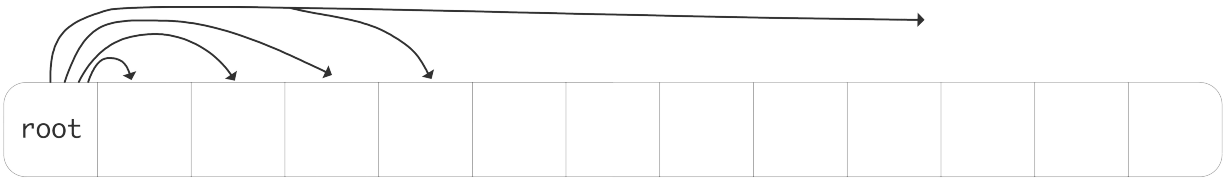


图 3.11：一个简单的广播

其他所有进程。虽然这描述了操作的语义，但实际上实现方式却大不相同。

消息传递所需的时间可以简单地建模为

$$\alpha + \beta n,$$

其中 α 是延迟，即在两个进程之间建立通信的一次性延迟，β 是每字节时间，或带宽的倒数，n 是发送的字节数。

在假设处理器一次只能发送一条消息的前提下，图 3.11 中的广播所需时间与处理器数量成正比。

练习 3.23. 涉及 p 个进程的广播总共需要多长时间？请分别给出 α 和 β 项。

缓解这一问题的一种方法是将广播结构设计成树状结构。如图所示 -
图 3.12。

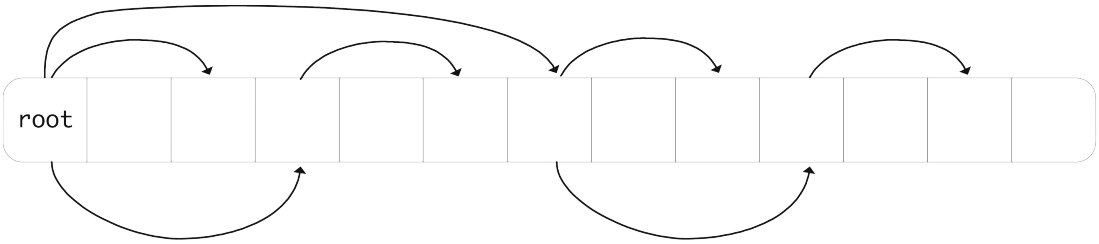


图 3.12：基于树的广播

Exercise 3.24. 通信时间现在如何依赖于处理器数量，分别针对 α 和 β 项。 α, β 项的下界会是多少?

Collectives 复杂度的理论在 HPC 书籍中第 7.1 节有更详细的描述；参见 e 也 [3]。

3.13 Collectivesand synchronization

除了屏障之外，Collectives 在处理器之间具有同步效果。例如，在

```
MPI_Bcast( ....data... root);
MPI_Send(....);
```

所有处理器上的发送操作将在根执行广播之后发生。相反，在 reduce 操作中，根可能需要等待其他处理器。这在图 3.13 中有所说明，该图展示了两个节点上进行归约操作的 TAU 跟踪，每个节点有两个六核插槽（处理器）。我们看到 1:

- 在每个插槽中，归约是线性累积；
- 在每个节点上，核心零和核心六随后合并它们的结果；
- 之后，最终的累积通过网络完成。

我们还看到两个节点并非完全同步，这对于 MPI 应用来说是正常的。因此，第一个节点上的核心 0 将处于空闲状态，直到它收到来自第二个节点上核心 12 的部分结果。

虽然集合操作在某种松散的意义上实现了同步，但无法对处理器之间集合操作之前和之后的事件做出任何断言：

```
...event 1...
MPI_Bcast(....);
...event 2....
```

考虑一个具体场景：

1. 这使用的是 mvapich 版本 1.6；在版本 1.9 中，节点内归约的实现已更改为模拟共享内存。

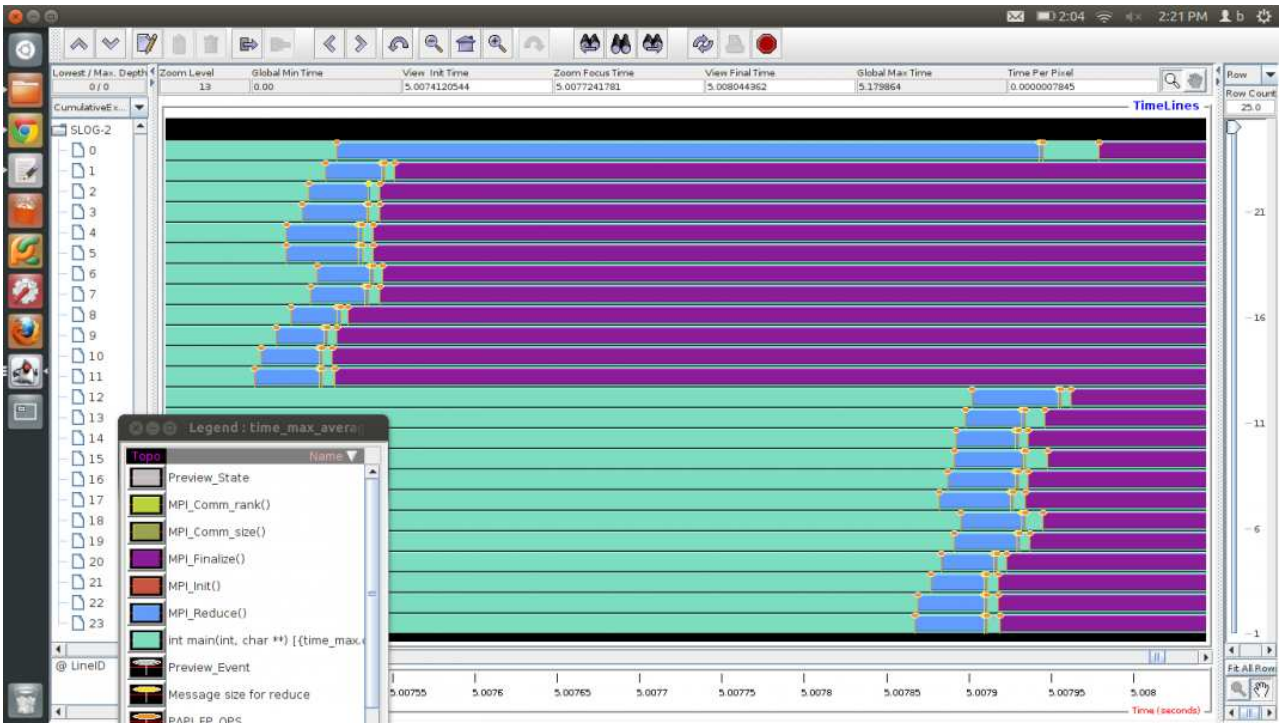


图 3.13：两个双插槽 12 核节点之间归约操作的跟踪

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

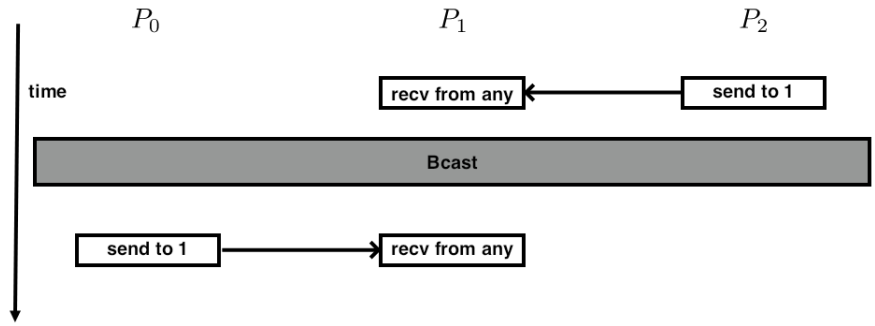
注意 `MPI_ANY_SOURCE` 在处理器 1 上的接收调用中的参数。一个明显的执行顺序可能是：

1. 来自 2 的发送被处理器 1 接收； 2. 所有人执行广播； 3. 来自 0 的发送被处理器 1 接收。

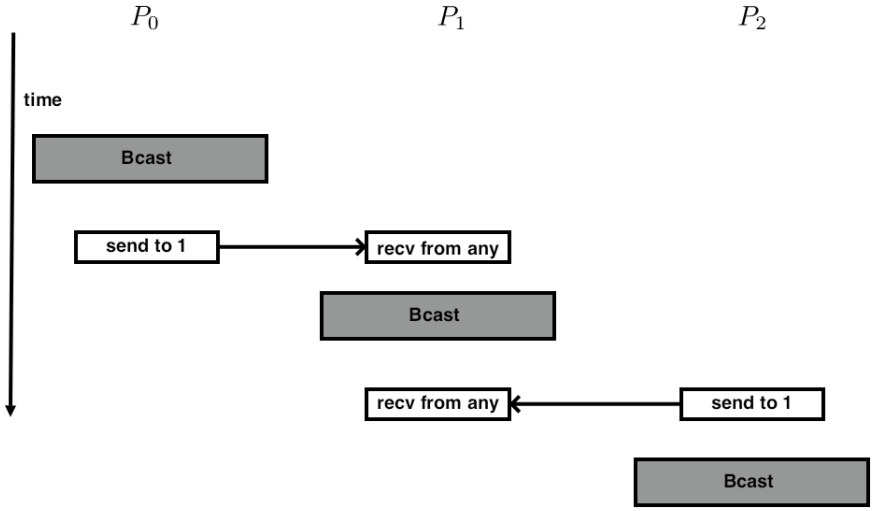
然而，同样可能出现以下执行顺序：

1. 处理器 0 开始广播，然后执行发送；

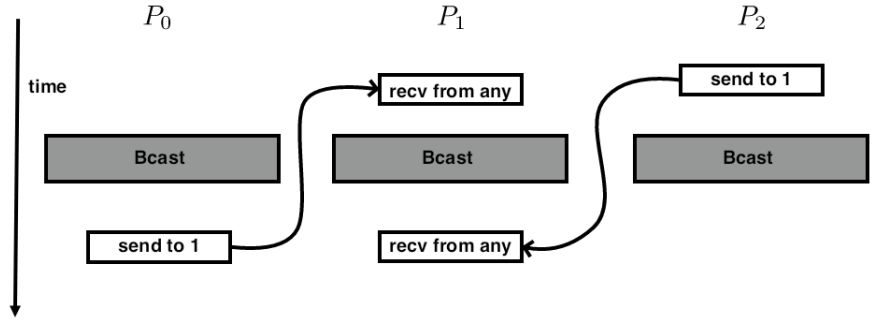
The most logical execution is:



However, this ordering is allowed too:



Which looks from a distance like:



换句话说，其中一条消息似乎 “ 回到了过去 ”。

图 3.14：发送和 collective 调用的可能时间顺序

2. 处理器 1 从 0 接收数据，然后执行其广播部分；3. 处理器 1 接收由 2 发送的数据，最后处理器 2 执行其广播部分 t.

这在图 3.14 中有所说明。

3.14 性能考虑

本节将讨论集体操作的多种实现方式及其性能影响。您可以使用 *SimGrid*（教程书章节，章节 -20）来测试这里描述的算法。

3.14.1 可扩展性

我们编写并行软件的动机有两个方面。首先，如果我们有一个问题需要解决，通常需要时间 T，那么我们希望使用 p 个处理器时，所需时间为 T/p。如果这是真的，我们称我们的并行方案为时间可扩展的。在实际中，我们通常接受一些小的额外项：正如你将在下面看到的，并行化通常会在运行时间中增加一个 $\log_2 p$ 项。

练习 3.25. 讨论以下算法的可扩展性：

- 你有一个浮点数数组。你需要计算每个数的正弦值
- 你有一个二维数组，表示区间 $[-2,2]^2$。你想绘制 *Mandelbrot set* 的图像，因此你需要计算每个点的颜色。
- 练习 2.6 的素数测试。

还有一种概念是并行算法可以空间可扩展：更多的处理器为你提供更多的内存，从而可以运行更大的问题。

练习 3.26. 讨论现代处理器设计背景下的空间可扩展性。

3.14.2 Complexity and scalability of collectives

3.14.2.1 Broadcast

朴素广播 编写一个广播操作，其中根进程对每个其他进程执行 `MPI_Send`。

从 α, β 的角度来看，这种方法的预期性能如何？

运行一些测试并确认。

简单环形 让根进程只发送给下一个进程，该进程再发送给它的邻居。该方案被称为 *bucket brigade*；另见第 4.1.5 节。

在 α, β 方面，这个的预期性能是什么？

运行一些测试并确认。

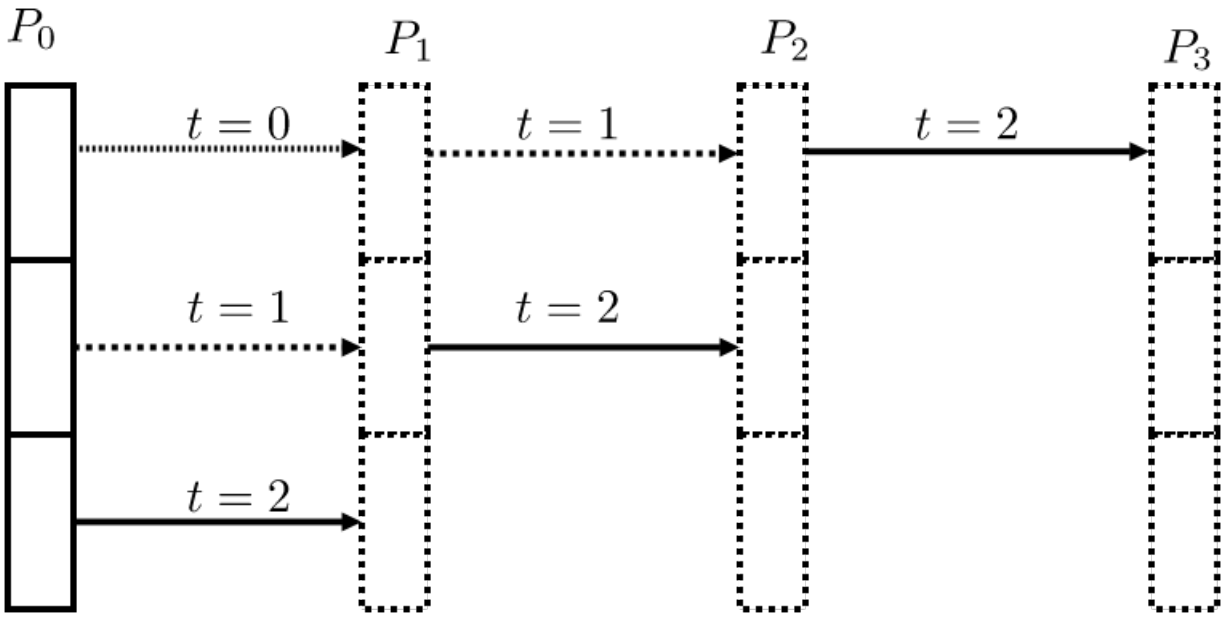


图 3.15：流水线式桶形接力

流水线环在环形广播中，每个进程需要接收完整消息后才能传递。我们可以通过将消息拆分成多个部分发送来提高效率。（见图 3.15。）这对于消息足够长以至于带宽成本主导延迟的情况尤其有利。

假设发送缓冲区长度大于 1。将发送缓冲区划分为若干块。根进程依次将这些块发送给下一个进程，每个进程将接收到的块继续发送下去。

从 α, β 的角度来看，这种方法的预期性能如何？为什么它比简单的环形更好？

运行一些测试并确认。

Recursive doubling 集合操作如 broadcast 可以通过 *implemented recursivedoubling* 实现，其中 root 发送给另一个进程，然后 root 和该进程再发送给另外两个进程，这四个进程再发送给另外四个，依此类推。然而，在实际的物理架构中，这种方案可以通过多种方式实现，且性能差异极大。

首先考虑进程 0 作为 root 的实现，它开始时发送给进程 1；然后它们发送给 2 和 3；这四个进程再发送给 4–7，依此类推。如果架构是一个线性处理器数组，这将导致 *contention*：多个消息想通过同一条线路。（这也与 *bisecectionbandwidth* 的概念相关。）

在以下分析中，我们将假设 *wormhole routing*：消息在网络中建立一条路径，预留必要的线路，并且发送时间与距离无关

网络。也就是说，任何消息的发送时间都可以建模为

$$T(n) = \alpha + \beta n$$

无论源和目的地，只要必要的连接可用。

Exercise 3.27. 分析如上所述的递归倍增广播的运行时间，采用虫洞路由。用阻塞的 MPI 发送和接收调用实现此广播。如果你有 SimGrid，可用多个参数运行测试。

另一种避免争用的方法是让每个倍增阶段将网络划分为独立的两半。也就是说，进程 0 发送给 $P/2$，之后这两个进程在网络的两半部分重复该算法，分别发送给 $P/4$ 和 $3P/4$。

Exercise 3.28. 分析此递归倍增的变体。编写代码并在 SimGrid 上测量运行时间。

练习 3.29. 重新审视练习 3.27 并将阻塞调用替换为非阻塞 `MPI_Isend` / `MPI_Irecv` 调用。确保测试数据是否正确传播。

MPI 实现通常有多种算法，它们会动态切换。有时你可以通过环境变量自行确定选择。

TACC 注释。 关于 *Intel MPI*，请参见 https://software.intel.com/en-us/mpi-developer-reference-linux-i-mpi-adjust-family-environment-variables。

3.15 复习问题

对于所有判断题，如果你回答某个陈述为假，请给出一句话的解释。

复习 3.30. 你将如何使用 MPI 集合操作实现以下场景?

- 让每个进程计算一个随机数。你想将这些数中的最大值打印到屏幕上。
- 每个进程再次计算一个随机数。现在你想用它们的最大值来缩放这些数。
- 让每个进程计算一个随机数。你想打印出最大值是在哪个处理器上计算的。

Review 3.31. MPI 集合操作至少可以分为以下几类 1. 有根与无根 2. 使用统一缓冲区长度与可变长度缓冲区 3. 阻塞与非阻塞。请举出每种类型的例子。

Review 3.32. 对错题：集合例程全部都是关于在进程间传递用户数据的。

Review3.33. 对错题：一个 `MPI_Scatter` 调用会将相同的数据放到每个进程上。

复习 3.34. 判断正误：使用选项 `MPI_IN_PLACE` 时，在 `MPI_Reduce` 中只需要为发送缓冲区分配空间。

复习 3.35. 判断正误：使用选项 `MPI_IN_PLACE` 时，您只需要在 `MPI_Gather` 中为发送缓冲区分配空间。

Review3.36. 给定一个分布式数组，每个处理器存储

```
double x[N]; // N can vary per processor
```

给出基于 MPI 的近似代码，计算数组中的最大值，并将结果保留在每个处理器上。

Review 3.37.

```
double data[Nglobal];int myfirst = /* something */,
 mylast = /* something */;
for (int i=myfirst; i<mylast; i++) {if (i>0 && i<N-1) {
process_point( data,i,Nglobal );}}
void process_point( double *data,int i,int N ) {
data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
data[i] = f(data[i-1],data[i],data[i+1]);}
```

这在时间上可扩展吗？这在空间上可扩展吗？缺少的 MPI 调用是什么？

Review 3.38.

```
double data[Nlocal+2]; // include left and right neighbor
int myfirst = /* something */, mylast = myfirst+Nlocal;
for (int i=0; i<Nlocal; i++) {if (i>0 && i<N-1) {
process_point( data,i,Nlocal );}
void process_point( double *data,int i0,int n ) {
int i = i0+1;
data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
data[i] = f(data[i-1],data[i],data[i+1]);}
```

这在时间上是可扩展的吗？这在空间上是可扩展的吗？缺少的 MPI 调用是什么？

Review3.39. 数据与前一个问题相同，给出归一化数组的代码，即缩放每个元素使得 $\|x\|_2 = 1$。

Review3.40. 就像 `MPI_Allreduce` 等价于 `MPI_Reduce` 后跟 `MPI_Bcast`,`MPI_Reduce_scatter` 等价于以下至少一种组合。选择那些等价的，并讨论时间或空间复杂度的差异：

1. `MPI_Reduce` 后跟 `MPI_Scatter`; 2. `MPI_Gather` 后跟 `MPI_Scatter`; 3. `MPI_Allreduce` 后跟 `MPI_Scatter`; 4. `MPI_Allreduce` 后跟一个本地操作（哪个？）；5. `MPI_Allgather` 后跟一个本地操作（哪个？）。

Review3.41. 想出至少两种进行广播的算法。比较它们的渐近行为。

第 4 章

MPI 主题：点对点

4.1 阻塞点对点操作

假设你有一个数字数组 $x_i : i = 0, \ldots, N$，你想计算

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \ldots, N-1.$$

如图 2.6 所示，我们给每个处理器分配了连续的 x_i 和 y_i 的子集。定义 i_p 为处理器 p 计算的 y 的第一个索引。（处理器 p 计算的最后一个索引是多少？该处理器计算了多少个索引？）

我们经常谈论并行计算中的 所有者计算 模型：每个处理器 “ 拥有 ” 某些数据项，并计算它们的值。当然，用于此计算的值不必是本地的，这就是通信需求产生的原因。

让我们研究处理器 p 如何计算它所拥有的 i 个值的 y_i。假设进程 p 也存储这些相同索引的值 x_i。现在，对于许多值 i，它可以评估计算

$$y_i = (x_{i-1} + x_i + x_{i+1})/3$$

本地（图 4.1 ）。

然而，在处理器 p 上计算第一个索引 i_p 的 y 时存在一个问题：

$$y_{i_p} = (x_{i_p-1} + x_{i_p} + x_{i_p+1})/3$$

左边的点，x_{i_p-1}，不存储在进程 p （它存储在 $p-1$ ），因此进程 p 无法立即使用它。（图 4.2 ）。最后一个索引的情况类似，p 试图计算的值只存在于 $p+1$。

你会看到需要处理器之间，或者技术上称为 点对点 的信息交换。MPI 通过匹配的发送和接收调用来实现这一点：

- 一个进程向另一个特定进程发送数据；
- 另一个进程从该源执行特定的接收。

我们现在将详细讨论发送和接收例程。

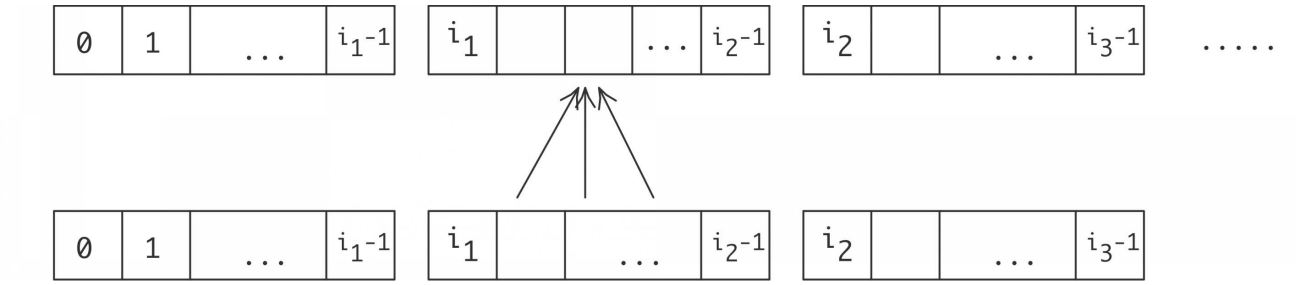


图 4.1：并行三点平均，内部点情况

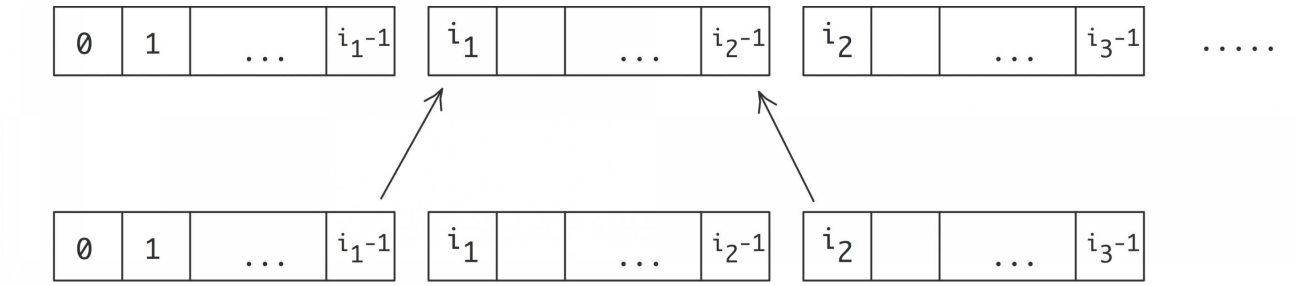


图 4.2：并行三点平均，边缘点情况

4.1.1 Example: ping-pong

仅两个进程之间信息交换的一个简单场景是 *ping-pong*：进程 A 向进程 B 发送数据，进程 B 再将数据发送回 A。这不是一个对应用程序特别相关的操作，尽管它经常被用作基准测试。这里我们讨论它是为了说明基本思想。

这意味着进程 A 执行代码

```
MPI_Send( /* to: */ B ..... );
MPI_Recv( /* from: */ B ... );
```

而进程 B 执行

```
MPI_Recv( /* from: */ A ... );
MPI_Send( /* to: */ A ..... );
```

由于我们以 SPMD 模式编程，这意味着我们的程序看起来像：

```
if ( /* I am process A */ ) {
  MPI_Send( /* to: */ B ..... );
  MPI_Recv( /* from: */ B ... );
} else if ( /* I am process B */ ) {
  MPI_Recv( /* from: */ A ... );
  MPI_Send( /* to: */ A ..... );
}
```

Remark 8 发送和接收调用的结构展示了 *MPI* 的对称性：每个目标进程都通过相同的发送调用被访问，无论它是运行在与发送者相同的多核芯片上，还是

图 4.1 MPI_Send

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Send (				
MPI_Send_c (				
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in send buffer	int MPI_Count	INTEGER	IN
datatype	datatype of each send buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
tag	message tag	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPL:

```
template<typename T >
void mpl::communicator::send
   ( const T scalar&,int dest,tag = tag(0) ) const
   ( const T *buffer,const layout< T > &,int dest,tag = tag(0) ) const
   ( iterT begin,iterT end,int dest,tag = tag(0) ) const
T : scalar type
begin : begin iterator
end : end iterator
```

Python:

```
Python native:
MPI.Comm.send(self, obj, int dest, int tag=0)
Python numpy:
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

在机房另一端的计算节点上，需要经过多个网络跳转才能到达。当然，任何有自尊的 *MPI* 实现都会针对发送方和接收方访问相同共享内存的情况进行优化。这意味着 *send/recv* 对被实现为从发送缓冲区到接收缓冲区的复制操作，而不是网络传输。

4.1.2 Send call

阻塞发送命令是 `MPI_Send` (figure4.1)。示例：

```
// sendandrecv.cdouble send_data = 1.;MPI_Send
( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
/* to: */ receiver, /* tag: */ 0,
/* communicator: */ comm);
```

send 调用包含以下元素。

缓冲区 发送缓冲区 由缓冲区 / 计数 / 数据类型三元组描述。详见 3.2.4 节的讨论。

目标 消息目标 是一个显式的进程秩 *(rank)* 用于发送。该秩是从零到 `MPI_Comm_size` 的数字。允许进程向自身发送，但这可能导致运行时死锁；详见 4.1.4 节的讨论。值 `MPI_PROC_NULL` 是允许的：使用该值作为目标不会发送或接收任何消息。

标签 接下来，消息可以有一个 标签。许多应用程序中，每个发送者在任一时刻只向给定接收者发送一条消息。对于同一发送者 / 接收者对之间存在多条同时消息的情况，标签可用于区分这些消息。

通常，标签值为零是安全的。实际上，面向对象的 MPI 接口通常将标签作为一个可选参数，默认值为零。如果您确实使用标签值，可以使用键 `MPI_TAG_UB` 查询可用的最大值；详见 15.1.2 节。

Communicator 最后，与绝大多数 MPI 调用一样，有一个 communicator 参数为发送事务提供上下文。为了匹配发送和接收操作，它们需要在同一个 communicator 中。

MPL 注释 *24*：缓冲区类型安全。

- 标量数据类型通过模板（以及 “ 基于参数的查找 ” ）处理：由编译器推导。
- 计数 > 1 在布局数据类型中声明。

MPLnote 25: Blocking send and receive. MPL 使用标签的默认值，并且可以推断缓冲区的类型。发送标量变为：

```
// sendscalar.cxx
if (comm_world.rank()==0) {
  double pi=3.14;
  comm_world.send(pi, 1);  // send to rank 1
  cout << "sent: " << pi << '\n';
} else if (comm_world.rank()==1) {
  double pi=0;
  comm_world.recv(pi, 0);  // receive from rank 0
  cout << "got : " << pi << '\n';
}
```

(参见注释 10.)

MPL 注释 *26*：发送数组。 MPL 可以发送静态数组而无需进一步的布局说明：

```
// sendarray.cxx
double v[2][2][2];
  comm_world.send(v, 1);  // send to rank 1
  comm_world.recv(v, 0);  // receive from rank 0
```

发送向量使用通用机制：

```
// sendbuffer.cxx
std::vector<double> v(8);
mpl::contiguous_layout<double> v_layout(v.size());
  comm_world.send(v.data(), v_layout, 1);  // send to rank 1
  comm_world.recv(v.data(), v_layout, 0);  // receive from rank 0
```

（参见注释 11.）

MPL 注释 *27*：迭代器布局。 非连续可迭代对象可以使用 `iterator_layout` 发送：

```
std::list<int> v(20, 0);
mpl::iterator_layout<int> l(v.begin(), v.end());
comm_world.recv(&(*v.begin()), l, 0);
```

4.1.3 接收调用

基本的阻塞接收命令是 `MPI_Recv`（图 4.2）。

一个例子：

```
double recv_data;MPI_Recv
( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
/* from: */ sender, /* tag: */ 0,/* communicator: */ comm,
/* recv status: */ MPI_STATUS_IGNORE);
```

这在结构上类似于发送调用，但有一些例外。

缓冲区 接收缓冲区 具有与发送调用相同的缓冲区 / 计数 / 数据参数。然而，这里的 *count* 参数表示缓冲区的大小，而不是消息的实际长度。这为传入消息的长度设置了上限。

- 对于接收长度未知的消息，使用 `MPI_Probe`；参见章节 4.4.1。
- 消息长度超过缓冲区大小将导致溢出错误，可能返回错误或终止程序；详见第 15.2.2 节。

接收消息的长度可以从状态对象中确定；详见第 4.3 节。

镜像接收调用的目标参数，`MPI_Send` 调用，`MPI_Recv` 具有消息源参数。该参数可以是特定的 rank，也可以是通配符 `MPI_ANY_SOURCE`。在后一种情况下，实际的消息源可以在消息接收后确定；参见第 4.3 节 {v18}。还允许使用源值 `MPI_PROC_NULL`，这会使接收立即成功但不接收任何数据。

MPL 注释 *28*：任意源。常数 *mpl*::any_source 等于 `MPI_ANY_SOURCE`（由 *constexpr*）。

图 4.2 MPI_Recv

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Recv (				
MPI_Recv_c (				
buf	initial address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in receive buffer	int MPI_Count	INTEGER	IN
datatype	datatype of each receive buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
tag	message tag or MPI_ANY_TAG	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

MPL:

```
template<typename T >status mpl::communicator::recv
( T &,int,tag = tag(0) ) const inline
( T *,const layout< T > &,int,tag = tag(0) ) const
( iterT  begin,iterT  end,int  source, tag  t = tag(0) ) constPython:
Comm.Recv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,
Status status=None)Python native:
recvbuf = Comm.recv(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
Status status=None)
```

Tag 类似于消息源，接收调用的消息标签可以是特定值或通配符，在这种情况下 MPI_ANY_TAG。

Python 注释 *13*：消息标签。Python 调用合理地使用默认 tag=0，但你可以指定你自己的标签值。在接收调用中，标签通配符是 *MPI.ANY_TAG*。

Communicator communicator 参数几乎不需多言。

状态 该 MPI_Recv 命令有一个发送调用所没有的参数： MPI_Status 对象，描述了 消息状态。这提供了关于接收消息的信息，例如如果你使用了源或标签的通配符。详见 4.3 节，了解更多关于状态对象的内容。

备注 9 如果你不关心状态，就像本书中许多示例那样，你可以指定常量 *MPI_STATUS_IGNORE*。注意 *MPI_Recv* 的签名将状态参数列为‘输出’；这

参数的“方向”当然只适用于你没有指定该常量的情况。

练习 4.1. 实现乒乓程序。使用 `MPI_Wtime` 添加计时器。对于 `status` receive 调用的参数，使用 `MPI_STATUS_IGNORE`。

- 运行多个乒乓（比如一千次），并将计时器放在循环外。第一次运行可能会更长；尝试忽略它。
- 先在同一节点上运行两个通信进程的代码，然后在不同节点上运行。你看到差异了吗?
- 然后修改程序以使用更长的消息。随着消息大小增加，时间是如何变化的?

作为额外奖励，你能做一个回归来确定 α, β 吗？（这个练习的骨架在名为 `pingpong` 的文件中。）

练习 4.2. 拿你的乒乓程序并修改它，使一半的处理器作为源，另一半作为目标。乒乓时间会增加吗？观察到的行为是否取决于你如何选择这两组?

4.1.4 阻塞通信的问题

你可能会认为 send 调用会将数据放到网络中的某个地方，发送代码可以在此调用后继续执行，如图 4.3 左侧所示。但这种理想情况并不现实：它假设

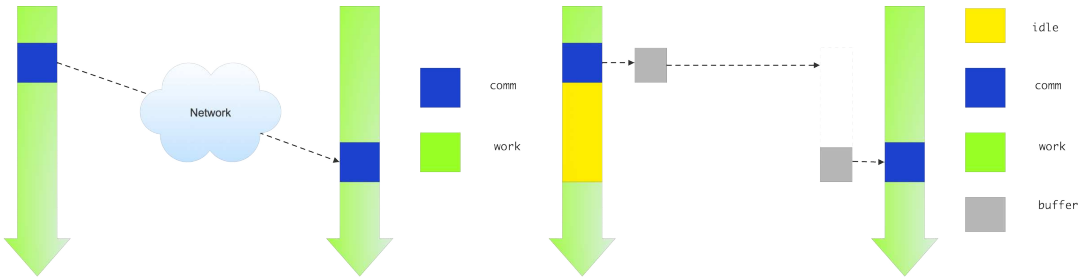


图 4.3：理想（左）和实际（右）发送 - 接收交互的示意图

网络中的某个地方有足够的缓冲区容量来容纳所有正在传输的消息。事实并非如此：数据驻留在发送方，发送调用会阻塞，直到接收方接收完所有数据。（对于小消息有一个例外，如下一节所述。）

使用 `MPI_Send` 和 `MPI_Recv` 被称为阻塞通信：当你的代码执行到发送或接收调用时，会阻塞直到调用成功完成。从技术上讲，阻塞操作被称为非本地，因为它们的执行依赖于进程本地之外的因素。参见第 5.4 节。

4.1.4.1 死锁

假设两个进程需要交换数据，考虑以下伪代码，它旨在在进程 0 和 1 之间交换数据：

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
receive(source=other);send(target=other);
```

想象两个进程执行这段代码。它们都发出了 send 调用 …… 然后无法继续，因为它们都在等待对方发出与它们的 receive 调用对应的 send 调用。这被称为死锁。

4.1.4.2 Eager vs rendezvous 协议

消息可以使用（至少）两种不同的协议发送：

1. Rendezvous 协议，2.
   Eager 协议。

rendezvous 协议 是最通用的。发送消息需要几个步骤 ：

1. 发送方发送一个头部，通常包含消息信封：描述消息的元数据；2. 接收方返回一个 “ 准备发送 ” 消息；3. 发送方发送实际数据。

这样做的目的是为大消息准备接收缓冲区空间。然而，这意味着发送方必须等待接收方的某个返回消息，使得该行为成为同步消息。

对于 eager 协议，考虑以下示例：

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
send(target=other);receive(source=other);
```

使用同步协议时，你应该会遇到死锁，因为发送调用将等待接收操作被发布。

然而，在实际中，这段代码通常是可行的。原因是 MPI 实现有时会发送小消息，而不管接收是否已经被发布。这被称为 *eager send*，它依赖于一定量可用缓冲区空间的存在。使用这种行为的大小有时被称为 *eager limit*。

为了说明 `MPI_Send` 中的 eager 和阻塞行为，考虑一个逐渐发送更大消息的示例。从屏幕输出中你可以看到落在 eager limit 之下的最大消息；之后代码因死锁而挂起。

```
// sendblock.c
other = 1-procno;
/* loop over increasingly large messages */
for (int size=1; size<2000000000; size*=10) {
  sendbuf = (int*) malloc(size*sizeof(int));
  recvbuf = (int*) malloc(size*sizeof(int));
  if (!sendbuf || !recvbuf) {
    printf("Out of memory\n"); MPI_Abort(comm,1);
```

```
    }MPI_Send(sendbuf,size,MPI_INT,other,0,comm);
MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
/* If control reaches this point,  the send call
did not block. If the send call blocks,
we do not reach this point,  and the program will hang.*/
if (procno==0)
printf("Send did not block for size %d\n",size);
free(sendbuf); free(recvbuf);}
```

```
!! sendblock.F90
     other = 1-mytid
     size = 1
     do
        allocate(sendbuf(size)); allocate(recvbuf(size))
        print *,size
        call MPI_Send(sendbuf,size,MPI_INTEGER,other,0,comm,err)
        call MPI_Recv(recvbuf,size,MPI_INTEGER,other,0,comm,status,err)
        if (mytid==0) then
           print *,"MPI_Send did not block for size",size
        end if
        deallocate(sendbuf); deallocate(recvbuf)
        size = size*10
        if (size>2000000000) goto 20
     end do
20   continue
```

```
## sendblock.py
size = 1
while size<2000000000:
    sendbuf = np.empty(size, dtype=int)
    recvbuf = np.empty(size, dtype=int)
    comm.Send(sendbuf,dest=other)
    comm.Recv(sendbuf,source=other)
    if procid<other:
        print("Send did not block for",size)
    size *= 10
```

如果你想让代码对所有消息大小表现出相同的阻塞行为，可以通过使用 `MPI_Ssend` 强制发送调用为阻塞，该调用序列与 `MPI_Send` 相同，但不允许 eager 发送。

```
// ssendblock.c
other = 1-procno;
sendbuf = (int*) malloc(sizeof(int));
recvbuf = (int*) malloc(sizeof(int));
size = 1;
MPI_Ssend(sendbuf,size,MPI_INT,other,0,comm);
MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
printf("This statement is not reached\n");
```

形式上，你可以如下描述死锁。绘制一个图，其中每个进程是一个节点，如果进程 A 在等待进程 B，则从 A 到 B 画一条有向弧。如果该有向图存在环路，则存在死锁。

上述示例中死锁的解决方案是先执行从 0 到 1 的发送，然后再从 1 到 0 （或反过来）。所以代码看起来像这样：

```
if ( /* I am processor 0 */ ) {
  send(target=other);
  receive(source=other);
} else {
  receive(source=other);
  send(target=other);
}
```

Eager 发送也会影响非阻塞发送。非阻塞发送后的 wait 调用将立即返回，
d无论任何接收调用，如果消息在 eager 限制之下：

代码：

```
// eageri.c
printf("Sending %lu elements\n",n);
MPI_Request request;
MPI_Isend(buffer,n,MPI_DOUBLE,processB,0,comm,&request);
MPI_Wait(&request,MPI_STATUS_IGNORE);
printf(".. concluded\n");
```

输出：

```
Setting eager limit to 5000
    ↪bytes
TACC:  Starting up job
    ↪4049189
TACC:  Starting parallel
    ↪tasks...
Sending 1 elements
.. concluded
Sending 10 elements
.. concluded
Sending 100 elements
.. concluded
Sending 1000 elements
^C[mpiexec@c207-029.frontera.tacc.utexas.edu]
    ↪Sending Ctrl-C to
    ↪processes as requested
```

eager 限制是实现相关的。例如，对于 *Intel MPI* 有变量 `I_MPI_EAGER_THRESHOLD` （旧版本）或 `I_MPI_SHM_EAGER_THRESHOLD`；对于 *mvapich2* 是 `MV2_IBA_EAGER_THRESHOLD`，而对于 *OpenMPI* 则是 `--mca` 选项 `btl_openib_eager_limit` 和 `btl_openib_rndv_eager_limit`。

4.1.4.3 序列化

阻塞通信还有第二个更为微妙的问题。考虑这样一种场景：每个处理器都需要将数据传递给其后继处理器，即下一个更高秩的处理器。基本思路是先发送给你的后继处理器，然后从你的前驱处理器接收。由于最后一个处理器没有后继处理器，因此它跳过发送，同样第一个处理器跳过接收。伪代码如下：

```
successor = mytid+1; predecessor = mytid-1;
if ( /* I am not the last processor */ )
  send(target=successor);
```

```
if ( /* I am not the first processor */ )
  receive(source=predecessor)
```

Exercise4.3. （课堂练习）每个学生右手持一张纸 —— 左手放在背后 —— 我们要执行：1. 把纸传给右边的邻居；2. 从左边的邻居接收纸。包括第一个和最后一个进程的边界条件，程序如下：1. 如果你不是最右边的学生，向右转并把纸传给右边的邻居。2. 如果你不是最左边的学生，向左转并从左边的邻居接收纸。

这段代码不会死锁。除了最后一个处理器外，所有处理器都在发送调用上阻塞，但最后一个处理器执行接收调用。因此，倒数第二个处理器可以完成发送，随后继续执行接收，这又使得另一个发送得以进行，依此类推。

从某种意义上说，这段代码实现了你的预期：它会终止（而不是永远挂起死锁）并正确交换数据。然而，执行现在遭遇了意外的串行化：任何时刻只有一个处理器处于活动状态，因此本应是并行的操作变成了

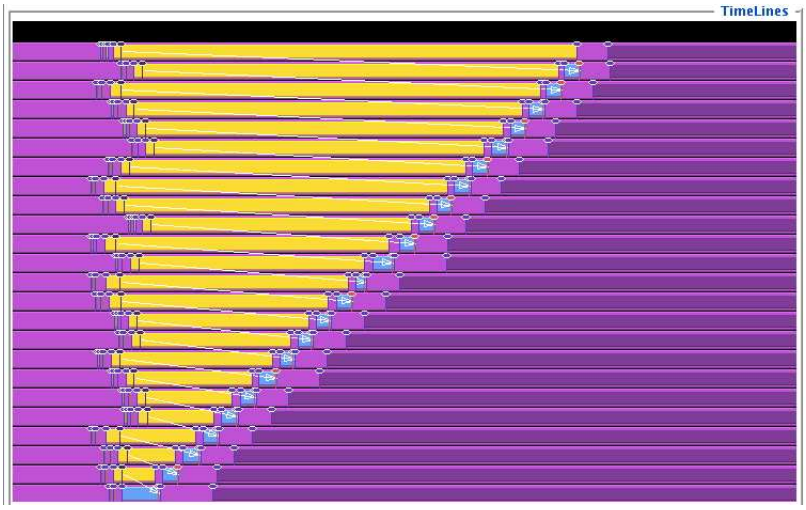


图 4.4：一个简单发送 - 接收代码的跟踪

一个顺序程序。这在图 4.4 中有所说明。

练习 4.4. 使用 `MPI_Send` 和 `MPI_Recv` 调用实现上述算法。运行代码，并使用 TAU 重现图 4.4 的跟踪输出。如果你没有 TAU，能否通过计时来展示这种串行化行为，例如在增加进程数量时运行？（本练习的骨架代码名为 `rightsend`。）

可以协调你的进程以获得高效且无死锁的执行，但这样做有点繁琐。

Exercise 4.5. 上述解决方案对每个处理器一视同仁。你能想出一个使用阻塞发送和接收但不受串行化行为影响的解决方案吗?

我们将在下一节探讨更好的解决方案。

4.1.5 Bucket brigade

前一个练习的问题在于一个概念上是并行的操作在执行时变成了串行。另一方面，有时操作本质上确实是串行的。一个例子是 *bucket brigade* 操作，其中一段数据依次传递给一系列处理器。

练习 4.6. 取练习 4.4 的代码并修改，使得来自进程零的数据能够传播到每个进程。具体来说，计算所有部分和 $\sum_{i=0}^{p} i^2$:

$$\begin{cases} x_0 = 1 & \text{on process zero} \\ x_p = x_{p-1} + (p+1)^2 & \text{on process } p \end{cases}$$

使用 `MPI_Send` 和 `MPI_Recv`；确保顺序正确。
思考题：这里涉及的所有量都是整数。使用整数数据类型是个好主意吗?
（本练习的骨架代码名为 `bucketblock`。）

备注 10 有一个 *MPI_Scan* 例程（第 *3.4* 节）执行相同的计算，但计算效率更高。因此，本练习仅用于说明原理。

4.1.6 Pairwise exchange

上面你已经看到，使用阻塞发送时，发送和接收调用的精确顺序至关重要。使用错误的顺序会导致死锁，或者在并行中效率极低。MPI 提供了一种解决此问题的方法，适用于许多场合：组合发送 / 接收调用 `MPI_Sendrecv`(图 4.3)。

The sendrecv 调用效果很好，如果每个进程恰好与一个发送者和一个接收者配对。你would then write

```
sendrecv( ....from... ...to... );
```

通过正确选择源和目标。例如，向右侧邻居发送数据:

```
MPI_Comm_rank(comm,&procno);
MPI_Sendrecv( ....
/* from: */ procno-1... ...
/* to:   */ procno+1... );
```

图 4.3 MPI_Sendrecv

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Sendrecv (MPI_Sendrecv_c (				
sendbuf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcount	number of elements in send buffer	int MPI_Count	INTEGER	IN
sendtype	type of elements in send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
sendtag	send tag	int	INTEGER	IN
recvbuf	initial address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcount	number of elements in receive buffer	int MPI_Count	INTEGER	IN
recvtype	type of elements receive buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
recvtag	receive tag or MPI_ANY_TAG	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

MPL:

```
template<typename T >status mpl::communicator::sendrecv
( const T & senddata, int dest,   tag sendtag,T & recvdata, int source,
 tag recvtag) const
( const T * senddata, const layout< T > & sendl, int dest,   tag sendtag,
T * recvdata, const layout< T > & recvl, int source, tag recvtag) const
( iterT1 begin1, iterT1 end1, int dest,   tag sendtag,iterT2 begin2,
 iterT2 end2, int source, tag recvtag) constPython:
```

```
Sendrecv(self,sendbuf, int dest, int sendtag=0,recvbuf=None,
 int source=ANY_SOURCE, int recvtag=ANY_TAG,
Status status=None)
```

该方案对除第一个和最后一个进程外的所有进程都是正确的。为了在这些进程上使用 sendrecv 调用，我们使用 MPI_PROC_NULL 来处理端点与之通信的不存在的进程。

```
MPI_Comm_rank( .... &procno );
if ( /* I am not the first processor */ )
  predecessor = procno-1;
else
  predecessor = MPI_PROC_NULL;
if ( /* I am not the last processor */ )
  successor = procno+1;
else
  successor = MPI_PROC_NULL;
sendrecv(from=predecessor,to=successor);
```

sendrecv 调用由所有处理器执行。

除最后一个处理器外，所有处理器都向其邻居发送；对于最后一个处理器，MPI_PROC_NULL 的目标值意味着“发送到空处理器”：实际上不进行发送。

L同样，从 MPI_PROC_NULL 接收成功且不改变接收缓冲区。对应的 MPI_Status 对象有源 MPI_PROC_NULL，标签 MPI_ANY_TAG，计数为零。

R备注 11 *The* *MPI_Sendrecv* 可以与普通的发送和接收调用互操作，既包括阻塞也包括非阻塞。因此，也可以用简单的发送或接收替换端点处的 *MPI_Sendrecv* 调用。

MPL 注释 *29*：发送 - 接收调用。 MPL 中的发送 - 接收调用在指定发送和接收缓冲区方面具有与单独的发送和接收调用相同的可能性：标量、布局、迭代器。然而，在九种可能的例程签名中，仅提供了发送和接收缓冲区以相同方式指定的版本。此外，发送和接收标签需要指定；它们没有默认值。

```
// sendrecv.cxx
mpl::tag_t t0(0);
comm_world.sendrecv
( mydata,sendto,t0,leftdata,
recvfrom,t0 );
```

```
// sendrecvarray.cxx
mpl::tag_t t0(0);
mpl::contiguous_layout<double>
    ↪twofloats(2);
comm_world.sendrecv
  ( mydata,twofloats,sendto,t0,
    leftdata,twofloats,recvfrom,t0 );
```

练习 4.7. 重新做练习 4.3，但使用 MPI_Sendrecv。制作一个跟踪。它看起来与串行的 send/recv 代码有区别吗？如果你没有 TAU，使用不同数量的进程运行你的代码，并证明运行时间基本保持不变。

此调用使得两个处理器之间交换数据变得简单：双方都将对方指定为目标和源。然而，目标和源之间不必存在这样的关系：可以从某种顺序中的前驱接收，并发送给该顺序中的后继；见图 4.5。

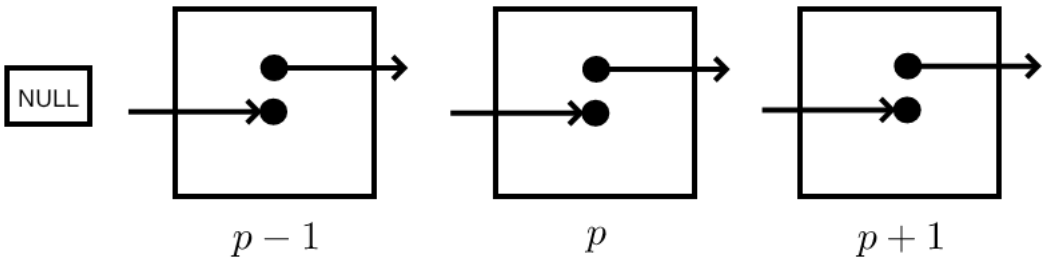


图 4.5：一个 MPI Sendrecv 调用

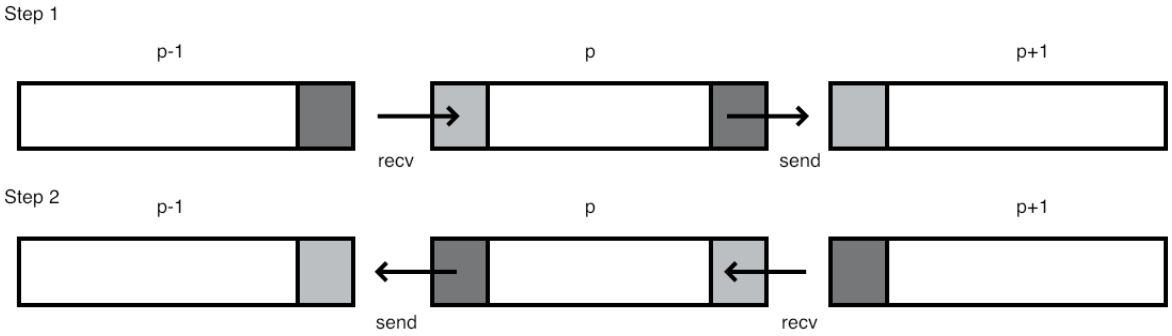


图 4.6：执行三点组合的两步发送 / 接收

.

对于上述三点组合方案，你需要将数据向左和向右移动，因此你需要两个 `MPI_Sendrecv` 调用；见图 4.6。

练习 4.8. 使用 `MPI_Sendrecv` 实现上述三点组合方案；每个处理器只需向其邻居发送一个数字。（该练习的骨架代码名为 `sendrecv`。）

Hints for this exercise:

- 每个进程执行一次发送和一次接收；如果某个进程需要跳过其中之一，可以在发送或接收的指定中将 `MPI_PROC_NULL` 指定为另一个进程。这样对应的操作将不会执行。
- 与简单的 send/recv 调用一样，进程必须匹配：如果进程 p 指定 p' 作为发送调用部分的目标，那么 p' 需要指定 p 作为接收调用部分的源。

以下练习让你使用 send-receive 调用实现一个排序算法 1。

练习 4.9. 一个非常简单的排序算法是 *swapsort* 或奇偶换位排序：处理器对成对比较数据，并在必要时交换。基本步骤称为比较并交换：在一对处理器中，每个处理器将其数据发送给对方；一个保留较小的值，另一个保留较大的值。为简化起见，本练习中每个处理器仅给出一个数字。

1. 有一个 `MPI_Compare_and_swap` 调用。不要使用它。

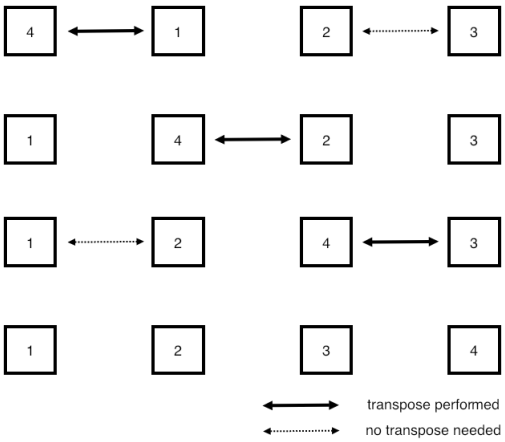


图 4.7：4 个元素的奇偶换位排序。

换位排序算法分为偶数和奇数阶段，其中在偶数阶段处理器 $2i$ 和 $2i+1$ 比较并交换数据，在奇数阶段处理器 $2i+1$ 和 $2i+2$ 进行比较和交换。你需要重复此过程 $P/2$ 次，其中 P 是处理器的数量；参见图 4.7。使用 `MPI_Sendrecv` 实现此算法。（如有需要，边界情况使用 `MPI_PROC_NULL` 。）使用 gather 调用打印排序过程开始和结束时分布式数组的全局状态。

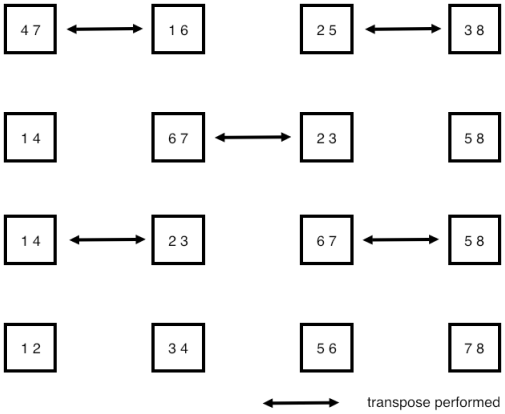


Figure 4.8: Odd-even transposition sort on 4 processes, holding 2 elements each.

备注 12 不可能像第 *`MPI_IN_PLACE`* 节 *3.3.2* 中那样使用缓冲区。相反，例程 *`MPI_Sendrecv_replace`* *(* 图 *4.4)* 只有一个缓冲区，既用作发送缓冲区也用作接收缓冲区。当然，这要求发送和接收消息都能放入该缓冲区。

练习 4.10. 将此练习扩展到每个进程持有相同数量元素且多于 1 个的情况。参考图 4.8 以获取灵感。这是巧合吗?

图 4.4 MPI_Sendrecv_replace

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Sendrecv_replace (				
MPI_Sendrecv_replace_c (				
buf	initial address of send and receive buffer	void*	TYPE(*), DIMENSION(..)	INOUT
count	number of elements in send and receive buffer	int MPI_Count	INTEGER	IN
datatype	type of elements in send and receive buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
sendtag	send message tag	int	INTEGER	IN
source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
recvtag	receive message tag or MPI_ANY_TAG	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

algorithmtakes the same number of steps as in the single scalar case? 以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

有非阻塞和持久版本的 `MPI_Sendrecv`： `MPI_Isendrecv`， `MPI_Sendrecv_init`， `MPI_Isendrecv_replace`， `MPI_Sendrecv_replace_init`。*MPI-4* 材料结束

4.2 非阻塞点对点操作

通信结构通常反映了操作的结构。对于一些规则的应用，我们也会得到规则的通信模式。再次考虑上述操作：

$$y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \ldots, N-1$$

Doing this in parallel induces communication, as pictured in figure 4.1.

我们注意到：

- 数据是一维的，并且我们有处理器的线性排序。
- 该操作涉及相邻的数据点，我们与相邻的处理器进行通信。

上面你已经看到如何在处理器对之间使用信息交换

- using `MPI_Send` and `MPI_Recv`，如果你小心的话；或者
- using `MPI_Sendrecv`，只要确实存在某种处理器配对。

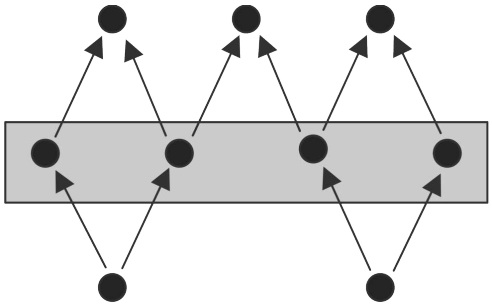


图 4.9：发送 / 接收模式不平衡的处理器

然而，在某些情况下，不可能、不高效或不方便进行这种确定性的发送和接收调用设置。图 4.9 展示了这样一种情况，其中处理器以一般图模式组织。在这里，处理器的发送和接收次数不必匹配。

在这种情况下，人们希望能够声明 “ 这些是预期的传入消息 ”，而不必按顺序等待它们。同样，人们希望声明传出消息，而不必按任何特定顺序执行。对发送和接收施加任何顺序很可能会遇到上述观察到的序列化行为，或者至少效率低下。

4.2.1 非阻塞发送和接收调用

在上一节中你已经看到，如果想避免死锁和性能问题，阻塞通信会使编程变得棘手。这些例程的主要优点是你可以完全控制数据的位置：如果发送调用返回，说明数据已成功接收，发送缓冲区可以用于其他用途或释放。

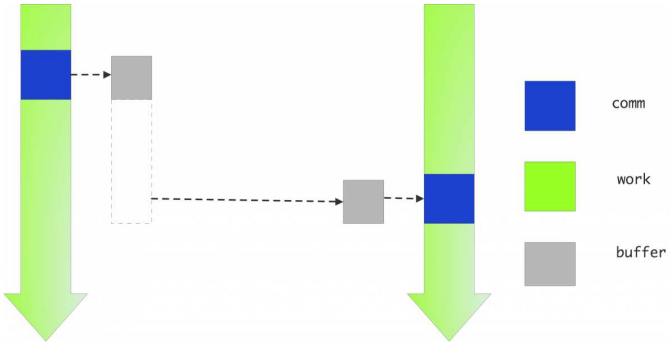


图 4.10：非阻塞发送

相比之下，非阻塞调用 `MPI_Isend`（图 4.5）和 `MPI_Irecv`（图 4.6）（其中 ‘I’ 代表 ‘*immediate*’ 或 ‘*incomplete*’ ）不会等待其对应操作完成：实际上它们告诉运行时系统 ‘ 这里有一些数据，请按如下方式发送 ’ 或 ‘ 这里有一些缓冲区空间，预计会有这样的数据到来 ’ 。这在图 4.10 中有所说明。

图 4.5 MPI_Isend

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Isend (				
MPI_Isend_c (				
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in send buffer	int MPI_Count	INTEGER	IN
datatype	datatype of each send buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
tag	message tag	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

MPL:

Python:

```
request = MPI.Comm.Isend(self, buf, int dest, int tag=0)
```

```
// isendandirecv.c
double send_data = 1.;
MPI_Request request;
MPI_Isend
  ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
        /* to: */ receiver, /* tag: */ 0,
        /* communicator: */ comm,
        /* request: */ &request);
MPI_Wait(&request,MPI_STATUS_IGNORE);
```

```
double recv_data;
MPI_Request request;
MPI_Irecv
  ( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
        /* from: */ sender, /* tag: */ 0,
        /* communicator: */ comm,
        /* request: */ &request);
MPI_Wait(&request,MPI_STATUS_IGNORE);
```

发出 MPI_Isend / MPI_Irecv 调用有时被称为 发布 发送 / 接收。

图 4.6 MPI_Irecv

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Irecv (				
MPI_Irecv_c (				
buf	initial address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in receive buffer	int MPI_Count	INTEGER	IN
datatype	datatype of each receive buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
tag	message tag or MPI_ANY_TAG	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

MPL:

```
template<typename T >
irequest mpl::communicator::irecv
   ( const T & data, int src, tag t = tag(0) ) const;
   ( const T * data, const layout< T > &  l, int  src, tag  t = tag(0) ) const;
   ( iterT  begin, iterT  end, int  src, tag  t = tag(0) ) const;
```

Python:

```
recvbuf = Comm.irecv(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG,
    Request request=None)
Python numpy:
Comm.Irecv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG,
    Request status=None)
```

4.2.2 请求完成：wait 调用

从 MPI_Isend / MPI_Irecv 的定义中，你可以看到非阻塞例程会产生一个 MPI_Request 对象。然后可以使用该请求来查询操作是否已完成。你可能还会注意到 MPI_Irecv 例程不会产生一个 MPI_Status 对象。这是合理的：状态对象描述的是实际接收到的数据，而在 MPI_Irecv 调用完成时还没有接收到数据。

等待请求完成可以使用多种例程。我们首先考虑 MPI_Wait（图 4.7）。它以请求作为输入，并输出一个 MPI_Status。如果你不需要状态对象，可以传递 MPI_STATUS_IGNORE。

```
// hangwait.c
if (procno==sender) {
  for (int p=0; p<nprocs-1; p++) {
    double send = 1.;
    MPI_Send( &send,1,MPI_DOUBLE,p,0,comm);
  }
```

图 4.7 MPI_Wait

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Wait (				
request	request	MPI_Request*	TYPE (MPI_Request)	INOUT
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

Python:

```
MPI.Request.Wait(type cls, request, status=None)
```

```
} else {
  double recv=0.;
  MPI_Request request;
  MPI_Irecv( &recv,1,MPI_DOUBLE,sender,0,comm,&request);
  do_some_work();
  MPI_Wait(&request,MPI_STATUS_IGNORE);
}
```

（注意，此示例使用了阻塞和非阻塞操作的混合：阻塞发送与非阻塞接收配对。）

请求是通过引用传递的，以便 wait 例程可以释放它：

- wait 调用释放请求对象，且
- 将变量的值设置为 MPI_REQUEST_NULL。

(详见章节 4.2.4。)

MPL 注释 *30*：来自非阻塞调用的请求。非阻塞例程的函数结果是 irequest 。注意：这不是像 C 接口中那样通过引用传递的参数。各种 wait 调用是 irequest 类的方法。

```
double recv_data;
mpl::irequest recv_request =
  comm_world.irecv( recv_data,sender );
recv_request.wait();
```

不能默认构造请求变量：

```
// DOES NOT COMPILE:
mpl::irequest recv_request;
recv_request = comm.irecv( ... );
```

这意味着不能按正常顺序先声明然后填充请求变量。

实现说明：wait 调用总是返回一个 status_t 对象；不对其赋值意味着会调用其析构函数。

现在我们详细讨论各种 wait 调用。这些是阻塞的；非阻塞版本见第 4.2.3 节。

图 4.8 MPI_Waitall

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Waitall (				
count	list length	int	INTEGER	IN
array_of_requests	array of requests	MPI_Request[]	TYPE (MPI_Request) (count)	INOUT
array_of_statuses	array of status objects	MPI_Status[]	TYPE (MPI_Status) (*)	OUT
)				

Python:

```
MPI.Request.Waitall(type cls, requests, statuses=None)
```

4.2.2.1 等待单个请求

MPI_Wait 等待单个请求。如果您确实在等待单个非阻塞通信完成，这是正确的例程。如果您在等待多个请求，可以在循环中调用此例程。

```
for (p=0; p<nrequests ; p++) // Not efficient!
  MPI_Wait(&request[p],&(status[p]));
```

然而，如果第一个请求的完成时间远晚于其他请求，这将效率低下：你的等待进程将有大量空闲时间。在这种情况下，使用以下例程之一。

4.2.2.2 等待所有请求

MPI_Waitall (图 4.8) 允许你等待多个请求，无论它们以何种顺序完成都无关紧要。使用此例程比上面的循环更易于编码，并且可能更高效。

```
// irecvloop.cMPI_Request requests =
(MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {int
left_p  = (p-1+nprocs) % nprocs, right_p = (p+1) % nprocs;
send_buffer[p] = nprocs-p;
MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);}
/* your useful code here */
MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);
```

T输出参数是一个数组或 MPI_Status 对象。如果你不需要状态对象，可以传递 MPI_STATUSES_IGNORE.

作为示例，我们实现练习 4.4，及其在图 4.4 中的跟踪，采用非阻塞执行和 `MPI_Waitall`。图 4.11 显示了该代码变体的跟踪。

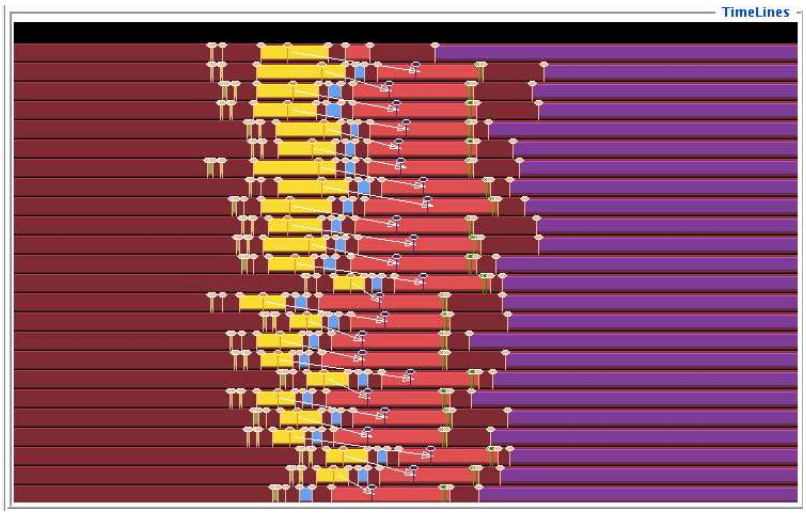


图 4.11：邻近处理器之间非阻塞发送的跟踪

练习 4.11. 重新审视练习 4.6 并考虑将阻塞调用替换为非阻塞调用。你能将 `MPI_Isend` / `MPI_Irecv` 调用和相应的 `MPI_Wait` 调用相隔多远？（该练习有一个名为 `bucketpipenonblock` 的骨架代码。）

练习 4.12. 创建两个正整数的分布式数组。取两者的差集：第一个数组需要被转换，去除其中存在于第二个数组的数字。你如何用 `MPI_Allgather` 调用来解决这个问题？为什么这样做不是一个好主意？请改用循环桶传递算法解决此练习。（该练习有一个名为 `setdiff` 的骨架代码。）

Python 注释 *14*：处理单个请求。非阻塞例程如 `MPI_Isend` 返回一个请求对象。`MPI_Wait` 是一个类方法，而不是请求对象的方法：

```
## irecvsingle.py
sendbuffer = np.empty( nprocs, dtype=int )
recvbuffer = np.empty( nprocs, dtype=int )

left_p = (procid-1) % nprocs
right_p = (procid+1) % nprocs
send_request = comm.Isend\
    ( sendbuffer[procid:procid+1],dest=left_p)
recv_request = comm.Irecv\
    ( sendbuffer[procid:procid+1],source=right_p)
MPI.Request.Wait(send_request)
MPI.Request.Wait(recv_request)
```

Python 注释 *15*：请求数组。请求数组（用于 waitall/some/any 调用）是一个普通的 Python

图 4.9 MPI_Waitany

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Waitany (					
	count	list length	int	INTEGER	IN
	array_of_requests	array of requests	MPI_Request[]	TYPE (MPI_Request) (count)	INOUT
	index	index of handle for operation that completed	int*	INTEGER	OUT
	status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

Python:

```
MPI.Request.Waitany( requests,status=None )
class method, returns index
```

列表：## irecvloop.pyrequests = []

```
sendbuffer = np.empty( nprocs, dtype=int )
recvbuffer = np.empty( nprocs, dtype=int )
for p in range(nprocs):
left_p = (p-1) % nprocs
right_p = (p+1) % nprocs
requests.append( comm.Isend\
( sendbuffer[p:p+1],dest=left_p) )
requests.append( comm.Irecv\
( sendbuffer[p:p+1],source=right_p) )
MPI.Request.Waitall(requests)
```

The MPI_Waitall method is again a class method.

4.2.2.3 等待任意请求

如果你需要所有非阻塞通信完成后才能继续执行程序的其他部分，那么 “waitall” 例程是很好的选择。然而，有时可以在每个请求满足时立即采取行动。在这种情况下，你可以使用 MPI_Waitany (图 4.9) 并编写：

```
for (p=0; p<nrequests; p++) {
  MPI_Irecv(buffer+index, /* ... */, requests+index);
}
for (p=0; p<nrequests; p++) {
  MPI_Waitany(nrequests,request_array,&index,&status);
  // operate on buffer[index]
}
```

注意该例程接受一个单一的 status 参数，通过引用传递，而不是一个 status 数组！

Fortran note 7: Indexof requests. index 参数是请求数组中的索引，该数组是一个 Fortran 数组，因此它使用基于 *1* 的索引。

```
!! irecvsource.F90
  if (mytid==ntids-1) then
     do p=1,ntids-1
        print *,"post"
        call MPI_Irecv(recv_buffer(p),1,MPI_INTEGER,p-1,0,comm,&
             requests(p),err)
     end do
     do p=1,ntids-1
        call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE,err)
        write(*,'("Message from",i3,":",i5)') index,recv_buffer(index)
     end do
```

```
!! waitnull.F90
  Type(MPI_Request),dimension(:),allocatable :: requests
  allocate(requests(ntids-1))
        call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
        if ( .not. requests(index)==MPI_REQUEST_NULL) then
             print *,"This request should be null:",index
```

```
!! waitnull.F90
  Type(MPI_Request),dimension(:),allocatable :: requests
  allocate(requests(ntids-1))
        call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
        if ( .not. requests(index)==MPI_REQUEST_NULL) then
             print *,"This request should be null:",index
```

MPL note 31: Request pools. 代替请求数组，使用一个 irequest_pool 对象，它表现得像一个请求向量，这意味着你可以 *push* 到它上面。

```
// irecvsource.cxx
mpl::irequest_pool recv_requests;
for (int p=0; p<nprocs-1; p++) {
  recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
}
```

你不能声明一个固定大小的池并分配元素。（为什么不行？你能找到解决方法吗？）

MPL 注释 *32*：等待任意。 irequest_pool 类具有方法 waitany, waitall, testany, testall, waitsome,testsome。

"any" 方法返回一个 *std::pair<mpl::test_result,size_t>*，其中 *test_result* 是一个 enum class，其值为：

- *completed*（用于任意 / 部分 / 全部完成），
- *no_completed*（用于无），
- *no_active_requests*（如果没有更多请求处于活动状态）。

```
  auto [success,index] = recv_requests.waitany();
  if ( success==mpl::test_result::completed ) {
        auto recv_status = recv_requests.get_status(index);
```

MPL 注释 *33*：请求处理。

```
auto [success,index] = recv_requests.waitany();
if ( success==mpl::test_result::completed ) {
      auto recv_status = recv_requests.get_status(index);
```

4.2.2.4 使用 *MPIWait any* 轮询

该 `MPI_Waitany` 例程可用于实现轮询：在进行其他工作时偶尔检查是否有传入消息。**代码：输出：**

```
// irecvsource.c
if (procno==nprocs-1) {
  int *recv_buffer;
  MPI_Request *request; MPI_Status status;
  recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));
  request = (MPI_Request*) malloc
    ((nprocs-1)*sizeof(MPI_Request));

  for (int p=0; p<nprocs-1; p++) {
    ierr = MPI_Irecv(recv_buffer+p,1,MPI_INT, p,0,comm,
                     request+p); CHK(ierr);
  }
  for (int p=0; p<nprocs-1; p++) {
    int index,sender;
    MPI_Waitany(nprocs-1,request,&index,&status);
    if (index!=status.MPI_SOURCE)
      printf("Mismatch index %d vs source %d\n",
             index,status.MPI_SOURCE);
    printf("Message from %d: %d\n",
           index,recv_buffer[index]);
  }
} else {
  ierr = MPI_Send(&procno,1,MPI_INT, nprocs-1,0,comm);
}
```

```
make[3]: `irecvsource' is up
    ↪to date.
process 1 waits 6s before
    ↪sending
process 2 waits 3s before
    ↪sending
process 0 waits 13s before
    ↪sending
process 3 waits 8s before
    ↪sending
process 5 waits 1s before
    ↪sending
process 6 waits 14s before
    ↪sending
process 4 waits 12s before
    ↪sending
Message from 5: 5
Message from 2: 2
Message from 1: 1
Message from 3: 3
Message from 4: 4
Message from 0: 0
Message from 6: 6
```

```
## irecvsource.py
if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1,dtype=int)
    requests = [ None ] * (nprocs-1)
    for sender in range(nprocs-1):
        requests[sender] = comm.Irecv(receive_buffer[sender:sender+1],source=sender)
    # alternatively: requests = [ comm.Irecv(s) for s in .... ]
    status = MPI.Status()
    for sender in range(nprocs-1):
        ind = MPI.Request.Waitany(requests,status=status)
        if ind!=status.Get_source():
            print("sender mismatch: %d vs %d" % (ind,status.Get_source()))
        print("received from",ind)
else:
    mywait = random.randint(1,2*nprocs)
    print("[%d] wait for %d seconds" % (procid,mywait))
```

```
time.sleep(mywait)
mydata = np.empty(1,dtype=int)
mydata[0] = procid
comm.Send([mydata,MPI.INT],dest=nprocs-1)
```

除了根进程外，每个进程都执行阻塞发送；根进程发布 MPI_Irecv 来自所有其他处理器的请求，然后循环执行 MPI_Waitany 直到所有请求都到达。使用 MPI_SOURCE 来测试 wait 调用的索引参数。

注意 MPI_STATUS_IGNORE 参数：我们对传入消息了解全部信息，因此不需要查询状态对象。与第 4.3.1 节中的示例形成对比。

4.2.2.5 等待某些请求

最后，MPI_Waitsome 非常类似于 MPI_Waitany，不同之处在于如果多个请求被满足，它会返回多个数字。现在 status 参数是一个 MPI_Status 对象数组。

4.2.2.6 Receive status of the wait calls

The MPI_Wait... 例程的输出是 MPI_Status 对象。如果您不关心状态信息，可以使用 MPI_STATUS_IGNORE 的值 MPI_Wait 和 MPI_Waitany，或者 MPI_STATUSES_IGNORE 的值 MPI_Waitall， MPI_Waitsome， MPI_Testall， MPI_Testsome。

备注 13 能够返回多个状态的例程，可以返回错误条件 *MPI_ERR_IN_STATUS*，表示其中一个状态出错。参见第 *4.3.3* 节。

练习 4.13.(Ther

这是名为 isendirecv 的本练习骨架代码。）现在使用非阻塞发送 / 接收例程来实现三点平均操作

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \ldots, N-1$$

在分布式数组上。对于第一个和最后一个进程，有两种方法：
1. 你可以使用 MPI_PROC_NULL 来处理 “ 缺失 ” 的通信；2. 你也可以完全跳过这些通信，但现在必须仔细计算请求。

4.2.2.7 Latency hiding / overlapping communication and computation

对于 Isend/Irecv 调用还有第二个动机：如果你的硬件支持，它可以在程序继续执行有用工作的同时进行通信：

```
// start nonblocking communication
MPI_Isend( ... ); MPI_Irecv( ... );
// do work that does not depend on incoming data....
// wait for the Isend/Irecv calls to finish
```

```
MPI_Wait( ... );
// now do the work that absolutely needs the incoming data....
```

这被称为 计算与通信重叠，或 延迟隐藏。另见 异步进展；章节 15.4。

不幸的是，许多通信涉及用户空间的活动，因此解决方案是让它由单独的线程处理。直到最近，处理器在执行这种多线程方面效率不高，因此真正的重叠仍是未来的承诺。一些网卡支持这种重叠，但这需要硬件、固件和 MPI 实现的复杂组合。

练习 4.14.(Ther

> 这是该练习的骨架，名为 `isendirecvarray`。) 取你在练习 4.13 中的代码并修改以使用延迟隐藏。可以在不需要邻居数据的情况下执行的操作，应在 `MPI_Isend` / `MPI_Irecv` 调用和相应的 `MPI_Wait` 调用之间执行。

Remark14 你现在已经见过各种发送类型：阻塞、非阻塞、同步。接收方能看到发送了哪种类型的消息吗？是否需要不同的接收例程？答案是，在接收端，没有什么能区分非阻塞或同步消息。`MPI_Recv` 调用可以匹配你迄今为止见过的任何发送例程，反之，用 `MPI_Send` 发送的消息也可以被 `MPI_Irecv` 接收。

4.2.2.8 *Buffer issues in nonblocking communication*

虽然使用非阻塞例程可以防止死锁，但它也引入了自身的问题。

- With a blocking send call, you could repeatedly fill the send buffer and send it off.

```
double *buffer;
for ( ... p ... ) {
   buffer = // fill in the data
   MPI_Send( buffer, ... /* to: */ p );
```

- 另一方面，当非阻塞发送调用返回时，实际的发送可能尚未执行，因此发送缓冲区可能不安全以供覆盖。类似地，当 recv 调用返回时，你不能确定预期的数据是否已经在其中。只有在相应的 wait 调用之后，你才能确定缓冲区已经被发送，或者已经接收到其内容。
- 因此，要使用非阻塞调用发送多个消息，您必须分配多个缓冲区。

```
double **buffers;
for ( ... p ... ) {
   buffers[p] = // fill in the data
   MPI_Send( buffers[p], ... /* to: */ p );
}
MPI_Wait( /* the requests */ );
```

```
// irecvloop.c
MPI_Request requests =
```

```
    (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {int
left_p  = (p-1+nprocs) % nprocs,right_p = (p+1) % nprocs;
send_buffer[p] = nprocs-p;
MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);}
/* your useful code here */
MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);
```

4.2.3 Waitand test 调用

The `MPI_Wait...` 例程是阻塞的。因此，如果接收进程在数据（或至少一些数据）实际接收之前无法执行任何操作，它们是一个很好的解决方案。 `MPI_Test...` 调用本身是非阻塞的：它们测试一个或多个请求是否已完成，否则立即返回。这也是一个本地操作：它不会强制进展。

备注 15 这些 *MPI_Test...* 例程类似于，但不同于 *MPI_Probe*，后者是阻塞的并强制进展；参见章节 *4.4.1*。

该 `MPI_Test` 调用可用于 管理者 - 工作者 模型：管理者进程创建任务，并将它们发送给任何已完成工作的工作者进程。（这使用了来自 `MPI_ANY_SOURCE` 的接收，以及随后对接收状态的 `MPI_SOURCE` 字段的测试。）在等待工作者时，管理者也可以做有用的工作，这需要定期检查传入消息。

伪代码：

```
while ( not done ) {
// create new inputs for a while....
// see if anyone has finished
MPI_Test( .... &index, &flag );
if ( flag ) {
// receive processed data and send new}
```

如果测试为真，请求将被释放并设置为 `MPI_REQUEST_NULL`，或者，在活动持久请求的情况下（章节 5.1），设置为非活动状态。

类似于 `MPI_Wait`，`MPI_Waitany`，`MPI_Waitall`，`MPI_Waitsome`，存在 `MPI_Test`（图 4.10),`MPI_Testany`, `MPI_Testall`, `MPI_Testsome`。

练习 4.15. 阅读 HPC 书籍第 7.5 节，并给出使用上述 `MPI_Test...` 调用习语的分布式稀疏矩阵 - 向量乘法的伪代码。讨论这种方法的优缺点。答案不会是非黑即白的：讨论你预期在何种情况下哪种方法更可取。

图 4.10 MPI_Test

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Test (				
request	communication request	MPI_Request*	TYPE (MPI_Request)	INOUT
flag	true if operation completed	int*	LOGICAL	OUT
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

Python:

```
reqest.Test()
```

图 4.11 MPI_Request_free

名称 参数名	说明	C 类型	F 类型	输入输 出
MPI_Request_free (				
request	communication request	MPI_Request*	TYPE (MPI_Request)	INOUT
)				

4.2.4 关于请求的更多内容

E非常非阻塞调用分配一个 MPI_Request 对象。与 MPI_Status 不同， MPI_Request 变量不是a实际上不是一个对象，而是一个（不透明的）指针。这意味着当你调用，例如，MPI_Irecv，MPI 将分配一个实际的请求对象，并返回其地址到 MPI_Request 变量。

相应地，调用 MPI_Wait 或 MPI_Test 释放该对象，并将句柄设置为 MPI_REQUEST_NULL。（对于持久通信有一个例外，请求仅被设置为 “ 非活动 ” ；参见 5.1 节。）因此，即使你知道操作已经成功，发出等待调用也是明智的。例如，如果所有接收调用都已完成，你知道对应的发送调用也已结束，严格来说不需要等待它们的请求。然而，省略等待调用会导致 内存泄漏。

另一种方法是调用 MPI_Request_free (图 4.11)，它将请求变量设置为 MPI_REQUEST_NULL，并在操作完成后标记该对象以便释放。理论上，可以发出一个非阻塞调用，然后立即调用 MPI_Request_free，省去任何等待调用。然而，这使得难以知道操作何时完成以及缓冲区何时可以安全重用 [26]。

您可以在不释放请求对象的情况下检查请求的状态，使用 MPI_Request_get_status(图 4.12)。对于多个状态，使用 MPI_Request_get_status_all，MPI_Request_get_status_some,MPI_Request_get_status_any，在 MPI-4.1 中。

图 4.12 MPI_Request_get_status

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Request_get_status (				
request	request	MPI_Request	TYPE (MPI_Request)	IN
flag	boolean flag, same as from MPI_TEST	int*	LOGICAL	OUT
status	status object if flag is true	MPI_Status*	TYPE (MPI_Status)	OUT
)				

4.3 Status 对象和通配符

在第 4.1.1 节中你看到 MPI_Recv 有一个类型为 MPI_Status 的 ‘status’ 参数，MPI_Send 该参数缺失。（各种 MPI_Wait... 例程也有一个 status 参数；参见第 4.2.1 节。）通常你会为这个参数指定 MPI_STATUS_IGNORE：通常你知道数据是什么以及它来自哪里。

然而，在某些情况下，接收方在调用接收时可能不知道消息的所有细节，因此 MPI 提供了一种查询消息 *status* 的方法：

- 如果你期望多个传入消息，按它们到达的顺序处理可能是最高效的。因此，你不会等待特定消息，而是在接收消息的描述中指定 MPI_ANY_SOURCE 或 MPI_ANY_TAG。现在你必须能够询问 ‘这条消息来自谁，里面有什么’。
- 也许你知道消息的发送者，但数据量未知。在这种情况下，你可以为接收缓冲区分配过多空间，消息接收后再查询其大小，或者你可以 ‘探测’ 传入消息，在得知发送了多少数据后再分配足够的空间。

T为此，receive 调用有一个 MPI_Status 参数。该 MPI_Status 对象是一个结构体（在 C 中是 struct，in F90 中是数组，F2008 中是派生类型），其成员可自由访问：

- MPI_SOURCE 给出消息的来源；参见第 4.3.1 节。
- MPI_TAG 给出接收消息时的标签；参见第 4.3.2 节。
- MPI_ERROR 给出 receive 调用的错误状态；参见第 4.3.3 节。
- 消息中的项数可以从 status 对象中推断，不是作为结构体成员，而是通过调用函数 MPI_Get_count；参见第 4.3.4 节。

Fortran 注释 8：*f08 中的 Status 对象。* 该 mpi_f08 模块将许多句柄（例如通信器）从 Fortran Integer 转换为 Type。通常通过 %val 成员来获取类型中的整数，但对于 status 对象来说，这更为困难。例程 MPI_Status_f2f08 和 MPI_Status_f082f 在它们之间进行转换。（值得注意的是，这些例程甚至在 C 语言中可用，它们操作 MPI_Fint，MPI_F08_status 参数。）

以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

既可获取也可设置，使用诸如 MPI_Status_get_source, MPI_Status_set_source 等例程。

End of MPI-4 material

Pythonnote 16: Status object. status 对象在传递给接收例程之前显式创建。它具有用于消息计数的常用查询方法：

```
## pingpongbig.py
status = MPI.Status()
comm.Recv( rdata,source=0,status=status)
count = status.Get_count(MPI.DOUBLE)
```

(count 函数无参数时返回以字节为单位的结果。)

然而，与 C/F 中 status 对象的字段可以直接访问不同，Python 也为这些字段提供了查询方法：

```
status.Get_source()
status.Get_tag()
status.Get_elements()
status.Get_error()
status.Is_cancelled()
```

如果您需要，它们甚至有 `Set` 变体。 https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Status.html

MPL 注释 *34*：状态对象。该 `mpl::status_t` 对象由接收（或等待）调用创建：

```
mpl::contiguous_layout<double> target_layout(count);
mpl::status_t recv_status =
  comm_world.recv(target.data(),target_layout, the_other);
recv_count = recv_status.get_count<double>();
```

4.3.1 源

在某些应用中，消息可能来自多个进程中的一个，这样是有意义的。在这种情况下，可以指定 `MPI_ANY_SOURCE` 作为源。要找出消息实际来自的 源 ， 您可以使用 `MPI_SOURCE` 状态对象中由 `MPI_Recv` 或 `MPI_Wait...` 调用后传递的 `MPI_Irecv` 字段。

```
MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
         &status);
sender = status.MPI_SOURCE;
```

在多种场景下，从 “ 任意源 ” 接收是有意义的。其中一种是 *manager-worker* 模型。manager 任务首先向 worker 任务发送数据，然后对第一个完成的进程的数据发出阻塞等待。

在 Fortran2008 风格中，source 是 `Status` type 的一个成员。

```
!! anysource.F90
  Type(MPI_Status)  :: status
        allocate(recv_buffer(ntids-1))
```

```
        do p=0,ntids-2
           call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
                MPI_ANY_SOURCE,0,comm,status)
           sender = status%MPI_SOURCE
```

在 Fortran90 风格中，源是 `Status` 数组中的一个索引。

```
!! anysource.F90
  integer :: status(MPI_STATUS_SIZE)
        allocate(recv_buffer(ntids-1))
        do p=0,ntids-2
           call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
                MPI_ANY_SOURCE,0,comm,status,err)
           sender = status(MPI_SOURCE)
```

MPL 注释 *35*：状态查询。 该 `status` 对象可以被查询：

```
int source = recv_status.source();
```

Likewise the source:

```
mpl::tag_t t = recv_status.tag();
```

4.3.2 Tag

在某些情况下，接收操作中可以使用标签通配符：`MPI_ANY_TAG`。消息的实际 *tag* 可以作为状态结构中的 `MPI_TAG` 成员被检索到。

这种情况并不多见。

- 来自单一源的消息，即使是非阻塞的，也是非超车的。这意味着消息可以通过它们的顺序来区分。
- 来自多个源的消息可以通过源字段来区分。
- 如果信息被编码在标签中，可能需要检索消息的 tag。
- 非超车论点不适用于混合计算的情况：两个线程可能发送没有 MPI 强制顺序的消息。参见第 45.1 节的示例。

MPL 注释 *36*：消息标签。MPL 在标签的处理上不同于其他 API：标签不是直接的整数，而是类 `tag_t` 的对象。

```
// sendrecv.cxx
mpl::tag_t t0(0);
comm_world.sendrecv
( mydata,sendto,t0,leftdata,
recvfrom,t0 );
```

类有几个方法，比如 `mpl::tag_t::any()`（用于接收调用中的 `MPI_ANY_TAG` 通配符）和 `mpl::tag_t::up()`（最大标签，从 `MPI_TAG_UB` 属性中找到）。

MPL 注释 *37*：标签类型。标签是 `int` 或 `enum` 类型：

```
    template<typename T >
  tag_t (T t);
  tag_t (int t);
```

Example:

```
// inttag.cxx
enum class pingpongtag : int { ping=1, pong=2 };
int pinger = 0, ponger = world.size()-1;
if (world.rank()==pinger) {
  world.send(x, ponger, pingpongtag::ping);
  world.recv(x, ponger, pingpongtag::pong);
} else if (world.rank()==ponger) {
  world.recv(x, pinger, pingpongtag::ping);
  world.send(x, pinger, pingpongtag::pong);
}
```

4.3.3 错误

对于返回单个状态的函数，任何错误都作为函数结果返回。对于返回多个状态的函数，例如 `MPI_Waitall`，如果其中一个接收操作出现错误，则结果为 `MPI_ERR_IN_STATUS`。在接收操作期间发生的任何错误都可以在状态结构的 `MPI_ERROR` 成员中找到。

4.3.4 Count

如果接收的数据量事先未知，接收元素的 *count* 可以通过 `MPI_Get_count` (图 4.13) 找到:

```
// count.c
if (procid==0) {
  int sendcount = (rand()>.5) ? N : N-1;
  MPI_Send( buffer,sendcount,MPI_FLOAT,target,0,comm );
} else if (procid==target) {
  MPI_Status status;
  int recvcount;
  MPI_Recv( buffer,N,MPI_FLOAT,0,0, comm, &status );
  MPI_Get_count(&status,MPI_FLOAT,&recvcount);
  printf("Received %d elements\n",recvcount);
}
```

图 4.13 MPI_Get_count

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Get_count (MPI_Get_count_c (				
status	return status of receive operation	const MPI_Status*	TYPE (MPI_Status)	IN
datatype	datatype of each receive buffer entry	MPI_Datatype	TYPE (MPI_Datatype)	IN
count	number of received entries	int* MPI_Count*	INTEGER	OUT
)				

MPL:

```
template<typename T>
int mpl::status::get_count () const

template<typename T>
int mpl::status::get_count (const layout<T> &l) const
```

Python:

```
status.Get_count( Datatype datatype=BYTE )
```

Code:

```
!! count.F90
  if (procid==0) then
     sendcount = N
     call random_number(fraction)
     if (fraction>.5) then
        print *,"One less" ; sendcount = N-1
     end if
     call MPI_Send(
    ↪buffer,sendcount,MPI_REAL,target,0,comm )
  else if (procid==target) then
    call MPI_Recv( buffer,N,MPI_REAL,0,0, comm, status )
    call MPI_Get_count(status,MPI_FLOAT,recvcount)
    print *,"Received",recvcount,"elements"
 end if
```

Output:

```
make[3]: `count' is up to
    ↪date.
TACC:  Starting up job
    ↪4051425
TACC:  Setting up parallel
    ↪environment for
    ↪MVAPICH2+mpispawn.
TACC:  Starting parallel
    ↪tasks...
 One less
 Received           9
    ↪elements
TACC:  Shutdown complete.
    ↪Exiting.
```

这可能是必要的，因为 count 参数传递给 MPI_Recv 的是缓冲区大小，而不是实际接收到的数据项数量的指示。

备注。

- 与 source 和 tag 不同，message count 并不是 status 结构的直接成员。
- 返回的 ‘count’ 是指定数据类型的元素数量。如果这是一个派生类型（章节 6.3），这与预定义数据类型元素的数量不同。对此，请使用 MPI_Get_elements（图 4.14）或 MPI_Get_elements_x，它返回基本元素的数量。

MPL 注释 *38*：接收计数。函数是 status 对象的一个方法。参数类型 get_count 。

图 4.14 MPI_Get_elements

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Get_elements (				
MPI_Get_elements_c (				
status	return status of receive operation	const MPI_Status*	TYPE (MPI_Status)	IN
datatype	datatype used by receive operation	MPI_Datatype	TYPE (MPI_Datatype)	IN
count	number of received basic elements	int* MPI_Count*	INTEGER	OUT
)				

通过模板处理：

```
// recvstatus.cxx
double pi=0;
auto s = comm_world.recv(pi, 0);  // receive from rank 0
int c = s.get_count<double>();
std::cout << "got : " << c << " scalar(s): " << pi << '\n';
```

4.3.5 示例：从任意源接收

考虑一个示例，其中最后一个进程接收来自所有其他进程的数据。我们可以将其实现为一个循环

```
for (int p=0; p<nprocs-1; p++)
  MPI_Recv( /* from source= */ p );
```

但如果消息乱序到达，这可能会导致空闲时间。

相反，我们使用 MPI_ANY_SOURCE 说明符为接收调用赋予通配符行为：将此值用作 “source” 值意味着我们接受来自通信器内任何源的消息，且消息仅通过标签值匹配。（注意，接收缓冲区的大小和类型不用于消息匹配！）

然后我们通过 MPI_Status 对象的 MPI_SOURCE 字段检索实际的源。

```
// anysource.cif (procno==nprocs-1) {/*
* The last process receives from every other process*/
int *recv_buffer;
recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));/*
* Messages can come in in any order, so use MPI_ANY_SOURCE*/
MPI_Status status;
```

```
  for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
                   &status); CHK(err);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=%d: %d\n",
             sender,recv_buffer[p]);
  }
  free(recv_buffer);
} else {
  /*
   * Each rank waits an unpredictable amount of time,
   * then sends to the last process in line.
   */
  float randomfraction = (rand() / (double)RAND_MAX);
  int randomwait = (int) ( nprocs * randomfraction );
  printf("process %d waits for %e/%d=%d\n",
           procno,randomfraction,nprocs,randomwait);
  sleep(randomwait);
  err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
}
```

```
## anysource.py
rstatus = MPI.Status()
comm.Recv(rbuf,source=MPI.ANY_SOURCE,status=rstatus)
print("Message came from %d" % rstatus.Get_source())
```

The *manager-worker* 模型是一种设计模式，提供了检查 `MPI_SOURCE` 的机会 field of the `MPI_Status` 对象描述接收到的数据。所有工作进程都模拟它们的 work 通过等待随机时间量，管理进程接受来自任何源的消息。

```
// anysource.c
if (procno==nprocs-1) {
  /*
   * The last process receives from every other process
   */
  int *recv_buffer;
  recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

  /*
   * Messages can come in in any order, so use MPI_ANY_SOURCE
   */
  MPI_Status status;
  for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
                   &status); CHK(err);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=%d: %d\n",
             sender,recv_buffer[p]);
  }
  free(recv_buffer);
} else {
  /*
   * Each rank waits an unpredictable amount of time,
```

图 4.15 MPI_Probe

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Probe (					
	source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
	tag	message tag or MPI_ANY_TAG	int	INTEGER	IN
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
	)				

```
    * then sends to the last process in line.*/
float randomfraction = (rand() / (double)RAND_MAX);
int randomwait = (int) ( nprocs * randomfraction );
printf("process %d waits for %e/%d=%d\n",procno,randomfraction,
nprocs,randomwait);sleep(randomwait);
err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);}
```

在第 49 章你可以用这个模型做编程项目。

4.4 关于点对点通信的更多内容

4.4.1 消息探测

MPI receive 调用指定一个接收缓冲区，其大小必须足够容纳任何发送的数据。如果你真的不知道发送了多少数据，并且不想过度分配接收缓冲区，可以使用 “probe” 调用。

例程 MPI_Probe (图 4.15) 和 MPI_Iprobe (图 4.16) （参见第 15.4 节）接受消息但不复制数据。相反，当探测告诉你有消息时，你可以使用 MPI_Get_count (第 4.3.4 节) 来确定其大小，分配足够大的接收缓冲区，并执行常规接收以复制数据。

```
// probe.c
if (procno==receiver) {
  MPI_Status status;
  MPI_Probe(sender,0,comm,&status);
  int count;
  MPI_Get_count(&status,MPI_FLOAT,&count);
  float recv_buffer[count];
  MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
} else if (procno==sender) {
  float buffer[buffer_size];
  ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(ierr);
}
```

Figure 4.16 MPI_Iprobe

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Iprobe (					
	source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
	tag	message tag or MPI_ANY_TAG	int	INTEGER	IN
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	flag	true if there is a matching message that can be received	int*	LOGICAL	OUT
	status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

图 4.17 MPI_Mprobe

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Mprobe (					
	source	rank of source or MPI_ANY_SOURCE	int	INTEGER	IN
	tag	message tag or MPI_ANY_TAG	int	INTEGER	IN
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	message	returned message	MPI_Message*	TYPE (MPI_Message)	OUT
	status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)					

在多线程环境中， MPI_Probe 调用存在一个问题：可能会发生以下情况。

> 1. 一个线程通过探测确定某条消息已经到达。 2. 它对该消息发出一个阻塞接收调用 ... 3. 但在探测和接收调用之间，另一个线程已经接收了该消息。 4. ... 导致第一个线程处于阻塞状态且没有消息可接收。

这是通过 MPI_Mprobe (图 4.17) 解决的，该图在成功探测后会将消息从匹配队列中移除：即可以被接收调用匹配的消息列表。匹配探测的线程现在通过一个类型为 MPI_Message 的对象对该消息发出 MPI_Mrecv (图 4.18) 调用。

4.4.2 错误

MPI 例程成功完成时返回 MPI_SUCCESS 。以下错误代码可能被返回 (详见第 15.2.1 节，发送和接收操作完成时的错误情况： MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_TAG, MPI_ERR_RANK.

图 4.18 MPI_Mrecv

Name Param name	Explanation	C type	F type	i nout
MPI_Mrecv (				
MPI_Mrecv_c (				
buf	initial address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in receive buffer	int MPI_Count	INTEGER	IN
datatype	datatype of each receive buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
message	message	MPI_Message*	TYPE (MPI_Message)	INOUT
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

4.4.3 Message envelope

除了其裸数据外，每条消息都有一个 *message envelope*。它包含足够的信息来区分不同的消息：源、目的地、标签、通信器。

4.5 复习问题

对于所有判断题，如果你认为某个陈述是错误的，请给出一句话的解释。

Review4.16. 描述一个涉及三个处理器的死锁场景。

Review4.17. 判断正误：使用 `MPI_Isend` 发送的消息可以通过另一个处理器上的 `MPI_Recv` 调用接收。

Review 4.18. 判断正误：使用 `MPI_Send` 发送的消息可以通过另一个处理器上的 `MPI_Irecv` 接收。

Review 4.19. 为什么 `MPI_Irecv` 调用没有 `MPI_Status` 参数?

Review4.20. 假设你正在测试乒乓时延。为什么通常不建议使用进程 0 和 1 作为源处理器和目标处理器？你能提出一个更好的猜测吗?

p

你能提出一个更好的猜测吗?

Review 4.21. ‘origin’ 、 ‘target’ 、 ‘fence’ 和 ‘window’ 这几个概念在单边通信中有什么关系？**Review 4.22.** 单边数据传输的三个例程是什么?

Review4.23. 在以下代码片段中，假设所有缓冲区都已分配了足够的大小。请对每个片段说明是否会死锁，并讨论性能问题。`for (int p=0; p<nprocs; p++)if (p!=procid)`

```
MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
```

```
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
```

~

```
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
```

```
int ireq = 0;
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Isend(sbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
```

```
int ireq = 0;
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
```

```
int ireq = 0;
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
```

Fortran 代码:

```
do p=0,nprocs-1if (p/=procid) then
call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)end ifend dodo p=0,
nprocs-1if (p/=procid) then
call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)end if
end do
```

```
do p=0,nprocs-1if (p/=procid) then
call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)end if
end dodo p=0,nprocs-1if (p/=procid) then
call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)end ifend do
```

```
ireq = 0
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Isend(sbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
           requests(ireq+1),ierr)
      ireq = ireq+1
   end if
end do
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
   end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
```

```
ireq = 0
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Irecv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
           requests(ireq+1),ierr)
      ireq = ireq+1
   end if
end do
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
   end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
```

```
// block5.F90
ireq = 0
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Irecv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
           requests(ireq+1),ierr)
      ireq = ireq+1
   end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
   end if
end do
```

复习 4.24. 考虑一个环形通信，其中

```
intnext = (mytid+1) % ntids,
prev = (mytid+ntids-1) % ntids;
```

并且每个进程发送到 *next*，并从 *prev* 接收。防止死锁的常规解决方案是同时使用 `MPI_Isend` 和 `MPI_Irecv`。发送和接收在 wait 调用时完成。但调用 wait 的顺序重要吗?

```
// ring3.cMPI_Request req1,req2;
MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,comm,&req1);
MPI_Isend(&x,1,MPI_DOUBLE,next,0,comm,&req2);
MPI_Wait(&req1,MPI_STATUS_IGNORE);
MPI_Wait(&req2,MPI_STATUS_IGNORE);
```

```
// ring4.cMPI_Request req1,req2;
MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,comm,&req1);
MPI_Isend(&x,1,MPI_DOUBLE,next,0,comm,&req2);
MPI_Wait(&req2,MPI_STATUS_IGNORE);
MPI_Wait(&req1,MPI_STATUS_IGNORE);
```

我们可以同时有一个非阻塞调用和一个阻塞调用吗? 这些场景会阻塞吗?

```
// ring1.c
MPI_Request req;
MPI_Issend(&x,1,MPI_DOUBLE,next,0,comm,&req);
MPI_Recv(&y,1,MPI_DOUBLE,prev,0,comm,
         MPI_STATUS_IGNORE);
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

```
// ring2.c
MPI_Request req;
MPI_Irecv(&y,1,MP
I_DOUBLE,prev,0,comm,&req);
MPI_Ssend(&x,1,MPI_DOUBLE,next,0,comm);
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

第 5 章

MPI 主题：通信模式

5.1 持久通信

你可以想象，建立一次通信会带来一些开销，如果相同的通信结构重复多次，这些开销可能被避免。

持久通信是一种处理重复通信事务的机制，其中事务的参数，如发送者、接收者、标签、根、缓冲区地址 / 类型 / 大小保持不变。只有缓冲区的内容可能在事务之间发生变化。

1. 对于非阻塞通信 `MPI_Ixxx`（包括点对点和集合通信）有一个持久变体 `MPI_Xxx_init`，调用序列相同。‘init’ 调用产生一个 `MPI_Request` 输出参数，可用于测试通信是否完成。2. ‘init’ 例程不启动实际通信：实际通信在 `MPI_Start`，或 `MPI_Startall` 用于多个请求时启动。3. 任何 MPI 的 ‘wait’ 调用都可以用来结束通信。4. 通信随后可以通过另一个 ‘start’ 调用重新启动。5. wait 调用不会释放请求对象，因为它可用于该事务的重复发生。请求对象仅在 `MPI_Request_free` 时释放。

```
MPI_Send_init( /* ... */ &request);
while ( /* ... */ ) {
  MPI_Start( request );
  MPI_Wait( request, &status );
}
MPI_Request_free( & request );
```

MPL 注释 *39*：持久请求。MPL 从持久 ‘init’ 例程返回一个 `prequest`，而不是一个 `irequest`(MPL 注释 30)：

```
template<typename T >
prequest send_init (const T &data, int dest, tag t=tag(0)) const;
```

同样，有一个 `prequest_pool` 而不是一个 `irequest_pool`（注释 31）。

图 5.1 MPI_Send_init

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Send_init (				
MPI_Send_init_c (				
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements sent	[int MPI_Count	INTEGER	IN
datatype	type of each element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
tag	message tag	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

Python:

```
MPI.Comm.Send_init(self, buf, int dest, int tag=0)
```

图 5.2 MPI_Startall

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Startall (				
count	list length	int	INTEGER	IN
array_of_requests	array of requests	MPI_Request[]	TYPE (MPI_Request) (count)	INOUT
)				

Python:

```
MPI.Prequest.Startall(type cls, requests)
```

5.1.1 持久点对点通信

主要的持久点对点例程是 MPI_Send_init（图 5.1），其调用序列与 MPI_Isend 相同，和 MPI_Recv_init，其调用序列与 MPI_Irecv 相同。

在以下示例中，使用持久通信实现了乒乓操作。由于我们在 “ping” 进程上对发送和接收都使用持久操作，因此我们使用 MPI_Startall（图 5.2）同时启动两者，并使用 MPI_Waitall 测试它们的完成情况。（对于启动单个持久传输，有 MPI_Start 。）

Code:

```
// persist.c
if (procno==src) {
  /*
   * Send ping, receive pong
   */
  MPI_Send_init
    (send,s,MPI_DOUBLE,tgt,0,comm,
     requests+0);
  MPI_Recv_init
    (recv,s,MPI_DOUBLE,tgt,0,comm,
     requests+1);
  for (int n=0; n<NEXPERIMENTS; n++) {
    fill_buffer(send,s,n);
    MPI_Startall(2,requests);
    MPI_Waitall(2,requests,
                MPI_STATUSES_IGNORE);
    int r = chck_buffer(send,s,n);
    if (!r) printf("buffer problem %d\n",s);
  }
} else if (procno==tgt) {
  /*
   * Receive ping, send pong
   */
  MPI_Send_init
    (recv,s,MPI_DOUBLE,src,0,comm,
     requests+0);
  MPI_Recv_init
    (recv,s,MPI_DOUBLE,src,0,comm,
     requests+1);
  for (int n=0; n<NEXPERIMENTS; n++) {
        // receive
    MPI_Start(requests+1);

    ↪MPI_Wait(requests+1,MPI_STATUS_IGNORE);
        // send
    MPI_Start(requests+0);

    ↪MPI_Wait(requests+0,MPI_STATUS_IGNORE);
  }
}
MPI_Request_free(requests+0);
MPI_Request_free(requests+1);
```

输出：

```
make[3]: `persist' is up to date.
TACC:  Starting up job 4328411
TACC:  Starting parallel tasks...
Pingpong size=1: t=1.2123e-04
Pingpong size=10: t=4.2826e-06
Pingpong size=100: t=7.1507e-06
Pingpong size=1000: t=1.2084e-05
Pingpong size=10000: t=3.7668e-05
Pingpong size=100000: t=3.4415e-04
Persistent size=1: t=3.8177e-06
Persistent size=10: t=3.2410e-06
Persistent size=100: t=4.0468e-06
Persistent size=1000: t=1.1525e-05
Persistent size=10000: t=4.1672e-05
Persistent size=100000: t=2.8648e-04
TACC:  Shutdown complete. Exiting.
```

(问问自己：为什么发送方使用 `MPI_Startall` 和 `MPI_Waitall`，但接收方使用 `MPI_Start` 和 `MPI_Wait` 两次？)

```
## persist.py
requests = [ None ] * 2
sendbuf = np.ones(size,dtype=int)
recvbuf = np.ones(size,dtype=int)
if procid==src:
    print("Size:",size)
```

图 5.3 MPI_Allreduce_init

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Allreduce_init (				
MPI_Allreduce_init_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in send buffer	int MPI_Count	INTEGER	IN
datatype	datatype of elements of send buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	operation	MPI_Op	TYPE(MPI_Op)	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
info	info argument	MPI_Info	TYPE (MPI_Info)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

```
    times[isize] = MPI.Wtime()
    for n in range(nexperiments):
        requests[0] = comm.Isend(sendbuf[0:size],dest=tgt)
        requests[1] = comm.Irecv(recvbuf[0:size],source=tgt)
        MPI.Request.Waitall(requests)
        sendbuf[0] = sendbuf[0]+1
    times[isize] = MPI.Wtime()-times[isize]
elif procid==tgt:
    for n in range(nexperiments):
        comm.Recv(recvbuf[0:size],source=src)
        comm.Send(recvbuf[0:size],dest=src)
```

与普通发送命令一样，其他发送模式也有持久变体：

- MPI_Bsend_init 对于缓冲通信，参见第 5.5 节；
- MPI_Ssend_init 对于同步通信，参见第 5.3.1 节；
- MPI_Rsend_init 对于就绪发送，参见第 15.8 节。

5.1.2 持久集合操作以下内容适用于最近发布的 *MPI-4* 标准，可能尚未被支持。

对于每个集合调用，都有一个持久变体。与持久点对点调用（第 5.1.1 节）类似，这些调用的调用序列大致与非持久变体相同，区别在于：

- 一个 MPI_Info 参数，可用于传递系统相关的提示；以及
- 增加了一个最终的 MPI_Request 参数。

(例如参见 MPI_Allreduce_init (图 5.3)。) 该请求（或来自多个调用的请求数组）随后可以被 MPI_Start (或 MPI_Startall) 用于启动实际通信。

```
// powerpersist1.c
double localnorm,globalnorm=1.;
MPI_Request reduce_request;
MPI_Allreduce_init
  ( &localnorm,&globalnorm,1,MPI_DOUBLE,MPI_SUM,
    comm,MPI_INFO_NULL,&reduce_request);
for (int it=0; ; it++) {
  /*
   * Matrix vector product
   */
  matmult(indata,outdata,buffersize);

// start computing norm of output vector
  localnorm = local_l2_norm(outdata,buffersize);
  double old_globalnorm = globalnorm;
  MPI_Start( &reduce_request );

// end computing norm of output vector
  MPI_Wait( &reduce_request,MPI_STATUS_IGNORE );
  globalnorm = sqrt(globalnorm);
// now `globalnorm' is the L2 norm of `outdata'
  scale(outdata,indata,buffersize,1./globalnorm);
}
MPI_Request_free( &reduce_request );
```

一些要点。

- 元数据数组，例如计数和数据类型，直到 `MPI_Request_free` 调用之前不得更改。
- 初始化调用是非本地的（针对持久集合通信的这种特殊情况），因此它可能会阻塞，直到所有进程都执行完毕。
- 可以初始化多个持久集合通信，在这种情况下，它们满足与普通集合通信相同的限制，特别是在顺序方面。因此，以下代码是不正确的：

```
// WRONG
if (procid==0) {
  MPI_Reduce_init( /* ... */ &req1);
  MPI_Bcast_init( /* ... */ &req2);
} else {
  MPI_Bcast_init( /* ... */ &req2);
  MPI_Reduce_init( /* ... */ &req1);
}
```

初始化之后，start 调用的顺序可以是任意的，并且在不同进程之间顺序也可以不同。

可用的持久集合通信有： `MPI_Barrier_init` `MPI_Bcast_init` `MPI_Reduce_init` `MPI_Allreduce_init` `MPI_Reduce_scatter_init` `MPI_Reduce_scatter_block_init` `MPI_Gather_init` `MPI_Gatherv_init` `MPI_Allgather_init` `MPI_Allgatherv_init` `MPI_Scatter_init` `MPI_Scatterv_init` `MPI_Alltoall_init` `MPI_Alltoallv_init` `MPI_Alltoallw_init` `MPI_Scan_init` `MPI_Exscan_init`

Remark 16 持久操作可以以任何顺序启动。然而，如果所有进程以相同顺序启动持久集合操作，则可能进行系统相关的优化。这可以通过在 *MPI-4.1* 中设置来声明

图 5.4 MPI_Psend_init

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Psend_init (				
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
partitions	number of partitions	int	INTEGER	IN
count	number of elements sent per partition	MPI_Count	INTEGER (KIND=MPI_COUNT_KIND)	IN
datatype	type of each element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
tag	message tag	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
info	info argument	MPI_Info	TYPE (MPI_Info)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

信息键 *mpi_assert_strict_persistent_collective_ordering* 到 *true*。（参见章节 *15.1.1* 关于信息对象。）该值需要在通信器的所有进程上设置一致。

End of MPI-4 material

5.1.3 持久邻居通信以下材料适用于最近发布的 *MP*

I-4 标准，可能尚未被支持。

存在邻域集合通信的持久版本；章节 11.2.2。

MPI_Neighbor_allgather_init, MPI_Neighbor_allgatherv_init, MPI_Neighbor_alltoall_init, MPI_Neighbor_alltoallv_init, MPI_Neighbor_alltoallw_init,*MPI-4* 材料结束

5.2 分区通信以下材料适用于最近发布的

的 *MPI-4* 标准，可能尚未被支持。

分区通信是持久通信的一种变体，意味着我们使用 init / start / wait 序列。不同之处在于，现在消息可以逐位构建。

- 普通的 MPI_Send_init 被替换为 MPI_Psend_init（图 5.4）。注意存在一个 MPI_Info 参数，类似于持久集合通信，但不同于持久发送和接收。
- 之后， MPI_Start 实际上并不启动传输；相反：
- 消息的每个分区都通过 MPI_Pready 单独声明为准备发送。
- MPI_Wait 调用完成该操作，表示所有分区均已发送。

图 5.5 MPI_Pready

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Pready (				
partition	partition to mark ready for transfer	int	INTEGER	IN
request	partitioned communication request	MPI_Request	TYPE (MPI_Request)	INOUT
)				

在多线程环境中，一个常见的场景是每个线程可以构建消息的各自部分。拥有分区消息意味着可以发送部分构建好的消息缓冲区，而无需等待所有线程完成。

指示消息的部分已准备好发送，可以通过以下调用之一完成：

- MPI_Pready (图 5.5) 用于单个分区；
- MPI_Pready_range 用于一系列分区；以及
- MPI_Pready_list 用于显式列举的分区列表。

该 MPI_Psend_init 调用返回一个 MPI_Request 对象，可用于测试完整操作的完成情况（参见章节 4.2.2 和 4.2.3 ）。

```
MPI_Request send_request;MPI_Psend_init
(sendbuffer,nparts,SIZE,MPI_DOUBLE,tgt,0,comm,MPI_INFO_NULL,
&send_request);for (int it=0; it<ITERATIONS; it++) {
MPI_Start(&send_request);for (int ip=0; ip<nparts; ip++) {
fill_buffer(sendbuffer,partitions[ip],partitions[ip+1],ip);
MPI_Pready(ip,send_request);}
MPI_Wait(&send_request,MPI_STATUS_IGNORE);}
MPI_Request_free(&send_request);
```

接收端在很大程度上是发送端的镜像：

```
double *recvbuffer = (double*)malloc(bufsize*sizeof(double));
MPI_Request recv_request;
MPI_Precv_init
  (recvbuffer,nparts,SIZE,MPI_DOUBLE,src,0,
   comm,MPI_INFO_NULL,&recv_request);
for (int it=0; it<ITERATIONS; it++) {
  MPI_Start(&recv_request); int r=1,flag;
  for (int ip=0; ip<nparts; ip++) // cycle this many times
    for (int ap=0; ap<nparts; ap++) { // check specific part
      MPI_Parrived(recv_request,ap,&flag);
      if (flag) {
        r *= chck_buffer
          (recvbuffer,partitions[ap],partitions[ap+1],ap);
```

图 5.6 MPI_Parrived

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Parrived (				
request	partitioned communication request	MPI_Request	TYPE (MPI_Request)	IN
partition	partition to be tested	int	INTEGER	IN
flag	true if operation completed on the specified partition, false if not	int*	LOGICAL	OUT
)				

```
        break; }
    }
  MPI_Wait(&recv_request,MPI_STATUS_IGNORE);
}
MPI_Request_free(&recv_request);
```

- 分区发送只能与分区接收匹配，因此我们从一个 MPI_Precv_init 开始。

- 分区到达可以通过 MPI_Parrived (图 5.6) 进行测试。
- 调用 MPI_Wait 完成操作，表示所有分区均已到达。

再次， MPI_Request 从 receive-init 调用返回的对象可用于测试完整接收操作的完成情况。

MPI-4 材料结束

5.3 同步和异步通信

将阻塞视为与另一个进程同步的一种形式是最容易理解的，但这并不完全正确。同步本身是一个概念，我们谈论同步通信时，指的是与另一个进程之间确实存在协调，而异步通信则指不存在这种协调。阻塞仅指程序等待直到用户数据可以安全重用；在同步情况下，阻塞调用意味着数据确实已传输，在异步情况下则仅意味着数据已传输到某个系统缓冲区。四种可能的情况如图 5.1 所示。

5.3.1 同步发送操作

MPI 有许多用于同步通信的例程，例如 MPI_Ssend。为了强调非阻塞和异步是不同的概念，有一个例程 MPI_Issend，它是同步的但非阻塞的。这些例程的调用序列与其非显式同步的变体相同，仅在语义上有所不同。

参见章节 4.1.4.2 中的示例。

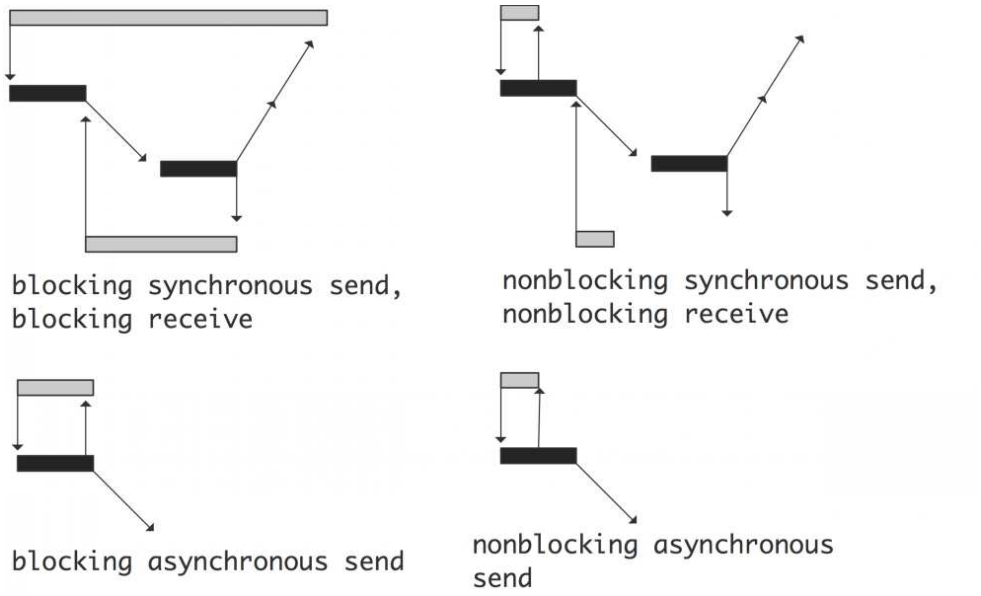


图 5.1：阻塞与同步性

5.4 局部和非局部操作

MPI 标准并未规定通信是否为缓冲的。如果消息是缓冲的，发送调用可以完成，即使尚未发布对应的接收。参见章节 4.1.4.2。因此，在标准通信中，发送操作是 非局部的：其完成可能依赖于对应的接收是否已发布。一个 局部操作 是指非非局部的操作。

另一方面，缓冲通信 (例程 `MPI_Bsend`, `MPI_Ibsend`, `MPI_Bsend_init` ； 章节 5.5) 是局部的：显式缓冲区的存在意味着发送操作无论接收是否已发布都可以完成。

同步发送 (例程 `MPI_Ssend`, `MPI_Issend`, `MPI_Ssend_init` ； 章节 15.8) 再次是非局部的（即使在非阻塞变体中），因为它只有在接收调用完成时才会完成。

最后， *readymode send* (`MPI_Rsend`, `MPI_Irsend`) 是非本地的，因为它的唯一正确用法是在对应的 receive 已经被发出时。

5.5 Buffered communication

到现在为止，你可能已经明白在 MPI 中管理缓冲区空间是很重要的：数据必须存放在某处，要么是在用户分配的数组中，要么是在系统缓冲区中。使用 *buffered communication* 是管理缓冲区空间的另一种方式。

1. 你分配自己的缓冲区空间，并将其附加到你的进程上。这个缓冲区不是发送缓冲区：它是用来替代 MPI 库内部或网络卡上使用的缓冲区空间；见图 5.2。如果有高带宽内存可用，你可以在那里创建你的缓冲区。

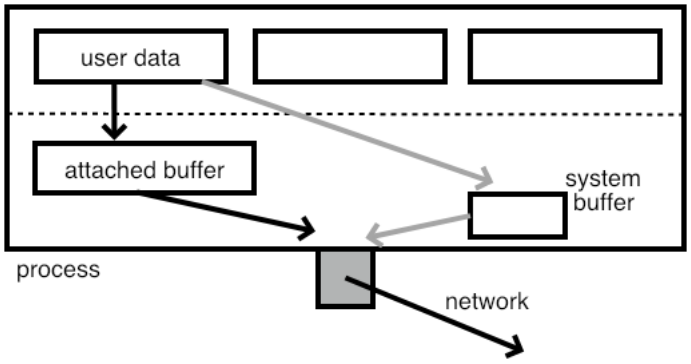


图 5.2：通过附加缓冲区路由的用户通信

图 5.7 MPI_Bsend

名称 参数名	说明	C 类型	F ty pe	i nout
MPI_Bsend (				
MPI_Bsend_c (				
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in send buffer	int MPI_Count	INTEGER	IN
datatype	datatype of each send buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
tag	message tag	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
)				

以下内容适用于最近发布的 *MPI-4* 标准，可能尚未被支持。

2. Abuffer 也可以直接附加到 communicator 或会话；见下文。*MPI-4* 内容结束

3. 你使用 `MPI_Bsend`（图 5.7）（或其 本地 变体 `MPI_Ibsend`）调用进行发送，使用其他正常的发送和接收缓冲区；

4. 当你完成缓冲发送后，解除缓冲区的附加。

缓冲发送的一个优点是它们是非阻塞的：由于有一个足够长以容纳消息的保证缓冲区，因此不必等待接收进程。

我们演示缓冲发送的使用：

```
// bufring.c
int bsize = BUFLEN*sizeof(float);float
*sbuf = (float*) malloc( bsize ),
*rbuf = (float*) malloc( bsize );
MPI_Pack_size( BUFLEN,MPI_FLOAT,comm,&bsize);
```

图 5.8 MPI_Buffer_attach

名称 参数名 说明		C 类型	F 类型	输入输出
MPI_Buffer_attach (				
MPI_Buffer_attach_c (				
buffer	initial buffer address	void*	TYPE(*), DIMENSION(..)	IN
size	buffer size, in bytes	int / MPI_Count	INTEGER	IN
)				

图 5.9 MPI_Comm_attach_buffer

```
bsize += MPI_BSEND_OVERHEAD;
float
  *buffer = (float*) malloc( bsize );

MPI_Buffer_attach( buffer,bsize );
err = MPI_Bsend(sbuf,BUFLEN,MPI_FLOAT,next,0,comm);MPI_Recv(rbuf,BUFLEN,MPI_FLOAT,prev,0,comm,MPI_STATUS_IGNORE)
                                                                                    ;
MPI_Buffer_detach( &buffer,&bsize );
```

5.5.1 Buffer treatment

如果你直接将缓冲区附加到 MPI 进程，使用 MPI_Buffer_attach（图 5.8）每个进程只能有一个缓冲区。以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

调用 MPI_Comm_attach_buffer （图 5.9）和 MPI_Comm_detach_buffer （图 5.10）（截至 MPI-4.1）可用于为每个通信器设置一个缓冲区。同样， MPI_Session_attach_buffer 和 MPI_Session_detach_buffer

还有： MPI_Comm_flush_buffer， MPI_Session_flush_buffer，以及一个全局函数 *MPI_Buffer_flush*。
MPI-4 内容结束
T缓冲区大小应足够容纳所有同时未完成的 MPI_Bsend 调用。你可以
c使用 MPI_Pack_size 计算所需的缓冲区大小；参见第 6.8 节。此外，还需要一个
MPI_BSEND_OVERHEAD 。参见上面的代码片段。
以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

在任何附加例程中指定 MPI_BUFFER_AUTOMATIC 作为 *buffer* 参数。该 size 参数随后被忽略。

MPI-4 材料结束

缓冲区通过 MPI_Buffer_detach（图 5.11）分离。该调用返回缓冲区的地址和大小；调用会阻塞，直到所有缓冲消息都已发送完毕。

注意， MPI_Buffer_attach 和 MPI_Buffer_detach 都带有 void* 缓冲区参数，但

图 5.10 MPI_Comm_detach_buffer

图 5.11 MPI_Buffer_detach

名称 参数名	说明	C 类型	F type	i nout
MPI_Buffer_detach (				
MPI_Buffer_detach_c (				
buffer_addr	initial buffer address	void*	TYPE(C_PTR)	OUT
size	buffer size, in bytes	int* MPI_Count*	INTEGER	OUT
)				

- 在 attach 例程中这是缓冲区的地址，
- 而在 detach 例程中它是缓冲区指针的地址。

这样做是为了让 detach 例程可以将缓冲区指针置零。

虽然缓冲发送是非阻塞的，类似于 MPI_Isend，但没有对应的等待调用。你可以通过强制发送来

```
MPI_Buffer_detach( &b, &n );
MPI_Buffer_attach( b, n );
```

MPL 注释 *40*：缓冲发送。 创建和附加缓冲区是通过 bsend_buffer 完成的，支持例程 bsend_size 有助于计算缓冲区大小：

```
// bufring.cxxvector<float> sbuf(BUFLEN), rbuf(BUFLEN);
int size{ comm_world.bsend_size<float>(mpl::contiguous_layout<float>(BUFLEN)) };
mpl::bsend_buffer buff(size);
comm_world.bsend(sbuf.data(),mpl::contiguous_layout<float>(BUFLEN), next);
```

常量： *mpl*::bsend_overhead 是 *constexpr*，对应 MPI 常量 MPI_BSEND_OVERHEAD。

MPL 注释 *41*：缓冲区附加和分离。有一个单独的附加例程，但通常由 bsend_buffer 的构造函数调用。同样，分离例程在缓冲区析构函数中调用。

```
void mpl::environment::buffer_attach (void *buff, int size);
std::pair< void *, int > mpl::environment::buffer_detach ();
```

5.5.2 Buffered sendcalls

可能的错误代码有

- MPI_SUCCESS 例程成功完成。
- MPI_ERR_BUFFER 缓冲区指针无效；这通常意味着您提供了一个空指针。
- MPI_ERR_INTERN 检测到 MPI 内部错误。

异步版本是 MPI_Ibsend，持久（参见章节 5.1 ）调用是 MPI_Bsend_init (图 5.12)。

图 5.12 MPI_Bsend_init

名称 参数名	说明	C 类型	F 类型	inout
MPI_Bsend_init (				
MPI_Bsend_init_c (				
buf	initial address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements sent	int / MPI_Count	INTEGER	IN
datatype	type of each element	MPI_Datatype	TYPE (MPI_Datatype)	IN
dest	rank of destination	int	INTEGER	IN
tag	message tag	int	INTEGER	IN
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

5.5.3 局部行为

attach 例程是局部的；detach 和 flush 例程是非局部的。

第 6 章

MPI 主题：数据类型

在到目前为止你所见过的示例中，每次发送的数据都是作为单一类型元素的连续缓冲区。实际上，你可能想发送非连续或异构数据。

- 作为非连续数据的一个例子，传输复数数组的实部意味着指定每隔一个数字。
- 当传输包含多种类型元素的 C 结构体或 Fortran 类型时，需要异构数据。

到目前为止你处理的数据类型被称为预定义数据类型；你为处理其他数据创建的数据类型被称为派生数据类型。

6.1 `MPI_Datatype` 数据类型

数据类型如 `MPI_INT` 是类型 `MPI_Datatype` 的值。该类型在不同语言中处理方式不同。

In C you can declare variables as

```
MPI_Datatype mytype;
```

Fortran 注 9：句柄的派生类型。 在 Fortran 2008 之前，数据类型变量存储在 `Integer` 变量中。随着 Fortran 2008 标准的引入，数据类型成为 Fortran 派生类型：

```
!! vector.F90
  Type(MPI_Datatype) :: newvectortype
```

从实现角度讲，这些类型恰好有一个成员，*MPI_VAL*，它是与早期 Fortran 版本中用于该数据类型的整数相同的整数。

Python note 17: 数据类型。 There is a 类

```
mpi4py.MPI.Datatype
```

with predefined values such as

```
mpi4py.MPI.Datatype.DOUBLE
```

它们本身是具有创建派生类型方法的对象；参见章节 6.3.1。

MPL 注释 *42*：数据类型处理。 MPL 主要通过 `layout` 类的子类来处理数据类型。布局是 MPL 例程基于数据类型的模板。

```
// sendlong.cxx
mpl::contiguous_layout<long long> v_layout(v.size());
  comm.send(v.data(), v_layout, 1); // send to rank 1
```

也适用于 float 和 double 的复数。

MPL 能推断其内部表示的数据类型包括枚举类型、固定大小的 C 数组以及 C++ 标准模板库中的模板类 `std::array`、 `std::pair` 和 `std::tuple`。唯一的限制是，C 数组和上述模板类所包含的数据元素类型必须是 MPL 能发送或接收的类型。

MPL 注释 *43*：原生 *MPI* 数据类型。如果您需要 `MPI_Datatype` 对象，该对象包含在 MPL 布局中，可以使用访问函数 `native_handle`。

6.2 预定义数据类型

MPI 有多种预定义的数据类型

- 首先是与宿主语言的简单数据类型对应的类型。名称设计得类似于 C 和 Fortran 的类型，例如 `MPI_FLOAT` 和 `MPI_DOUBLE` 对应于 `float` 和 `double` 在 C 中， 而 `MPI_REAL` 和 `MPI_DOUBLE_PRECISION` 对应于 `Real` 和 `Double precision` 在 Fortran 中。
- 类型 `MPI_PACKED` 和 `MPI_BYTE` 不对应于语言类型。
- 类型 `MPI_Aint` (以及 Fortran 的 kind `MPI_ADDRESS_KIND`) 用于远程内存访问 (RMA) 窗口；参见第 9.3.1 节。
- 类型 `MPI_Offset` (以及对应的 Fortran `MPI_OFFSET_KIND` kind) 用于定义 `MPI_Offset` 量， 用于文件 I/O ；参见第 10.2.2 节。
- 类型 `MPI_Count` 描述缓冲区；参见章节 6.4。
- 类型 `MPI_CHAR` 对应一个字符，这与 C `char` 不同：它可以超过一个字节。此外，MPI 在不同架构之间通信时会转换本地字符表示。

6.2.1 C/C++

这里我们以 C/C++ 为例，说明用于声明变量的类型与该类型在 MPI 通信例程中的对应关系：

```
long int i;
MPI_Send(&i,1,MPI_LONG,target,tag,comm);
```

参见表 6.1。

- 对 C99 类型有一定但不完全的支持；参见表 6.2。

C 类型	MPI 类型
`char`	`MPI_CHAR`
`unsigned char`	`MPI_UNSIGNED_CHAR`
`char`	`MPI_SIGNED_CHAR`
`short`	`MPI_SHORT`
`unsigned short`	`MPI_UNSIGNED_SHORT`
`int`	`MPI_INT`
`unsigned int`	`MPI_UNSIGNED`
`long int`	`MPI_LONG`
`unsigned long int`	`MPI_UNSIGNED_LONG`
`long long int`	`MPI_LONG_LONG_INT`
`float`	`MPI_FLOAT`
`double`	`MPI_DOUBLE`
`long double`	`MPI_LONG_DOUBLE`
`unsigned char`	`MPI_BYTE`
（不对应于 C 类型）	`MPI_PACKED`

表 6.1：C 语言中的预定义数据类型

C9 类型	MPI 类型
_Bool	`MPI_C_BOOL`
`float` *_Complex*	`MPI_C_COMPLEX`
	`MPI_C_FLOAT_COMPLEX`
`double` *_Complex*	`MPI_C_DOUBLE_COMPLEX`
`long double` *_Complex*	`MPI_C_LONG_DOUBLE_COMPLEX`

Table 6.2: C99 synonym types.

- 支持 C11 固定宽度整数类型；见表 6.3。
- 该 `MPI_LONG_INT` 类型不是整数类型，而是 `long` 和 `int` 打包在一起；见第 3.10.1.1 节。
- 见第 6.2.4 节，了解 `MPI_Aint` 及更多关于字节计数的信息。

6.2.2 Fortran

表 6.4 列出了标准 Fortran 类型和常见扩展。右表中的所有类型不必全部支持；例如 `MPI_INTEGER16` 可能不存在，在这种情况下它将等同于 `MPI_DATATYPE_NULL`。

The default integer type `MPI_INTEGER` is equivalent to `INTEGER(KIND=MPI_INTEGER_KIND)`.

C11 类型	MPI 类型
int8_t	*MPI_INT8_T*
int16_t	*MPI_INT16_T*
int32_t	*MPI_INT32_T*
int64_t	*MPI_INT64_T*
uint8_t	*MPI_UINT8_T*
uint16_t	*MPI_UINT16_T*
uint32_t	*MPI_UINT32_T*
uint64_t	*MPI_UINT64_T*

表 6.3：C11 固定宽度整数类型。

		MPI_INTEGER1
MPI_CHARACTER	Character(Len=1)	MPI_INTEGER2
MPI_INTEGER		MPI_INTEGER4
MPI_REAL		MPI_INTEGER8
MPI_DOUBLE_PRECISION		MPI_INTEGER16
MPI_COMPLEX		MPI_REAL2
MPI_LOGICAL		MPI_REAL4
MPI_BYTE		MPI_REAL8
MPI_PACKED		MPI_DOUBLE_COMPLEX
		Complex(Kind=Kind(0.d0))

Table 6.4: Standard Fortran types (left) and common extension (right)

6.2.2.1 *Fortran90 kind-defined types*

如果您的 Fortran90 代码使用 `KIND` 来定义具有指定精度的标量类型，您需要使用以下例程来实现 *Fortran* 标量类型的 *MPI* 等价物：

- `MPI_Type_create_f90_integer` (图 6.1)
- `MPI_Type_create_f90_real` (图 6.2)
- `MPI_Type_create_f90_complex` (图 6.3)。

Example of an integer kind;

```
INTEGER ( KIND = SELECTED_INT_KIND(15) ) , &
    DIMENSION(100) :: array
INTEGER :: root , error
Type(MPI_Datatype) :: integertype

CALL MPI_Type_create_f90_integer( 15 , integertype , error )
CALL MPI_Bcast ( array , 100 , &
 integertype , root , MPI_COMM_WORLD , error )
! error parameter optional in f08, both routines.
```

图 6.1 MPI_Type_create_f90_integer

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Type_create_f90_integer (					
	r	decimal exponent range, i.e., number of decimal digits	int	INTEGER	IN
	newtype	the requested MPI datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

图 6.2 MPI_Type_create_f90_real

Name	Param name	Explanation	C 类型	F 类型	输入输出
MPI_Type_create_f90_real (					
	p	precision, in decimal digits	int	INTEGER	IN
	r	decimal exponent range	int	INTEGER	IN
	newtype	the requested MPI datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

代码：

```
!! kindsend.F90
  Integer,parameter :: digits=16
  Integer,parameter :: ip = Selected_Int_Kind(digits)
  Integer (kind=ip) :: data
  Type(MPI_Datatype) :: mpi_ip
  Call MPI_Type_create_f90_integer(digits,mpi_ip)
  if (rank==0) then
     print *,"Fortran type has range",range(data)
     call MPI_Send( data,1,mpi_ip, 1,0,comm )
  else if (rank==1) then
     call MPI_Recv( data,1,mpi_ip, 0,0,comm,
    ↪MPI_STATUS_IGNORE )
```

Output:

```
Fortran type has range
   ↪     18
 Sending:  729000000000000
Received:  729000000000000
```

真实类型的示例：

```
REAL ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
DIMENSION(100) :: array
CALL MPI_Type_create_f90_real( 15 , 300 , realtype , error )
```

复杂类型的示例：

```
COMPLEX ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &DIMENSION(100) :: array
CALL MPI_Type_create_f90_complex( 15 , 300 , complextype , error )
```

Remark 17 因此创建的 *MPI* 类型是预定义数据类型，因此无需提交或释放它们。

图 6.3 MPI_Type_create_f90_complex

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Type_create_f90_complex (					
	p	precision, in decimal digits	int	INTEGER	IN
	r	decimal exponent range	int	INTEGER	IN
	newtype	the requested MPI datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

6.2.3 Python

Python note 18: 预定义数据类型。 本节 6.2.3 讨论了 Python 中的预定义数据类型。

In python, all buffer data comes from *Numpy*.

mpi4py 类型	NumPy 类型
MPI.INT	np.intc
	np.int32
MPI.LONG	np.int64
MPI.FLOAT	np.float32
MPI.DOUBLE	np.float64

在此表中我们看到 Numpy 有三种整数类型，一种对应于 C `int`s，另外两种明确指定位数。曾经有一种 `np.int` 类型，但自 *Numpy1.20* 起已被弃用。

示例：代码：

```
## inttype.py
sizeofint = np.dtype('int32').itemsize
print("Size of numpy int32: {}".format(sizeofint))
sizeofint = np.dtype('intc').itemsize
print("Size of C int: {}".format(sizeofint))
```

Output:

```
Size of numpy int32: 4
Size of C int: 4
```

```
## allgatherv.py
mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)
```

6.2.3.1 类型对应 *MPI/Python*

Above we saw that the number of bytes of a Numpy type can be deduced from

```
sizeofint = np.dtype('intc').itemsize
```

可以推导出与 MPI 类型对应的 Numpy 类型：

```
## typesize.py
datatype = MPI.FLOAT
typecode = MPI._typecode(datatype)
assert typecode is not None # check MPI datatype is built-in
dtype = np.dtype(typecode)
```

6.2.4 字节寻址类型

到目前为止，我们主要是在发送的上下文中讨论数据类型。 MPI_Aint 类型不仅用于发送，更用于描述对象的大小，例如 MPI_Win 对象的大小（章节 9.1 ）或 MPI_Type_create_hindexed 中的字节位移。

地址具有类型 MPI_Aint。地址范围的起始由 MPI_BOTTOM 给出。另见 MPI_Sizeof(章节 6.2.5) 和 MPI_Get_address (章节 6.3.6) 例程。

类型为 MPI_Aint 的变量可以作为 MPI_AINT 发送：

```
MPI_Aint address;
MPI_Send( address,1,MPI_AINT, ... );
```

参见章节 9.5.3 了解示例。

为了防止字节计算中的溢出错误，有支持例程 MPI_Aint_add

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
```

类似地 MPI_Aint_diff。

Fortran 注释 *10*：*Fortran* 中的字节计数类型。 相当于 MPI_Aint 的 Fortran 类型是 kind 为 MPI_ADDRESS_KIND 的整数：

```
integer(kind=MPI_ADDRESS_KIND) :: winsize
```

使用这种整数类型来计算窗口的大小还需要能够查询该窗口中数据类型的大小。详见第 6.2.5 节。

示例用法见 MPI_Win_create：

```
call MPI_Sizeof(windowdata,window_element_size,ierr)
window_size = window_element_size*500
call MPI_Win_create( windowdata,window_size,window_element_size,... )
```

Python 注释 *19*：*numpy* 类型的大小。 这里有一个查找 *numpy* 数据类型字节大小的好方法：

```
## putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

图 6.4 MPI_Type_match_size

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Type_match_size (				
typeclass	generic type specifier	int	INTEGER	IN
size	size, in bytes, of representation	int	INTEGER	IN
datatype	datatype with correct type, size	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

6.2.5 匹配 languagetypetoMPI 类型

在某些情况下，您可能想找到与编程语言中的某种类型对应的 MPI 类型。

- 在 C++ 函数和类可以是模板化的，这意味着类型并不完全确定：

```
template<typename T> {
class something<T> {
public:
  void dosend(T input) {
    MPI_Send( &input,1,/* ????? */ );
  };
};
```

（注意，在 MPL 中几乎不需要这样做，因为 MPI 调用在那里是模板化的。）

- Petsc 安装使用通用标识符 PetscScalar （或 PetscReal）以及依赖配置的实现。

- 数据类型的大小并不总是静态已知的，例如当使用 Fortran KIND 关键字时。

H这里有一些 MPI 机制可以解决这个问题。

6.2.5.1 C 语言中的类型匹配

C 语言中的数据类型可以通过 MPI_Type_match_size（图 6.4）转换为 MPI 类型，其中 *typeclass* 参数是以下之一 MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER, MPI_TYPECLASS_COMPLEX。

图 6.5 MPI_Type_size

名称	参数名	说明	C 类型	F 类型	i nout
MPI_Type_size (					
MPI_Type_size_c (					
	datatype	datatype to get information on	MPI_Datatype	TYPE (MPI_Datatype)	IN
	size	datatype size	int* MPI_Count*	INTEGER	OUT
	)				

图 6.6 MPI_Sizeof

Name	参数名	说明	C 类型	F 类型	输入输出
MPI_Sizeof (					
	)				

Code:

```
// typematch.c
float x5;
double x10;
int s5,s10;
MPI_Datatype mpi_x5,mpi_x10;

MPI_Type_match_size
  (MPI_TYPECLASS_REAL,sizeof(x5),&mpi_x5);
MPI_Type_match_size
  (MPI_TYPECLASS_REAL,sizeof(x10),&mpi_x10);
MPI_Type_size(mpi_x5,&s5);
printf("float:  size=%d, mpi size=%d\n",
         sizeof(x5),s5);
MPI_Type_size(mpi_x10,&s10);
printf("double: size=%d, mpi size=%d\n",
         sizeof(x10),s10);
```

Output:

```
mpiexec -n 1 ./typematch
float:  size=4, mpi size=4
double: size=8, mpi size=8
```

MPI 对结构类型所占空间的查询方式有多种。首先 MPI_Type_size(图 6.5) 计算 数据类型大小 为类型中数据所占用的字节数。这意味着在 *MPI* 向量数据类型 中，它不计算间隙。

```
// typesize.c
MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Type_size(newtype,&size);
ASSERT( size==(count*bs)*sizeof(double) );
```

6.2.5.2 *Fortran* 中的类型匹配

在 Fortran 中，可以通过语言中的数据类型大小获得 MPI_Sizeof (图 6.6) （注

非可选的错误参数！该例程在 MPI-4 中已被弃用：建议使用 `storage_size` （报告位数）和 / 或 `c_sizeof` （来自 `iso_c_binding` 模块，报告字节数）。

```
!! matchkind.F90
     call MPI_Sizeof(x10,s10,ierr)
     call MPI_Type_match_size(MPI_TYPECLASS_REAL,s10,mpi_x10)
     call MPI_Type_size(mpi_x10,s10)
     print *,"10 positions supported, MPI type size is",s10
```

Petsc 有其自己的翻译机制；参见章节 32.2。

6.3 派生数据类型

MPI 允许你创建自己的数据类型，这在某种程度上（但不完全是 …… ）类似于在编程语言中定义结构体。如果你想在一条消息中发送多个项目，MPI 数据类型大多是有用的。

仅使用预定义数据类型存在两个问题，正如你迄今所见。

- MPI 通信例程只能发送单一数据类型的倍数：即使它们在内存中是连续的，也无法发送不同类型的项目。理论上可以使用 `MPI_BYTE` 数据类型，但这并不可取。
- 通常也无法发送非连续内存中的同一类型项目。当然，你可以发送包含所需项目的连续内存区域，但这会浪费带宽和接收端的内存空间。

使用 MPI 数据类型，你可以通过多种方式解决这些问题。

- 你可以创建一个新的连续数据类型，它由另一数据类型的元素数组组成。发送这种类型的一个元素与发送组件类型的多个元素本质上没有区别。
- 你可以创建一个向量数据类型，它由组件类型的规则间隔块元素组成。这是发送非连续数据问题的第一种解决方案。
- 对于不规则间隔的数据，有索引数据类型，您可以为组件类型的元素块指定一个索引位置数组。每个块的大小可以不同。
- 结构体数据类型可以容纳多种数据类型。

您可以结合这些机制来获得不规则间隔的异构数据，等等。

6.3.1 基本调用

创建新数据类型的典型调用顺序如下：

- 您需要一个用于数据类型的变量；其类型为 `MPI_Datatype`。
- 有一个 create 调用，随后是一个 ‘commit’ 调用，MPI 在此执行内部记录和优化；我们将在下面详细讨论这一点。
- 类型需要被 ‘committed’ 。之后：

- 该数据类型被使用，可能多次使用；
- 当数据类型不再需要时，必须释放以防止内存泄漏；参见章节 6.3.1.2。

在代码中：

```
MPI_Datatype newtype;
MPI_Type_something( < oldtype specifications >, &newtype );
MPI_Type_commit( &newtype );
/* code that uses your new type */
MPI_Type_free( &newtype );
```

In Fortran2008:

```
Type(MPI_Datatype) :: newvectortype
call MPI_Type_something( <oldtype specification>, &
          newvectortype)
call MPI_Type_commit(newvectortype)
!! code that uses your type
call MPI_Type_free(newvectortype)
```

Python note 20: Derivedtype handling. 各种类型创建例程是 datatype 类的方法，之后 commit 和 free 是新类型上的方法。

```
## vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

MPL note 44: Derived type handling. 在 MPL 中，类型创建例程位于主命名空间，针对 datatype 进行了模板化。

```
// vector.cxx
vector<double>
  source(stride*count);
if (procno==sender) {
  mpl::strided_vector_layout<double>
    newvectortype(count,1,stride);
  comm_world.send
    (source.data(),newvectortype,the_other);
}
```

The commit call is part of the type creation, and freeing is done in the destructor.

MPL note 45: Layouts.

```
namespace mpl {
  template <typename T> class layout; // Basisklasse
```

图 6.7 MPI_Type_commit

名称	参数名	说明	C 类型	F 类型	i nout
MPI_Type_commit (					
	datatype	datatype that is committed	MPI_Datatype*	TYPE (MPI_Datatype)	INOUT
)					

MPL:

```
Done as part of the type create call.
```

```
  template <typename T> class null_layout; // MPI_DATATYPE_NULL
  template <typename T> class empty_layout; // leere Nachricht
  template <typename T> class contiguous_layout; // MPI_Type_contiguous
  template <typename T> class vector_layout; // MPI_Type_contiguous
  template <typename T> class strided_vector_layout; // MPI_Type_vector
  template <typename T> class indexed_layout; // MPI_Type_indexed
  template <typename T> class hindexed_layout; // MPI_Type_create_hindexed
  template <typename T> class indexed_block_layout; // MPI_Type_create_indexed_block
  template <typename T> class hindexed_block_layout;  // MPI_Type_create_hindexed_block
  template <typename T> class iterator_layout; // MPI_Type_create_hindexed_block
  template <typename T> subarray_layout; // MPI_Type_create_subarray
  class heterogeneous_layout; // MPI_Type_create_struct
}
```

6.3.1.1 *Create* 调用

The `MPI_Datatype` 变量通过调用以下例程之一来获取其值:

- `MPI_Type_contiguous` 用于连续数据块; 章节 6.3.2 ;
- `MPI_Type_vector` 用于规则步长数据; 章节 6.3.3 ;
- `MPI_Type_create_subarray` 用于更高维块的子集; 章节 6.3.4 ;
- `MPI_Type_create_struct` 用于异构不规则数据; 章节 6.3.7 ;
- `MPI_Type_indexed` 和 `MPI_Type_hindexed` 用于不规则步长数据; 章节 6.3.6。

These calls take an existing type, whether predefined or also derived, and produce a new type.

6.3.1.2 *Commitand free*

有必要调用 `MPI_Type_commit` (图 6.7) 在一个新数据类型上, 这使得 MPI 为该数据类型执行索引计算。

当你不再需要该数据类型时, 你调用 `MPI_Type_free` (图 6.8)。 (这通常在面向对象 API 中不需要。) 这具有以下效果:

- 数据类型标识符的定义将被更改为 `MPI_DATATYPE_NULL`。
- 任何已经开始使用该数据类型的通信都将成功完成。
- 以该数据类型定义的数据类型仍然可以使用。

图 6.8 MPI_Type_free

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Type_free (				
datatype	datatype that is freed	MPI_Datatype*	TYPE (MPI_Datatype)	INOUT
)				

MPL:

```
Done in the destructor.
```

图 6.9 MPI_Type_contiguous

名称 参数名	说明	C 类型	F ty pe	i nout
MPI_Type_contiguous (				
MPI_Type_contiguous_c (				
count	replication count	int MPI_Count	INTEGER	IN
oldtype	old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype	new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

Python:

```
Create_contiguous(self, int count)
```

6.3.2 连续类型

最简单的派生类型是 “ 连续 ” 类型，由 MPI_Type_contiguous (图 6.9) 构造。

A 连续类型描述了一个预定义或先前定义类型的数组。没有区别 -
发送一个连续类型的元素和发送多个组成类型的元素之间的区别。如图 6.1 所示。

```
// contiguous.c
MPI_Datatype newvectortype;
if (procno==sender) {
  MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,receiver,0,comm);
  MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,sender,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(count==recv_count);
}
```

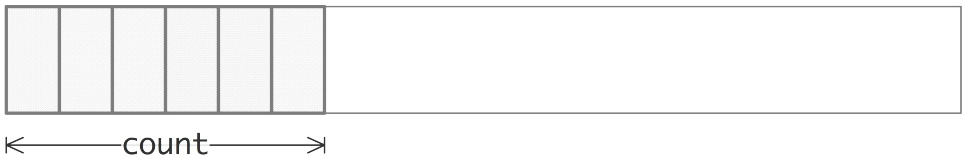


图 6.1：一个连续数据类型由组成类型的元素构建而成

```
!! contiguous.F90
  integer :: newvectortype
  if (mytid==sender) then
     call MPI_Type_contiguous(count,MPI_DOUBLE_PRECISION,newvectortype)
     call MPI_Type_commit(newvectortype)
     call MPI_Send(source,1,newvectortype,receiver,0,comm)
     call MPI_Type_free(newvectortype)
  else if (mytid==receiver) then
     call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
          recv_status)
     call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
     !ASSERT(count==recv_count);
  end if
```

```
## contiguous.py
source = np.empty(count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newcontiguoustype = MPI.DOUBLE.Create_contiguous(count)
    newcontiguoustype.Commit()
    comm.Send([source,1,newcontiguoustype],dest=the_other)
    newcontiguoustype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

MPL 注释 *46*：连续类型。MPL 接口广泛使用 `contiguous_layout`，因为它是声明非标量缓冲区的主要方式；参见注释 11。

MPL 注释 *47*：连续组合。连续布局只能使用预定义类型或其他连续布局作为它们的 “ 旧 ” 类型。要为其他布局创建连续类型，使用 `vector_layout`：

```
// contiguous.cxx
mpl::contiguous_layout<int> type1(7);
mpl::vector_layout<int> type2(8,type1);
```

（与 `strided_vector_layout` 形成对比；注意 48。）

6.3.3 Vector type

最简单的非连续数据类型是 ‘vector’ 类型，由 `MPI_Type_vector`（图 6.10）构造。vector 类型描述了一系列大小相等、间隔固定的块。这一点如图所示

图 6.10 MPI_Type_vector

Name	Param name	Explanation	C type	F type	inout
MPI_Type_vector (					
MPI_Type_vector_c (					
	count	number of blocks	int MPI_Count	INTEGER	IN
	blocklength	number of elements in each block	int MPI_Count	INTEGER	IN
	stride	number of elements between start of each block	int MPI_Count	INTEGER	IN
	oldtype	old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
	newtype	new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)					

Python:

```
MPI.Datatype.Create_vector(self, int count, int blocklength, int stride)
```

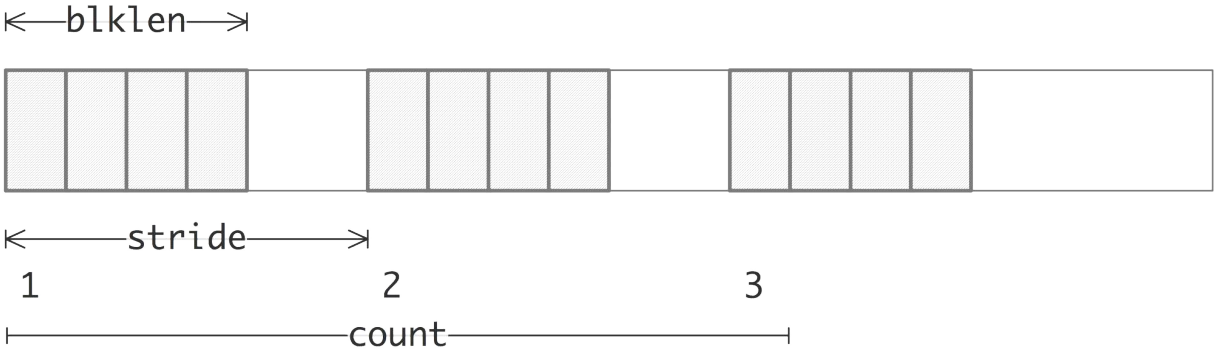


Figure 6.2: 一个向量数据类型是由组成类型的带间隔块元素构建而成的

在图 6.2 中。

向量数据类型首次非平凡地展示了数据类型在发送方和接收方可以不同。如果发送方发送 `b` 个长度为 `l` 的块，接收方可以将它们作为 `bl` 个连续元素接收，无论是作为连续数据类型，还是作为预定义类型的连续缓冲区；见图 6.3。接收方不知道发送方数据类型的步长。

在此示例中，向量类型仅在发送方创建，以便发送数组的步进子集；接收方将数据作为连续块接收。

```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
MPI_Datatype newvectortype;
if (procno==sender) {
```

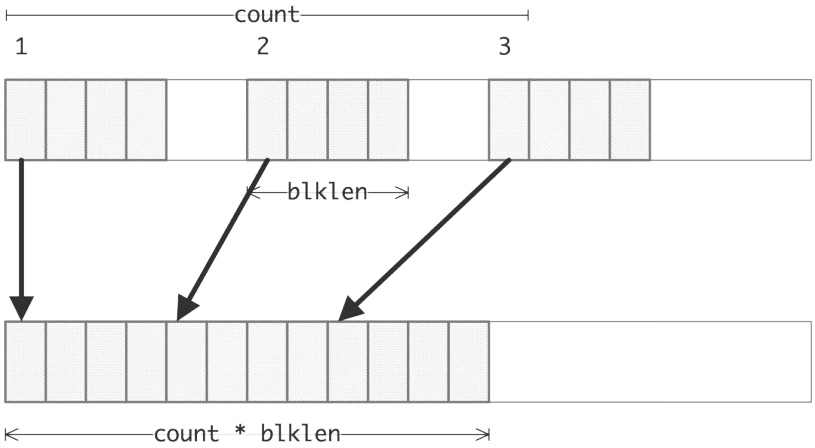


Figure 6.3: 发送向量数据类型并以预定义类型接收 d or contiguous

```
  MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(recv_count==count);
}
```

We illustrate Fortran2008:

```
if (mytid==sender) then
call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
newvectortype)call MPI_Type_commit(newvectortype)
call MPI_Send(source,1,newvectortype,receiver,0,comm)
call MPI_Type_free(newvectortype)
if ( .not. newvectortype==MPI_DATATYPE_NULL) thenprint *,
"Trouble freeing datatype"elseprint *,
"Datatype successfully freed"end ifelse if (mytid==receiver) then
call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
recv_status)
call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
end if
```

在传统的 Fortran90 模式下，代码保持不变，只是类型声明为 `Integer`:

```
!! vector.F90integer :: newvectortypeinteger :: recv_count
call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
newvectortype,err)call MPI_Type_commit(newvectortype,err)
```

Python 注释 *21*：向量类型。向量创建例程是 datatype 类的一个方法。有关一般讨论，请参见第 6.3.1 节。

```
## vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

MPL 注释 *48*：向量类型。MPL 有 `strided_vector_layout` 类作为向量类型的等价物：

```
// vector.cxx
vector<double>
  source(stride*count);
if (procno==sender) {
  mpl::strided_vector_layout<double>
    newvectortype(count,1,stride);
  comm_world.send
    (source.data(),newvectortype,the_other);
}
```

（参见注释 47 关于非步长向量。）

6.3.3.1 二维数组

图 6.4 显示了不规则数据的一个来源：对于采用 列主存储 的矩阵，一列在内存中是连续存储的。然而，这样的矩阵的一行并不连续；其元素之间的间隔是一个 步长，等于列的长度。

练习 6.1. 如果整个矩阵的大小为 $M \times N$，子矩阵为 $m \times n$，你会如何描述子矩阵的内存布局？

作为此数据类型的一个例子，考虑转置矩阵的例子，例如在 C 和 Fortran 数组之间转换。假设一个处理器有一个以 C 行主布局存储的矩阵，并且它需要将一列发送给另一个处理器。如果矩阵声明为

```
int M,N; double mat[M][N]
```

那么一列有 M 个单元素块，块之间间隔 N 个位置。换句话说：

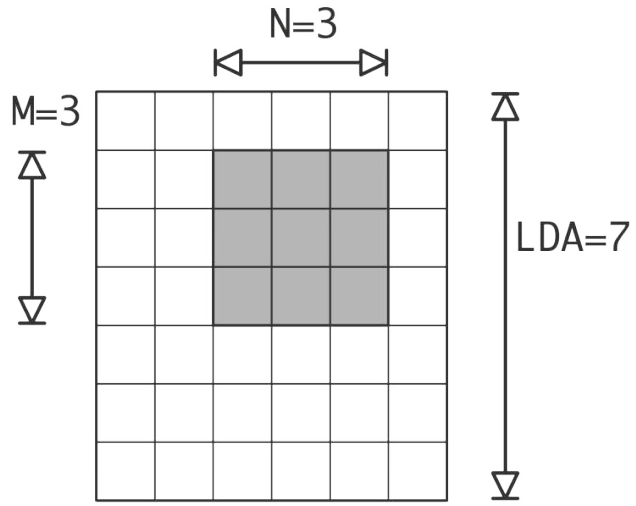


图 6.4：列主存储中矩阵一行和一列的内存布局

```
MPI_Datatype MPI_column;
MPI_Type_vector(
    /* count= */ M, /* blocklength= */ 1, /* stride= */ N,
    MPI_DOUBLE, &MPI_column );
```

发送第一列很简单：

```
MPI_Send( mat, 1,MPI_column, ... );
```

T第二列就稍微复杂一点：你现在需要挑选具有相同步幅的元素，但
s从 `A[0][1]` 开始。

```
MPI_Send( &(mat[0][1]), 1,MPI_column, ... );
```

你可以通过将索引表达式替换为 `mat+1` 来使其稍微高效一些（但更难阅读）。

Exercise 6.2. 假设你有一个大小为 $4N \times 4N$ 的矩阵，并且你想发送元素 `A[4*i][4*j]`，以及 $i, j = 0, \ldots, N-1$。你如何用一次传输发送这些元素?

Exercise6.3. 在处理器零上分配一个矩阵，使用 Fortran 列主序存储。使用 P 个 sendrecv 调用，将该矩阵的行分发到各个处理器。

Pythonnote 22: Sending fromthe middle of a matrix. 在 C 和 Fortran 中，很容易对数组中间的数据应用派生类型，例如从 C 矩阵中提取任意列，或从 Fortran 矩阵中提取行。虽然 Python 可以轻松描述数组的切片，但通常它会复制这些切片而不是取地址。因此，有必要将矩阵转换为缓冲区并计算字节偏移量：

```
## rowcol.py
rowsize = 4; colsize = 5
```

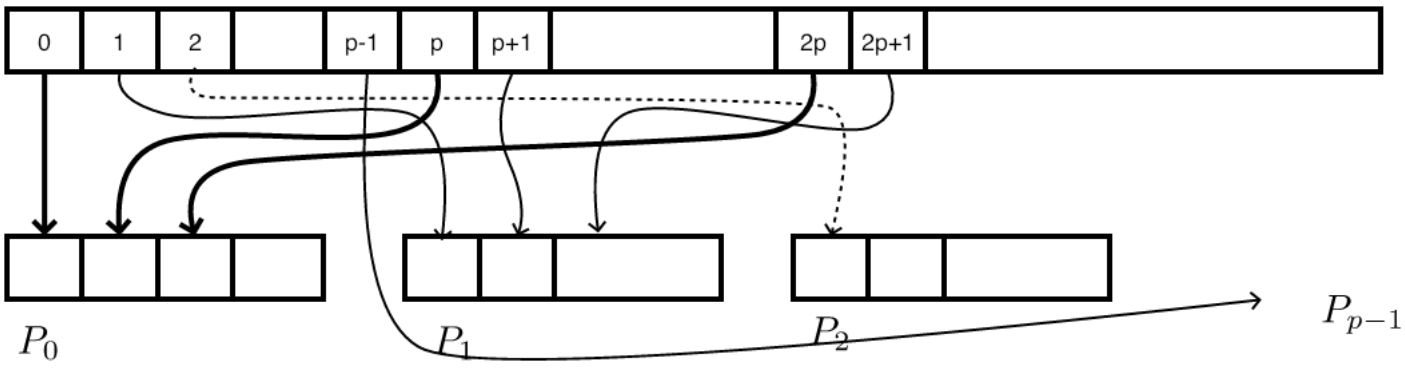


图 6.5：从进程零向所有其他进程发送步进数据

```
coltype = MPI.INT.Create_vector(4, 1, 5)coltype.Commit()columntosend = 2
comm.Send\( [np.frombuffer(matrix.data, intc,
offset=columntosend*np.dtype('intc').itemsize), 1,coltype],receiver)
```

练习 6.4. 让处理器 0 拥有一个长度为 $10P$ 的数组 x，其中 P 是处理器的数量。元素 $0, P, 2P, \ldots, 9P$ 应该发送到处理器零，$1, P+1, 2P+1, \ldots$ 发送到处理器 1，依此类推。

- 将此编码为一系列的 send/recv 调用，发送时使用向量数据类型，接收时使用连续缓冲区。
- 为简化起见，跳过与零的发送。若要包含该情况，最优雅的解决方案是什么？
- 为了测试，定义数组为 $x[i] = i$。

（此练习的骨架代码名为 `stridesend`。）

练习 6.5. 编写代码比较从数组中发送一个跨距子集所需的时间：手动将元素复制到一个较小的缓冲区，或者使用向量数据类型。你发现了什么？你可能需要在相当大的数组上进行测试。

6.3.4 子数组类型

向量数据类型可以通过递归使用来处理维度大于 2 的数组中的块。然而，这样做很繁琐。相反，有一个显式的子数组类型 `MPI_Type_create_subarray`（图 6.11）。它描述了数组的维度和范围，以及子数组的起始点（“左上角”）和范围。

MPL 注 49：子数组布局。该模板 `subarray_layout` 类由一个包含全局大小 / 子块大小 / 第一个坐标三元组的向量构造而成。

```
mpl::subarray_layout<int>(
```

图 6.11 MPI_Type_create_subarray

Name 参数名	说明	C 类型	F 类型	输入输出
MPI_Type_create_subarray (				
MPI_Type_create_subarray_c (				
ndims	number of array dimensions	int	INTEGER	IN
array_of_sizes	number of elements of type oldtype in each dimension of the full array	[const int[] MPI_Count[]	INTEGER (ndims)	IN
array_of_subsizes	number of elements of type oldtype in each dimension of the subarray	[const int[] MPI_Count[]	INTEGER (ndims)	IN
array_of_starts	starting coordinates of the subarray in each dimension	[const int[] MPI_Count[]	INTEGER (ndims)	IN
order	array storage order flag	int	INTEGER	IN
oldtype	old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype	new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

Python:

```
MPI.Datatype.Create_subarray
   (self, sizes, subsizes, starts, int order=ORDER_C)
```

```
 { {ny, ny_1, ny_0}, {nx, nx_1, nx_0} }
);
```

练习 6.6. 假设你的处理器数量是 $P = Q^3$，并且每个进程都有一个大小相同的数组。使用 `MPI_Type_create_subarray` 将所有数据收集到根进程。使用一系列的发送和接收调用；`MPI_Gather` 在这里不适用。（本练习的骨架代码名为 `cubegather`。）

Fortran 注释 *11*：子数组。Fortran 通过数组切片自然支持子数组。

```
!! section.F90integer, parameter :: siz=20real,
dimension(siz,siz) :: matrix = [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ]real,
dimension(2,2) :: submatrixif (procno==0) then
call MPI_Send(matrix(1:2,1:2),4,MPI_REAL,1,0,comm)
else if (procno==1) then
call MPI_Recv(submatrix,4,MPI_REAL,0,0,comm,MPI_STATUS_IGNORE)
if (submatrix(2,2)==22) thenprint *, "Yay"elseprint *, "nay...."end if
end if
```

然而，非阻塞操作存在一个细微之处：对于非连续缓冲区，会创建一个临时对象，该对象在 MPI 调用后被释放。对于阻塞发送这是正确的，但对于非阻塞操作，临时对象必须保留直到 wait 调用。

```
!! sectionisend.F90integer :: sizreal,dimension(:,:),
allocatable :: matrixreal,dimension(2,2) :: submatrixsiz = 20
allocate( matrix(siz,siz) )
matrix = reshape( [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ], (/siz,siz/) )
call MPI_Isend(matrix(1:2,1:2),4,MPI_REAL,1,0,comm,request)
call MPI_Wait(request,MPI_STATUS_IGNORE)deallocate(matrix)
```

在 MPI-3 中，变量 `MPI_SUBARRAYS_SUPPORTED` 表示支持此机制：

```
if ( .not. MPI_SUBARRAYS_SUPPORTED ) then
print *,"This code will not work"
call MPI_Abort(comm,0)end if
```

参数的可能性是 `order` `MPI_ORDER_C` 和 `MPI_ORDER_FORTRAN`。然而，这与元素遍历的顺序无关；它决定了子数组边界的解释方式。举个例子，我们用数字填充一个 4×4 按 C 顺序排列的数组 $0\cdots 15$，并以两种方式发送 $[0,1]\times[0\cdots 4]$ 切片，先是 C 顺序，然后是 Fortran 顺序：

```
// row2col.c
#define SIZE 4
  int
    sizes[2], subsizes[2], starts[2];
  sizes[0] = SIZE; sizes[1] = SIZE;
  subsizes[0] = SIZE/2;  subsizes[1] = SIZE;
  starts[0] = starts[1] = 0;
    MPI_Type_create_subarray
      (2,sizes,subsizes,starts,
       MPI_ORDER_C,MPI_DOUBLE,&rowtype);
    MPI_Type_create_subarray
      (2,sizes,subsizes,starts,
       MPI_ORDER_FORTRAN,MPI_DOUBLE,&coltype);
```

The receiver r接收以下内容，格式化以突出其中的位置 th e numbers originate:

```
Received C order:
    0.000 1.000 2.000
3.000 4.000 5.000 6.000
7.000Received F order:
0.000 1.000 4.000 5.000
8.000 9.000 12.000 13.000
```

图 6.12 MPI_Type_indexed

名称 参数名	说明	C 类型	F 类型	inout
MPI_Type_indexed (				
MPI_Type_indexed_c (				
count	number of blocks---also number of entries in array_of_displacements and array_of_blocklengths	int MPI_Count	INTEGER	IN
array_of_blocklengths	number of elements per block	const int[] MPI_Count[]	INTEGER (count)	IN
array_of_displacements	displacement for each block, in multiples of oldtype	const int[] MPI_Count[]	INTEGER (count)	IN
oldtype	old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype	new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

Python:

```
MPI.Datatype.Create_indexed(self, blocklengths,displacements )
```

6.3.5 Distributedarray 类型

每个维度可以独立地分布为 MPI_DISTRIBUTE_BLOCK, MPI_DISTRIBUTE_CYCLIC,MPI_DISTRIBUTE_NONE,

With the cyclic distribution, the amount of cyclicity can be indicated by setting *dargs*[*id*] to a certain number.

对于块分布，块可以在 *dargs*[*id*] 中显式设置，但 MPI_DISTRIBUTE_DFLT_DARG 会导致找到一个均匀分布。

排序可以是 MPI_ORDER_C 或 MPI_ORDER_FORTRAN .

6.3.6 Indexed type

索引数据类型，由 MPI_Type_indexed (图 6.12) 构造，可以从单一数据类型的数组中发送任意位置的元素。您需要提供一个索引位置数组，以及一个块长度数组，每个索引对应一个单独的块长度。发送的元素总数是所有块长度的总和。

以下示例选择位于素数索引位置的项。

```
// indexed.c
displacements = (int*) malloc(count*sizeof(int));
blocklengths = (int*) malloc(count*sizeof(int));
source = (int*) malloc(totalcount*sizeof(int));
target = (int*) malloc(targetbuffersize*sizeof(int));
```

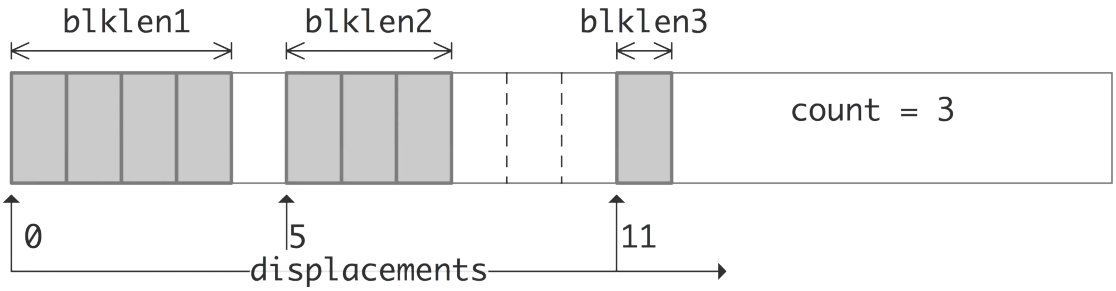


图 6.6：MPI Indexed 数据类型的元素

```
MPI_Datatype newvectortype;
if (procno==sender) {
  MPI_Type_indexed(count,blocklengths,displacements,MPI_INT,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,targetbuffersize,MPI_INT,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_INT,&recv_count);
  ASSERT(recv_count==count);
}
```

对于 Fortran，我们展示一次传统语法：

```
!! indexed.F90integer :: newvectortype;ALLOCATE(indices(count))
ALLOCATE(blocklengths(count))ALLOCATE(source(totalcount))
ALLOCATE(targt(count))if (mytid==sender) then
call MPI_Type_indexed(count,blocklengths,indices,MPI_INT,&
newvectortype,err)call MPI_Type_commit(newvectortype,err)
call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
call MPI_Type_free(newvectortype,err)
else if (mytid==receiver) then
call MPI_Recv(targt,count,MPI_INT,sender,0,comm,&recv_status,
err)call MPI_Get_count(recv_status,MPI_INT,recv_count,err)
!    ASSERT(recv_count==count);end if
```

```
## indexed.py
displacements = np.empty(count,dtype=int)
blocklengths = np.empty(count,dtype=int)
source = np.empty(totalcount,dtype=np.float64)
```

```
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newindextype = MPI.DOUBLE.Create_indexed(blocklengths,displacements)
    newindextype.Commit()
    comm.Send([source,1,newindextype],dest=the_other)
    newindextype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

MPLnote 50: Indexed type. 在 MPL 中， `indexed_layout` 基于表示块长度 / 块位置的二元组向量。

```
// indexed.cxx
const int count = 5;
mpl::contiguous_layout<int>
  fiveints(count);
mpl::indexed_layout<int>
  indexed_where{ { {1,2}, {1,3}, {1,5}, {1,7}, {1,11} } };

if (procno==sender) {
  comm_world.send( source_buffer.data(),indexed_where, receiver );
} else if (procno==receiver) {
  auto recv_status =
    comm_world.recv( target_buffer.data(),fiveints, sender );
  int recv_count = recv_status.get_count<int>();
  assert(recv_count==count);
}
```

MPL 注释 *51*：*gatherv* 的布局。对于 `MPI_Gatherv` / `MPI_Alltoallv` 的大小 / 位移数组通过 `layouts` 对象处理，该对象基本上是 `layout` 对象的向量。

```
mpl::layouts<int> receive_layout;
for ( int iproc=0,loc=0; iproc<nprocs; iproc++ ) {
  auto siz = size_buffer.at(iproc);
  receive_layout.push_back
        ( mpl::indexed_layout<int>( {{ siz,loc }} ) );
  loc += siz;
}
```

MPL 注释 *52*：索引块类型。对于所有块长度相同的情况，使用 `indexed_block_layout`：

```
// indexedblock.cxx
mpl::indexed_block_layout<int>
  indexed_where(  1, {2,3,5,7,11} );
comm_world.send( source_buffer.data(),indexed_where, receiver );
```

你也可以 `MPI_Type_create_hindexed` 它描述了单个旧类型的块，但索引位置以字节为单位，而不是旧类型的倍数。

```
int MPI_Type_create_hindexed
 (int count, int blocklens[],
    MPI_Aint indices[],MPI_Datatype old_type,MPI_Datatype *newtype)
```

图 6.13 MPI_Type_create_hindexed_block

名称 参数名	说明	C 类型	F 类型	inout
MPI_Type_create_hindexed_block (				
MPI_Type_create_hindexed_block_c (				
count	number of blocks---also number of entries in array_of_displacements	int MPI_Count	INTEGER	IN
blocklength	number of elements in each block	int MPI_Count	INTEGER	IN
array_of_displacements	byte displacement of each block	const MPI_Aint[] MPI_Count[]	INTEGER (KIND=MPI_ADDRESS_KIND) (count)	IN
oldtype	old datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
newtype	new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

图 6.14 MPI_Get_address

Name Param name	说明	C 类型	F 类型	输入输出
MPI_Get_address (				
location	location in caller memory	const void*	TYPE(*), DIMENSION(..)	IN
address	address of location	MPI_Aint*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
)				

一个稍微简单一点的版本， MPI_Type_create_hindexed_block (图 6.13) 假设块长度是常数。

T这里是 hindexed 和上面之间的一个重要区别 MPI_Type_indexed: that one described offsets froma base location; these routines describes absolute memory addresses. You can use thisto send f or instance the elements of a linked list. Youwould traverse the list, recording the addressesofthe elements with MPI_Get_address (图 6.14)。 (例程 MPI_Address 已被弃用。)

在 C++ 中，如果组件类型是指针，你可以使用它来发送一个 *std::<vector>*，即来自 *C++* 标准库 的向量对象。

6.3.7 Struct type

结构类型由 MPI_Type_create_struct (图 6.15) 创建，可以包含多种数据类型。（例程 MPI_Type_struct 在 MPI-3 中已被弃用。）规范包含一个 ‘count’ 参数，用于指定单个结构中有多少个块。例如，

```
struct {
int i;
float x,y;
```

图 6.15 MPI_Type_create_struct

Name	Param name	Explanation	C type	F type	inout
MPI_Type_create_struct (					
MPI_Type_create_struct_c (					
	count	number of blocks---also number of entries in arrays array_of_types, array_of_displacements, and array_of_blocklengths	[int MPI_Count	INTEGER	IN
	array_of_blocklengths	number of elements in each block	[const int[] MPI_Count[]	INTEGER (count)	IN
	array_of_displacements	byte displacement of each block	[const MPI_Aint[] MPI_Count[]	INTEGER (KIND=MPI_ADDRESS_KIND) (count)	IN
	array_of_types	type of elements in each block	const MPI_Datatype[]	TYPE (MPI_Datatype) (count)	IN
	newtype	new datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
	)				

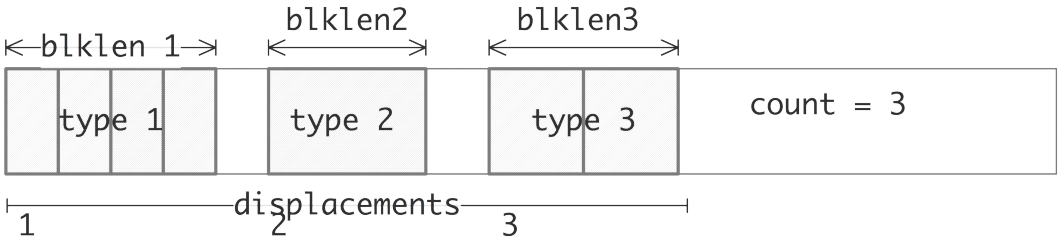


Figure 6.7: MPI Struct 数据类型的元素

```
} point;
```

有两个块，一个是单个整数，另一个是两个浮点数。这在图 6.7 中有所说明。

count The number of blocks in this datatype. The blocklengths, displacements, types arguments have to be at least of this length.

blocklengths 数组，包含每个数据类型块的长度。**displacements** 数组，描述每个数据类型块的相对位置。**types** 数组，包含数据类型；新类型中的每个块都是单一数据类型；可以有多个由相同类型组成的块。

在这个例子中，与之前的不同，发送方和接收方都创建结构类型。使用结构时，不再可能以派生类型发送并以简单类型数组接收。（只要它们具有相同的 *datatypesignature*，就可以以一种结构类型发送并以另一种结构类型接收。）

```
// struct.c
struct object {
  char c;
  double x[2];
  int i;
};
MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen]; MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];

/*
 * where are the components relative to the structure?
 */
MPI_Aint current_displacement=0;

// one character
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;

// two doubles
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x) - (size_t)&myobject;

// one int
blocklengths[2] = 1; types[2] = MPI_INT;
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;

MPI_Type_create_struct(structlen,blocklengths,displacements,types,&newstructuretype);
MPI_Type_commit(&newstructuretype);
if (procno==sender) {
  MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
} else if (procno==receiver) {
  MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,MPI_STATUS_IGNORE);
}
MPI_Type_free(&newstructuretype);
```

注意本例中的 `displacement` 计算，涉及一些不太优雅的指针算术。以下 Fortran 代码使用了 `MPI_Get_address`，这更优雅，实际上也是 Fortran 中进行地址计算的唯一方式。

```
!! struct.F90
  Type object
     character :: c
     real*8,dimension(2) :: x
     integer :: i
  end type object
  type(object) :: myobject
  integer,parameter :: structlen = 3
  type(MPI_Datatype) :: newstructuretype
  integer,dimension(structlen) :: blocklengths
  type(MPI_Datatype),dimension(structlen) :: types;
  MPI_Aint,dimension(structlen) :: displacements
```

```
MPI_Aint :: base_displacement, next_displacement
if (procno==sender) then
   myobject%c = 'x'
   myobject%x(0) = 2.7; myobject%x(1) = 1.5
   myobject%i = 37

!! component 1: one character
blocklengths(1) = 1; types(1) = MPI_CHAR
call MPI_Get_address(myobject,base_displacement)
call MPI_Get_address(myobject%c,next_displacement)
displacements(1) = next_displacement-base_displacement

!! component 2: two doubles
blocklengths(2) = 2; types(2) = MPI_DOUBLE
call MPI_Get_address(myobject%x,next_displacement)
displacements(2) = next_displacement-base_displacement

!! component 3: one int
blocklengths(3) = 1; types(3) = MPI_INT
call MPI_Get_address(myobject%i,next_displacement)
displacements(3) = next_displacement-base_displacement

if (procno==sender) then
   call MPI_Send(myobject,1,newstructuretype,receiver,0,comm)
else if (procno==receiver) then
   call MPI_Recv(myobject,1,newstructuretype,sender,0,comm,MPI_STATUS_IGNORE)
end if
call MPI_Type_free(newstructuretype)
```

这样写是不正确的

```
displacement[0] = 0;
displacement[1] = displacement[0] + sizeof(char);
```

因为你不知道 编译器 如何在内存中布局结构体 [1]

.

如果你想发送多个结构体，你必须更多地关注结构体中的填充。你可以通过添加一个额外的类型 `MPI_UB` 来解决结构体的 “ 上界 ” 问题：

```
displacements[3] = sizeof(myobject); types[3] = MPI_UB;
MPI_Type_create_struct(struclen+1,.....);
```

MPL 注释 *53*：结构体类型标量。可以将 MPI 结构体类型描述为一组位移，这些位移应用于任何符合规范的项目集合。另一方面，MPL `heterogeneous_layout` 则包含实际数据。因此你可以写成

```
// structscalar.cxx
char c; double x; int i;
if (procno==sender) {
  c = 'x'; x = 2.4; i = 37; }
mpl::heterogeneous_layout object( c,x,i );
if (procno==sender)
```

1. 作业问题：语言标准对此有什么规定?

```
  comm_world.send( mpl::absolute,object,receiver );
else if (procno==receiver)
  comm_world.recv( mpl::absolute,object,sender );
```

这里， `absolute` 表示缺少隐式缓冲区：布局是绝对的，而不是相对描述。

MPL note 54: Struct type general. 比标量更复杂的数据 t akes more work:

```
// struct.cxxchar c; vector<double> x(2); int i;
if (procno==sender) {c = 'x'; x[0] = 2.7; x[1] = 1.5; i = 37; }
mpl::heterogeneous_layout object( c,
mpl::make_absolute(x.data(),mpl::vector_layout<double>(2)),
i );if (procno==sender) {
comm_world.send( mpl::absolute,object,receiver );
} else if (procno==receiver) {
comm_world.recv( mpl::absolute,object,sender );}
```

注意除了上述提到的 `absolute` 之外的 `make_absolute` 。

6.4 大数据类型

TMPI 发送和接收调用中的 `size` 参数是整数类型，这意味着它的最大值是（平台 - 相关的，但通常是：） $2^{31}-1$。如今的计算机已经足够强大，这成为了一个限制。根据 MPI-4 标准，这个问题已经通过允许更大计数参数类型 `MPI_Count` 得到解决。其实现方式在某种程度上依赖于语言。

以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

MPL 注释 *55*：大计数。

6.4.1 C

对于每个例程，例如 `MPI_Send` 带有一个整数计数，存在一个对应的 `MPI_Send_c` 带有一个 countoftype `MPI_Count`。

```
MPI_Count buffersize = 1000;
double *indata,*outdata;
indata = (double*) malloc( buffersize*sizeof(double) );
outdata = (double*) malloc( buffersize*sizeof(double) );
MPI_Allreduce_c(indata,outdata,buffersize,
                MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
```

代码：

```
// pingpongbig.c
assert( sizeof(MPI_Count)>4 );
for ( int power=3; power<=10; power++) {
MPI_Count length=pow(10,power);
buffer = (double*)malloc(
↪length*sizeof(double) );MPI_Ssend_c
(buffer,length,MPI_DOUBLE,processB,0,
comm);MPI_Recv_c
(buffer,length,MPI_DOUBLE,

         processB,0,comm,MPI_STATUS_IGNORE);
```

输出：

```
make[3]: `pingpongbig' is up to date.
Ping-pong between ranks 0--1, repeated 10
↪timesMPI Count has 8 bytesSize: 10^3,
 (repeats=10000)
Time 1.399211e-05 for size 10^3: 1.1435
↪Gb/secSize: 10^4, (repeats=10000)
Time 4.077882e-05 for size 10^4: 3.9236
↪Gb/secSize: 10^5, (repeats=1000)
Time 1.532863e-04 for size 10^5: 10.4380
↪Gb/secSize: 10^6, (repeats=1000)
Time 1.418844e-03 for size 10^6: 11.2768
↪Gb/secSize: 10^7, (repeats=100)
Time 1.443470e-02 for size 10^7: 11.0844
↪Gb/secSize: 10^8, (repeats=100)
Time 1.540918e-01 for size 10^8: 10.3834
↪Gb/secSize: 10^9, (repeats=10)
Time 1.813220e+00 for size 10^9: 8.8241
↪Gb/secSize: 10^10, (repeats=10)
Time 1.846741e+01 for size 10^10: 8.6639
↪Gb/sec
```

6.4.2 Fortran

count 参数可以声明为

```
use mpi_f08
Integer(kind=MPI_COUNT_KIND) :: count
```

由于 Fortran 支持多态性，可以使用相同的例程名称。

The legit way of coding:

```
!! typecheckkind.F90
  integer(8) :: source,n=1
  call MPI_Init()
  call MPI_Send(source,n,MPI_INTEGER8, &
       1,0,MPI_COMM_WORLD)
```

… 但你可以看到底层是什么：

```
!! typecheck8.F90
  integer(8) :: source,n=1
  call MPI_Init()
  call MPI_Send(source,n,MPI_INTEGER8, &
       1,0,MPI_COMM_WORLD)
```

除非使用 `mpi_f08` 模块，否则无法使用此类型的例程。*MPI-4* 材料结束

```
!! pingpongbig.F90
  integer :: power,countbytes
  Integer(KIND=MPI_COUNT_KIND) :: length
  call MPI_Sizeof(length,countbytes,ierr)
  if (procno==0) &
       print *,"Bytes in count:",countbytes
        length = 10**power
        allocate( senddata(length),recvdata(length) )
           call MPI_Send(senddata,length,MPI_DOUBLE_PRECISION, &
                processB,0, comm)
           call MPI_Recv(recvdata,length,MPI_DOUBLE_PRECISION, &
                processB,0, comm,MPI_STATUS_IGNORE)
```

6.4.3 Count datatype

该 `MPI_Count` 数据类型被定义为足够大以容纳

- 普通的 4 字节整数类型；
- 该 `MPI_Aint` 类型，章节 6.2.4 和 6.2.4；
- the `MPI_Offset` 类型，第 10.2.2 节。

C/C `size_t` 中的类型定义为足够大以包含 `++` 的输出，换句话说，足够大以测量任何对象。 `sizeof`

6.4.4 MPI 3 temporarysolution

使用派生类型已经可以发送大消息：要发送一个包含 10 个 40 元素的大数据类型，你可以

- 创建一个包含 10_{20} 个元素的连续类型，且
- 发送该类型的 10_{20} 个元素。

这通常有效，但并不完美。例如，例程 `MPI_Get_elements` 返回发送的基本元素总数（而不是 `MPI_Get_count` 它会返回派生类型元素的数量）。由于其输出参数是整数类型，无法存储正确的值。

MPI-3 标准通过引入 `MPI_Count` datatype 和带有 `_x` 扩展的新例程解决了这个问题，这些例程返回该类型的计数。以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

鉴于 “ 扩展 ” 的例程，这个解决方案不再需要，并且从 MPI-4.1 起被弃用。*MPI-4* 内容结束

Let us consider an example.

分配超过 4Gbyte 的缓冲区并不难 :

```
// vectorx.c
float *source=NULL,*target=NULL;
int mediumsize = 1<<30;
int nblocks = 8;
size_t datasize = (size_t)mediumsize * nblocks * sizeof(float);
if (procno==sender) {
  source = (float*) malloc(datasize);
```

我们使用发送派生类型元素的技巧：

```
MPI_Datatype blocktype;
MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
MPI_Type_commit(&blocktype);
if (procno==sender) {
  MPI_Send(source,nblocks,blocktype,receiver,0,comm);
```

我们对接收调用也使用相同的技巧，但现在我们捕获状态参数，该参数稍后会告诉我们发送了多少个基本类型的元素：

```
} else if (procno==receiver) {
  MPI_Status recv_status;
  MPI_Recv(target,nblocks,blocktype,sender,0,comm,
    &recv_status);
```

当我们查询缓冲区中有多少基本元素时（记住在接收调用中缓冲区长度是接收元素数量的上限），我们是否需要一个比整数更大的计数器。MPI 为此引入了一种类型 `MPI_Count`，以及新的例程如 `MPI_Get_elements_x`(图 4.14) 返回这种类型的计数：

```
MPI_Count recv_count;
MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```

Remark 18 计算一个大数进行分配并非完全简单。

```
// getx.c
int gig = 1<<30;
int nblocks = 8;
size_t big1 = gig * nblocks * sizeof(double);
size_t big2 = (size_t)1 * gig * nblocks * sizeof(double);
size_t big3 = (size_t) gig * nblocks * sizeof(double);
size_t big4 = gig * nblocks * (size_t) ( sizeof(double) );
size_t big5 = sizeof(double) * gig * nblocks;
;
```

输出为：

```
size of size_t = 8
0 68719476736 68719476736 0 68719476736
```

显然，不仅操作是从左到右进行，类型转换也是这样：只有当一个操作数是 `size_t` 时，计算的子表达式才会被转换为 `size_t`。

上面，我们实际上并没有创建一个大于 2G 的数据类型，但如果你这样做了，可以通过 `MPI_Type_get_extent_x` (图 6.17) 和 `MPI_Type_get_true_extent_x` (图 6.17) 查询其范围。

Python 注释 *23*：大数据。由于 python 具有无限大小的整数，因此不需要例程的 “x” 变体。在内部，`MPI.Status.Get_elements` 是基于 `MPI_Get_elements_x` 实现的。

图 6.16 MPI_Type_get_extent

名称 参数名 说明		C 类型	F 类型	输入输出
MPI_Type_get_extent (				
MPI_Type_get_extent_c (				
datatype	datatype to get information on	MPI_Datatype	TYPE (MPI_Datatype)	IN
lb	lower bound of datatype	MPI_Aint* / MPI_Count*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
extent	extent of datatype	MPI_Aint* / MPI_Count*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
)				

6.5 Typemaps 和类型匹配

对于派生类型，你已经看到发送方和接收方的类型不必完全匹配。然而，当构造发送缓冲区和解包接收缓冲区时，缓冲区中的连续类型必须匹配。

发送和接收缓冲区中的类型也需要与底层架构的数据类型匹配，有两个例外。 MPI_PACKED 和 MPI_BYTE 类型可以匹配任何底层类型。然而，这仍然不意味着只在发送方或接收方使用这些类型，而另一方使用特定类型是一个好主意。

6.6 类型范围

关于类型大小的相关问题，参见第 6.2.5 节。

6.6.1 范围和真实范围

T数据类型范围，用 MPI_Type_get_extent (图 6.16) 测量，严格来说是从第一个
t到该类型的最后一个数据项的距离，也就是包括类型中的间隙。它以字节为单位测量，因此
o输出参数的类型是 MPI_Aint。

在以下示例中（另见图 6.8 ）我们测量了向量类型的 extent。注意 extent 并不是步长乘以块数，因为那样会计算一个 “ 尾部间隙 ” 。

```
MPI_Aint lb,asize;
MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Type_get_extent(newtype,&lb,&asize);ASSERT( lb==0 );
ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
MPI_Type_free(&newtype);
```

类似地，使用 MPI_Type_get_extent 计算 struct 中由 对齐 问题引起的间隙。

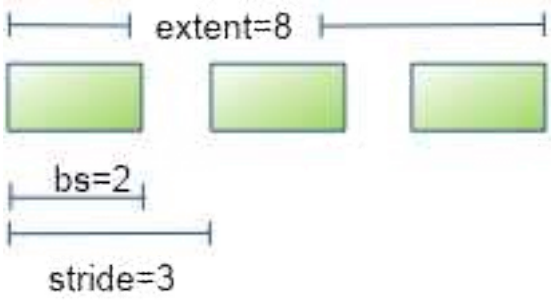


图 6.8：向量数据类型的范围

```
size_t size_of_struct = sizeof(struct object);MPI_Aint typesize,
typelb;MPI_Type_get_extent(newstructuretype,&typelb,&typesize);
assert( typesize==size_of_struct );
```

参见章节 6.3.7 中定义结构类型的代码。

备注 19 例程 *MPI_Type_get_extent* 替代了已弃用的函数 *MPI_Type_extent*, *MPI_Type_lb*,*MPI_Type_ub*。

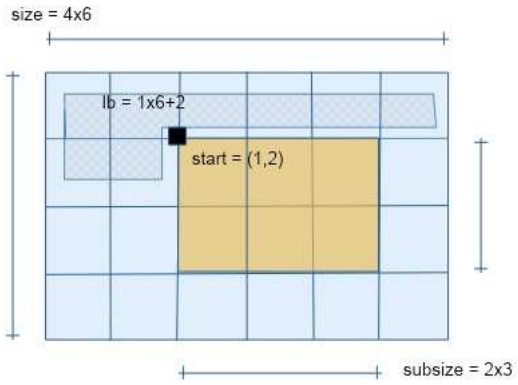


图 6.9：子数组数据类型的真实下界和范围

子数组数据类型 不必从缓冲区的第一个元素开始，因此 *extent* 是对涉及数据量的夸大。实际上，下界是零，*extent* 等于取子数组的块的大小。例程 MPI_Type_get_true_extent (图 6.17) 返回下界，指示数据的起始位置，以及从该点开始的 extent。这在图 6.9 中有所说明。

图 6.17 MPI_Type_get_true_extent

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Type_get_true_extent (					
MPI_Type_get_true_extent_c (					
	datatype	datatype to get information on	MPI_Datatype	TYPE (MPI_Datatype)	IN
	true_lb	true lower bound of datatype	MPI_Aint* / MPI_Count*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
	true_extent	true extent of datatype	MPI_Aint* / MPI_Count*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
)					

代码：

```
// trueextent.c
int sender = 0, receiver = 1, the_other = 1-procno;
int sizes[2] = {4,6},subsizes[2] = {2,3},starts[2] = {1,2};
  MPI_Datatype subarraytype;
  MPI_Type_create_subarray
    (2,sizes,subsizes,starts,
     MPI_ORDER_C,MPI_DOUBLE,&subarraytype);
  MPI_Type_commit(&subarraytype);

  MPI_Aint true_lb,true_extent,extent;
   MPI_Type_get_true_extent
    (subarraytype,&true_lb,&true_extent);
  MPI_Aint
    comp_lb = sizeof(double) *
        ( starts[0]*sizes[1]+starts[1] ),
    comp_extent = sizeof(double) *
        ( sizes[1]-starts[1] // first row
          + starts[1]+subsizes[1] // last row
          + ( subsizes[0]>1 ? subsizes[0]-2 : 0 )*sizes[1]
    ↪);
  ASSERT(true_lb==comp_lb);
  ASSERT(true_extent==comp_extent);
  MPI_Send(source,1,subarraytype,the_other,0,comm);
  MPI_Type_free(&subarraytype);
```

输出：

```
In basic array of 192 bytes
find sub array of 48 bytes
Found lb=64, extent=72
Computing lb=64 extent=72
Non-true lb=0, extent=192,
    ↪computed=192
Finished
received:  8.500  9.500
    ↪10.500 14.500 15.500
    ↪16.500
1,2
1,3
1,4
2,2
2,3
2,4
```

还有 “大数据” 例程 MPI_Type_get_extent_x MPI_Type_get_true_extent_x 其输出为 MPI_Count。

以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

C 例程 MPI_Type_get_extent_c MPI_Type_get_true_extent_c 也输出一个 MPI_Count。*MPI-4* 材料结束

图 6.18 MPI_Type_create_resized

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Type_create_resized (				
MPI_Type_create_resized_c (				
oldtype	input datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
lb	new lower bound of datatype	MPI_Aint / MPI_Count	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
extent	new extent of datatype	MPI_Aint / MPI_Count	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
newtype	output datatype	MPI_Datatype*	TYPE (MPI_Datatype)	OUT
)				

6.6.2 范围调整

一个类型部分由其下界和范围（extent）来表征，或者等价地由下界和上界来表征。有点神奇的是，你实际上可以改变这些来实现特殊效果。这在以下情况下是需要的:

- 某些 gather/scatter 操作的情况；参见第 6.6.2.2 节中的示例。
- 当缓冲区中派生项的数量超过一个时。参见第 6.6.2.1 节中的示例。

解决方案所依赖的技术点是你可以用 MPI_Type_create_resized(图 6.18) 对类型进行 ‘重设大小’，从而赋予它不同的范围，同时不影响其中实际包含的数据量。

我们可以如下描述数据类型所占用的空间（‘真实范围’）和 ‘范围’。如果发送计数大于 1，或者你散布某种数据类型:

1. 指针设置在第一个数据项；然后 2. 对于要发送的数据

类型的每个实例：(a) 按数据类型描述发送数据；(b) 指针

按数据类型的范围前进

(接收和聚集的行为类似，但数据流向相反。)

6.6.2.1 Example 1

在迄今为止的派生类型示例中，我们总是使用发送计数为 1。如果使用更大的计数会发生什么?

考虑一个向量类型，发送计数为 2。

```
MPI_Type_vector( count,bs,stride,oldtype,&one_n_type );
MPI_Type_contiguous( 2,&one_n_type,&two_n_type );
```

Contrast this with a twice-as-large vector type: It is clear that

```
MPI_Type_vector( 2*count,bs,stride,oldtype,&two_n_type );
```

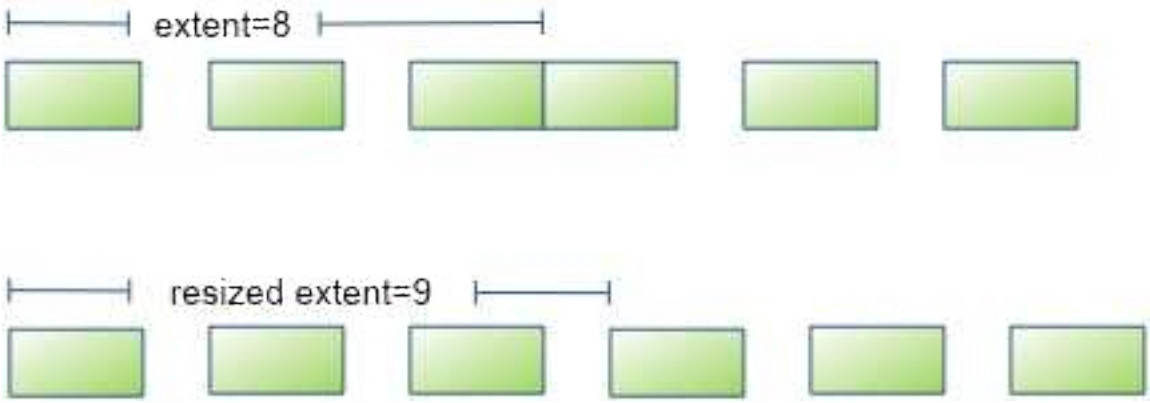


图 6.10：两个向量的连续类型，调整范围大小前后。

差异如图 6.10 所示，上面展示了使用发送计数为 2 的结果，下面展示了期望的效果。

为了展示如何通过调整范围大小来解决这个问题，我们来看一个具体的例子，考虑从包含连续整数的缓冲区发送多个派生类型：

```
// vectorpadsend.c
for (int i=0; i<max_elements; i++) sendbuffer[i] = i;
MPI_Type_vector(count,blocklength,stride,MPI_INT,&stridetype);
MPI_Type_commit(&stridetype);
MPI_Send( sendbuffer,ntypes,stridetype, receiver,0, comm );
```

我们接收进一个连续缓冲区：

```
MPI_Recv( recvbuffer,max_elements,MPI_INT, sender,0, comm,&status );
int count; MPI_Get_count(&status,MPI_INT,&count);
printf("Receive %d elements:",count);
for (int i=0; i<count; i++) printf(" %d",recvbuffer[i]);
printf("\n");
```

输出为：

```
Receive 6 elements: 0 2 4 5 7 9
```

接下来，我们调整类型大小以添加末尾的间隙。此过程如图 6.1 0.

Resizing the type looks like:

```
MPI_Type_get_extent(stridetype,&l,&e);
printf("Stride type l=%ld e=%ld\n",l,e);
e += ( stride-blocklength) * sizeof(int);
MPI_Type_create_resized(stridetype,l,e,&paddedtype);
MPI_Type_get_extent(paddedtype,&l,&e);
printf("Padded type l=%ld e=%ld\n",l,e);
MPI_Type_commit(&paddedtype);
MPI_Send( sendbuffer,ntypes,paddedtype, receiver,0, comm );
```

以及相应的输出，包括查询范围，内容如下：

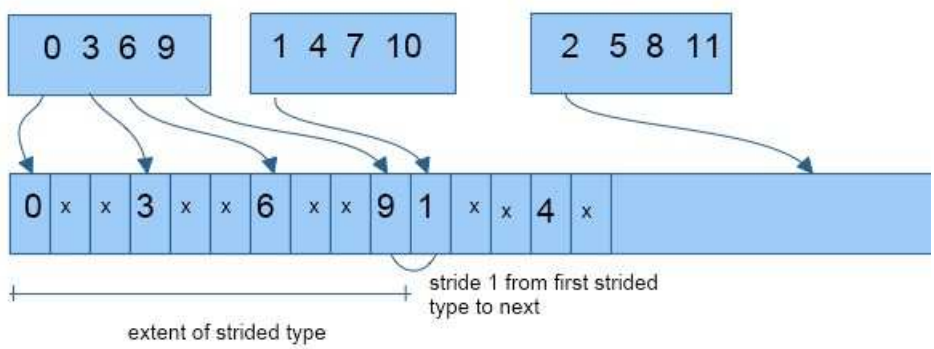


图 6.11：聚集分步类型的布局。

```
Strided type l=0 e=20
Padded  type l=0 e=24
Receive 6 elements: 0 2 4 6 8 10
```

关于范围例程，我们有两个观察：

1. 下界和范围参数的类型是 `MPI_Aint`，这意味着它们以字节为单位测量；2. 下界通常保持不变：本书中没有给出任何更改下界的例子。

6.6.2.2 示例 2

另一个例子，我们回顾练习 6.4（和图 6.5）中，每个进程创建一个整数缓冲区，这些整数将在 gather 调用中交错：分散的数据是通过单独的事务发送的。是否有可能在一次 gather 或 scatter 调用中处理所有这些交错的数据包？

这里的问题是 MPI 在 scatter 中使用发送类型的 extent，或者在 gather 中使用接收类型的 extent：如果该类型从第一个元素到最后一个元素的大小是 20 字节，那么在 scatter 中数据将以 20 字节间隔读取，或者在 gather 中以 20 字节间隔写入。这忽略了类型中的“间隙”！（参见练习 6.4。）

```
int *mydata = (int*) malloc( localsize*sizeof(int) );
for (int i=0; i<localsize; i++)
  mydata[i] = i*nprocs+procno;
MPI_Gather( mydata,localsize,MPI_INT,
    /* rest to be determined */ );
```

一个普通的 gather 调用当然不会交错，而是将数据首尾相接地放置：

```
MPI_Gather( mydata,localsize,MPI_INT,
            gathered,localsize,MPI_INT, // abutting
            root,comm );
```

```
gather 4 elements from 3 procs:0
3 6 9 1 4 7 10 2 5 8 11
```

使用 strided 类型仍然是将数据首尾相接，但现在在 gather 缓冲区中存在未写入的间隙：

```
MPI_Gather( mydata,localsize,MPI_INT,
            gathered,1,stridetype, // abut with gaps
            root,comm );
```

This is illustrated in figure 6.11. A sample printout of the result would be:

0 1879048192 1100361260 3 3 0 6 0 0 9 1 198654

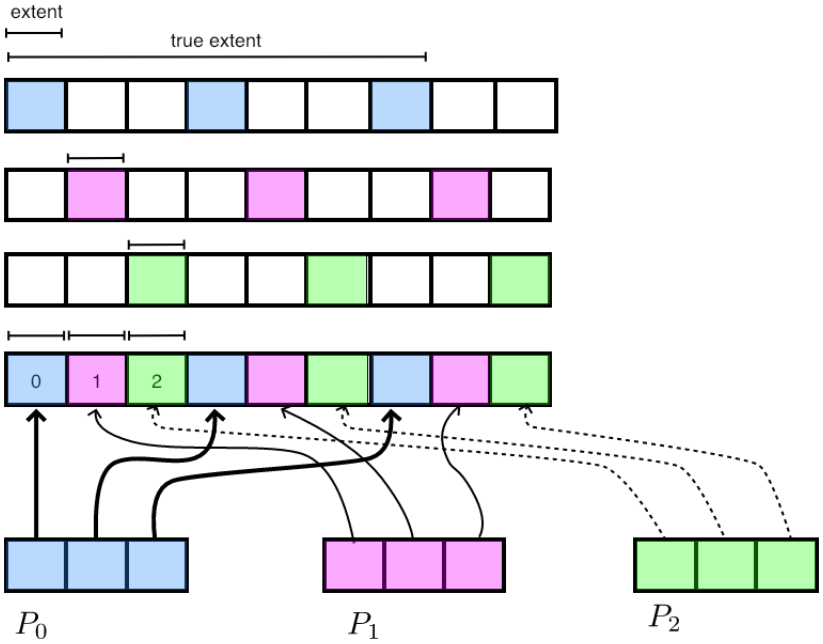


图 6.12：来自具有调整大小范围数据的交错 gather

诀窍是使用 `MPI_Type_create_resized` 使类型的范围仅为一个 int 长度：

```
// interleavegather.c
MPI_Datatype interleavetype;
MPI_Type_create_resized(stridetype,0,sizeof(int),&interleavetype);
MPI_Type_commit(&interleavetype);
MPI_Gather( mydata,localsize,MPI_INT,
              gathered,1,interleavetype, // shrunk extent
              root,comm );
```

现在数据以相同的步幅写入，但起始点等于缩小后的范围：

```
0 1 2 3 4 5 6 7 8 9 10 11
```

这在图 6.12 中说明。

Fortrannote12: 范围是 *Aint*。下界和范围参数的类型是 `Integer(kind=MPI_Address_kind)`：

```
!! stridescatter.F90
```

```
integer(kind=MPI_Address_kind) :: l,e
call MPI_Type_get_extent(scattertype,l,e)e = c_sizeof(i)
call MPI_Type_create_resized(scattertype,l,e,interleavetype)
call MPI_Type_commit(interleavetype)
```

练习 6.7. 重写练习 6.4，使用 gather 而不是单独的消息。

MPL 注释 *56*：范围调整。调整数据类型大小不会生成新类型，而是在 “ 原地 ” 进行调整：

```
void layout::resize(ssize_t lb, ssize_t extent);
```

6.6.2.3 示例：动态向量

在向量类型中必须明确指定有多少个块，这会让你（有点）困扰吗？如果你能创建一个 “ 带填充的块 ”，然后发送任意数量的这些块，那就好了。

嗯，你可以通过调整类型大小来引入填充，使其稍微大一些。

```
// stridestretch.cMPI_Datatype oneblock;
MPI_Type_vector(1,1,stride,MPI_DOUBLE,&oneblock);
MPI_Type_commit(&oneblock);MPI_Aint block_lb,block_x;
MPI_Type_get_extent(oneblock,&block_lb,&block_x);
printf("One block has extent: %ld\n",block_x);MPI_Datatype paddedblock;
MPI_Type_create_resized(oneblock,0,stride*sizeof(double),&paddedblock);
MPI_Type_commit(&paddedblock);
MPI_Type_get_extent(paddedblock,&block_lb,&block_x);
printf("Padded block has extent: %ld\n",block_x);
// now send a bunch of these padded blocks
MPI_Send(source,count,paddedblock,the_other,0,comm);
```

对此问题还有第二种解决方案，使用结构类型。这种方法不使用重设大小，而是指示一个位移，直到结构的末尾。我们通过在该位移处放置一个类型 `MPI_UB` 来实现：

```
int blens[2]; MPI_Aint displs[2];MPI_Datatype types[2],
 paddedblock;blens[0] = 1; blens[1] = 1;
displs[0] = 0; displs[1] = 2 * sizeof(double);
types[0] = MPI_DOUBLE; types[1] = MPI_UB;
MPI_Type_struct(2, blens, displs, types, &paddedblock);
MPI_Type_commit(&paddedblock);MPI_Status recv_status;
MPI_Recv(target,count,paddedblock,the_other,0,comm,&recv_status);
```

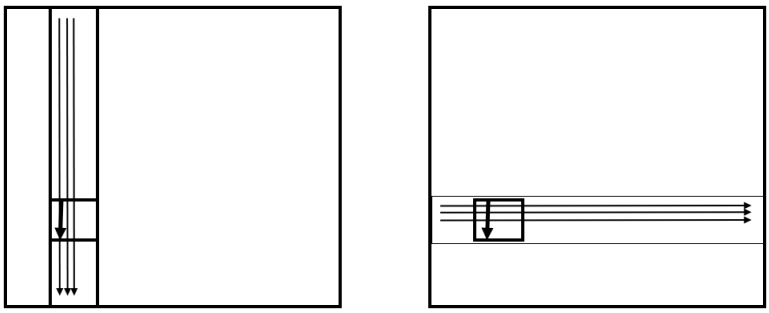


Figure 6.13: Transposing a 1D partitioned array

6.6.2.4 示例：转置

转置数据是 FFT 等操作的重要部分。我们将分步骤进行讲解。参见图 6.13。

The source data can be described as a vector type defined as:

- 有 b 个块，
- 块大小为 b，
- 间隔为数组的全局 i 大小。

```
## transposeblock.py
MPI_Datatype sourceblock;
MPI_Type_vector( blocksize_j,blocksize_i,isize,MPI_INT,&sourceblock);
MPI_Type_commit( &sourceblock);
```

目标类型更难描述。首先我们注意到，源类型中的每个连续块都可以描述为一个具有以下特征的向量类型：

- b 块，
- 大小为 1 每个，
- 以矩阵的全局 j 大小为步长。

```
MPI_Datatype targetcolumn;
MPI_Type_vector( blocksize_i,1,jsize, MPI_INT,&targetcolumn);
MPI_Type_commit( &targetcolumn );
```

对于接收进程的完整类型，我们现在需要将 b 行一起打包。

Exercise6.8. 完成代码。

- `targetcolumn` 类型的跨度是多少？
- 块中第一个元素的间距是多少？因此你如何调整 `targetcolumn` 类型的大小？

6.7 Reconstructing types

可以从一个数据类型中找出它是如何构造的。这使用了例程 `MPI_Type_get_envelope` 和 `MPI_Type_get_contents`。第一个例程返回 *combiner* （其值如

图 6.19 MPI_Pack

Name Param name	Explanation	C type	F type	i nout
MPI_Pack (				
MPI_Pack_c (				
inbuf	input buffer start	const void*	TYPE(*), DIMENSION(..)	IN
incount	number of input data items	int MPI_Count	INTEGER	IN
datatype	datatype of each input data item	MPI_Datatype	TYPE (MPI_Datatype)	IN
outbuf	output buffer start	void*	TYPE(*), DIMENSION(..)	OUT
outsize	output buffer size, in bytes	int MPI_Count	INTEGER	IN
position	current position in buffer, in bytes	int* MPI_Count*	INTEGER	INOUT
comm	communicator for packed message	MPI_Comm	TYPE (MPI_Comm)	IN
)				

MPI_COMBINER_VECTOR) 和参数的数量；然后使用第二个例程来检索实际的参数。

6.8 打包

派生数据类型的原因之一是处理非连续数据。在较旧的通信库中，这只能通过将数据从其原始容器打包到缓冲区中来完成，同样在接收端将其解包到目标数据结构中。

MPI 提供了这种打包功能，部分是为了与此类库兼容，部分也是出于灵活性的考虑。与以原子方式传输数据的派生数据类型不同，打包例程将数据顺序添加到缓冲区，解包时也按顺序取出数据。

这意味着可以打包一个整数，描述其余打包消息中有多少个浮点数。相应地，解包例程可以先检查第一个整数，并根据它解包正确数量的浮点数。

MPI offers the following:

- 该 MPI_Pack 命令将数据添加到发送缓冲区；
- 该 MPI_Unpack 命令从接收缓冲区检索数据；
- 缓冲区以 MPI_PACKED 的数据类型发送。

W使用 MPI_Pack (图 6.19) 可以一次向缓冲区添加一个数据元素。该 position 参数 i由打包例程每次更新。

相反， MPI_Unpack (图 6.20) 一次从缓冲区检索一个元素。您需要指定

图 6.20 MPI_Unpack

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Unpack (MPI_Unpack_c (				
inbuf	input buffer start	const void*	TYPE(*), DIMENSION(..)	IN
insize	size of input buffer, in bytes	[int MPI_Count	INTEGER	IN
position	current position in bytes	[int* MPI_Count*	INTEGER	INOUT
outbuf	output buffer start	void*	TYPE(*), DIMENSION(..)	OUT
outcount	number of items to be unpacked	[int MPI_Count	INTEGER	IN
datatype	datatype of each output data item	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator for packed message	MPI_Comm	TYPE (MPI_Comm)	IN
)				

the MPI datatype.

打包缓冲区以 MPI_PACKED 数据类型发送或接收。发送例程使用 position 参数来指定发送的数据量，但接收例程事先不知道该值，因此必须指定一个上限。

图 6.21 MPI_Pack_size

名称 参数名	说明	C 类型	F 类型	inout
MPI_Pack_size (				
MPI_Pack_size_c (				
incount	count argument to packing call	int MPI_Count	INTEGER	IN
datatype	datatype argument to packing call	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator argument to packing call	MPI_Comm	TYPE (MPI_Comm)	IN
size	upper bound on size of packed message, in bytes	int* MPI_Count*	INTEGER	OUT
)				

Code:

```
if (procno==sender) {position = 0;
MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
for (int i=0; i<nsends; i++) {
double value = rand()/(double)RAND_MAX;
printf("[%d] pack %e\n",procno,value);
MPI_Pack(&value,1,MPI_DOUBLE,buffer,buflen,&position,
comm);}MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,
comm);MPI_Send(buffer,position,MPI_PACKED,other,0,comm);
} else if (procno==receiver) {int irecv_value;
double xrecv_value;
MPI_Recv(buffer,buflen,MPI_PACKED,other,0,comm,
MPI_STATUS_IGNORE);position = 0;
MPI_Unpack(buffer,buflen,&position,&nsends,1,MPI_INT,
comm);for (int i=0; i<nsends; i++) {
MPI_Unpack(buffer,buflen,&position,&xrecv_value,1,
MPI_DOUBLE,comm);
printf("[%d] unpack %e\n",procno,xrecv_value);}
MPI_Unpack(buffer,buflen,&position,&irecv_value,1,MPI_INT,
comm);ASSERT(irecv_value==nsends);}
```

Output:

```
[0] pack 8.401877e-01
[0] pack 3.943829e-01
[0] pack 7.830992e-01
[0] pack 7.984400e-01
[0] pack 9.116474e-01
[0] pack 1.975514e-01
```

你可以用 `MPI_Pack_size` (图 6.21) 预先计算所需缓冲区的大小。

Code:

```
// pack.c
for (int i=1; i<=4; i++) {
  MPI_Pack_size(i,MPI_CHAR,comm,&s);
  printf("%d chars: %d\n",i,s);
}
for (int i=1; i<=4; i++) {
  MPI_Pack_size(i,MPI_UNSIGNED_SHORT,comm,&s);
  printf("%d unsigned shorts: %d\n",i,s);
}
for (int i=1; i<=4; i++) {
  MPI_Pack_size(i,MPI_INT,comm,&s);
  printf("%d ints: %d\n",i,s);
}
```

Output:

```
1 chars: 1
2 chars: 2
3 chars: 3
4 chars: 4
1 unsigned shorts: 2
2 unsigned shorts: 4
3 unsigned shorts: 6
4 unsigned shorts: 8
1 ints: 4
2 ints: 8
3 ints: 12
4 ints: 16
```

练习 6.9. 假设你有一个 “ 数组结构 ”

```
struct aos {
  int length;
  double *reals;
  double *imags;
};
```

包含动态创建的数组。编写代码以发送和接收此结构。

6.9 复习问题

对于所有判断题，如果你回答某个陈述是错误的，请给出一句话的解释。

1. 给出两个 MPI 派生数据类型的例子。描述它们使用了哪些参数?
2. 给出一个实际例子，其中发送方使用的类型与接收方在相应接收调用中使用的类型不同。请指出涉及的类型。

3. 仅限 Fortran。对还是错？(a) 数组索引可以是 d 发送和接收缓冲区数组之间的不同。
   (b) 允许发送数组的部分。
   (c) 你需要先将多维数组 `Reshape` 成线性形状，才能发送它。
   (d) 当一个可分配数组被维度化和分配后，MPI 在用作发送缓冲区时，会将其视为普通的静态数组。

   (e) 如果你将可分配数组用作接收缓冲区，它会被分配：并被填充入接收的数据。

4. 仅 Fortran：当你想用可分配数组作为接收缓冲区时，如何处理这种情况 buffer，但它尚未被分配，且你不知道传入数据的大小?

第 7 章

MPI 主题：Communicators

Communicator 是描述一组进程的对象。在许多应用中，所有进程紧密协作，您只需要的唯一 communicator 是 `MPI_COMM_WORLD`，它描述了您的作业启动的所有进程的组。

本章将介绍创建新的 MPI 进程组的方法：原始 world communicator 的子组。第 8 章讨论了动态进程管理，虽然它不扩展 `MPI_COMM_WORLD` ，但确实扩展了可用进程的集合。该章还讨论了 “ 会话模型 ”，这是构造 communicator 的另一种方式。

7.1 基本 communicators

有三个预定义的 communicators：

- `MPI_COMM_WORLD` 包含所有由 `mpiexec` (或某个相关程序) 一起启动的进程。
- `MPI_COMM_SELF` 是仅包含当前进程的通信器。
- `MPI_COMM_NULL` 是无效的通信器。该值产生于
  – 当通信器被释放时；参见第 7.3 节；– 作为构造通信器的例程的错误返回值；– 对于创建的笛卡尔通信器之外的进程（第 11.1.1 节）；– 在非派生进程查询其父进程时（第 7.6.3 节）。

这些值是常量，尽管不一定是编译时常量。因此，它们不能用于 switch 语句、数组声明或 `constexpr` 求值。

如果你不想重复写 `MPI_COMM_WORLD` ，你可以将该值赋给类型为 `MPI_Comm` 的变量。

示例：

```
// C:
#include <mpi.h>
MPI_Comm comm = MPI_COMM_WORLD;
```

图 7.1 MPI_Comm_dup

Name Param name	Explanation	C type	F type	i nout
MPI_Comm_dup (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
newcomm	copy of comm	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

MPL:

```
Done as part of the copy assignment operator.
```

```
!! Fortran 2008 interface
use mpi_f08
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
```

```
!! Fortran legacy interface
#include <mpif.h>
Integer :: comm = MPI_COMM_WORLD
```

Python note 24: Communicator types.

```
comm = MPI.COMM_WORLD
```

MPL note 57: 预定义通信器。The `environment` namespace hasthe equivalents of `MPI_COMM_WORLD`and `MPI_COMM_SELF`:

```
const communicator& mpl::environment::comm_world();
const communicator& mpl::environment::comm_self();
```

使用 `MPI_COMM_NULL` 的方式有所不同。

MPL 注释 58：原始 *communicator* 句柄。 如果您需要 `MPI_Comm` 包含在 MPL 中的对象 `communicator`，有一个访问函数 *native_handle*。

您可以使用 `MPI_Comm_set_name` 为您的 communicators 命名，这可以提高出现错误时错误信息的质量。

7.2 复制 communicators

使用 `MPI_Comm_dup` (图 7.1) 您可以精确复制一个通信器（参见第 7.2.2 节的应用）。还有一个非阻塞变体 `MPI_Comm_idup` (图 7.2)。

这些调用不会传播信息提示（章节 15.1.1 和 15.1.1.2 ）；要实现这一点，请使用 `MPI_Comm_dup_with_info` 和 `MPI_Comm_idup_with_info`；详见章节 15.1.1.2。

Pythonnote 25: Communicator duplication. 重复的通信器作为复制例程的输出创建：

图 7.2 MPI_Comm_idup

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_idup (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
newcomm	copy of comm	MPI_Comm*	TYPE (MPI_Comm)	OUT
request	communication request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

```
newcomm = comm.Dup()
```

MPL note 59: Communicator duplication. Communicators 可以被复制，但仅限于初始化期间。复制赋值已被删除。因此：

```
// LEGAL:
mpl::communicator init = comm;
// WRONG:
mpl::communicator init;
init = comm;
```

7.2.1 Communicator comparing

你可能会想 “ 完全相同的副本 ” 具体是什么意思。为此，可以将 communicator 视为一个上下文标签，你可以将其附加到发送和接收等操作上。重要的是这个标签，而不是 communicator 中包含了哪些进程。如果发送和接收具有相同的 communicator 上下文，它们 “ 属于同一组 ”。反之，一个 communicator 中的发送不能匹配由 MPI_Comm_dup 复制创建的另一个 communicator 中的接收。

测试两个 communicator 是否真正相同，不仅仅是测试它们是否包含相同的进程。调用 MPI_Comm_compare 会返回 MPI_IDENT，如果两个 communicator 值相同，而不是一个通过复制从另一个派生：

Code:

```
// commcompare.c
int result;
MPI_Comm copy = comm;
MPI_Comm_compare(comm,copy,&result);
printf("assign:    comm==copy: %d \n",
       result==MPI_IDENT);
printf("            congruent: %d \n",
       result==MPI_CONGRUENT);
printf("            not equal: %d \n",
       result==MPI_UNEQUAL);

MPI_Comm_dup(comm,&copy);
MPI_Comm_compare(comm,copy,&result);
printf("duplicate: comm==copy: %d \n",
       result==MPI_IDENT);
printf("            congruent: %d \n",
       result==MPI_CONGRUENT);
printf("            not equal: %d \n",
       result==MPI_UNEQUAL);
```

Output:

```
assign:    comm==copy: 1
            congruent: 0
            not equal: 0
duplicate: comm==copy: 0
            congruent: 1
            not equal: 0
```

实际上不相同的通信器可以是

- 由相同的进程组成，顺序相同，结果为 `MPI_CONGRUENT`；
- 仅由相同的进程组成，但顺序不同，结果为 `MPI_SIMILAR`；
- 不同，结果为 `MPI_UNEQUAL`。

不允许与 `MPI_COMM_NULL` 进行比较。

MPL 注释 *60*：通信器比较。

代码：

```
const mpl::communicator &comm =
mpl::environment::comm_world();
MPI_Comm
world_extract = comm.native_handle(),
world_given = MPI_COMM_WORLD;

int result;
MPI_Comm_compare(world_extract,world_given,&result);
cout << "Compare raw comms: " << "\n"
     << "identical: "
     << (result==MPI_IDENT) << "\n"
     << "congruent: "
     << (result==MPI_CONGRUENT) << "\n"
     << "unequal  : "
     << (result==MPI_UNEQUAL) << "\n";
```

Output:

```
Compare raw comms:
identical: true
congruent: false
unequal  : false
```

7.2.2 用于库使用的通信器复制

复制通信器看似毫无意义，但实际上对于软件库的设计非常有用。想象一下你有一个代码

```
MPI_Isend(...); MPI_Irecv(...);
// library call
MPI_Waitall(...);
```

并假设库中有接收调用。现在库中的接收可能无意中捕获了外部环境中发送的消息。

让我们考虑一个例子。首先，这里是库存储调用程序的 communicator 的代码：

```
// commdupwrong.cxx
class library {private:
MPI_Comm comm;int procno,nprocs,
other;MPI_Request request[2];
public:
library(MPI_Comm incomm) {
comm = incomm;
MPI_Comm_rank(comm,&procno);
other = 1-procno;};
int communication_start();
int communication_end();};
```

这模拟了一个执行简单消息交换的主程序，并且它调用了两次库例程。用户并不知道，库也发出了发送和接收调用，结果它们相互干扰了。

Here

- 主程序执行发送，
- 库调用 function_start 执行一次发送和一次接收；因为接收可以匹配任意发送，它与第一个发送配对；
- 主程序执行接收，这将与库调用的发送配对；
- 主程序和库都执行一次等待调用，并且在两种情况下所有请求都成功完成，只是不是你预期的方式。

为防止这种混淆，库应使用 MPI_Comm_dup 复制外部通信器，并相对于其副本发送所有消息。现在用户代码的消息永远无法到达库软件，因为它们位于不同的通信器上。

```
// commdupright.cxx
class library {private:
MPI_Comm comm;int procno,nprocs,
other;MPI_Request request[2];
public:
library(MPI_Comm incomm) {
MPI_Comm_dup(incomm,&comm);
MPI_Comm_rank(comm,&procno);
```

```
    other = 1-procno;
  };
  ~library() {
    MPI_Comm_free(&comm);
  }
  int communication_start();
  int communication_end();
};
```

注意前面的示例如何在 `MPI_Comm_free` cal 在 C++ 析构函数中执行。

```
## commdup.py
class Library():
    def __init__(self,comm):
        # wrong: self.comm = comm
        self.comm = comm.Dup()
        self.other = self.comm.Get_size()-self.comm.Get_rank()-1
        self.requests = [ None ] * 2
    def __del__(self):
        if self.comm.Get_rank()==0: print(".. freeing communicator")
        self.comm.Free()
    def communication_start(self):
        sendbuf = np.empty(1,dtype=int); sendbuf[0] = 37
        recvbuf = np.empty(1,dtype=int)
        self.requests[0] = self.comm.Isend( sendbuf, dest=other,tag=2 )
        self.requests[1] = self.comm.Irecv( recvbuf, source=other )
    def communication_end(self):
        MPI.Request.Waitall(self.requests)

mylibrary = Library(comm)
my_requests[0] = comm.Isend( sendbuffer,dest=other,tag=1 )
mylibrary.communication_start()
my_requests[1] = comm.Irecv( recvbuffer,source=other )
MPI.Request.Waitall(my_requests,my_status)
mylibrary.communication_end()
```

7.3 Sub-communicators

在许多场景中，你会将一个大型任务分配到所有可用的处理器上。然而，你的任务可能包含两个或更多可以视为独立任务的部分。在这种情况下，将处理器相应地划分为子组是有意义的。

假设你正在运行一个模拟，输入被生成，对其进行计算，计算结果被分析或图形化渲染。你可以考虑将处理器分为三组，分别对应生成、计算和渲染。只要你仅执行发送和接收，这种划分是可行的。然而，如果某一组进程需要执行集体操作，你不希望其他组参与其中。因此，你确实希望这三组彼此独立：你希望它们处于不同的 communicator 中。

为了创建这样的进程子集，MPI 提供了从 MPI_COMM_WORLD(或其他 communicator) 中取出子集并将该子集转换为新的 communicator 的机制。

现在你明白为什么 MPI 集体调用需要一个 communicator 参数。集体操作涉及该 communicator 的所有进程。如果只有 world communicator 存在，就不需要这样的参数，但通过创建包含所有可用进程子集的 communicator，你可以对该子集执行集体操作。

用法如下：

- 你可以使用诸如 MPI_Comm_dup (第 7.2 节)，MPI_Comm_split(第 7.4 节)，MPI_Comm_create (第 7.5 节)，MPI_Intercomm_create(第 7.6 节)，MPI_Comm_spawn(第 8.1 节)；
- 你使用该 communicator 一段时间；
- 并且你在完成后调用 MPI_Comm_free ；这也将通信器变量设置为 MPI_COMM_NULL。一个类似的例程， MPI_Comm_disconnect 等待所有未完成的通信结束。两者都是集合操作。

7.3.1 场景：分布式线性代数

出于可扩展性的原因（参见 HPC 书籍，第 7.2.3 节），矩阵通常应以二维方式分布，即每个进程接收一个既不是完整列块也不是完整行块的子块。这意味着处理器本身至少在逻辑上被组织成一个二维网格。操作随后涉及行或列内的归约或广播。为此，处理器的行或列需要在一个子通信器中。

7.3.2 场景：气候模型

气候模拟代码有几个组件，例如对应陆地、空气、海洋和冰层。你可以想象，每个组件需要一套不同的方程和算法来进行模拟。然后你可以划分你的进程，每个子集模拟气候的一个组件，偶尔与其他组件通信。

7.3.3 场景：快速排序

流行的快速排序算法通过将数据分成两个子集来工作，每个子集可以单独排序。如果你想进行并行排序，可以通过创建两个子通信器来实现，在这些子通信器上排序数据，并递归地创建更多子通信器。

7.3.4 共享内存

在单边通信的上下文中，通信器拆分有一个重要的应用，即根据进程是否访问相同的共享内存区域对进程进行分组；参见第 7.4.1 节。

图 7.3 MPI_Comm_split

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Comm_split (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
color	control of subset assignment	int	INTEGER	IN
key	control of rank assignment	int	INTEGER	IN
newcomm	new communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

7.3.5 进程生成

最后，新创建的通信器不一定总是初始 MPI_COMM_WORLD 的子集。MPI 可以动态生成新进程（参见第 8 章），这些进程在一个 MPI_COMM_WORLD 中启动。此外，还会创建另一个通信器，将旧世界和新世界连接起来，以便您可以与新进程通信。

7.4 拆分通信器

上面我们看到几种场景，划分 MPI_COMM_WORLD 成不相交的子通信器是有意义的。命令 MPI_Comm_split（图 7.3）使用 “color” 来定义这些子通信器：旧通信器中所有具有相同 color 的进程最终会被放入一个新的通信器中。旧通信器仍然存在，因此进程现在有两个不同的上下文来进行通信。

新通信子中进程的排名由 “key” 值决定：在子通信子中，key 值最低的进程被赋予最低的排名，依此类推。大多数情况下，没有理由使用与全局排名不同的相对排名，因此 MPI_Comm_rank 全局通信子的值是一个不错的选择。对于相同 key 值的进程，使用原始通信子的排名来打破平局。因此，指定零作为 key 值也会保留原始的进程顺序。

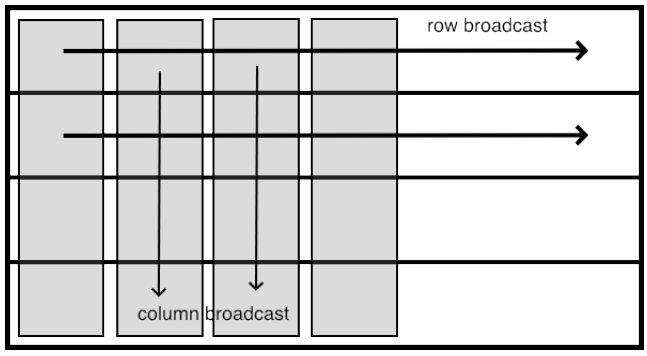


图 7.1：子通信子中的行和列广播

这里是一个 communicator 分割的示例。假设你的处理器排列成一个二维网格：

```
MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;
```

你现在可以为每一列创建一个 communicator：

```
MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, mytid, &column_comm );
```

并在该列中进行广播：

```
MPI_Bcast( data, /* stuff */, column_comm );
```

由于程序的 SPMD 特性，你现在在每个处理器列中并行执行广播操作。这类操作常见于密集线性代数。

练习 7.1. 将你的进程组织成一个网格，并为行和列创建子通信器。为此，计算每个进程的行号和列号。在行和列通信器中，计算 rank。例如，在一个 2×3 处理器网格上，你应该得到：

```
Global ranks:  Ranks in row:  Ranks in colum:0 1 2 0 1 2 0 0 0 3 4 5
0 1 2 1 1 1
```

检查行通信器中的 rank 是否为列号，另一种情况亦然。

反过来也要检查。

在不同数量的进程上运行你的代码，例如行数和列数为 2 的幂，或为素数。（此练习有一个名为 `procgrid` 的骨架代码。）

备注 20 将 *color* 参数设置为 *MPI_UNDEFINED* 的进程，接收的通信器值为 *MPI_COMM_NULL*，也就是说，它不会成为任何创建的子通信器的一部分。

Python 注释 *26*：*Comm split key* 是可选的。 在 Python 中， ‘key’ 参数是可选的：**代码：**

```
## commsplit.py
mydata = procid

# communicator modulo 2
color = procid%2
mod2comm = comm.Split(color)
procid2 = mod2comm.Get_rank()

# communicator modulo 4 recursively
color = procid2 % 2
mod4comm = mod2comm.Split(color)
procid4 = mod4comm.Get_rank()
```

输出：

```
Proc 0 -> 0 -> 0
Proc 2 -> 1 -> 0
Proc 6 -> 3 -> 1
Proc 4 -> 2 -> 1
Proc 3 -> 1 -> 0
Proc 7 -> 3 -> 1
Proc 1 -> 0 -> 0
Proc 5 -> 2 -> 1
```

MPL 注释 *61*：通信器拆分。 在 MPL 中，拆分通信器是作为通信器构造函数的重载之一完成的；

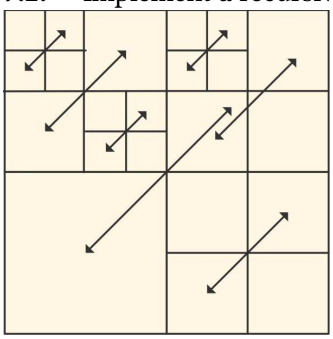
```
// commsplit.cxx
// create sub communicator modulo 2
int color2 = procno % 2;
mpl::communicator comm2
  ( mpl::communicator::split, comm_world, color2 );
auto procno2 = comm2.rank();

// create sub communicator modulo 4 recursively
int color4 = procno2 % 2;
mpl::communicator
  comm4( mpl::communicator::split, comm2, color4 );
auto procno4 = comm4.rank();
```

实现说明： `communicator::split` 标识符是类 `communicator::split_tag` 的一个对象，该类本身是 `communicator` 的一个空子类：

```
class split_tag {};
static constexpr split_tag split{};
```

作为通信器拆分的另一个例子，考虑用于矩阵转置的递归算法。处理器被组织成一个方形网格。矩阵以 2×2 块形式划分。

Exercise 7.2. Implement a recursive algorithm for matrix transposition:



- 交换块 $(1,2)$ 和 $(2,1)$；然后
- 将处理器划分为四个子通信器，并在每个子通信器上递归应用此算法；
    - 如果通信器只有一个进程，则就地转置矩阵。每个进程处理一个元(as素)

7.4.1 按类型拆分

还有一个例程 `MPI_Comm_split_type`（图 7.4）使用类型而非键来拆分通信器。

这里的 `split_type` 参数必须来自以下（简短）列表：

- `MPI_COMM_TYPE_SHARED`：将通信器拆分为共享内存区域的进程子通信器。我们将在第 12.1 节中看到其实际应用。

以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

图 7.4 MPI_Comm_split_type

名称	参数名	说明	C 类型	F 类型	i nout
MPI_Comm_split_type (					
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	split_type	type of processes to be grouped together	int	INTEGER	IN
	key	control of rank assignment	int	INTEGER	IN
	info	info argument	MPI_Info	TYPE (MPI_Info)	INOUT
	newcomm	new communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

Python:

```
MPI.Comm.Split_type(
  self, int split_type, int key=0, Info info=INFO_NULL)
```

- MPI_COMM_TYPE_HW_GUIDED (MPI-4)：使用来自 info 的值进行拆分 MPI_Get_hw_resource_types. 函数 MPI_Get_hw_resource_info (截至 MPI-4.1) 返回一个 MPI_Info 对象，包含可用硬件资源的键 / 值对。（参见第 15.1.1 节，了解如何解包 info 对象。）
- MPI_COMM_TYPE_HW_UNGUIDED (MPI-4)：类似于 MPI_COMM_TYPE_HW_GUIDED，但生成的通信器应严格是原始通信器的子集。在无法满足此条件的进程上，将返回 MPI_COMM_NULL 。
- MPI_COMM_TYPE_RESOURCE_GUIDED(MPI-4.1)：这通过 – 硬件属性拆分通信器。对于这种情况，有信息键 mpi_hw_resource_type。一个可能的键值 mpi_shared_memory 产生与使用拆分类型 MPI_COMM_TYPE_SHARED 相同的拆分效果。– pset 名称。对于这种情况，有信息键 mpi_pset_name。如果通信器不是从会话派生的，拆分后的通信器将是 MPI_COMM_NULL。

MPI-4 材料结束

Remark21 *MPI* 的 OpenMPI 实现有许多非标准的拆分类型，例如 *OMPI_COMM_TYPE_SOCKET* ；参见 *https://www.open-mpi.org/doc/v4.1/man3/MPI_Comm_split_type.3.php*

7.5 通信器和组

你在第 7.4 节中看到，可以派生出包含另一个通信器部分进程的通信器。这里有一个更通用的机制，使用 MPI_Group 对象。

Using groups, it takes the following steps to create a new communicator:

1. 使用 MPI_Comm_group (图 7.5) 访问通信器对象的 MPI_Group 。

图 7.5 MPI_Comm_group

Name Param name	Explanation	C type	F type	i nout
MPI_Comm_group (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
group	group corresponding to comm	MPI_Group*	TYPE (MPI_Group)	OUT
)				

图 7.6 MPI_Comm_create

Name 参数名	说明	C 类型	F type	i nout
MPI_Comm_create (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
group	group, which is a subset of the group of comm	MPI_Group	TYPE (MPI_Group)	IN
newcomm	new communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

2. 使用接下来讨论的各种例程来形成一个新组。注意：即使在不会成为新通信器一部分的进程上，也需要形成该组。3. 使用 MPI_Comm_create (图 7.6)，在旧通信器上进行集合操作，创建一个新的通信器对象。

4. 在不属于子组的秩上，结果通信器的值将是 MPI_COMM_NULL。

还有一个例程 MPI_Comm_create_group，只需要在构成新通信器的组上调用。

7.5.1 进程组

组通过 MPI_Group_incl (图 7.7)、 MPI_Group_excl (图 7.8) 以及其他一些例程进行操作。

```
MPI_Comm_group (comm, group)
MPI_Comm_create (MPI_Comm comm,MPI_Group group, MPI_Comm newcomm)
```

```
MPI_Group_union(group1, group2, newgroup)
MPI_Group_intersection(group1, group2, newgroup)
MPI_Group_difference(group1, group2, newgroup)
```

```
MPI_Group_size(group, size)
MPI_Group_rank(group, rank)
```

图 7.7 MPI_Group_incl

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Group_incl (				
group	group	MPI_Group	TYPE (MPI_Group)	IN
n	number of elements in array ranks (and size of newgroup)	int	INTEGER	IN
ranks	ranks of processes in group to appear in newgroup	const int[]	INTEGER(n)	IN
newgroup	new group derived from above, in the order defined by ranks	MPI_Group*	TYPE (MPI_Group)	OUT
)				

图 7.8 MPI_Group_excl

名称 参数名	说明	C 类型	F ty pe	i nout
MPI_Group_excl (				
group	group	MPI_Group	TYPE (MPI_Group)	IN
n	number of elements in array ranks	int	INTEGER	IN
ranks	array of integer ranks of processes in group not to appear in newgroup	const int[]	INTEGER(n)	IN
newgroup	new group derived from above, preserving the order defined by group	MPI_Group*	TYPE (MPI_Group)	OUT
)				

某些 MPI 类型， `MPI_Win` 和 `MPI_File`，是在通信器上创建的。虽然你不能直接从对象中提取该通信器，但你可以通过 `MPI_Win_get_group` 和 `MPI_File_get_group` 获取组。

有一个预定义的空组 `MPI_GROUP_EMPTY`，它可以用作组构造例程的输入，或者作为零交集等操作的结果。这不同于 `MPI_GROUP_NULL`，后者是对组的无效操作的输出，或 `MPI_Group_free` 的结果。

MPL 注释 *62*：原始组句柄。 如果你需要 MPL 组中包含的 `MPI_Datatype` 对象，有一个访问函数 *native_handle*。

7.5.2 示例

假设你想将 world communicator 拆分成一个管理进程，其余进程为工作进程。

```
// portapp.cMPI_Comm comm_work;{MPI_Group world_group,
work_group;MPI_Comm_group( comm_world,&world_group );
int manager[] = {0};
MPI_Group_excl( world_group,1,manager,&work_group );
MPI_Comm_create( comm_world,work_group,&comm_work );
MPI_Group_free( &world_group ); MPI_Group_free( &work_group );}
```

练习 7.3. 编写一个进行规模研究的代码：你的代码需要包含一个针对逐渐增大子集的循环 MPI_COMM_WORLD。

```
for (int subsize=1; subsize<=worldsize; subsize++) {
  MPI_Comm subcomm;
  // form `subcomm' to be of size `subsize'
  MPI_Allreduce( /* stuff */ subcomm );
  }
```

仔细确定各个通信器和组调用由哪个进程执行；特别是在正确的进程上执行 MPI_Comm_free 和 MPI_Group_free 。

7.6 Intercommunicators

在某些场景中，可能希望有一种方式在 communicators 之间进行通信。例如，一个应用程序可以有明确功能分离的模块（预处理器、模拟、后处理器），它们需要成对地流式传输数据。另一个例子是，动态生成的进程（第 8.1 节）拥有自己的 MPI_COMM_WORLD 值，但仍需要与生成它们的进程进行通信。本节将讨论 *inter-communicator* 机制，以满足此类用例。

当然，可以通过拥有一个重叠它们的 communicator 来实现不相交 communicators 之间的通信，但这会很复杂：由于 “inter” 通信发生在重叠的 communicator 中，必须将其顺序转换为两个工作 communicator 的顺序。直接用这些 communicators 来表达消息会更简单，这正是 *inter-communicator* 中发生的情况。

调用 MPI_Intercomm_create (图 7.9) 涉及以下 communicators：

- 两个本地通信器，在此上下文中称为 *intra-communicator*：每个通信器中有一个进程作为本地领导者，与远程领导者相连；
- 对等通信器，通常 MPI_COMM_WORLD，包含本地通信器；
- 一个 *inter-communicator*，允许子通信器的领导者与另一个子通信器通信。

尽管 intercommunicator 只连接两个进程，但它在 peer communicator 上是集体的。

图 7.2：intercommunicator 设置中 ranks 的示意图

图 7.9 `MPI_Intercomm_create`

名称 参数名	说明	C 类型	F ty pe	i nout
`MPI_Intercomm_create (`				
`local_comm`	`local intra-communicator`	`MPI_Comm`	`TYPE (MPI_Comm)`	`IN`
`local_leader`	`rank of local group leader in local_comm`	`int`	`INTEGER`	`IN`
`peer_comm`	``peer'' communicator; significant only at the local_leader	`MPI_Comm`	`TYPE (MPI_Comm)`	`IN`
`remote_leader`	`rank of remote group leader in peer_comm; significant only at the local_leader`	`int`	`INTEGER`	`IN`
`tag`	`tag`	`int`	`INTEGER`	`IN`
`newintercomm`	`new inter-communicator`	`MPI_Comm*`	`TYPE (MPI_Comm)`	`OUT`
`)`				

7.6.1 互通信器点对点通信

本地领导者现在可以相互通信。

- 发送者指定目标为另一个子通信器中另一个领导者的本地编号；
- 同样，接收方在其子通信器中将发送方的本地编号指定为源。

I 从某种意义上说，这种设计是合理的：处理器以其自然的、本地的编号被引用。另一方面，这意味着每个组都需要知道另一个组的本地排序是如何安排的。使用复杂的 *key* 值使这变得困难。

```
if (i_am_local_leader) {
```

```
    if (color==0) {interdata = 1.2;int inter_target = local_number_of_other_leader;
printf("[%d] sending interdata %e to %d\n",procno, interdata, inter_target);
MPI_Send(&interdata,1,MPI_DOUBLE,inter_target,0,intercomm);} else {
MPI_Status status;
MPI_Recv(&interdata,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,intercomm,&status);
int inter_source = status.MPI_SOURCE;
printf("[%d] received interdata %e from %d\n",procno, interdata, inter_source);
if (inter_source!=local_number_of_other_leader)fprintf(stderr,
"Got inter communication from unexpected %d; s/b %d\n", inter_source,
local_number_of_other_leader);}}
```

7.6.2 互通信器集合操作

互通信器可以用于诸如广播等集合操作。

- 在发送组中，根进程传递 `MPI_ROOT` 作为 ‘root’ 值；其他所有进程使用 `MPI_PROC_NULL`。
- 在接收组中，所有进程使用一个 ‘root’ 值，该值是根组中根进程的秩。注意：这不是全局秩！

Gather 和 scatter 的行为类似；allgather 不同：组 A 的所有发送缓冲区按秩顺序连接，并放置在组 B 的所有进程上。

如果两个进程组相对于彼此异步工作，可以使用 Intercommunicators；另一个应用是容错（章节 15.5）。

```
if (color==0) { // sending group: the local leader sends
  if (i_am_local_leader)
    root = MPI_ROOT;
  else
    root = MPI_PROC_NULL;
} else { // receiving group: everyone indicates leader of other group
  root = local_number_of_other_leader;
}
if (DEBUG) fprintf(stderr,"[%d] using root value %d\n",procno,root);
MPI_Bcast(&bcast_data,1,MPI_INT,root,intercomm);
```

7.6.3 互通信器查询

S你之前见过的一些针对内部通信器的操作在互通信器上表现不同：
m互通信器：

- `MPI_Comm_size` 返回的是本地组的大小，而不是互通信器的大小。

图 7.10 MPI_Comm_get_parent

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_get_parent (					
	parent	the parent communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

图 7.11 MPI_Comm_test_inter

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_test_inter (					
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	flag	true if comm is an inter-communicator	int*	LOGICAL	OUT
)					

- MPI_Comm_rank 返回本地组中的秩。
- MPI_Comm_group 返回本地组。

派生进程可以通过 MPI_Comm_get_parent (图 7.10) （参见第 8.1 节中的示例）找到它们的父通信器。在其他进程上，这将返回 MPI_COMM_NULL。

测试一个通信器是内部通信器还是互通信器：MPI_Comm_test_inter (图 7.11)。

MPI_Comm_compare 适用于互通信器。

通过互通信器连接的进程可以使用 MPI_Comm_remote_size (图 7.12) 查询 “ 另一方 ” 通信器的大小。实际的组可以通过 MPI_Comm_remote_group (图 7.13) 获得。

虚拟拓扑（第 11 章）不能用互通信器创建。要设置虚拟拓扑，首先使用函数 MPI_Intercomm_merge (图 7.14) 将互通信器转换为内部通信器。

图 7.12 MPI_Comm_remote_size

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_remote_size (					
	comm	inter-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	size	number of processes in the remote group of comm	int*	INTEGER	OUT
)					

Python:

```
Intercomm.Get_remote_size(self)
```

图 7.13 MPI_Comm_remote_group

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_remote_group (					
	comm	inter-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	group	remote group corresponding to comm	MPI_Group*	TYPE (MPI_Group)	OUT
)					

Python:

```
Intercomm.Get_remote_group(self)
```

7.7 复习问题

对于所有判断题，如果你回答某个陈述是错误的，请给出一句话的解释。

1. 对错题：在每个通信器中，进程的编号是从零开始连续的。 2. 如果一个进程属于两个通信器，则它在两个通信器中的秩相同。 3. 任何不是 `MPI_COMM_WORLD` 的通信器都是它的严格子集。 4. 由 `MPI_Comm_split` 派生的子通信器是互不相交的。 5. 如果两个进程在某个通信器中的秩是 $p < q$，且它们属于同一个子通信器，那么它们在子通信器中的秩 p'，q' 也遵守 $p' < q'$。

图 7.14 MPI_Intercomm_merge

名称 参数名	说明	C 类型	F 类型	inout
MPI_Intercomm_merge (				
intercomm	inter-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
high	ordering of the local and remote groups in the new intra-communicator	int	LOGICAL	IN
newintracomm	new intra-communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

第 8 章

MPI 主题：进程管理

在本课程中，到目前为止我们只考虑了运行 MPI 程序的 SPMD 模型。在某些罕见情况下，您可能希望以 MPMD 模式运行，而不是 SPMD。这可以通过操作系统级别，使用 `mpiexec` 机制的选项来实现，或者您可以使用 MPI 内置的进程管理。如果您对后者感兴趣，请继续阅读。

8.1 进程生成

MPI 的第一个版本不包含任何进程管理例程，尽管早期的 *PVM* 项目确实具有该功能。进程管理后来在 MPI-2 中被添加。

与您可能想象的不同，新添加的进程不会成为 `MPI_COMM_WORLD` 的一部分；相反，它们获得自己的通信器，并且在这个新组和现有组之间建立了一个 *inter-communicator* (章节 7.6)。第一个例程是 `MPI_Comm_spawn` (图 8.1)，它尝试启动单个命名可执行文件的多个副本。启动这些代码时的错误会以整数数组的形式返回，或者如果您很有把握，可以指定 `MPI_ERRCODES_IGNORE`。

是否有机会生成新的可执行文件尚不清楚；毕竟，`MPI_COMM_WORLD` 包含了你所有可用的处理器。你可能可以告诉你的作业启动器为几个额外的进程保留空间，但这取决于安装情况（见下文）。然而，有一个标准机制用于查询是否已保留了这样的空间。属性 `MPI_UNIVERSE_SIZE`，通过 `MPI_Comm_get_attr` (第 15.1.2 节) 检索，将告诉你可用的主机总数。

如果不支持此选项，你可以自行确定想要生成多少进程。然而，如果你超出了硬件资源，你的多任务操作系统（几乎所有人使用的都是某种 Unix 变体）将使用时间片轮转来启动生成的进程，但你不会获得任何性能提升。

8.1.1 命令行参数

The *argv* argument contains the *commandline arguments* passed to the spawned process.

- 这个数组需要以空字符结尾，以便确定其长度。

图 8.1 MPI_Comm_spawn

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Comm_spawn (					
	command	name of program to be spawned	const char*	CHARACTER	IN
	argv	arguments to command	char*[]	CHARACTER(*)	IN
	maxprocs	maximum number of processes to start	int	INTEGER	IN
	info	a set of key-value pairs telling the runtime system where and how to start the processes	MPI_Info	TYPE (MPI_Info)	IN
	root	rank of process in which previous arguments are examined	int	INTEGER	IN
	comm	intra-communicator containing group of spawning processes	MPI_Comm	TYPE (MPI_Comm)	IN
	intercomm	inter-communicator between original group and the newly spawned group	MPI_Comm*	TYPE (MPI_Comm)	OUT
	array_of_errcodes	one code per process	int[]	INTEGER(*)	OUT
	)				

Python:

```
MPI.Intracomm.Spawn(self,command, args=None, int maxprocs=1,
 Info info=INFO_NULL,int root=0, errcodes=None)
returns an intracommunicator
```

- 如果生成的进程不接受命令行参数，可以使用一个值为 `MPI_ARGV_NULL` 可以使用，在 C 和 Fortran 中均可使用。在 C 中，这与 *NULL* 相同。
- 与主程序的 *argv* 参数不同，spawn 调用中传入的 *argv* 参数不包含可执行文件的名称。

8.1.2 示例：workmanager

这里是一个 work manager 的示例。首先我们查询有多少空间可用于新进程，使用标志来查看是否支持此选项：

```
int universe_size, *universe_size_attr,uflag;
MPI_Comm_get_attr
  (comm_world,MPI_UNIVERSE_SIZE,
   &universe_size_attr,&uflag);
if (uflag) {
  universe_size = *universe_size_attr;
} else {
  printf("This MPI does not support UNIVERSE_SIZE.\nUsing world size");
  universe_size = world_n;
```

```
}int work_n = universe_size - world_n;if (world_p==0) {
printf("A universe of size %d leaves room for %d workers\n",
universe_size,work_n);
printf(".. spawning from %s\n",procname);}
```

(参见章节 15.1.2 关于该解引用行为。)

然后我们实际生成进程:

```
const char *workerprogram = "./spawnapp";
MPI_Comm_spawn(workerprogram,MPI_ARGV_NULL,
               work_n,MPI_INFO_NULL,
               0,comm_world,&comm_inter,NULL);
```

```
## spawnmanager.py
try :
    universe_size = comm.Get_attr(MPI.UNIVERSE_SIZE)
    if universe_size is None:
        print("Universe query returned None")
        universe_size = nprocs + 4
    else:
        print("World has {} ranks in a universe of {}"\
              .format(nprocs,universe_size))
except :
    print("Exception querying universe size")
    universe_size = nprocs + 4
nworkers = universe_size - nprocs

itercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)
```

一个进程可以通过使用 `MPI_Comm_get_parent` 来检测它是生成进程还是被生成进程：生成的 intercommunicator 在父进程上是 `MPI_COMM_NULL` 。

```
// spawnapp.c
MPI_Comm comm_parent;
MPI_Comm_get_parent(&comm_parent);
int is_child = (comm_parent!=MPI_COMM_NULL);
if (is_child) {
  int nworkers,workerno;
  MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
  MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
  printf("I detect I am worker %d/%d running on %s\n",
         workerno,nworkers,procname);
```

生成的程序看起来非常像一个常规的 MPI 程序，具有自己的初始化和结束调用。

```
// spawnworker.c
MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
MPI_Comm_get_parent(&parent);
```

```
## spawnworker.py
parentcomm = comm.Get_parent()
nparents = parentcomm.Get_remote_size()
```

生成的进程最终拥有自己的值 `MPI_COMM_WORLD`，但管理者和工作者无论如何都能找到彼此。spawn 例程返回父进程的互通信器；子进程可以通过 `MPI_Comm_get_parent`（章节 7.6.3）找到它。生成进程的数量可以通过父通信器上的 `MPI_Comm_remote_size` 获得。

```
Running spawnapp with usize=12, wsize=4%%
%% manager output%%
A universe of size 12 leaves room for 8 workers
.. spawning from c209-026.frontera.tacc.utexas.edu
%%%% worker output%%


Worker deduces 8 workers and 4 parents
I detect I am worker 0/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 1/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 2/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 3/8 running on c209-027.frontera.tacc.utexas.edu
I detect I am worker 4/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 5/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 6/8 running on c209-028.frontera.tacc.utexas.edu
I detect I am worker 7/8 running on c209-028.frontera.tacc.utexas.edu
```

8.1.3 MPI 启动与 universe

你可以用以下方式启动该程序的单个副本

```
mpiexec -n 1 spawnmanager
```

但使用包含多个主机的 hostfile。

TACCnote. Intel MPI 要求你传递一个选项 `-usize` 到 `mpiexec`，以指示 comm universe 的大小。使用 TACC 作业启动器 `ibrun` 执行以下操作：

```
export FI_MLX_ENABLE_SPAWN=yes
# specific
MY_MPIRUN_OPTIONS="-usize 8" ibrun -np 4 spawnmanager
# more generic
MY_MPIRUN_OPTIONS="-usize ${SLURM_NPROCS}" ibrun -np 4 spawnmanager
# using mpiexec:
mpiexec -np 2 -usize ${SLURM_NPROCS} spawnmanager
```

图 8.2 MPI_Open_port

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Open_port (				
info	implementation-specific information on how to establish an address	MPI_Info	TYPE (MPI_Info)	IN
port_name	newly established port	char*	CHARACTER	OUT
)				

8.1.4 MPMD

I你可以使用 MPI_Comm_spawn_multiple 来生成多个可执行文件，而不是只生成一个。
c进程可以通过属性 MPI_APPNUM 检索它是哪个可执行文件；详见第 15.1.2 节。

命令行参数的处理方式与 MPI_Comm_spawn （第 8.1.1 节）类似，只是现在是一个字符串数组的数组。如果不是所有可执行文件都接受多个 spawn 的命令行参数，则可以传递值 MPI_ARGVS_NULL 。如果只有某些可执行文件不接受参数，则需要为它们传递一个长度为 1 的数组，数组中仅包含空终止符。

8.2 Socket-style communications

可以与拥有自己 world communicator 的正在运行的 MPI 程序建立连接。

服务器进程与 MPI_Open_port 建立端口，并调用 MPI_Comm_accept 接受对其端口的连接。

- *client* 进程在 MPI_Comm_connect 调用中指定该端口。这建立了连接。

8.2.1 服务器调用

服务器调用 MPI_Open_port (图 8.2)，生成一个端口名。端口名由系统生成并复制到长度最多为 MPI_MAX_PORT_NAME 的字符缓冲区中。

服务器随后需要调用 MPI_Comm_accept (图 8.3)，在客户端进行连接调用之前。这是在调用通信器上的集合操作。它返回一个允许与客户端通信的互通信器（章节 7.6 ）。

```
MPI_Comm intercomm;
char myport[MPI_MAX_PORT_NAME];
MPI_Open_port( MPI_INFO_NULL,myport );
int portlen = strlen(myport);
MPI_Send( myport,portlen+1,MPI_CHAR,1,0,comm_world );
printf("Host sent port <<%s>>\n",myport);
MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
printf("host accepted connection\n");
```

端口可以通过 MPI_Close_port 关闭。

图 8.3 MPI_Comm_accept

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Comm_accept (				
port_name	port name	const char*	CHARACTER	IN
info	implementation-dependent information	MPI_Info	TYPE (MPI_Info)	IN
root	rank in comm of root node	int	INTEGER	IN
comm	intra-communicator over which call is collective	MPI_Comm	TYPE (MPI_Comm)	IN
newcomm	inter-communicator with client as remote group	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

图 8.4 MPI_Comm_connect

Name 参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_connect (				
port_name	network address	const char*	CHARACTER	IN
info	implementation-dependent information	MPI_Info	TYPE (MPI_Info)	IN
root	rank in comm of root node	int	INTEGER	IN
comm	intra-communicator over which call is collective	MPI_Comm	TYPE (MPI_Comm)	IN
newcomm	inter-communicator with server as remote group	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

8.2.2 Client calls

服务器生成端口名称后，客户端需要使用 `MPI_Comm_connect` (图 8.4) 连接该端口，同样通过字符缓冲区指定端口。connect 调用是其通信器上的集合操作。

```
char myport[MPI_MAX_PORT_NAME];
if (work_p==0) {
  MPI_Recv( myport,MPI_MAX_PORT_NAME,MPI_CHAR,
            MPI_ANY_SOURCE,0, comm_world,MPI_STATUS_IGNORE );
  printf("Worker received port <<%s>>\n",myport);
}
MPI_Bcast( myport,MPI_MAX_PORT_NAME,MPI_CHAR,0,comm_work );

/*
 * The workers collective connect over the inter communicator
 */
MPI_Comm intercomm;
MPI_Comm_connect( myport,MPI_INFO_NULL,0,comm_work,&intercomm );
if (work_p==0) {
  int manage_n;
  MPI_Comm_remote_size(intercomm,&manage_n);
  printf("%d workers connected to %d managers\n",work_n,manage_n);
```

图 8.5 MPI_Publish_name

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Publish_name (				
service_name	a service name to associate with the port	const char*	CHARACTER	IN
info	implementation-specific information	MPI_Info	TYPE (MPI_Info)	IN
port_name	a port name	const char*	CHARACTER	IN
)				

}

如果命名端口不存在（或已关闭），`MPI_Comm_connect` 会引发类 `MPI_ERR_PORT` 的错误。

客户端可以断开与 `MPI_Comm_disconnect` 的连接。

在 5 个进程上运行上述代码，结果为：

```
# exchange port name:
Host sent port <<tag#0$OFA#000010e1:0001cde9:0001cdee$rdma_port#1024$rdma_host#10:16:225:0:1:205:199:25
Worker received port <<tag#0$OFA#000010e1:0001cde9:0001cdee$rdma_port#1024$rdma_host#10:16:225:0:1:205:

# Comm accept/connect
host accepted connection
4 workers connected to 1 managers

# Send/recv over the intercommunicator
Manager sent 4 items over intercomm
Worker zero received data
```

8.2.3 已发布的 servicenames

比上述端口机制更优雅的是，可以发布一个命名服务，带有 `MPI_Publish_name`（图 8.5），然后其他进程可以发现该服务。

```
// publishapp.c
MPI_Comm intercomm;
char myport[MPI_MAX_PORT_NAME];
MPI_Open_port( MPI_INFO_NULL,myport );
MPI_Publish_name( service_name, MPI_INFO_NULL, myport );
MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
```

工作进程通过以下方式连接到互通信器

```
char myport[MPI_MAX_PORT_NAME];
MPI_Lookup_name( service_name,MPI_INFO_NULL,myport );
MPI_Comm intercomm;
MPI_Comm_connect( myport,MPI_INFO_NULL,0,comm_work,&intercomm );
```

为此，需要有一个 *name server* 正在运行。

图 8.6 MPI_Unpublish_name

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Unpublish_name (				
service_name	a service name	const char*	CHARACTER	IN
info	implementation-specific information	MPI_Info	TYPE (MPI_Info)	IN
port_name	a port name	const char*	CHARACTER	IN
)				

图 8.7 MPI_Comm_join

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Comm_join (				
fd	socket file descriptor	int	INTEGER	IN
intercomm	new inter-communicator	MPI_Comm*	TYPE (MPI_Comm)	OUT
)				

Intel note. 启动 *hydra* 名称服务器并使用相应的 mpi 启动器：

```
hydra_nameserver &
MPIEXEC=mpiexec.hydra
```

有一个环境变量，但似乎不需要。

```
export I_MPI_HYDRA_NAMESERVER=`hostname`:8008
```

也可以将名称服务器作为参数传递给作业启动器 .

在 arun 结束时，服务应使用 `MPI_Unpublish_name` (图 8.6) 取消发布。取消发布不存在或已取消发布的服务会返回错误代码 `MPI_ERR_SERVICE`。

MPI 不保证连接尝试的公平性。也就是说，连接尝试不一定按照发起的顺序被满足，其他连接尝试的竞争可能会阻止某个特定连接尝试被满足。

8.2.4 Unix 套接字

也可以使用 *intercommunicator* 从 Unix*socket* 创建，使用 `MPI_Comm_join` (图 8.7)。

8.3 会话

初始化 MPI 最常见的方法，使用 `MPI_Init` (或 `MPI_Init_thread`) 和 `MPI_Finalize`，称为 *worldmodel* ，其描述如下：

1. There is a single call to `MPI_Init` or `MPI_Init_thread`;
2. There is a single call to `MPI_Finalize`;
3. With very few exceptions, all MPI calls appear in between the initialize and finalize calls.

图 8.8 MPI_Session_init

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Session_init (				
info	info object to specify thread support level and MPI implementation specific resources	MPI_Info	TYPE (MPI_Info)	IN
errhandler	error handler to invoke in the event that an error is encountered during this function call	MPI_Errhandler	TYPE (MPI_Errhandler)	IN
session	new session	MPI_Session*	TYPE (MPI_Session)	OUT
)				

该模型存在一些缺点:

1. 在 MPI_Init 期间没有错误处理。2. MPI 不能被终结后重新启动；3. 如果多个库同时激活，它们不能初始化或终结 MPI，而必须基于子通信器；参见章节 7.2.2。4. 没有线程安全的方式初始化 MPI：库不能安全地执行

```
MPI_Initialized(&flag);
if (!flag) MPI_Init(0,0);
```

如果它在多线程环境中运行。以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

除了所有 MPI 都被 MPI_Init （或 MPI_Init_thread ）和 MPI_Finalize 括起来的 world 外，还有会话模型，其中诸如库等实体可以独立启动 / 结束它们的 MPI 会话。

这两种模型可以在同一个程序中使用，但它们混合使用时存在限制。

8.3.1 会话模型简述

在会话模型中，每个会话独立启动和结束 MPI，赋予每个会话一个独立的 MPI_COMM_WORLD。然后，世界模型成为启动 MPI 的另一种方式。除了启动多个会话，每个会话在自己的一组进程上，可能是相同或重叠的，你还可以使用世界模型创建的 MPI_COMM_WORLD 之外，创建会话。

你不能在单个调用中混合来自不同会话的对象，来自会话和世界模型的对象，或来自会话和 MPI_Comm_get_parent 或 MPI_Comm_join 的对象。

8.3.2 会话创建

一个 MPI 会话通过 MPI_Session_init (图 8.8) 和 MPI_Session_finalize 初始化和结束，有点类似于 MPI_Init 和 MPI_Finalize。

```
MPI_Session the_session;
MPI_Session_init
  ( session_request_info,MPI_ERRORS_ARE_FATAL,
    &the_session );
MPI_Session_finalize( &the_session );
```

鉴于上述推理，此调用是线程安全的。

8.3.2.1 会话信息

The `MPI_Info` 传递给 `MPI_Session_init` 的对象可以为 null，或者可以用来请求线程级别：

```
// session.c
MPI_Info session_request_info = MPI_INFO_NULL;
MPI_Info_create(&session_request_info);
char thread_key[] = "mpi_thread_support_level";
MPI_Info_set(session_request_info,thread_key,
"MPI_THREAD_MULTIPLE");
```

Other info keys can be implementation-dependent, but the key *thread_support* is pre-defined.

信息键可以通过 `MPI_Session_get_info` 再次检索：

```
MPI_Info session_actual_info;
MPI_Session_get_info( the_session,&session_actual_info );
char thread_level[100]; int info_len = 100, flag;
MPI_Info_get_string( session_actual_info,
                     thread_key,&info_len,thread_level,&flag );
```

8.3.2.2 会话错误处理器

错误处理程序参数接受预定义的错误处理程序（章节 15.2.2 ）或由 `MPI_Session_create_errhandler` 创建的错误处理程序。

8.3.3 进程集和通信器

会话包含多个进程集。进程集通过统一资源标识符（ URI ）表示，其中 URI *mpi://WORLD* 和 *mpi://SELF* 始终被定义。

你使用 `MPI_Session_get_num_psets` 和 `MPI_Session_get_nth_pset` 查询 ‘psets’：

Code:

```
int npsets;
MPI_Session_get_num_psets
  ( the_session,MPI_INFO_NULL,&npsets );
if (mainproc)
  printf("Number of process sets: %d\n",npsets);
for (int ipset=0; ipset<npsets; ipset++) {
  int len_pset; char name_pset[MPI_MAX_PSET_NAME_LEN];
  MPI_Session_get_nth_pset
    ( the_session,MPI_INFO_NULL,
      ipset,&len_pset,name_pset );
  if (mainproc)
    printf("Process set %2d: <<%s>>\n",
           ipset,name_pset);
```

Output:

```
mpiexec -n 2 ./session
Could not obtain thread
    ↪level,flag=0
Number of process sets: 2
Process set  0:
    ↪<<mpi://WORLD>>
Process set  1:
    ↪<<mpi://SELF>>
Found WORLD as pset 0
World has 2 processes
```

以下部分代码创建了一个与会话模型中等效的通信器 `MPI_COMM_WORLD`：

```
MPI_Group world_group = MPI_GROUP_NULL;
MPI_Comm  world_comm  = MPI_COMM_NULL;
MPI_Group_from_session_pset
  ( the_session,world_name,&world_group );
MPI_Comm_create_from_group
  ( world_group,"victor-code-session.c",
    MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,
    &world_comm );
MPI_Group_free( &world_group );
int procid = -1, nprocs = 0;
MPI_Comm_size(world_comm,&nprocs);
MPI_Comm_rank(world_comm,&procid);
```

然而，比较来自 session 和 world 模型的通信器（带有 `MPI_Comm_compare`），或来自不同 session 的通信器，是未定义行为。

从一个进程集获取 info 对象（章节 15.1.1）： `MPI_Session_get_pset_info`。该 info 对象总是包含键 *mpi_size*。

8.3.4 示例

作为 session 使用的一个示例，我们声明一个库类，其中每个库对象启动并结束其自己的 session：

```
// sessionlib.cxx
class Library {
private:
  MPI_Comm world_comm; MPI_Session session;
public:
  Library() {
    MPI_Info info = MPI_INFO_NULL;
    MPI_Session_init
      ( MPI_INFO_NULL,MPI_ERRORS_ARE_FATAL,&session );
    char world_name[] = "mpi://WORLD";
    MPI_Group world_group;
```

```
    MPI_Group_from_session_pset
( session,world_name,&world_group );
MPI_Comm_create_from_group
( world_group,"world-session",MPI_INFO_NULL,
MPI_ERRORS_ARE_FATAL,&world_comm );
MPI_Group_free( &world_group );};
~Library() { MPI_Session_finalize(&session); };
```

现在我们创建一个主程序，使用 world 模型，激活两个库，通过参数向它们传递数据：

```
int main(int argc,char **argv) {
Library lib1,lib2;MPI_Init(0,0);
MPI_Comm world = MPI_COMM_WORLD;
int procno,nprocs;
MPI_Comm_rank(world,&procno);
MPI_Comm_size(world,&nprocs);
auto sum1 = lib1.compute(procno);
auto sum2 = lib2.compute(procno+1);
```

N注意，主程序与任一库之间，或两个库之间都不会有 mpi 调用，但在此场景中这似乎是合理的。
l库，但在此场景中这似乎是合理的。
MPI-4 材料结束

8.4 init/finalize 之外可用的功能

`MPI_Initialized` `MPI_Finalized` `MPI_Get_version` `MPI_Get_library_version` `MPI_Info_create` `MPI_Info_create_env` `MPI_Info_set` `MPI_Info_delete` `MPI_Info_get` `MPI_Info_get_valuelen` `MPI_Info_get_nkeys` `MPI_Info_get_nthkey` `MPI_Info_dup` `MPI_Info_free` `MPI_Info_f2c` `MPI_Info_c2f` `MPI_Session_create_errhandler` `MPI_Session_call_errhandler` `MPI_Errhandler_free` `MPI_Errhandler_f2c` `MPI_Errhandler_c2f` `MPI_Error_string` `MPI_Error_class`

同样所有以 `MPI_Txxx` 开头的例程。

第 9 章

MPI 主题：单边通信

上面，你看到了两边式的点对点操作：它们需要发送方和接收方的配合。这种配合可以是松散的：你可以用 `MPI_ANY_SOURCE` 作为发送方发布接收，但必须同时有发送和接收调用。这种两边性可能有限制。考虑接收进程是数据的动态函数的代码：

```
x = f();
p = hash(x);
MPI_Send( x, /* to: */ p );
```

问题现在是：p 如何知道要发布接收，其他所有人又如何知道不发布?

在本节中，你将看到单边通信例程，其中一个进程可以执行 “put” 或 “get” 操作，将数据写入或从另一个处理器读取数据，而不需要另一个处理器的参与。

在单边 MPI 操作中，标准中称为远程内存访问（Remote Memory Access，RMA）操作，或在其他文献中称为远程直接内存访问（Remote Direct Memory Access，RDMA），仍然涉及两个进程：*origin*，即发起传输的进程，无论是 “put” 还是 “get”，以及被访问内存的 *target*。与两边操作不同，target 不会执行与 origin 操作对应的动作。

这并不意味着 origin 可以在任意时间访问 target 上的任意数据。首先，MPI 中的单边通信仅限于访问 target 上特定声明的内存区域：target 声明一块可被其他进程访问的内存区域。这被称为 *window*。window 限制了 origin 进程访问 target 内存的方式：你只能从 window 中 “get” 数据或向 window 中 “put” 数据；其他内存对其他进程不可达。在 origin 端没有这样的限制；任何数据都可以作为 “put” 的源或 “get” 操作的接收者。

替代使用 window 的方法是使用 *distributed shared memory* 或 *virtual shared memory*：内存是分布式的，但表现得像是共享的。所谓的 Partitioned Global Address Space (PGAS) 语言，如 Unified Parallel C (UPC)，采用这种模型。

W在单边通信中，MPI 有两种模式：active RMA 和 passive RMA。在 *activeRMA* 中，或者 *a*主动目标同步，目标设置了其窗口可被访问的时间段（ ‘epoch’ ）的边界 window 可以被访问。此模式的主要优点是源程序可以执行许多

小规模传输，在后台进行聚合。这适用于以批量同步并行（ BSP ）模式结构化的应用程序，具有超级步。Active RMA 的行为类似于带有结束 `MPI_Waitall` 的异步传输。

在被动 *RMA*，或被动目标同步中，目标进程不限制其窗口何时可以被访问。（基于 PGAS 模型的语言如 UPC 即采用此模型：数据可以随意读取或写入。）虽然直观上能够在任意时间写入和读取目标很有吸引力，但这存在问题。例如，它需要目标上的远程代理，这可能干扰主线程的执行，或者相反，它可能不会在最佳时间被激活。被动 RMA 也非常难以调试，并且可能导致竞态条件。

9.1 Windows

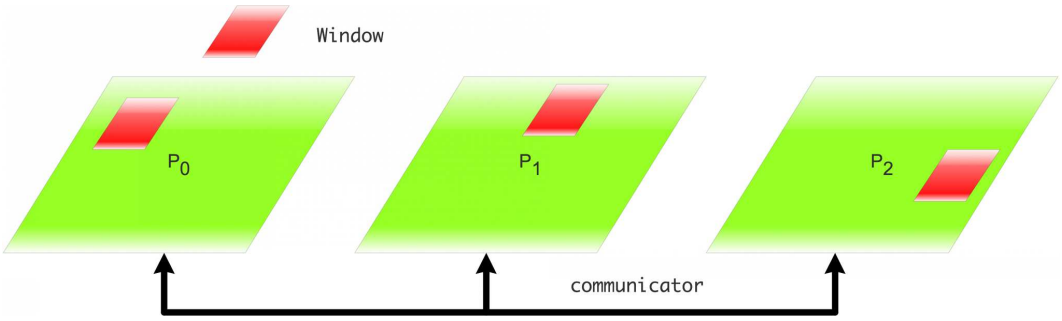


图 9.1：用于单边数据访问的窗口的集合定义

在单边通信中，每个处理器可以使一块称为 *window* 的内存区域可用于单边传输。这存储在类型为 `MPI_Win` 的变量中。一个进程可以将其自身内存中的任意项（不限于任何 window ）放入另一个进程的 window，或者从另一个进程的 window 中获取内容到自己的内存中。

window 可以描述如下：

- window 是在一个通信器上定义的，因此创建调用是集合性的；见图 9.1。
- window 大小可以在每个进程上单独设置。允许大小为零，但由于 window 创建是集合性的，无法跳过创建调用。
- 你可以为 window 设置一个 “displacementunit”：这是用作索引单位的字节数。例如，如果你使用 `sizeof(double)` 作为位移单位，访问位置 8 的 `MPI_Put` 将对应第 8 个 double。这比必须指定第 64 个字节更简单。
- 窗口是 put 操作中数据的目标，或 get 操作中数据的源；见图 9.2。
- 窗口可以关联内存，因此需要用 `MPI_Win_free` 显式释放。

典型调用包括：

图 9.1 MPI_Win_create

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Win_create (				
MPI_Win_create_c (				
base	initial address of window	void*	TYPE(*), DIMENSION(..)	IN
size	size of window in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
disp_unit	local unit size for displacements, in bytes	int MPI_Aint	INTEGER	IN
info	info argument	MPI_Info	TYPE (MPI_Info)	IN
comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
win	window object	MPI_Win*	TYPE(MPI_Win)	OUT
)				

Python:

```
MPI.Win.Create
   (memory, int disp_unit=1,
    Info info=INFO_NULL, Intracomm comm=COMM_SELF)
```

```
MPI_Info info;
MPI_Win window;
MPI_Win_allocate( /* size info */, info, comm, &memory, &window );
// do put and get calls
MPI_Win_free( &window );
```

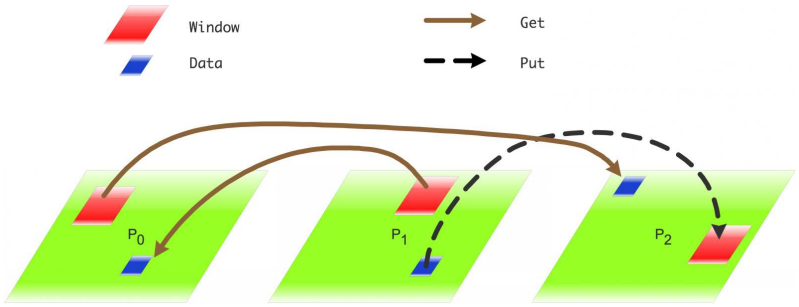


Figure 9.2: Put and get between process memory and windows

9.1.1 窗口的创建和释放

窗口的内存乍一看是用户空间中的普通数据。你可以通过多种方式将数据与窗口关联：

1. 你可以将用户缓冲区传递给 `MPI_Win_create`（图 9.1）。该缓冲区可以是普通数组，也可以通过 `MPI_Alloc_mem` 创建。（在前一种情况下，可能无法锁定窗口；参见第 9.4 节。）

图 9.2 MPI_Win_allocate

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Win_allocate (				
MPI_Win_allocate_c (				
size	size of window in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
disp_unit	local unit size for displacements, in bytes	int MPI_Aint	INTEGER	IN
info	info argument	MPI_Info	TYPE (MPI_Info)	IN
comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
baseptr	initial address of window	void*	TYPE(C_PTR)	OUT
win	window object returned by call	MPI_Win*	TYPE(MPI_Win)	OUT
)				

2. 你可以让 MPI 进行分配，这样 MPI 可以针对内存的放置执行各种优化。用户代码随后从 MPI 接收指向数据的指针。这又可以通过两种方式完成：

- 使用 `MPI_Win_allocate`（图 9.2）一次调用创建数据和窗口。
- 如果一个通信器位于共享内存上（参见第 7.4.1 节），你可以使用 `MPI_Win_allocate_shared` 在该共享内存中创建一个窗口。这对于 MPI 共享内存 非常有用；参见第 12 章。

3. 最后，你可以使用 `MPI_Win_create_dynamic` 创建一个推迟分配的窗口；参见第 9.5.3 节。

首先，`MPI_Win_create` 从指向内存的指针创建一个窗口。数据数组不得 PARAMETER 或 `static const`。

大小参数以字节为单位。在 C 语言中，这可以通过 `sizeof` 操作符完成；

```
// putfencealloc.cMPI_Win the_window;
int *window_data;
MPI_Win_allocate(2*sizeof(int),sizeof(int),
MPI_INFO_NULL,comm,&window_data,&the_window);
```

有关在 Fortran 中进行此计算，请参见第 15.3.1 节。

Python 注释 *27*：位移字节计算。计算字节位移时，这里有一种查找 *numpy* 数据类型大小的好方法：

```
## putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

图 9.3 MPI_Alloc_mem

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Alloc_mem (				
size	size of memory segment in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
info	info argument	MPI_Info	TYPE (MPI_Info)	IN
baseptr	pointer to beginning of memory segment allocated	void*	TYPE(C_PTR)	OUT
)				

接下来，可以通过使用 MPI 获得内存 `MPI_Win_allocate`，其输出为数据指针。注意 C 函数签名中的 `void*`；仍然需要传递指向指针的指针：

```
double *window_data;
MPI_Win_allocate( ... &window_data ... );
```

例程 `MPI_Alloc_mem`（图 9.3）仅执行 `MPI_Win_allocate` 的分配部分，之后需要 `MPI_Win_create`。

- 一个 `MPI_ERR_NO_MEM` 错误表示无法分配内存。以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

  - 可以通过指定 `MPI_Info` 键为 *mpi_minimum_memory_alignment* 来对分配的内存进行对齐。
  - 分配内存的类型可以通过 info 键进行控制；参见第 9.5.2 节。
  - 一个 info 键 *mpi_accumulate_granularity* 可用于区分累积操作中更注重吞吐量还是延迟。该键的整数值表示同步之间的字节数；这也可以表示原子写入的字节数（MPI-4.1）。

MPI-4 内容结束
这段内存通过 `MPI_Free_mem` 释放：

```
// getfence.cint *number_buffer = NULL;
MPI_Alloc_mem( /* size: */ 2*sizeof(int),
MPI_INFO_NULL, &number_buffer);
MPI_Win_create
( number_buffer,2*sizeof(int),sizeof(int),
MPI_INFO_NULL, comm, &the_window);
MPI_Win_free(&the_window);
MPI_Free_mem(number_buffer);
```

（注意 free 调用中没有 & 符号！）
如果没有特殊的内存区域，这些调用会简化为 *malloc* 和 *free*；SGI 是一个存在此类内存的例子。

窗口通过调用集合操作来释放 `MPI_Win_free`（图 9.4），它设置了窗口句柄 -

图 9.4 MPI_Win_free

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Win_free (				
win	window object	MPI_Win*	TYPE(MPI_Win)	INOUT
)				

dle to *MPI_WIN_NULL*。此调用必须仅在所有 RMA 操作完成后进行，具体由 MPI_Win_fence,MPI_Win_wait, MPI_Win_complete, MPI_Win_unlock 决定，视情况而定。如果窗口内存是通过 MPI 内部调用 MPI_Win_allocate 或 MPI_Win_allocate_shared 分配的，则会被释放。用于窗口的用户内存可以在 MPI_Win_free 调用后释放。

关于窗口内存的更多讨论见第 9.5.1 节。

Python 注释 *28*：窗口缓冲区。与 C 语言不同，python 的 window allocate 调用不会返回指向缓冲区内存的指针，而是返回一个 *MPI.memory* 对象。如果你需要裸内存，有以下选项：

- 窗口对象暴露了 Python 缓冲区接口。因此你可以做一些 Python 式的操作，比如

```
mview = memoryview(win)
array = numpy.frombuffer(win, dtype='i4')
```

- 如果你真的想要原始基指针（作为整数），你可以做以下任意操作：

```
base, size, disp_unit = win.atts
base = win.Get_attr(MPI.WIN_BASE)
```

- 你可以使用 mpi4py 内置的 memoryview/buffer 类类型，但我不推荐，使用上面提到的 NumPy 会更好：

```
mem = win.tomemory() # type(mem) is MPI.memory, similar to memoryview, but quite
↪limited in functionalitybase = mem.addresssize = mem.nbytes
```

9.1.2 地址运算

使用 windows 需要对地址进行一定的运算，意思是 MPI_Aint。参见 MPI_Aint_add 和 MPI_Aint_diff 在第 6.2.4 节。

9.2 Activetarget synchronization:epochs

单边通信相比双边通信有一个明显的复杂点：如果你调用 put 而不是 send，接收方怎么知道数据已经到达？这个让目标知道当前状态的过程称为 “同步”，并且有多种机制实现。首先我们将考虑 *activetarget synchronization*。这里目标知道传输可能发生的时间（通信时期），但不会进行任何与数据相关的调用。

图 9.5 MPI_Win_fence

名称 参数名	说明	C 类型	F 类型	inout
MPI_Win_fence (				
assert	program assertion	int	INTEGER	IN
win	window object	MPI_Win	TYPE(MPI_Win)	IN
)				

Python:

```
win.Fence(self, int assertion=0)
```

本节我们将介绍第一种机制，即使用 *fence* 操作： `MPI_Win_fence` (图 9.5)。该操作是在窗口的通信器上进行的集合操作。（另一种更复杂的主动目标同步机制将在第 9.2.2 节讨论。）

两个 fence 之间的间隔称为一个 *epoch*。大致来说，在一个 epoch 中你可以进行单边通信调用，而在结束的 fence 之后，所有这些通信都完成。

```
MPI_Win_fence(0,win);
MPI_Get( /* operands */, win);
MPI_Win_fence(0, win);
// the `got' data is available
```

在两个 fence 之间，窗口是暴露的，在此期间你不应本地访问它。如果你确实需要本地访问，可以使用 RMA 操作。此外，只有一个远程进程可以执行 `put` ；允许多个 `accumulate` 访问。

Fence 与其他窗口调用一起，是集体操作。这意味着它们暗示了进程之间某种程度的同步。考虑：

```
MPI_Win_fence( ... win ... ); // start an epoch
if (mytid==0) // do lots of work
MPI_Win_fence( ... win ... ); // end the epoch
```

并假设所有进程几乎同时执行第一个 fence。零号进程在执行第二个 fence 调用之前会先做一些工作，但所有其他进程可以立即调用它。然而，它们不能完成第二个 fence 调用，直到所有单边通信完成，这意味着它们要等待零号进程。

作为进一步的限制，不能在单个 epoch 中混合 `MPI_Get` 与 `MPI_Put` 或 `MPI_Accumulate` 调用。因此，我们可以将一个 epoch 描述为源端的 访问 *epoch* ，以及目标端的 暴露 *epoch* 。

9.2.1 Fence 断言

你可以通过 `assert` 参数向系统提供关于该 epoch 与之前和之后的 epoch 的各种提示。

- `MPI_MODE_NOSTORE` 该值可以针对每个进程指定，也可以不指定。
- `MPI_MODE_NOPUT` 该值可以针对每个进程指定，也可以不指定。

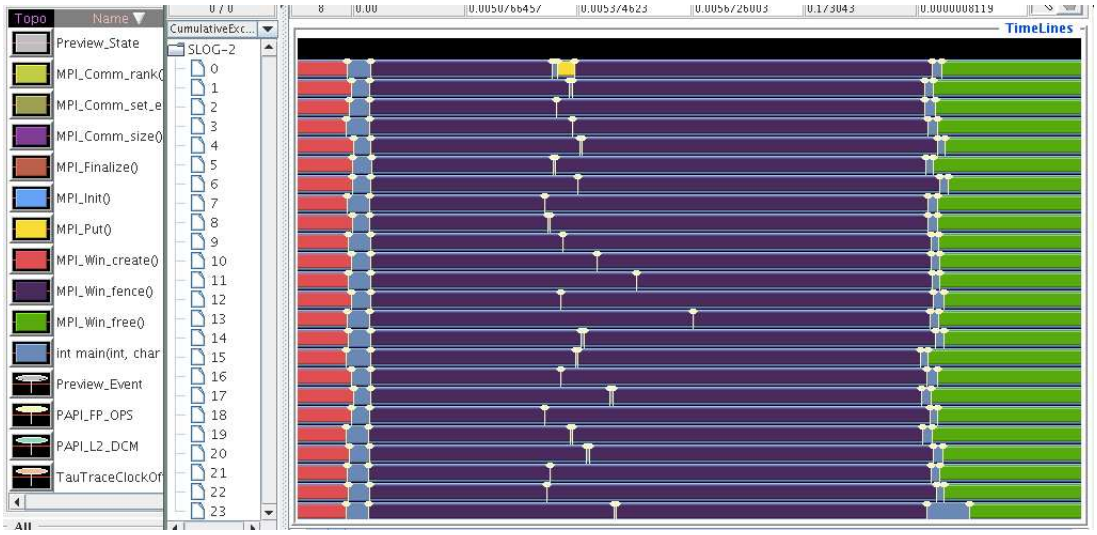


图 9.3：单边通信时期的跟踪，其中进程零仅发起单边传输

- `MPI_MODE_NOPRECEDE` 此值必须在所有进程上指定，或者不相同。
- `MPI_MODE_NOSUCCEED` 此值必须在所有进程上指定，或者不相同。

示例：

```
MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
MPI_Get( /* operands */, win);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

断言是一个整数参数：你可以通过相加或使用逻辑或来组合断言。值为零总是正确的。更多信息，请参见第 9.6 节。

9.2.2 非全局 target 同步

“fence” 机制（第 9.2 节）使用对窗口通信器的全局同步，使程序具有类似 BSP 的特性。因此，它适用于进程大体同步的应用，但如果处理器不同步，可能导致性能低效。此外，全局同步可能有硬件支持，使其限制性不如初看起来那么大。

有一种更细粒度的机制，仅对处理器组使用同步。它需要四个不同的调用，分别用于开始和结束 epoch，针对 target 和 origin 各两个。

你使用 *exposureepoch* 的 `MPI_Win_post` / `MPI_Win_wait` 来开始和完成：

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

I换句话说，这会将你的窗口变成远程访问的 *target*。有一个非阻塞版本 `MPI_Win_test` of `MPI_Win_wait`.

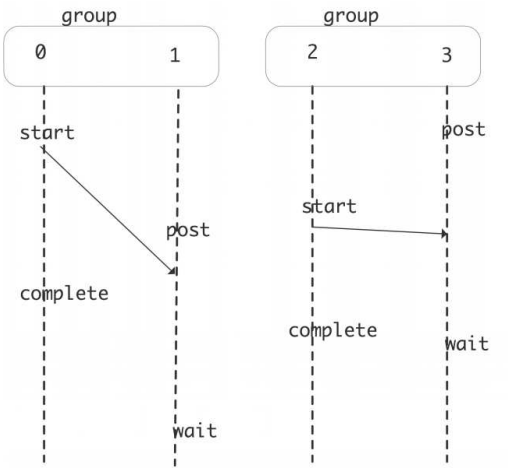


图 9.4：细粒度活动目标同步中的窗口锁调用

你开始并完成一个 访问 *epoch* 通过 `MPI_Win_start` / `MPI_Win_complete`：

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
```

换句话说，这些调用界定了对远程窗口的访问，当前处理器是远程访问的 *origin* 。

在下面的代码片段中，单个处理器将数据放到另一个处理器上。注意它们各自有自己的组定义，且接收进程只执行 post 和 wait 调用。

```
// postwaitwin.c
MPI_Comm_group(comm,&all_group);
if (procno==origin) {
  MPI_Group_incl(all_group,1,&target,&two_group);
// access
  MPI_Win_start(two_group,0,the_window);
  MPI_Put( /* data on origin: */   &my_number, 1,MPI_INT,
           /* data on target: */   target,0,   1,MPI_INT,
   the_window);
  MPI_Win_complete(the_window);
}

if (procno==target) {
  MPI_Group_incl(all_group,1,&origin,&two_group);
// exposure
  MPI_Win_post(two_group,0,the_window);
  MPI_Win_wait(the_window);
}
```

这两对操作都声明了一个处理器组；参见第 7.5.1 节了解如何从通信器中获取这样的组。在源处理器上，您需要指定一个包含将要交互的目标的组；在目标处理器上，您需要指定一个包含可能的源的组。

图 9.6 MPI_Put

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Put (				
MPI_Put_c (				
origin_addr	initial address of origin buffer	const void*	TYPE(*), DIMENSION(..)	IN
origin_count	number of entries in origin buffer	int MPI_Count	INTEGER	IN
origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	rank of target	int	INTEGER	IN
target_disp	displacement from start of window to target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
target_count	number of entries in target buffer	int MPI_Count	INTEGER	IN
target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
win	window object used for communication	MPI_Win	TYPE(MPI_Win)	IN
)				

Python:

```
win.Put(self, origin, int target_rank, target=None)
```

9.3 Put,get, accumulate

我们现在来看用于执行单边操作的前三个例程：Put 、 Get 和 Accumulate 调用。（我们将在第 9.3.7 节中讨论所谓的 “ 原子 ” 操作。）这些调用在某种程度上类似于 Send 、 Receive 和 Reduce，当然不同的是只有一个进程发起调用。由于所有工作都由一个进程完成，其调用序列同时包含了对源（调用进程）和目标（受影响的其他进程）数据的描述。

与双边情况一样， `MPI_PROC_NULL` 可以用作目标秩。

Accumulate 例程有一个 `MPI_Op` 参数，可以是任何常用运算符，但不包括用户自定义运算符（参见第 3.10.1 节）。

9.3.1 Put

The `MPI_Put` (图 9.6) 调用可以被视为单边发送。因此，它需要指定

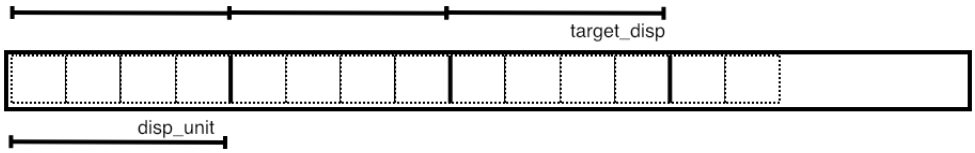
- the target rank
- 要从源头发送的数据，以及
- 要写入目标的位置。

源端数据的描述是常见的缓冲区 / 计数 / 数据类型三元组。然而，目标端数据的描述更为复杂。它有一个计数和一个数据类型，但另外它相对于目标窗口起始位置有一个位移。这个位移可以被给出

以字节为单位，因此其类型是 MPI_Aint，但严格来说它是窗口定义中指定的位移单位的倍数。

具体来说，数据从

$$\texttt{window\_base} + \texttt{target\_disp} \times \texttt{disp\_unit}.$$



这是一次单独的 put 操作。注意，window create 和 window fence 调用是集合操作，因此必须在 create 调用中使用的通信器的所有处理器上执行。

```
// putfence.c
MPI_Win the_window;
MPI_Win_create
  (&window_data,2*sizeof(int),sizeof(int),
   MPI_INFO_NULL,comm,&the_window);
MPI_Win_fence(0,the_window);
if (procno==0) {
  MPI_Put
    ( /* data on origin: */   &my_number, 1,MPI_INT,
      /* data on target: */   other,1,    1,MPI_INT,
      the_window);
}
MPI_Win_fence(0,the_window);
MPI_Win_free(&the_window);
```

Fortrannote 13: 位移单位。变量 disp_unit 被声明为 “kind” 的整数 MPI_ADDRESS_KIND：

```
!! putfence.F90
integer(kind=MPI_ADDRESS_KIND) :: target_displacement
target_displacement = 1
call MPI_Put( my_number, 1,MPI_INTEGER, &other,
target_displacement, &1,MPI_INTEGER, &the_window)
```

> 在 Fortran2008 之前，指定字面常量（例如 0 ）可能导致奇怪的运行时错误；解决方法是指定一个类型正确的零值变量。使用 mpi_f08 模块后，这种做法不再被允许。相反，你会得到类似如下的错误

```
error #6285: There is no matching specific subroutine for this generic subroutine call.   [MPI
```

Python 注释 29：*MPI* 单边传输例程。 MPI_Put (以及 Get 和 Accumulate) 最少接受源缓冲区和目标进程。位移默认是零。

练习 9.1. 重新审视练习 4.3 并使用 MPI_Put 解决它。（该练习的骨架文件名为 rightput。）

图 9.7 MPI_Get

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Get (				
MPI_Get_c (				
origin_addr	initial address of origin buffer	void*	TYPE(*), DIMENSION(..)	OUT
origin_count	number of entries in origin buffer	int / MPI_Count	INTEGER	IN
origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	rank of target	int	INTEGER	IN
target_disp	displacement from window start to the beginning of the target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
target_count	number of entries in target buffer	int / MPI_Count	INTEGER	IN
target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
win	window object used for communication	MPI_Win	TYPE(MPI_Win)	IN
)				

Python:

```
win.Get(self, origin, int target_rank, target=None)
```

练习 9.2. 编写代码，其中：

- 进程 0 计算一个随机数 r
- 如果 $r < .5$，则 zero 在窗口 1 上写入；
- 如果 $r \geq .5$，则 zero 在窗口 2 上写入。

(T以下是在该练习下的代码框架 name randomput.)

9.3.2 Get

该 MPI_Get (图 9.7) 调用非常相似。

示例：

```
MPI_Win_fence(0,the_window);
if (procno==0) {
  MPI_Get( /* data on origin: */   &my_number, 1,MPI_INT,
           /* data on target: */   other,1,    1,MPI_INT,
           the_window);
}
MPI_Win_fence(0,the_window);
```

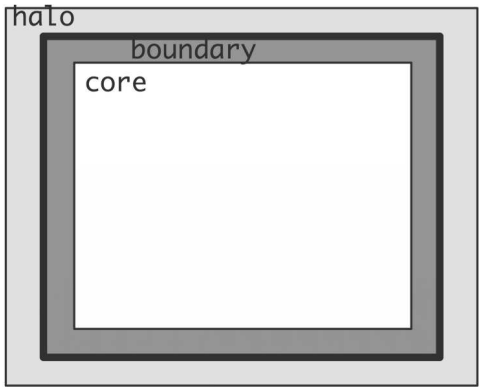
我们在不参与的进程上创建一个空窗口。

```
## getfence.py
if procid==0 or procid==nprocs-1:
```

```
    win_mem = np.empty( 1,dtype=np.float64 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    win = MPI.Win.Create( None,comm=comm )

# put data on another process
win.Fence()
if procid==0 or procid==nprocs-1:
    putdata = np.empty( 1,dtype=np.float64 )
    putdata[0] = mydata
    print("[%d] putting %e" % (procid,mydata))
    win.Put( putdata,other )
win.Fence()
```

9.3.3 Put 和 get 示例：halo update

作为示例，我们来看一下 *halo update*。数组 `A` 使用本地值和来自相邻处理器的 halo 进行更新，这些 halo 通过 Put 或 Get 操作获得。



在第一个版本中，我们将计算和通信分开。每次迭代有两个栅栏。在循环体中两个栅栏之间，我们执行 `MPI_Put` 操作；在第二个栅栏和下一次迭代的第一个栅栏之间只有计算，因此我们添加了 `MPI_MODE_NOPRECEDE` 和 `MPI_MODE_NOSUCCEED` 断言。`MPI_MODE_NOSTORE` 断言声明本地窗口未被更新：Put 操作仅作用于远程窗口。

```
for ( .... ) {
  update(A);
  MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put( ... );
  MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
  }
```

有关断言的更多内容，请参见第 9.6 节。

接下来，我们将更新分为核心部分（corepart），这部分可以纯粹从本地值完成，以及边界部分（boundary），这部分需要本地值和 halo 值。核心部分的更新可以与 halo 的通信重叠进行。

```
for ( .... ) {
  update_boundary(A);
  MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get( ... );
  update_core(A);
```

图 9.8 MPI_Accumulate

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Accumulate (					
MPI_Accumulate_c (					
	origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
	origin_count	number of entries in buffer	int / MPI_Count	INTEGER	IN
	origin_datatype	datatype of each entry	MPI_Datatype	TYPE (MPI_Datatype)	IN
	target_rank	rank of target	int	INTEGER	IN
	target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
	target_count	number of entries in target buffer	int / MPI_Count	INTEGER	IN
	target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
	op	reduce operation	MPI_Op	TYPE(MPI_Op)	IN
	win	window object	MPI_Win	TYPE(MPI_Win)	IN
	)				

Python:

```
MPI.Win.Accumulate(self, origin, int target_rank, target=None, Op op=SUM)
```

```
    MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
    }
```

MPI_MODE_NOPRECEDE 和 MPI_MODE_NOSUCCEED 断言仍然成立，但 Get 操作意味着我们不是在第二个栅栏中使用 MPI_MODE_NOSTORE，而是在第一个栅栏中使用 MPI_MODE_NOPUT。

9.3.4 Accumulate

第三个单边例程是 MPI_Accumulate（图 9.8），它对正在放置的结果执行归约操作。

Accumulate 是一种具有远程结果的原子归约操作。这意味着在同一时期内对单个目标的多次累加会得到正确的结果。与 MPI_Reduce 一样，操作数累加的顺序是未定义的。

可用相同的预定义运算符，但不支持用户自定义运算符。还有一个额外的运算符：MPI_REPLACE，其效果是仅保留最后到达的结果。

练习 9.3. 使用单边通信实现 “ 全收集 ” 操作：每个处理器存储一个数字，您希望每个处理器构建一个包含所有处理器数值的数组。请注意，处理器收集自身数值不需要特殊处理：处理器与自身之间的 “ 通信 ” 是完全合法的。

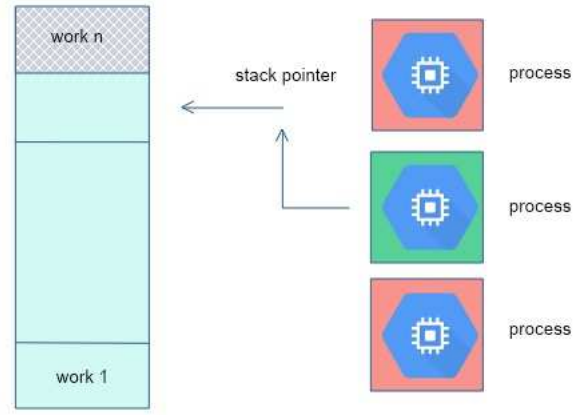


图 9.5：具有共享栈指针的工作描述符池

下一练习请参见图 9.5。

练习 9.4. 实现

实现一个共享计数器：

- 一个进程维护一个 counter;
- 迭代：其他所有人在随机时刻更新此计数器。
- 当计数器不再为正时，所有人停止迭代。

这里的问题是数据同步：每个人看到的计数器是否相同 way?

9.3.5 RMA 操作的顺序性和一致性

关于一个 epoch 内部发生的事情，几乎没有保证。

- Get 和 Put/Accumulate 操作之间没有顺序保证：如果两者都执行，无法保证 Get 是在更新之前还是之后获取的值。
- 多个 Put 操作之间没有顺序保证。使用 Accumulate 更安全。

以下操作在一个 epoch 内是定义良好的：

- 不要使用多个 Put 操作，改用带有 `MPI_REPLACE` 的 Accumulate。
- `MPI_Get_accumulate` 使用 `MPI_NO_OP` 是安全的。
- 默认情况下，来自同一源的多个 Accumulate 操作按程序顺序执行。若要允许重排序，例如使所有读取操作发生在所有写入操作之后，请在创建窗口时使用 info 参数；详见章节 9.5.4。

9.3.6 基于请求的操作

类似于 `MPI_Isend` 存在基于请求的单边操作： `MPI_Rput` (图 9.9) 同样 `MPI_Rget` 和 `MPI_Raccumulate` 和 `MPI_Rget_accumulate`。这些仅适用于被动目标同步。任何 `MPI_Win_flush`... 调用也会终止这些传输。

图 9.9 MPI_Rput

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Rput (				
MPI_Rput_c (				
origin_addr	initial address of origin buffer	const void*	TYPE(*), DIMENSION(..)	IN
origin_count	number of entries in origin buffer	int MPI_Count	INTEGER	IN
origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	rank of target	int	INTEGER	IN
target_disp	displacement from start of window to target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
target_count	number of entries in target buffer	int MPI_Count	INTEGER	IN
target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
win	window object used for communication	MPI_Win	TYPE(MPI_Win)	IN
request	RMA request	MPI_Request*	TYPE (MPI_Request)	OUT
)				

9.3.7 原子操作

单边调用被认为是在 MPI 中模拟共享内存，但 put 和 get 调用对于某些共享数据的场景来说是不够的。考虑以下场景：

- 一个进程存储一张工作描述符表，以及指向第一个未处理描述符的指针；
- 每个进程读取指针，读取相应的描述符，并递增指针；并且
- 读取了描述符的进程随后执行相应的任务。

问题在于读取和更新指针不是一个原子操作，因此可能多个进程获得相同的值；反之，指针的多次更新可能导致工作描述符被跳过。这些不同的整体行为，取决于底层事件的精确时序，被称为竞态条件。

在 MPI-3 中添加了一些原子例程。(图 9.10) 和 `MPI_Get_accumulate` (图 9.11) 都是原子地从指定窗口检索数据，并应用一个操作符，将目标上的数据与源上的数据结合。与 Put 和 Get 不同，在同一时期内进行多个原子操作是安全的。

这两个例程执行相同的操作：返回操作前的数据，然后对目标数据进行原子更新，但 `MPI_Get_accumulate` 在数据类型处理上更灵活。更简单的例程，`MPI_Fetch_and_op`，仅对单个元素操作，允许更快的实现，特别是通过硬件支持。

图 9.10 MPI_Fetch_and_op

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Fetch_and_op (				
origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)	OUT
datatype	datatype of the entry in origin, result, and target buffers	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	rank of target	int	INTEGER	IN
target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
op	reduce operation	MPI_Op	TYPE(MPI_Op)	IN
win	window object	MPI_Win	TYPE(MPI_Win)	IN
)				

图 9.11 MPI_Get_accumulate

Name Param name	解释	C 类型	F 类型	输入输出
MPI_Get_accumulate (				
MPI_Get_accumulate_c (				
origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
origin_count	number of entries in origin buffer	int MPI_Count	INTEGER	IN
origin_datatype	datatype of each entry in origin buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)	OUT
result_count	number of entries in result buffer	int MPI_Count	INTEGER	IN
result_datatype	datatype of each entry in result buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	rank of target	int	INTEGER	IN
target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
target_count	number of entries in target buffer	int MPI_Count	INTEGER	IN
target_datatype	datatype of each entry in target buffer	MPI_Datatype	TYPE (MPI_Datatype)	IN
op	reduce operation	MPI_Op	TYPE(MPI_Op)	IN
win	window object	MPI_Win	TYPE(MPI_Win)	IN
)				

使用 MPI_NO_OP 作为 MPI_Op 将这些例程变成一个原子 Get。同样，使用 MPI_REPLACE 将它们变成一个原子 Put。

练习 9.5. 重新做练习 9.4 使用 MPI_Fetch_and_op。问题依然是确保所有进程对共享计数器有相同的视图。使 fetch-and-op 有条件执行是否可行？有没有办法无条件执行？鉴于多个进程可以同时更新计数器，‘break’ 测试应该是什么？

示例。 一个根进程有一张数据表；其他进程通过 被动目标同步 通过 MPI_Win_lock 执行原子获取和更新该数据。

```
// passive.cxx
if (procno==repository) {
// Repository processor creates a table of inputs
// and associates that with the window
}
if (procno!=repository) {
  float contribution=(float)procno,table_element;
  int loc=0;
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
// read the table element by getting the result from adding zero
  MPI_Fetch_and_op
    (&contribution,&table_element,MPI_FLOAT,
     repository,loc,MPI_SUM,the_window);
  MPI_Win_unlock(repository,the_window);
}
```

```
## passive.py
if procid==repository:
    # repository process creates a table of inputs
    # and associates it with the window
    win_mem = np.empty( ninputs,dtype=np.float32 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    # everyone else has an empty window
    win = MPI.Win.Create( None,comm=comm )
if procid!=repository:
    contribution = np.empty( 1,dtype=np.float32 )
    contribution[0] = 1.*procid
    table_element = np.empty( 1,dtype=np.float32 )
    win.Lock( repository,lock_type=MPI.LOCK_EXCLUSIVE )
    win.Fetch_and_op( contribution,table_element,repository,0,MPI.SUM)
    win.Unlock( repository )
```

最后， MPI_Compare_and_swap (图 9.12) 如果目标数据等于某个比较值，则交换源和目标数据。

9.3.7.1 *A case study in atomic operations*

让我们考虑一个示例，其中一个由 *counter_process* 标识的进程拥有一张工作描述符表，所有进程，包括计数器进程，都从中取项目进行处理。为了避免重复工作，

图 9.12 MPI_Compare_and_swap

Name Param name	Explanation	C type	F type	输入输出
MPI_Compare_and_swap (				
origin_addr	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
compare_addr	initial address of compare buffer	const void*	TYPE(*), DIMENSION(..)	IN
result_addr	initial address of result buffer	void*	TYPE(*), DIMENSION(..)	OUT
datatype	datatype of the element in all buffers	MPI_Datatype	TYPE (MPI_Datatype)	IN
target_rank	rank of target	int	INTEGER	IN
target_disp	displacement from start of window to beginning of target buffer	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
win	window object	MPI_Win	TYPE(MPI_Win)	IN
)				

计数器进程有一个计数器，指示最高编号的可用项。我们模拟的这个应用部分是：

1. 一个进程读取计数器，以找到一个可用的工作项；以及 2. 随后将计数器减一。

我们在独立内存模型下初始化窗口内容：

```
// countdownop.c
MPI_Win_fence(0,the_window);
if (procno==counter_process)
MPI_Put(&counter_init,1,MPI_INT,
counter_process,0,1,MPI_INT,the_window);
MPI_Win_fence(0,the_window);
```

我们首先考虑简单方法，即用 `MPI_Get` 和 `MPI_Put` 字面执行上述方案：

```
// countdownput.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
counter_process,0,1,MPI_INT,
the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
int decrement = -1;
counter_value += decrement;MPI_Put
( &counter_value,   1,MPI_INT,
counter_process,0,1,MPI_INT,
the_window);}
```

```
MPI_Win_fence(0,the_window);
```

如果只有一个进程对 `i_am_available` 有真实值，则该方案是正确的：该进程 “ 拥有 ” 当前计数器值，并且通过 `MPI_Put` 操作正确更新计数器。然而，如果有多个进程存在，它们会获得重复的计数器值，且更新也是不正确的。如果我们运行此程序，会发现计数器并没有被所有 “put” 调用的总数所递减。

练习 9.6. 假设只有一个进程可用，三个栅栏中间的那个的作用是什么？它可以省略吗？

我们可以通过使用 `MPI_Accumulate` 来修正计数器的递减，因为它是原子的：同一时期内的多个更新都会被处理。

```
// countdownacc.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
counter_process,0,1,MPI_INT,
the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
int decrement = -1;MPI_Accumulate
( &decrement,       1,MPI_INT,
counter_process,0,1,MPI_INT,MPI_SUM,
the_window);}
MPI_Win_fence(0,the_window);
```

该方案仍然存在进程会获取重复计数器值的问题。真正的解决方法是将 “get” 和 “put” 操作合并为一个原子操作；在这种情况下为 `MPI_Fetch_and_op`：

```
MPI_Win_fence(0,the_window);
int
  counter_value;
if (i_am_available) {
  int
    decrement = -1;
  total_decrement++;
  MPI_Fetch_and_op
    ( /* operate with data from origin: */   &decrement,
      /* retrieve data from target:     */   &counter_value,
      MPI_INT, counter_process, 0, MPI_SUM,
      the_window);
}
MPI_Win_fence(0,the_window);
if (i_am_available) {
  my_counter_values[n_my_counter_values++] = counter_value;
}
```

现在，如果有多个访问，每个访问都在一个原子且不可分割的操作中检索计数器值并更新它。

图 9.13 MPI_Win_lock

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Win_lock (				
lock_type	either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED	int	INTEGER	IN
rank	rank of locked window	int	INTEGER	IN
assert	program assertion	int	INTEGER	IN
win	window object	MPI_Win	TYPE(MPI_Win)	IN
)				

Python:

```
MPI.Win.Lock(self,int rank, int lock_type=LOCK_EXCLUSIVE,
 int assertion=0)
```

9.4 Passivetarget synchronization

在 *passive target synchronization* 中，只有 origin 是主动参与的：target 不进行任何同步调用。这意味着 origin 进程远程锁定 target 上的窗口，执行单边传输，然后通过解锁再次释放该窗口。

在访问时期，也称为被动目标时期（在这种情况下，“暴露时期”的概念在被动目标同步中没有意义），进程可以启动并完成一次单边传输。通常它会使用 `MPI_Win_lock`（图 9.13）锁定窗口：

```
if (rank == 0) {
  MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);
  MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
  MPI_Win_unlock (1, win);
}
```

备注 22 对于未由 *MPI_Alloc_mem*（可能是内部）创建的窗口，即除 *MPI_Win_create* 之外的所有窗口，锁定窗口的可能性不被保证。

9.4.1 Lock types

启动一个访问时期需要一个锁，也就是说，origin 需要获得访问 target 的能力。你可以使用 `MPI_Win_lock` 在特定进程上获取锁，或者使用 `MPI_Win_lock_all` 在所有进程（在一个通信器中）上获取锁。与 `MPI_Win_fence` 不同，这不是一个集合调用。此外，可以通过 `MPI_Win_lock` 同时拥有多个活动的访问时期。

两种锁类型是：

- `MPI_LOCK_SHARED`：多个进程可以访问同一等级上的窗口。如果多个进程执行 `MPI_Get` 调用，则没有问题；但对于 `MPI_Put` 和类似调用，则存在一致性问题；见下文。
- `MPI_LOCK_EXCLUSIVE`：一个 origin 对某个目标上的窗口获得独占访问权限。与 sharedlock 不同，这没有一致性问题。

你只能在 `MPI_Win_lock` 中指定锁类型；`MPI_Win_lock_all` 始终是 shared。

图 9.14 MPI_Win_unlock

名称	参数名	说明	C 类型	F 类型	i nout
MPI_Win_unlock (					
	rank	rank of window	int	INTEGER	IN
	win	window object	MPI_Win	TYPE(MPI_Win)	IN
	)				

图 9.15 MPI_Win_lock_all

名称	参数名	说明	C 类型	F 类型	i nout
MPI_Win_lock_all (					
	assert	program assertion	int	INTEGER	IN
	win	window object	MPI_Win	TYPE(MPI_Win)	IN
	)				

要解锁一个窗口，使用 MPI_Win_unlock（图 9.14），分别 MPI_Win_unlock_all。

练习 9.7. 使用被动目标同步调查原子更新。使用 MPI_Win_lock 和排他锁，这意味着每个进程只有在绝对必要时才获取锁。

- 除了一个进程外，所有进程都更新一个窗口：

```
int one=1;
MPI_Fetch_and_op(&one, &readout,MPI_INT, repo,
 zero_disp, MPI_SUM,the_win);
```

- 而剩余的进程则自旋，直到其他进程完成它们的操作更新。

对于后一个过程，使用原子操作来读取共享值。你能用共享锁替换独占锁吗？（此练习的骨架名为 lockfetch。）

练习 9.8. 如练习 9.7，但现在使用共享锁：所有进程同时获取锁，并在需要时保持锁。这里的问题是，窗口缓冲区和本地变量之间的一致性现在不再由栅栏或释放锁来强制。使用 MPI_Win_flush_local 来强制窗口（在另一个进程上）和本地变量之间的一致性，来自 MPI_Fetch_and_op。（此练习的骨架名为 lockfetchshared。）

9.4.2 锁定所有

要锁定窗口组中所有进程的窗口，使用 MPI_Win_lock_all（图 9.15）。这不是一个集合调用：“所有”部分指的是一个进程正在锁定所有进程的窗口。

图 9.16 MPI_Win_flush_local

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Win_flush_local (					
	rank	rank of target window	int	INTEGER	IN
	win	window object	MPI_Win	TYPE(MPI_Win)	IN
	)				

- 断言值可以是零，或者 MPI_MODE_NOCHECK，这断言没有其他进程会获取竞争锁。
- 没有“locktype”参数：这是一个共享锁。

The corresponding unlock is MPI_Win_unlock_all.

T‘lock/unlock all’ 的预期用法是它们包围一个包含 get/put 和 flush 的扩展时期 calls.

9.4.3 被动目标同步中的完成和一致性

在单边传输中，应理清数据的多个实例，以及影响它们一致性的各种完成。

- 用户数据。这是传递给 RMA 调用的缓冲区。例如，在一次 MPI_Put 调用之后，但仍处于访问时期，用户缓冲区尚不安全重用。确保缓冲区已被传输称为本地完成。
- 窗口数据。虽然这可能是公开可访问的，但不一定总是与内部副本保持一致。
- 远程数据。即使 MPI_Put 成功，也不能保证另一个进程已经接收到数据。成功的传输是一个远程完成。

如前所述，RMA 操作是非阻塞的，因此我们需要机制来确保操作已完成，并确保用户和窗口数据的一致性。

被动目标时期中 RMA 操作的完成通过 MPI_Win_unlock 或 MPI_Win_unlock_all 来保证，类似于主动目标同步中 MPI_Win_fence 的使用。

如果被动目标时期持续时间较长，且未使用 unlock 操作来确保完成，则可使用以下调用。

备注 23 在主动目标同步中使用 *flush* 例程（或通常在被动目标时期之外）时，您可能会收到一条消息

```
Wrong synchronization of RMA calls
```

9.4.3.1 本地完成

调用 MPI_Win_flush_local (图 9.16) 确保所有针对给定目标的操作在源端完成。例如，对于调用 MPI_Get 或 MPI_Fetch_and_op，本地结果在 MPI_Win_flush_local 之后可用。

图 9.17 MPI_Win_flush

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Win_flush (				
rank	rank of target window	int	INTEGER	IN
win	window object	MPI_Win	TYPE(MPI_Win)	IN
)				

随着 MPI_Win_flush_local_all 本地操作在所有目标上完成。这通常会与 MPI_Win_lock_all (第 9.4.2 节) 一起使用。

9.4.3.2 远程完成

调用 MPI_Win_flush (图 9.17) 和 MPI_Win_flush_all 会完成目标上所有未完成的 RMA 操作，使其他进程能够访问其数据。这对于 MPI_Put 操作非常有用，但也可以用于诸如 MPI_Fetch_and_op 的原子操作。

9.4.3.3 Window synchronization

在独立内存模型下，用户代码可以持有一个与内部窗口数据不一致的缓冲区。调用 MPI_Win_sync 会同步窗口的私有副本和公共副本。

9.5 关于窗口内存的更多内容

9.5.1 内存模型

你可能认为窗口内存与传递给 MPI_Win_create 的缓冲区相同，或者与从 MPI_Win_allocate (章节 9.1.1) 获得的缓冲区相同。这不一定正确，实际情况称为内存模型。有两种内存模型：

- 在统一内存模型下，进程空间中的缓冲区确实是窗口内存，或者至少它们保持一致。这意味着在一个时期完成后，你可以从缓冲区读取窗口内容。为了实现这一点，窗口需要使用 MPI_Win_allocate_shared 创建。该内存模型是 MPI 共享内存所必需的；见第 12 章。
- 在分离内存模型下，进程空间中的缓冲区是私有窗口，而 put/get 操作的目标是公共窗口，两者不同且不保持一致。在该模型下，你需要显式执行 get 操作来读取窗口内容。

您可以使用 MPI_Win_get_attr 调用并带有 MPI_WIN_MODEL 关键字来查询窗口的模型：

```
// window.cint *modelstar,flag;
MPI_Win_get_attr(the_window,MPI_WIN_MODEL,&modelstar,&flag);
int model = *modelstar;if (procno==0)
printf("Window model is unified: %d\n",model==MPI_WIN_UNIFIED);
```

可能的取值有：

- MPI_WIN_SEPARATE,
- MPI_WIN_UNIFIED,

For more on attributes, see section 9.5.5.
The following material is for the recently released MPI-4 standard and may not be supported yet.

9.5.2 分配类型

有时希望指明由 MPI_Win_allocate 或 MPI_Alloc_mem 返回的内存类型，尤其是在 *accelerators* 参与时。此功能在 MPI-4.1 中添加。

MPI-4.1 中有两个 info 键用于查询 / 请求支持，或断言使用，内存分配种类：

- mpi_memory_alloc_kinds：该值是由 MPI 实现支持的内存分配种类的逗号分隔列表。它可以用在输入 MPI_Info 对象的 MPI_Session_init， MPI_Comm_spawn，或 MPI_Comm_spawn_multiple，以请求支持某些分配种类。如果用作查询，它会报告给定会话中支持的分配种类，等等。这可能包括用户未请求的种类。
- mpi_assert_memory_alloc_kinds：通过设置此项，用户告诉 MPI 所有缓冲区（在给定的通信器、会话等上）仅使用所指示类型的内存。

一些内存分配种类的 info 值是预定义的：

- mpi：由 MPI 库分配的内存。
- system：由标准系统分配器返回的内存。
- default: 来自支持的分配器之一的内存。

Info 值可以有限制符：

```
kind1:restrict1,kind2:restrict2
```

Restrict values are:

- alloc_mem
- win_allocate
- win_allocate_shared

MPI-4 材料结束

9.5.3 动态附加内存

在章节 9.1.1 中，我们研究了创建窗口及其内存的简单方法。

也可以创建大小动态设置的窗口。使用 MPI_Win_create_dynamic (图 9.18) 创建动态窗口，并使用 MPI_Win_attach (图 9.19) 将内存附加到窗口。

乍一看，代码看起来像是将一个 MPI_Win_create 调用拆分为窗口的单独创建和缓冲区的声明：

```
// windynamic.c
MPI_Win_create_dynamic(MPI_INFO_NULL,comm,&the_window);
if (procno==data_proc)
  window_buffer = (int*) malloc( 2*sizeof(int) );
MPI_Win_attach(the_window,window_buffer,2*sizeof(int));
```

图 9.18 MPI_Win_create_dynamic

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Win_create_dynamic (				
info	info argument	MPI_Info	TYPE (MPI_Info)	IN
comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
win	window object returned by the call	MPI_Win*	TYPE(MPI_Win)	OUT
)				

图 9.19 MPI_Win_attach

Name 参数名	说明	C 类型	F 类型	输入输 出
MPI_Win_attach (				
win	window object	MPI_Win	TYPE(MPI_Win)	IN
base	initial address of memory to be attached	void*	TYPE(*), DIMENSION(..)	IN
size	size of memory to be attached in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
)				

（其中 *window_buffer* 表示已分配的内存。）

然而，在 RMA 操作中，窗口的寻址方式有一个重要的区别。对于所有其他窗口模型，位移参数是相对于缓冲区起始位置以单位计量的，而这里的位移是一个绝对地址。这意味着我们需要获取窗口缓冲区的地址 `MPI_Get_address` 并将其传达给其他进程：

```
MPI_Aint data_address;
if (procno==data_proc) {
  MPI_Get_address(window_buffer,&data_address);
}
MPI_Bcast(&data_address,1,MPI_AINT,data_proc,comm);
```

数据的位置，也就是位移参数，随后以缓冲区起始的绝对位置加上字节数的方式给出；换句话说，位移单位 是 1。在这个例子中，我们使用 `MPI_Get` 来找到窗口缓冲区中的第二个整数：

```
MPI_Aint disp = data_address+1*sizeof(int);
MPI_Get( /* data on origin: */           retrieve, 1,MPI_INT,
 /* data on target: */ data_proc,disp,     1,MPI_INT,
 the_window);
```

注释。

附加的内存可以通过 `MPI_Win_detach` （图 9.20 ）释放。

- 上述片段显示，源进程拥有窗口缓冲区的实际地址。如果缓冲区未附加到窗口，使用该地址是错误的。

图 9.20 MPI_Win_detach

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Win_detach (				
win	window object	MPI_Win	TYPE(MPI_Win)	IN
base	initial address of memory to be detached	const void*	TYPE(*), DIMENSION(..)	IN
)				

- 特别是，必须确保在对窗口执行 RMA 操作之前，attach 调用已经完成。

9.5.4 Windowusage hints

以下键可以作为 info 参数传递：

- *no_locks*：如果设置为 true，则不会在此窗口上使用被动目标同步（章节 9.4 ）。
- *accumulate_ordering*：一个由逗号分隔的关键字列表 rar, raw, war, waw 可以被指定。这表示对 MPI_Accumulate 或 MPI_Get_accumulate 的读写可以重新排序，但受某些约束限制。
- *accumulate_ops*：值 *same_op* 表示 并发 *Accumulate* 调用使用相同的操作符； *same_op_no_op* 表示相同的操作符或 MPI_NO_OP。

9.5.5 窗口信息

The MPI_Info 参数（参见章节 15.1.1 关于信息对象）可用于传递实现相关的i信息。

A numbe当窗口被创建时，会存储若干属性 eated.

- MPI_WIN_BASE 用于获取指向窗口区域起始位置的指针：

```
void *base;MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag)
```

- MPI_WIN_SIZE and MPI_WIN_DISP_UNIT for obtaining the size and *window displacement unit*:

```
MPI_Aint *size;
MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag),
int *disp_unit;
MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag),
```

- MPI_WIN_CREATE_FLAVOR for determining the type of create call used:

```
int *create_kind;
MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag)
```

  with possible values:
  - MPI_WIN_FLAVOR_CREATE 如果窗口是用 MPI_Win_create;–
    MPI_WIN_FLAVOR_ALLOCATE 如果窗口是用 MPI_Win_allocate;

  - MPI_WIN_FLAVOR_DYNAMIC 如果窗口是用 MPI_Win_create_dynamic 创建的。在这种情况下，基地址是 MPI_BOTTOM 且大小为零；- MPI_WIN_FLAVOR_SHARED 如果窗口是用 MPI_Win_allocate_shared 创建的；

- MPI_WIN_MODEL 用于查询窗口内存模型；参见第 9.5.1 节。

获取与窗口关联的进程组（参见第 7.5 节）：

```
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
```

窗口信息对象（参见第 15.1.1 节）可以被设置和获取：

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
```

```
int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
```

9.6 断言

这些例程

- (Active target synchronization) MPI_Win_fence, MPI_Win_post, MPI_Win_start;
- (Passive target synchronization) MPI_Win_lock, *MPI_Win_lockall*,

通过一个参数传递断言，该断言可以关于 epoch 之前、之后和期间的活动。值 zero 始终被允许，但你可以通过指定以下一个或多个选项（在 C/C++ 中通过按位或组合，或在 Fortran 中使用 IOR ）来使程序更高效。

- MPI_Win_start 支持选项：- MPI_MODE_NOCHECK 匹配

  对 MPI_Win_post 的调用在调用 MPI_Win_start 时，所有目标进程上的调用已经完成。只有当每个匹配的 post 调用中都指定了 nocheck 选项时，start 调用中才能指定 nocheck 选项。这类似于 “ready-send” 的优化，当握手在代码中是隐式的时，可能节省一次握手。（然而，ready-send 是由常规接收匹配的，而 start 和 post 都必须指定 nocheck 选项。）支持以下选项：

- MPI_Win_post
  - MPI_MODE_NOCHECK the matching calls to MPI_Win_start have not yet occurred on any origin processes when the call to MPI_Win_post is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
  - MPI_MODE_NOSTORE the local window was not updated by local stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
  - MPI_MODE_NOPUT 本地窗口在 post 调用之后，直到随后的（ wait ）同步之前，不会被 put 或 accumulate 调用更新。这可能避免在 wait 调用时需要缓存同步。支持以下选项：
- MPI_Win_fence
  - MPI_MODE_NOSTORE the local window was not updated by local stores (or local get or receive calls) since last synchronization.

– MPI_MODE_NOPUT 本地窗口在 fence 调用之后不会被 put 或 accumulate 调用更新，直到随后的 (fence) 同步。– MPI_MODE_NOPRECEDE fence 不会完成任何本地发起的 RMA 调用序列。如果该断言由窗口组中的任何进程给出，则必须由组中所有进程给出。

– MPI_MODE_NOSUCCEED fence 不会启动任何本地发起的 RMA 调用序列。如果该断言由窗口组中的任何进程给出，则必须由组中所有进程给出。

- MPI_Win_lock 并且 MPI_Win_lock_all 支持以下选项：– MPI_MODE_NOCHECK 没有其他进程持有，或将尝试获取冲突锁，

  当调用者持有窗口锁时。这在通过其他方式实现互斥时非常有用，但仍然需要附加到锁定和解锁调用的相干操作。

9.7 实现

你可能想知道单边通信是如何实现的 [1]。处理器能以某种方式访问另一个处理器的数据吗？不幸的是，不能。

主动目标同步是通过双边通信实现的。想象第一个 fence 操作什么也不做，除非它结束了之前的单边操作。Put 和 Get 调用本身不涉及通信，除了标记它们与哪些处理器交换数据。真正发生一切的是结束的 fence：首先一个全局操作确定哪些目标需要发起发送或接收调用，然后执行实际的发送和接收。

练习 9.9. 假设在一个时期内只执行 Get 操作。简述这些操作如何被转换为发送 / 接收对。这里的问题是发送方如何得知它们需要发送。证明你可以用一个 MPI_Reduce_scatter 调用来解决这个问题。

前一段提到需要一个集合操作来确定双边通信流量。由于集合操作会引入一定程度的同步，你可能想要限制这种操作。

练习 9.10. 论证 windowpost/wait/start/complete 操作机制仍然需要集体操作，但这负担较轻。

被动目标同步完全需要另一种机制。这里目标进程需要有一个后台任务（进程、线程、守护进程等）运行，监听锁定窗口的请求。这可能会比较昂贵。

1. 关于此主题的更多内容，请参见 [27].

9.8 复习问题

找出此代码中的所有错误。

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define MASTER 0

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  int r, p;
  MPI_Comm_rank(comm, &r);
  MPI_Comm_size(comm, &p);
  printf("Hello from %d\n", r);
  int result[1] = {0};
  //int assert = MPI_MODE_NOCHECK;
  int assert = 0;
  int one = 1;
  MPI_Win win_res;
  MPI_Win_allocate(1 * sizeof(MPI_INT), sizeof(MPI_INT), MPI_INFO_NULL, comm, &result[0],
    ↪&win_res);
  MPI_Win_lock_all(assert, win_res);
  if (r == MASTER) {
    result[0] = 0;
    do{
      MPI_Fetch_and_op(&result, &result , MPI_INT, r, 0, MPI_NO_OP, win_res);
      printf("result: %d\n", result[0]);
    } while(result[0] != 4);
    printf("Master is done!\n");
  } else {
    MPI_Fetch_and_op(&one, &result, MPI_INT, 0, 0, MPI_SUM, win_res);
  }
  MPI_Win_unlock_all(win_res);
  MPI_Win_free(&win_res);
  MPI_Finalize();
  return 0;
```

第 10 章

MPI 主题：文件 I/O

本章讨论了 MPI 的 I/O 支持，旨在缓解并行文件访问中固有的问题。让我们先探讨这些问题。这个故事部分取决于你运行的是哪种并行计算机。以下是你可能遇到的一些硬件场景：

- 在工作站网络中，每个节点将拥有一个带有自己文件系统的独立驱动器。
- 在许多集群中，会有一个共享文件系统，它表现得好像每个进程都可以访问每个文件。
- 集群节点可能有也可能没有私有文件系统。

Based on this, the following strategies are possible, even before we start talking about MPI I/O.

- 一个进程可以使用 `MPI_Gather` 收集所有数据并写出。这样做至少有三个问题：它使用了网络带宽进行收集，可能需要根进程大量内存，并且集中写入是一个瓶颈。
- 在没有共享文件系统的情况下，写操作可以通过让每个进程创建一个唯一的文件并在运行后合并这些文件来实现并行。这使得 I/O 对称，但收集所有文件是一个瓶颈。
- 即使有共享文件系统，这种方法也是可行的，但它可能会给文件系统带来很大压力，且后处理可能是一项重要任务。
- 使用共享文件系统时，每个进程打开同一个已存在的文件进行读取，并使用各自的文件指针获取其唯一数据，是没有问题的。
- …… 但让每个进程打开同一个文件进行输出可能不是一个好主意。例如，如果两个进程尝试写入文件末尾，可能需要对它们进行同步，并同步文件系统的刷新。

出于这些原因，MPI 提供了许多例程，使得可以从大量进程中读取和写入单个文件，为每个进程分配其自己的明确定义的数据访问位置。这些位置可以使用 MPI *deriveddatatype* 来表示源数据（即内存中的数据）和目标数据（即磁盘上的数据）。因此，在一次对通信子进行的集合调用中，每个进程可以访问内存中不连续的数据，并将其放置在磁盘上不连续的位置。

有专门用于文件 I/O 的库，如 *hdf5*、 *netcdf* 或 *silo*。然而，这些库通常会向文件添加头信息，可能无法被后处理应用程序理解。使用 MPI I/O，您可以完全控制写入文件的内容。（查看文件的一个有用工具是 unix 实用程序 `od`。）

图 10.1 MPI_File_open

名称 参数名	说明	C 类型	F 类型	i nout
MPI_File_open (				
comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
filename	name of file to open	const char*	CHARACTER	IN
amode	file access mode	int	INTEGER	IN
info	info object	MPI_Info	TYPE (MPI_Info)	IN
fh	new file handle	MPI_File*	TYPE (MPI_File)	OUT
)				

Python:

```
Open(type cls, Intracomm comm, filename,
int amode=MODE_RDONLY, Info info=INFO_NULL)
```

TACC 注释。每个节点都有一个私有的 `/tmp` 文件系统（通常是闪存存储），你可以向其写入文件。注意事项：

- 由于这些驱动器与共享文件系统分开，你无需担心文件服务器的压力。
- 这些临时文件系统在你的作业完成后会被清除，因此你必须在作业脚本中进行后处理。
- 这些本地驱动器的容量相当有限；具体数字请参见用户指南。

10.1 文件处理

MPI 有一个用于文件的数据类型：`MPI_File`。这有点像传统的文件句柄，因为它有打开、关闭、读 / 写和定位操作。然而，与传统文件处理不同，传统文件处理在并行中意味着每个进程有一个句柄，而这个句柄是集体的：MPI 进程表现得好像共享一个文件句柄。

你可以用 `MPI_File_open`（图 10.1）打开文件。这个例程是集体的，即使只有某些进程会通过读或写调用访问文件。同样，`MPI_File_close` 也是集体的。

MPL 注释 *63*：文件打开。 文件是类型为 *mpl::file* 的对象。你可以使用默认构造函数和 *open* 方法，或者在构造函数中打开文件：

```
// filewrite.cxx
mpl::file mpifile
  (comm_world,"filewrite.dat",
   mpl::file::access_mode::create | mpl::file::access_mode::write_only
   );
mpifile.close();
```

- 打开失败会抛出一个 *mpl::io_failure* 异常；该异常的 *what* 方法会返回一个错误字符串。复制构造函数和复制赋值构造函数已被删除。

Python note 30: File open is class method. 注意打开文件时稍显不同寻常的语法：

```
mpifile = MPI.File.Open(comm,filename,mode)
```

> 尽管文件是在一个 communicator 上打开的，但它是 `MPI.File` 类的类方法，而不是 communicator 对象的方法。后者作为参数传入。

文件访问模式：

- `MPI_MODE_RDONLY`: 只读，
- `MPI_MODE_RDWR`: 读写，
- `MPI_MODE_WRONLY`: 仅写，
- `MPI_MODE_CREATE`: 如果文件不存在则创建，
- `MPI_MODE_EXCL`: 如果文件已存在则报错，
- `MPI_MODE_DELETE_ON_CLOSE`: 关闭时删除文件，
- `MPI_MODE_UNIQUE_OPEN`: 文件不会被 *concurrently* 在其他地方打开，
- `MPI_MODE_SEQUENTIAL`: 文件将只被顺序访问，
- `MPI_MODE_APPEND`: 将所有文件指针的初始位置设置为文件末尾。

这些模式可以相加或按位或。

作为一个小示例：**代码：**

```
// filewrite.c
MPI_File mpifile;
MPI_File_open
  (comm,"filewrite.dat",
   MPI_MODE_CREATE | MPI_MODE_WRONLY,MPI_INFO_NULL,
   &mpifile);
MPI_File_write_at
  (mpifile,/* offset: */ procno*sizeof(int),
   &procno,1, MPI_INT,MPI_STATUS_IGNORE);
MPI_File_close(&mpifile);
```

Output:

```
Finished: all 4 correct
octal dump:
0000000    000000  000000
    ↪000001  000000  000002
    ↪ 000000  000003  000000
0000020
```

你可以用 `MPI_File_delete` 删除一个文件。

缓冲区可以通过 `MPI_File_sync` 刷新，这是一个集合调用。

10.2 文件读写

基本的文件操作，在 open 和 close 调用之间，是类似 POSIX 的非集合调用

- `MPI_File_seek` (图 10.2)。 *whence* 参数可以是：– `MPI_SEEK_SET` 指针被设置为 offset。– `MPI_SEEK_CUR` 指针被设置为当前位置加上 offset。– `MPI_SEEK_END` 指针被设置为文件末尾加上 offset。
- `MPI_File_write` (图 10.3)。此例程将指定数据写入指定位置
  使用 currentfile 视图。写入的项目数量通过 `MPI_Status` 参数返回；该参数的所有其他字段未定义。如果文件是用 `MPI_MODE_SEQUENTIAL` 打开的，则不能使用它。

图 10.2 MPI_File_seek

名称 参数名	说明	C 类型	F 类型	i nout
MPI_File_seek (				
fh	file handle	MPI_File	TYPE (MPI_File)	INOUT
offset	file offset	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)	IN
whence	update mode	int	INTEGER	IN
)				

图 10.3 MPI_File_write

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_File_write (				
MPI_File_write_c (				
fh	file handle	MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in buffer	int MPI_Count	INTEGER	IN
datatype	datatype of each buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

- 如果所有进程在相同的逻辑时间执行写操作，最好使用集体调用 MPI_File_write_all。
- MPI_File_read (图 10.4) 此例程尝试从指定位置读取指定数据
  在当前文件视图中指定。读取的项目数量返回在 MPI_Status 参数中；该参数的所有其他字段未定义。如果文件是用 MPI_MODE_SEQUENTIAL 打开的，则不能使用此函数。
- 如果所有进程在相同的逻辑时间执行读操作，最好使用集体调用 MPI_File_read_all(图 10.5)。

为了线程安全，最好将定位和读 / 写操作结合起来：

- MPI_File_read_at: 结合读和定位。集体变体是 MPI_File_read_at_all。
- MPI_File_write_at: 结合写和定位。集体变体是 MPI_File_write_at_all; sec-tion 10.2.2.

向并行文件写入和从并行文件读取与发送和接收非常相似：

- 进程使用预定义数据类型或派生数据类型来描述数组中哪些元素写入文件，或从文件读取。
- 在最简单的情况下，您通过使用偏移量读取或写入文件中的数据，或者先执行一次 seek 操作。

图 10.4 MPI_File_read

名称 参数名	Explanation	C 类型	F 类型	输入输出
MPI_File_read (				
MPI_File_read_c (				
fh	file handle	MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in buffer	int / MPI_Count	INTEGER	IN
datatype	datatype of each buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

图 10.5 MPI_File_read_all

名称 参数名 说明		C 类型	F 类型	输入输出
MPI_File_read_all (				
MPI_File_read_all_c (				
fh	file handle	MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer	void*	TYPE(*), DIMENSION(..)	OUT
count	number of elements in buffer	int / MPI_Count	INTEGER	IN
datatype	datatype of each buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

- 但你也可以设置一个 ‘file view’ 来明确描述文件中将涉及的元素。

MPL note 64: File writing. Routines with the obvious names exist:

```
mpifile.write_at
  ( /* offset: */ procno*sizeof(int),
    /* data: */ procno );
```

此外 *read*, *write*, *iread*, *iwrite*, *read_at*, *write_at*, *iread_at*, *iwrite_at*, *read_all*, *write_all*, *iread_all*, *iwrite_all*,*read_ordered*, *write_ordered*, *iread_ordered*, *iwrite_ordered*, *read_shared*, *write_shared*,*iread_shared*, *iwrite_shared*, *read_at_all*, *write_at_all*, *iread_at_all*, *iwrite_at_all*, *read_all_begin*, *write_all_begin*, *read_all_end*, *write_all_end*, *read_at_all_begin*, *write_at_all_begin*, *read_at_all_end*, *write_at_all_end*, *read_ordered_begin*, *write_ordered_begin*, *read_ordered_end*, *write_ordered_end*,

图 10.6 MPI_File_iwrite

名称 参数名	说明	C 类型	F 类型	i nout
MPI_File_iwrite (				
MPI_File_iwrite_c (				
fh	file handle	MPI_File	TYPE (MPI_File)	INOUT
buf	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in buffer	[int MPI_Count	INTEGER	IN
datatype	datatype of each buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
request	request object	MPI_Request*	TYPE (MPI_Request)	OUT
)				

10.2.1 非阻塞读 / 写

就像有阻塞和非阻塞发送一样，也有非阻塞写和读

eads: MPI_File_iwrite（图 10.6），MPI_File_iread 操作及其集合版本 MPI_File_iwrite_all, MPI_File_iread_all.

同样是 MPI_File_iwrite_at, MPI_File_iwrite_at_all, MPI_File_iread_at., MPI_File_iread_at_all.

这些例程输出一个 MPI_Request 对象，然后可以用 MPI_Wait 或 MPI_Test 进行测试。

非阻塞集合 I/O 函数类似于其他非阻塞集合（第 3.11 节）：当所有进程完成集合操作时，请求被满足。

还有 分离集合，其功能类似于非阻塞集合 I/O，但带有请求 / 等待机制：

MPI_File_write_all_begin / MPI_File_write_all_end（类似地 MPI_File_read_all_begin / MPI_File_read_all_end），其中第二个例程会阻塞，直到集合写入 / 读取完成。

同样 MPI_File_iread_shared, MPI_File_iwrite_shared。

10.2.2 单独文件指针，连续写入

集合打开调用后，每个进程持有一个单独文件指针，它可以单独定位到共享文件中的某个位置。让我们探讨这种模式。

写数据到文件的最简单方法类似于发送调用：指定一个缓冲区，使用常规的 count/datatype 规格，并给出文件中的目标位置。例程 MPI_File_write_at（图 10.7）以绝对方式给出该位置，参数类型为 MPI_Offset，其以字节为单位计数。

练习 10.1. 在每个进程上创建一个长度为 nwords=3 的缓冲区，并使用 MPI_File_write_at 将这些缓冲区作为序列写入一个文件。（该练习的骨架代码名为 blockwrite。）

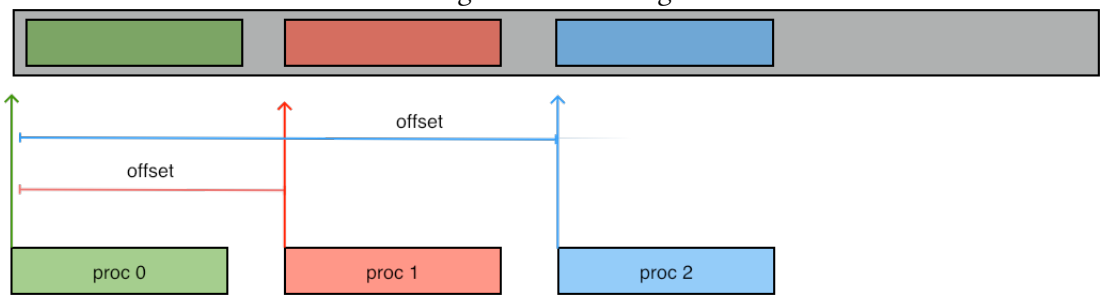
图 10.7 MPI_File_write_at

名称 参数名 说明		C 类型	F 类型	输入输出
MPI_File_write_at (				
MPI_File_write_at_c (				
fh	file handle	MPI_File	TYPE (MPI_File)	INOUT
offset	file offset	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)	IN
buf	initial address of buffer	const void*	TYPE(*), DIMENSION(..)	IN
count	number of elements in buffer	int / MPI_Count	INTEGER	IN
datatype	datatype of each buffer element	MPI_Datatype	TYPE (MPI_Datatype)	IN
status	status object	MPI_Status*	TYPE (MPI_Status)	OUT
)				

Python:

```
MPI.File.Write_at(self, Offset offset, buf, Status status=None)
```

Figure 10.1: Writing at an offset



你也可以使用 MPI_File_seek 调用来定位文件指针，并使用 MPI_File_write 在指针位置写入。写调用本身也会移动文件指针，因此写入连续元素的单独调用不需要使用 MPI_SEEK_CUR 进行寻址调用。

练习 10.2. 重写练习 10.1 的代码，使用一个循环，每次迭代只写入一个项目到文件。注意不需要显式地移动文件指针。

练习 10.3. 构造一个包含连续整数 $0, \ldots, WP$ 的文件，其中 W 是某个整数，P 是进程数。每个进程 p 写入数字 $p, p+W, p+2W, \ldots$。使用一个循环，每次迭代

1. 使用 MPI_File_write 写入一个数字，2. 使用 MPI_File_seek 移动文件指针，参数为 *whence*，值为 MPI_SEEK_CUR。

图 10.8 MPI_File_set_view

名称 参数名	说明	C 类型	F 类型	i nout
MPI_File_set_view (				
fh	file handle	MPI_File	TYPE (MPI_File)	INOUT
disp	displacement	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)	IN
etype	elementary datatype	MPI_Datatype	TYPE (MPI_Datatype)	IN
filetype	filetype	MPI_Datatype	TYPE (MPI_Datatype)	IN
datarep	data representation	const char*	CHARACTER	IN
info	info object	MPI_Info	TYPE (MPI_Info)	IN
)				

Python:

```
mpifile = MPI.File.Open( .... )mpifile.Set_view(self,
Offset disp=0, Datatype etype=None, Datatype filetype=None,
datarep=None, Info info=INFO_NULL)
```

10.2.3 文件视图

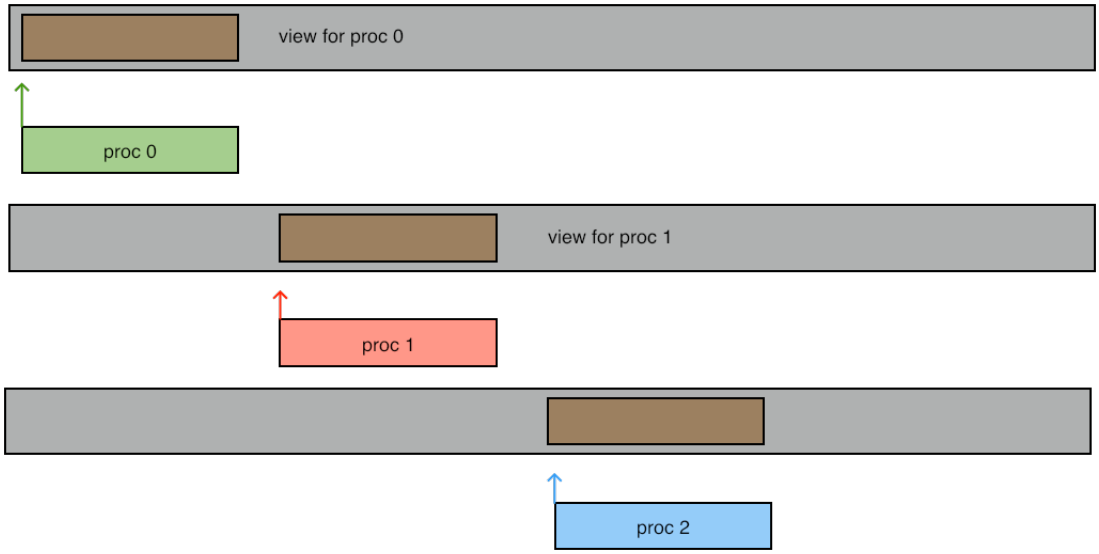
之前的写法足以写入文件中的简单连续块。然而，你也可以访问文件中的非连续区域。为此，你可以使用 `MPI_File_set_view`（图 10.8）。该调用是集体的，即使并非所有进程都访问文件。

- 位移参数以字节为单位。它在不同进程间可以不同。在顺序文件如磁带或网络流中，设置位移没有意义；对于这些，`MPI_DISPLACEMENT_CURRENT` 值可以被使用。
- `etype` 描述文件的数据类型，所有进程上需要保持一致。
- `filetype` 描述该进程如何查看文件，因此在不同进程间可以不同。
- `datarep` 字符串可以具有以下值：– `native`：磁盘上的数据以与内存中完全相同的格式表示；– `internal`：磁盘上的数据以 MPI 实现所使用的任何内部格式表示；– `external`：磁盘上的数据使用 XDR 可移植数据格式表示。
- The `info` 参数是一个 `MPI_Info` 对象，或 `MPI_INFO_NULL`。详见章节 15.1.1.3 了解更多文件信息。（参见 *T3PIO* [21]，这是一个辅助设置该对象的工具。）

```
// scatterwrite.c
MPI_File_set_view
  (mpifile,
   offset,MPI_INT,scattertype,
   "native",MPI_INFO_NULL);
```

Exercise 10.4.

图 10.2：在视图上写入



（本练习有一个骨架，名为 `viewwrite`。）以与练习 10.1 相同的方式写文件，但现在使用 `MPI_File_write` 并使用 `MPI_File_set_view` 设置一个视图来确定数据写入的位置。

通过将视图设置为派生数据类型，可以获得非常有创意的效果。

Fortran 注释 14：偏移量字面量。 在 Fortran 中，必须确保位移参数的 ‘kind’ 是 `MPI_OFFSET_KIND`。特别是，不能将字面量零 ‘0’ 指定为位移；应使用 `0_MPI_OFFSET_KIND` 代替。

更多： `MPI_File_set_size`, `MPI_File_get_size` `MPI_File_preallocate`, `MPI_File_get_view`。

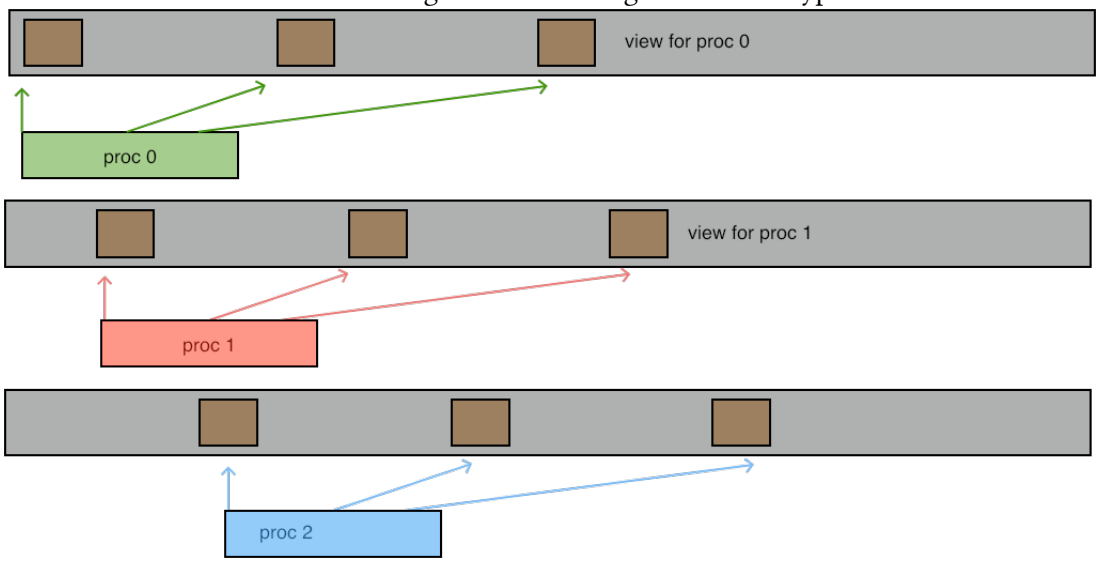
10.2.4 共享文件指针

可以有一个文件指针在用于打开文件的通信子中的所有进程之间共享（因此是相同的）。该文件指针由 `MPI_File_seek_shared` 设置。对于读写操作，有两组例程：

- 单独访问使用 `MPI_File_read_shared` 和 `MPI_File_write_shared` 完成。非阻塞变体是 `MPI_File_iread_shared` 和 `MPI_File_iwrite_shared`。
- 集体访问使用 `MPI_File_read_ordered` 和 `MPI_File_write_ordered` 完成，这些操作按秩的升序执行。

共享文件指针要求所有进程使用相同的视图。此外，由于需要维护共享指针，这些操作的效率较低。

Figure 10.3: Writing at a derived type



10.3 一致性

一个进程可以读取另一个进程之前写入的数据。为此，当然需要强制时间顺序，例如使用 `MPI_Barrier`，或者使用从写进程到读进程的零字节发送。

然而，文件也需要声明为 *atomic*： `MPI_File_set_atomicity`。

10.4 常量

`MPI_SEEK_SET` 曾被称为 `SEEK_SET`，这与 C++ 库产生了冲突。必须通过

```
make"CPPFLAGS=-DMPICH_IGNORE_CXX_SEEK -DMPICH_SKIP_MPICXX
```

和类似的。

10.5 错误处理

默认情况下，MPI 使用 `MPI_ERRORS_ARE_FATAL`，因为并行错误几乎不可能恢复。另一方面，文件处理错误不那么严重：如果找不到文件，可以放弃该操作。因此，文件操作的默认错误处理程序是 `MPI_ERRORS_RETURN`。

默认的 I/O 错误处理程序可以通过 `MPI_File_get_errhandler` 和 `MPI_File_set_errhandler` 分别查询和设置，传递 `MPI_FILE_NULL` 作为参数。

10.6 复习问题

Exercise 10.5. 对 / 错？在你的 *SLURM* 作业结束后，你可以从登录节点复制你写入到 `\tmp` 的文件。

Exercise 10.6. 对 / 错？文件视图（ `MPI_File_set_view` ）的目的是

- 将 MPI 派生类型写入文件；没有它们你只能写连续的缓冲区；
- 防止集体写入中的冲突；它们对单独写入不是必需的。

Exercise 10.7. 如果进行哪些更改，序列 `MPI_File_seek_shared`, `MPI_File_read_shared` 可以被替换为 `MPI_File_seek`, `MPI_File_read` ？

第 11 章

MPI 主题：拓扑结构

一个通信器描述了一组进程，但你的计算结构可能并不是每个进程都会与每个其他进程通信。例如，在一个数学上定义在笛卡尔二维网格上的计算中，进程本身表现得像是二维有序的，并与南 / 北 / 东 / 西的邻居通信。如果 MPI 了解你的应用，它可能会针对这一点进行优化，例如通过重新编号 rank，使得通信的进程在你的集群中物理上更接近。

向 MPI 声明计算结构的机制称为虚拟拓扑。定义了以下类型的拓扑：

- `MPI_UNDEFINED`: 该值适用于未显式指定拓扑的通信器。
- `MPI_CART`: 该值适用于笛卡尔拓扑，其中进程表现得像是有序的在多维 ‘砖块’ 中；参见章节 11.1.
- `MPI_GRAPH`: 该值描述了 MPI-1 中定义的图拓扑；章节 11.2.4。这不必要地增加了负担，因为每个进程都需要知道整个图，因此应被视为过时；应使用类型 `MPI_DIST_GRAPH` 。
- `MPI_DIST_GRAPH`: 该值描述了分布式图拓扑，其中每个进程只描述与自身相连的进程图中的边；参见章节 11.2。

这些值可以通过例程 `MPI_Topo_test` 发现。

11.1 笛卡尔网格拓扑

一个 *Cartesian grid* 是一个结构，通常是二维或三维的点阵结构，每个维度上每个点都有两个邻居。因此，如果一个 Cartesian grid 的尺寸是 $K \times M \times N$，它的点的坐标是 (k,m,n)，依此类推 $0 \leq k < K$。大多数点有六个邻居 $(k \pm 1,m,n), (k,m \pm 1,n), (k,m,n \pm 1)$；例外的是边缘点。一个边缘点通过 *wraparound connections* 连接的网格称为 *periodic grid*。

MPI 有一个 ‘Cartesian communicator’ 构造（即类型为 `MPI_CART` 的通信器；见上文），用于使进程不仅按其秩线性组织，而且仿佛它们被组织在一个 Cartesian grid 中。

MPL 注释 *65*：笛卡尔通信器。 有一个单独的类 `cartesian_communicator`。

此外，还有一个类 `dimensions` 描述笛卡尔网格的形状及其周期性。 `size(int)` 方法检索给定维度的大小。

辅助例程 `MPI_Dims_create` 帮助寻找给定维度的网格，尝试最小化直径。 - 最小化直径。

Code:

```
// cartdims.c
int *dimensions = (int*)
    ↪malloc(dim*sizeof(int));
for (int idim=0; idim<dim; idim++)
  dimensions[idim] = 0;
MPI_Dims_create(nprocs,dim,dimensions);
```

Output:

```
mpicc -o cartdims cartdims.o
Cartesian grid size: 3 dim: 1
 3
Cartesian grid size: 3 dim: 2
 3 x  1
Cartesian grid size: 4 dim: 1
 4
Cartesian grid size: 4 dim: 2
 2 x  2
Cartesian grid size: 4 dim: 3
 2 x  2 x  1
Cartesian grid size: 12 dim: 1
12
Cartesian grid size: 12 dim: 2
 4 x  3
Cartesian grid size: 12 dim: 3
 3 x  2 x  2
Cartesian grid size: 12 dim: 4
 3 x  2 x  2 x  1
```

I如果组件中的维度数组非零，则该组件不被修改。当然，
s指定的维度必须能整除输入的节点数。

MPL note 66: Dims create. `dims_create` 例程接收一个仅指定了维度的 `dimensions` 对象，并创建一个填充了尺寸的对象。

```
mpl::cartesian_communicator::dimensions brick(3);
brick = mpl::dims_create(nprocs,brick);
```

要使某些维度具有周期性，初始的 `dimensions` 对象需要用周期值 `periodic` 或 `non_periodic` 创建。

```
mpl::cartesian_communicator::dimensions pbrick
  ( { mpl::cartesian_communicator::non_periodic,
      mpl::cartesian_communicator::periodic,
      mpl::cartesian_communicator::non_periodi                c } );
pbrick = mpl::dims_create(nprocs,pbrick);
```

11.1.1 Cartesian topology communicator

笛卡尔拓扑通过给出 `MPI_Cart_create` (图 11.1) 沿每个轴的处理器网格大小，以及该轴上网格是否是周期性的来指定。

图 11.1 MPI_Cart_create

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Cart_create (					
	comm_old	input communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	ndims	number of dimensions of Cartesian grid	int	INTEGER	IN
	dims	integer array of size ndims specifying the number of processes in each dimension	const int[]	INTEGER (ndims)	IN
	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension	const int[]	LOGICAL (ndims)	IN
	reorder	ranking may be reordered (true) or not (false)	int	LOGICAL	IN
	comm_cart	communicator with new Cartesian topology	MPI_Comm*	TYPE (MPI_Comm)	OUT
	)				

图 11.2 MPI_Topo_test

Name	参数名	说明	C type	F type	inout
MPI_Topo_test (					
	comm	communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	status	topology type of communicator comm	int*	INTEGER	OUT
	)				

```
MPI_Comm cart_comm;
int *periods = (int*) malloc(dim*sizeof(int));
for ( int id=0; id<dim; id++ ) periods[id] = 0;
MPI_Cart_create( comm,dim,dimensions,periods,
0,&cart_comm );
```

(笛卡尔网格的进程数可以少于输入通信器：任何未包含的进程将作为输出得到 MPI_COMM_NULL。)

MPLnote67:Cartesian communicatorcreate. 实际的 Cartesian communicator 有一个构造函数，接受一个 dimensions 对象作为输入。

```
mpl::cartesian_communicator cart_comm( comm_world,brick );
```

For a gi在通信器中，您可以使用 M 测试它的类型。 PI_Topo_test (figure 11.2):

```
int world_type,cart_type;
```

图 11.3 MPI_Cart_coords

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Cart_coords (				
comm	communicator with Cartesian structure	MPI_Comm	TYPE (MPI_Comm)	IN
rank	rank of a process within group of comm	int	INTEGER	IN
maxdims	length of vector coords in the calling program	int	INTEGER	IN
coords	integer array (of size maxdims) containing the Cartesian coordinates of specified process	int[]	INTEGER (maxdims)	OUT
)				

```
MPI_Topo_test( comm,&world_type);
MPI_Topo_test( cart_comm,&cart_type );
if (procno==0) {
  printf("World comm type=%d, Cart comm type=%d\n",
         world_type,cart_type);
  printf("no topo        =%d, cart top      =%d\n",
         MPI_UNDEFINED,MPI_CART);
}
```

对于笛卡尔通信器，您可以使用 MPI_Cartdim_get 和 MPI_Cart_get 来获取其信息：

```
int dim;
MPI_Cartdim_get( cart_comm,&dim );
int *dimensions = (int*) malloc(dim*sizeof(int));
int *periods    = (int*) malloc(dim*sizeof(int));
int *coords     = (int*) malloc(dim*sizeof(int));
MPI_Cart_get( cart_comm,dim,dimensions,periods,coords );
```

MPL 注释 *68*：获取维度对象。 可以从通信器中提取 *dimensions* 对象

```
mpl::cartesian_communicator::dimensions
  dimensions = cart_comm.get_dimensions();
```

之后可以提取维度和周期性：

```
// cartcoord.cxx
int dsize = dimensions.size(idim);
auto p = dimensions.periodicity(idim);
```

11.1.2 Cartesian vs worldrank

笛卡尔通信器中的每个点都有一个坐标和一个秩。秩到笛卡尔坐标的转换由 MPI_Cart_coords (图 11.3) 完成，坐标到秩的转换由 MPI_Cart_rank (图 11.4) 完成。在这两种情况下，这种转换都可以在任何进程上完成；对于后者

图 11.4 MPI_Cart_rank

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Cart_rank (				
comm	communicator with Cartesian structure	MPI_Comm	TYPE (MPI_Comm)	IN
coords	integer array (of size ndims) specifying the Cartesian coordinates of a process	const int[]	INTEGER(*)	IN
rank	rank of specified process	int*	INTEGER	OUT
)				

例程注意，如果网格在出错的坐标上不是周期性的，则笛卡尔网格外的坐标是错误的。

```
// cart.cMPI_Comm comm2d;
int periodic[ndim]; periodic[0] = periodic[1] = 0;
MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
if (comm2d==MPI_COMM_NULL) {
printf("Process %d not included\n",procno);} else {
MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
printf("I am %d: (%d,%d); originally %d\n",rank_2d,
coord_2d[0],coord_2d[1],procno);
```

The reorder 参数到 MPI_Cart_create 指示进程在新通信器中的秩是否可以与旧通信器中不同。

MPL note 69: Rank to coord translation. cartesiancommunicator 的 coordinates 方法返回一个类似向量的对象，描述进程坐标：

```
for ( int ip=0; ip<nprocs; ip++ ) {
  mpl::cartesian_communicator::vector
        coord = cart_comm.coordinates(ip);
  print("[{:2}] coord: [",ip);
  for ( int id=0; id<dim; id++ )
        print("{},",coord[id]);
  print("]\n");
```

11.1.3 笛卡尔通信

笛卡尔网格中一个常见的通信模式是对沿一个坐标轴相邻的进程进行 MPI_Sendrecv 。

举例来说，考虑一个在第一维度上是周期性的三维网格 :

```
// cartcoord.c
for ( int id=0; id<dim; id++)
```

```
    periods[id] = id==0 ? 1 : 0;
MPI_Cart_create
( comm,dim,dimensions,periods,0,
&period_comm );
```

我们在维度 0 和 1 上移动进程 0。在维度 0 中，我们得到一个环绕的源，以及一个作为行优先顺序中下一个进程的目标；在维度 1 中，我们得到 `MPI_PROC_NULL` 作为源，以及一个合法的目标。

Code:

```
int pred,succ;
MPI_Cart_shift
        (period_comm,/* dim: */ 0,/* up: */ 1,
         &pred,&succ);
printf("periodic dimension 0:\n  src=%d, tgt=%d\n",
       pred,succ);
MPI_Cart_shift
        (period_comm,/* dim: */ 1,/* up: */ 1,
         &pred,&succ);
printf("non-periodic dimension 1:\n  src=%d, tgt=%d\n",
       pred,succ);
```

Output:

```
Grid of size 6 in 3
    ↪dimensions:
   3 x  2 x  1
Shifting process 0.
periodic dimension 0:
  src=4, tgt=2
non-periodic dimension 1:
  src=-1, tgt=1
```

MPLnote 70: Cartesian shifting. 例程 `cartesian_communicator::shift` 接受一个维度和一个方向，并将源和目标作为一个 `shifted_ranks` 对象，基本上是两个整数的元组：

```
int pred,succ;mpl::shift_ranks shifted = cart_comm.shift
( /* dim: */ 1,/* up: */ 1 );
pred = shifted.source; succ = shifted.destination;
print("non-periodic: src={}, tgt={}\n",pred,succ);
```

11.1.4 子网格中的通信器

例程 `MPI_Cart_sub`（图 11.5）类似于 `MPI_Comm_split`，它将一个通信器拆分成不相交的子通信器。在这种情况下，它将一个笛卡尔通信器拆分成不相交的笛卡尔通信器，每个对应于维度的一个子集。该子集继承了原始通信器的大小和周期性。

代码:

```
MPI_Cart_sub( period_comm,remain,&hyperplane );
if (procno==0) {
MPI_Topo_test( hyperplane,&topo_type );
MPI_Cartdim_get( hyperplane,&hyperdim );
printf("hyperplane has dimension %d, type %d\n",
hyperdim,topo_type);
MPI_Cart_get( hyperplane,dim,dims,period,coords );
printf("  periodic: ");for (int id=0; id<2; id++)
printf("%d,",period[id]);printf("\n");
```

Output:

```
Grid of size 6 in 3
↪dimensions:3 x  2 x  1
hyperplane has dimension 2,
↪type 2periodic: 1,0,
```

图 11.5 MPI_Cart_sub

Name	参数名	说明	C 类型	F 类型	i nout
MPI_Cart_sub (					
	comm	communicator with Cartesian structure	MPI_Comm	TYPE (MPI_Comm)	IN
	remain_dims	the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false)	const int[]	LOGICAL(*)	IN
	newcomm	communicator containing the subgrid that includes the calling process	MPI_Comm*	TYPE (MPI_Comm)	OUT
)					

11.1.5 重排序

该 MPI_Cart_create 例程有可能对秩进行重排序。如果应用此操作，例程 MPI_Cart_map 将给出该操作的结果。给定与 MPI_Cart_create 相同的参数，它返回调用进程的重排序秩。

11.2 分布式图拓扑

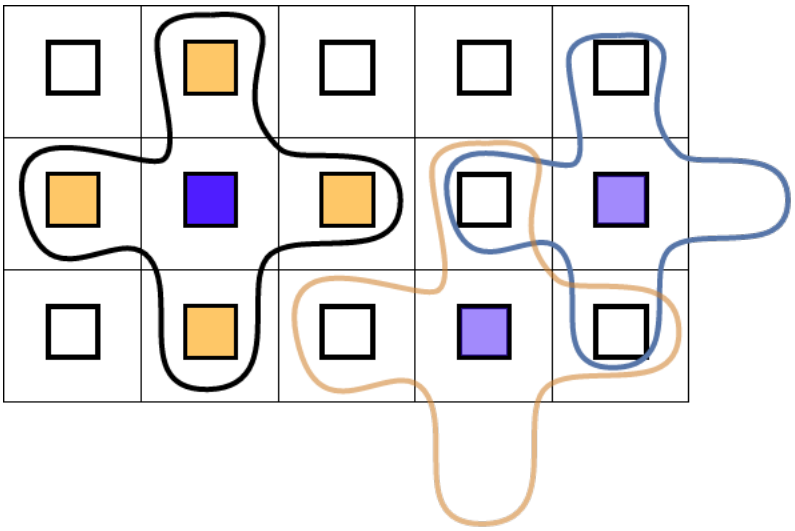


图 11.1：分布式图拓扑示意图，其中每个节点有四个邻居

在许多基于网格的计算中（这里的网格一词采用其数学上的有限元方法（ FEM ）意义），一个网格点会收集来自其周围网格点的信息。在对网格进行合理分布的情况下

进程，这意味着每个进程将从多个邻居进程收集信息。邻居的数量取决于该进程。例如，在二维网格中（并假设计算采用五点模板），大多数进程与四个邻居通信；边缘上的进程与三个邻居通信，角落的进程与两个邻居通信。

这样的拓扑结构如图 11.1 所示。

MPI 的图拓扑概念和邻域集合操作，为表达这种通信结构提供了一种优雅的方式。使用图拓扑而非较旧、更简单方法的原因有多种。

- MPI 被允许重新排序进程，使得集群中的网络邻近性对应于代码结构中的邻近性。
- 普通的集合操作不能直接用于图问题，除非为每个图邻域采用子通信器。然而，这样的调度会导致死锁或串行化。
- 处理图问题的常规方法是通过非阻塞通信。然而，由于用户指明了它们被发布的明确顺序，某些进程可能会发生拥塞。
- 集合操作可以对数据进行流水线处理，而发送 / 接收操作需要完整传输它们的数据。
- 集合操作可以使用生成树，而发送 / 接收使用直接连接。

Thus the minimal description of a process graph contains for each process:

- 度：邻居进程的数量；以及
- 要通信的进程的秩。

然而，这忽略了通信并不总是对称的：也许你接收的进程不是你发送的对象。更糟的是，也许只有这一对偶关系的一侧容易描述。因此，有两个例程：

- `MPI_Dist_graph_create_adjacent` 假设一个进程既知道它发送给谁，也知道谁会发送给它。这是程序员需要指定的最多工作，但最终是最高效的。
- `MPI_Dist_graph_create` 只在每个进程上指定它是哪个的源；也就是说，这个进程将发送给谁。因此，需要一定量的处理 —— 包括通信 —— 来构建相反的信息，即将发送给某个进程的 rank。

11.2.1 图的构建

有两种用于进程图的构建例程。这些例程相当通用，允许任何进程指定拓扑的任何部分。当然，在实际中，你大多会让每个进程描述它自己的邻居结构。

例程 `MPI_Dist_graph_create_adjacent` 假设一个进程既知道它发送给谁，也知道谁会发送给它。这意味着通信图中的每条边都被表示了两次，因此内存占用是严格必要量的两倍。然而，构建图时不需要通信。

第二种创建例程，`MPI_Dist_graph_create`（图 11.6），可能更容易使用，特别是在程序的通信结构是对称的情况下，这意味着一个进程发送给的邻居与它接收的邻居相同。现在你只需在每个进程上指定它是哪个源；也就是说，这个进程将发送给谁。[1]。因此，需要进行一定量的处理 —— 包括通信 —— 来构建反向信息，即将发送给某个进程的进程的秩。

MPL 注释 *71*：分布式图创建。类 *mpl::dist_graph_communicator* 只有一个与 `MPI_Dist_graph_create` 对应的构造函数。

图 11.1 描述了常见的五点模板结构。如果我们让每个进程只描述自己，我们得到以下内容：

- `nsources`= 1 因为调用进程在图中只描述一个节点：它自己。
- `sources` 是一个长度为 1 的数组，包含调用进程的秩。
- `degrees` 是一个长度为 1 的数组，包含该进程的度（可能是：4）。
- `destinations` 是一个长度为该进程度的数组，可能仍然是 4。该数组的元素是邻居节点的 rank；严格来说，是该进程将要发送到的节点。
- `weights` 是一个声明目的地相对重要性的数组。对于一个无权图使用 `MPI_UNWEIGHTED`。如果图是加权的，但源节点的度为零，可以传递一个空数组作为 `MPI_WEIGHTS_EMPTY`。
- `reorder`（`int` 在 C 中，`LOGICAL` 在 Fortran 中）指示是否允许 MPI 重新排列进程以实现更高的局部性。

生成的通信器包含原始通信器的所有进程，且 rank 相同。换句话说 `MPI_Comm_size` 和 `MPI_Comm_rank` 在图通信器上给出与其构造的内部通信器相同的值。要获取关于分组的信息，使用 `MPI_Dist_graph_neighbors` 和 `MPI_Dist_graph_neighbors_count`；参见章节 11.2.3。

举例来说，我们构建一个非对称图，即顶点 v_1 和 v_2 之间的边 $v_1 \to v_2$ 并不意味着存在边 $v_2 \to v_1$。

1. 我不同意这个设计决策。指定你的源通常比指定你的目标更容易。

图 11.6 MPI_Dist_graph_create

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Dist_graph_create (					
	comm_old	input communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	n	number of source nodes for which this process specifies edges	int	INTEGER	IN
	sources	array containing the n source nodes for which this process specifies edges	const int[]	INTEGER(n)	IN
	degrees	array specifying the number of destinations for each source node in the source node array	const int[]	INTEGER(n)	IN
	destinations	destination nodes for the source nodes in the source node array	const int[]	INTEGER(*)	IN
	weights	weights for source to destination edges	const int[]	INTEGER(*)	IN
	info	hints on optimization and interpretation of weights	MPI_Info	TYPE (MPI_Info)	IN
	reorder	the ranks may be reordered (true) or not (false)	int	LOGICAL	IN
	comm_dist_graph	communicator with distributed graph topology added	MPI_Comm*	TYPE (MPI_Comm)	OUT
	)				

MPL:

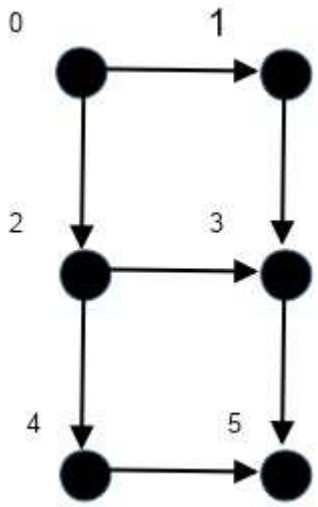
```
dist_graph_communicator(const communicator &old_comm,
const source_set &ss, const dest_set &ds, bool reorder=true)where:
class dist_graph_communicator::source_set : private set< pair<int,int> >
class dist_graph_communicator::dest_set   : private set< pair<int,int> >
```

Python:

```
MPI.Comm.Create_dist_graph
    (self, sources, degrees, destinations, weights=None, Info info=INFO_NULL, bool reorder=False)
returns graph communicator
```

Code:

```
// graph.c
for ( int i=0; i<=1; i++ ) {
  int neighb_i = proci+i;
  if (neighb_i<0 || neighb_i>=idim)
    continue;
  int j = 1-i;
  int neighb_j = procj+j;
  if (neighb_j<0 || neighb_j>=jdim)
        continue;
  destinations[ degree++ ] =
        PROC(neighb_i,neighb_j,idim,jdim);
}
MPI_Dist_graph_create
  (comm,
   /* I specify just one proc: me */ 1,
   &procno,&degree,destinations,weights,
   MPI_INFO_NULL,0,
   &comm2d
   );
```



Here we gather the coordinates of the source neighbors:

Code:

```
int indegree,outdegree,
  weighted;
MPI_Dist_graph_neighbors_count
  (comm2d,
   &indegree,&outdegree,
   &weighted);
int
  my_ij[2] = {proci,procj},
  other_ij[4][2];
MPI_Neighbor_allgather
  ( my_ij,2,MPI_INT,
    other_ij,2,MPI_INT,
    comm2d );
```

输出：

```
[ 0 = (0,0)] has  2 outbound: 1, 2,
0 inbound:[ 1 = (0,1)] has  1 outbound: 3,
1 inbound: (0,0)=0
[ 2 = (1,0)] has  2 outbound: 3, 4,
1 inbound: (0,0)=0
[ 3 = (1,1)] has  1 outbound: 5,
2 inbound: (0,1)=1 (1,0)=2
[ 4 = (2,0)] has  1 outbound: 5,
1 inbound: (1,0)=2
[ 5 = (2,1)] has  0 outbound:
2 inbound: (1,1)=3 (2,0)=4
```

然而，我们不能依赖源是有序的，因此以下代码段执行了对源邻居的显式查询：

Code:

```
int indegree=4, sources[indegree],
  inweights[indegree],weighted;
int outdegree=4, targets[outdegree],
  outweights[outdegree];
MPI_Dist_graph_neighbors_count
  (comm2d,
   &indegree,&outdegree,
   &weighted);
MPI_Dist_graph_neighbors
  (comm2d,
   indegree,sources,inweights,
   outdegree,targets,outweights
   );
```

Output:

```
0 inbound:
1 inbound: 0
1 inbound: 0
2 inbound: 1 2
1 inbound: 2
2 inbound: 4 3
```

Pythonnote 31: Graph communicators. 图通信器的创建是 `Comm` 类的一个方法，图通信器是一个函数返回结果：

```
graph_comm = oldcomm.Create_dist_graph(sources, degrees, destinations)
```

weights 、 info 和 reorder 参数都有默认值。

MPL note 72: Graph communicators. 构造函数 `dist_graph_communicator`

```
dist_graph_communicator
(const communicator &old_comm, const source_set &ss,
const dest_set &ds, bool reorder = true);
```

是对 `MPI_Dist_graph_create_adjacent` 的封装。

MPLnote 73: 图通信器查询。方法 `indegree`, `outdegree` 是对 `MPI_Dist_graph_neighbors_count` 的封装。源和目标可以通过 `inneighbors` 和 `outneighbors` 查询，这些是对 `MPI_Dist_graph_neighbors` 的封装。

11.2.2 邻居集合操作

我们现在可以使用图拓扑执行一个 gather 或 allgather `MPI_Neighbor_allgather` (图 11.7)，该操作仅结合与调用进程直接相连的进程。

邻居集合操作具有与常规集合操作相同的参数列表，但它们适用于图通信器。

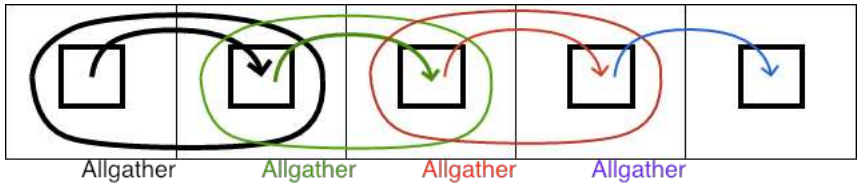


图 11.2：使用邻居集合操作解决右发送练习

图 11.7 MPI_Neighbor_allgather

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Neighbor_allgather (MPI_Neighbor_allgather_c (				
sendbuf	starting address of send buffer	const void*	TYPE(*), DIMENSION(..)	IN
sendcount	number of elements sent to each neighbor	[int MPI_Count	INTEGER	IN
sendtype	datatype of send buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
recvbuf	starting address of receive buffer	void*	TYPE(*), DIMENSION(..)	OUT
recvcount	number of elements received from each neighbor	[int MPI_Count	INTEGER	IN
recvtype	datatype of receive buffer elements	MPI_Datatype	TYPE (MPI_Datatype)	IN
comm	communicator with topology structure	MPI_Comm	TYPE (MPI_Comm)	IN
)				

练习 11.1. 重新审视练习 4.3 并使用 MPI_Dist_graph_create 解决它。参考图 11.2 以获得灵感。

使用度值为 1。（此练习的骨架名为 rightgraph。）

前一个练习可以使用以下度值完成：

- 1，表示每个进程仅与另一个进程通信；或者
- 2，表示你实际上是从两个进程收集数据。

在后一种情况下，结果不会像传统的 gather 那样按进程号递增的顺序存放在接收缓冲区中。相反，你需要使用 MPI_Dist_graph_neighbors 来确定它们的顺序；参见第 11.2.3 节。

另一个邻居集合操作是 MPI_Neighbor_alltoall。

向量变体是 MPI_Neighbor_allgatherv 和 MPI_Neighbor_alltoallv。

有一个异构（多数据类型）变体： MPI_Neighbor_alltoallw .

列表是： MPI_Neighbor_allgather, MPI_Neighbor_allgatherv, MPI_Neighbor_alltoall, MPI_Neighbor_alltoallv, MPI_Neighbor_alltoallw.

Nonblocking: MPI_Ineighbor_allgather, MPI_Ineighbor_allgatherv, MPI_Ineighbor_alltoall, MPI_Ineighbor_alltoallv, MPI_Ineighbor_alltoallw.

出于不明原因，没有 *MPI_Neighbor_allreduce*。

图 11.8 MPI_Dist_graph_neighbors_count

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Dist_graph_neighbors_count (					
	comm	communicator with distributed graph topology	MPI_Comm	TYPE (MPI_Comm)	IN
	indegree	number of edges into this process	int*	INTEGER	OUT
	outdegree	number of edges out of this process	int*	INTEGER	OUT
	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise	int*	LOGICAL	OUT
)					

图 11.9 MPI_Dist_graph_neighbors

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Dist_graph_neighbors (					
	comm	communicator with distributed graph topology	MPI_Comm	TYPE (MPI_Comm)	IN
	maxindegree	size of sources and sourceweights arrays	int	INTEGER	IN
	sources	processes for which the calling process is a destination	int[]	INTEGER (maxindegree)	OUT
	sourceweights	weights of the edges into the calling process	int[]	INTEGER(*)	OUT
	maxoutdegree	size of destinations and destweights arrays	int	INTEGER	IN
	destinations	processes for which the calling process is a source	int[]	INTEGER (maxoutdegree)	OUT
	destweights	weights of the edges out of the calling process	int[]	INTEGER(*)	OUT
)					

11.2.3 查询

有两个例程用于查询进程的邻居： `MPI_Dist_graph_neighbors_count` (图 11.8) 和 `MPI_Dist_graph_neighbors` (图 11.9)。

虽然这些信息似乎可以从图的构造中推导出来，但这并不完全正确，原因有两个。

1. 使用非邻接版本 `MPI_Dist_graph_create`，只指定了出度和目标；该调用随后提供入度和源；
2. 如上所述，gather 调用中数据放置在接收缓冲区的顺序不是由 create 调用决定的，只能通过这种方式查询。

11.2.4 Graph topology (deprecated)

原始的 MPI-1 具有图拓扑接口 `MPI_Graph_create`，该接口要求每个进程指定完整的进程图。由于这不可扩展，应视为已弃用。请改用分布式图拓扑（章节 11.2）。

其他遗留例程：`MPI_Graph_neighbors`, `MPI_Graph_neighbors_count`, `MPI_Graph_get`, `MPI_Graphdims_get`。

11.2.5 重新排序

类似于 `MPI_Cart_map` 例程（章节 11.1.5），例程 `MPI_Graph_map` 为调用进程提供了重新排序的秩。

第 12 章

MPI 主题：共享内存

一些程序员认为 MPI 在共享内存上效率不高，因为所有操作看起来都是通过网络调用完成的。这是不正确的：许多 MPI 实现都有检测共享内存并加以利用的优化，因此数据是被复制的，而不是通过通信层传输。（相反，像 *OpenMP* 这样的共享内存编程系统实际上可能存在与线程处理相关的低效。）使用 MPI 在共享内存上的主要低效之处在于进程实际上不能共享数据。

单边 MPI 调用（第 9 章）也可以用来模拟共享内存，意思是一个源进程可以在目标进程不主动参与的情况下访问其数据。然而，这些调用并不区分真正的共享内存和跨网络的一边访问。

本章将探讨 MPI 如何与实际共享内存的存在进行交互。（此功能在 MPI-3 标准中添加。）这依赖于 `MPI_Win` 窗口概念，但除此之外使用对其他进程内存的直接访问。

12.1 识别共享内存

MPI 的单边例程对进程持非常对称的看法：每个进程都可以访问每个其他进程的窗口（在通信子内）。当然，实际上性能会有所不同，取决于源进程和目标进程是否实际上位于同一共享内存上，或者它们是否只能通过网络通信。出于这个原因，MPI 使得通过共享内存域对进程进行分组变得容易，使用 `MPI_Comm_split_type`（参见第 7.4.1 节）和类型 `MPI_COMM_TYPE_SHARED`。

通过共享内存进行划分：

Code:

```
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type
(MPI_COMM_WORLD,
MPI_COMM_TYPE_SHARED,procno,
info,&sharedcomm);MPI_Comm_size
(sharedcomm,&new_nprocs);
MPI_Comm_rank
(sharedcomm,&new_procno);
```

Output:

```
make[3]: `commsplittype' is
    ↪up to date.
TACC:  Starting up job
    ↪4356245
TACC:  Starting parallel
    ↪tasks...
There are 10 ranks total
[0] is processor 0 in a
    ↪shared group of 5,
    ↪running on
    ↪c209-010.frontera.tacc.utexas.edu
[5] is processor 0 in a
    ↪shared group of 5,
    ↪running on
    ↪c209-011.frontera.tacc.utexas.edu
TACC:  Shutdown complete.
    ↪Exiting.
```

Exercise 12.1. 编写一个程序，使用 `MPI_Comm_split_type` 来分析一次运行：1. 有多少个节点；2. 每个节点上有多少个进程。如果你在不均匀分布的情况下运行此程序，比如 3 个节点上有 10 个进程，你会发现什么分布?

```
Nodes: 3; processes: 10
TACC:  Starting up job 4210429
TACC:  Starting parallel tasks...
There are 3 nodes
Node sizes: 4 3 3
TACC:  Shutdown complete. Exiting.
```

MPLnote74: 按 *sharedmemory* 拆分。 类似于普通通信器拆分（幻灯片 61 ）：`communicator::split_shared`.

```
// commsplittype.cxxmpl::communicator shared_comm
( mpl::communicator::split_shared_memory, world_comm );
intonnode_procno = shared_comm.rank(),
onnode_nprocs = shared_comm.size();
```

但请注意：共享内存目前不可用，因为 windows 尚未实现。

12.2 Windows 的共享内存

存在于相同物理共享内存上的进程应该能够通过复制来移动数据，而不是通过 MPI 的发送 / 接收调用 —— 当然这些调用在底层也会执行复制操作。为了实现这种用户级复制：

图 12.1 MPI_Win_allocate_shared

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Win_allocate_shared (					
MPI_Win_allocate_shared_c (					
	size	size of local window in bytes	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
	disp_unit	local unit size for displacements, in bytes	int MPI_Aint	INTEGER	IN
	info	info argument	MPI_Info	TYPE (MPI_Info)	IN
	comm	intra-communicator	MPI_Comm	TYPE (MPI_Comm)	IN
	baseptr	address of local allocated window segment	void*	TYPE(C_PTR)	OUT
	win	window object returned by the call	MPI_Win*	TYPE(MPI_Win)	OUT
	)				

1. 我们需要创建一个带有 MPI_Win_allocate_shared 的共享内存区域。这会创建一个具有 统一内存模型 (参见第 9.5.1 节) 的窗口；2. 我们需要获取指向进程在此共享空间中区域的指针；这是通过 MPI_Win_shared_query 完成的。

Remark24 从 *MPI-4.1* 开始， *MPI_Win_shared_query* 可以用于来自 *MPI_Win_allocate* 和 *MPI_Win_create* 的内存，只要这实际上是共享内存上的一个窗口。只有 *MPI_Win_allocate_shared* 保证产生这样的共享内存。

12.2.1 指向共享窗口的指针

第一步是在一个节点上的进程上创建一个窗口（在单边 MPI 的意义上；见第 9.1 节）。使用 MPI_Win_allocate_shared (图 12.1) 调用大概会将内存放置在进程运行的 socket 附近。

```
// sharedbulk.cMPI_Win node_window;
MPI_Aint window_size; double *window_data;
if (onnode_procid==0)
window_size = sizeof(double);
else window_size = 0;MPI_Win_allocate_shared
( window_size,sizeof(double),MPI_INFO_NULL,
nodecomm,&window_data,&node_window);
```

由 MPI_Win_allocate_shared 分配的内存在线程间是连续的。这使得地址计算成为可能。然而，如果集群节点具有非统一内存访问（ NUMA ）结构，例如如果两个 socket 各自直接连接有内存，这将增加延迟

对于某些进程。为防止这种情况，关键字 *alloc_shared_noncontig* 可以在 true 中设置为 MPI_Info 对象。

以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

I在连续情况下， *mpi_minimum_memory_alignment* info 参数（第 9.1.1 节）仅适用于
m第一个进程上的内存；在非连续情况下，它适用于所有进程。
MPI-4 内容结束

```
// numa.cMPI_Info window_info;MPI_Info_create(&window_info);
MPI_Info_set(window_info,"alloc_shared_noncontig","true");
MPI_Win_allocate_shared( window_size,sizeof(double),window_info,
nodecomm,&window_data,&node_window);MPI_Info_free(&window_info);
```

让我们来探讨一下。我们创建一个共享窗口，每个进程存储恰好一个 double，即 8 字节。以下代码片段查询窗口位置，并打印到进程 0 上窗口的字节距离。

```
for (int p=1; p<onnode_nprocs; p++) {
MPI_Aint window_sizep; int windowp_unit; double *winp_addr;
MPI_Win_shared_query( node_window,p,&window_sizep,&windowp_unit,
 &winp_addr );distp = (size_t)winp_addr-(size_t)win0_addr;
if (procno==0)printf("Distance %d to zero: %ld\n",p,(long)distp);
```

W使用默认策略时，这些窗口是连续的，因此距离是 8 字节的倍数。
N非连续分配则不是这样：

策略：共享窗口分配的默认行为

```
Distance 1 to zero: 8
Distance 2 to zero: 16
Distance 3 to zero: 24
Distance 4 to zero: 32
Distance 5 to zero: 40
Distance 6 to zero: 48
Distance 7 to zero: 56
Distance 8 to zero: 64
Distance 9 to zero: 72
```

策略：允许非连续共享窗口分配

```
Distance 1 to zero: 4096
Distance 2 to zero: 8192
Distance 3 to zero: 12288
Distance 4 to zero: 16384
Distance 5 to zero: 20480
Distance 6 to zero: 24576
Distance 7 to zero: 28672
Distance 8 to zero: 32768
Distance 9 to zero: 36864
```

这里的解释是每个窗口都放置在其自己的小页面上，在这个特定系统中，其大小为 4K。

备注 25 *C* 语言中的取地址符号（*&*）不是一个物理地址，而是一个虚拟地址。页面在物理内存中的位置由页表决定。

图 12.2 MPI_Win_shared_query

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Win_shared_query (				
MPI_Win_shared_query_c (				
win	shared memory window object	MPI_Win	TYPE(MPI_Win)	IN
rank	rank in the group of window win or MPI_PROC_NULL	int	INTEGER	IN
size	size of the window segment	MPI_Aint*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
disp_unit	local unit size for displacements, in bytes	int* / MPI_Aint*	INTEGER	OUT
baseptr	address for load/store access to window segment	void*	TYPE(C_PTR)	OUT
)				

12.2.2 查询共享结构

尽管上面创建的窗口是共享的，但这并不意味着它是连续的。因此，有必要检索指向每个进程中你想要通信的区域的指针：MPI_Win_shared_query(图 12.2)。

```
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query( node_window,0,&window_size0,
&window_unit, &win0_addr );
```

12.2.3 热方程示例

作为一个示例，考虑一维热方程。在每个进程上我们创建一个包含三个点的局部区域：

```
// sharedshared.c
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&shared_baseptr,&shared_window);
```

12.2.4 共享大块数据

在诸如 光线追踪 等应用中，存在一个只读的大型数据对象（场景中要渲染的对象），所有进程都需要它。在传统的 MPI 中，这需要在每个进程上冗余存储，导致巨大的内存需求。使用 MPI 共享内存，我们可以在每个节点上只存储一次该数据对象。使用如上 MPI_Comm_split_type 来为每个 NUMA 域找到一个通信器，我们将该对象存储在该节点通信器的进程零上。

练习 12.2. 让 “ 共享 ” 数据起源于 MPI_COMM_WORLD 中的进程零。然后：

- 为每个共享内存域创建一个通信器；
- 为所有节点上编号为零的进程创建一个通信器；

- 将共享数据广播到每个节点上的进程零。（此练习的骨架代码名为 `shareddata`。）

第 13 章

MPI 主题：混合计算

虽然 MPI 标准本身没有提及线程 —— 进程是主要的计算单位 —— 但允许使用线程。下面我们将讨论为此存在的相关规定。

将线程和其他共享内存模型与 MPI 结合使用，当然会引出如何处理竞态条件的问题。以下是一个涉及 MPI 的数据竞争代码示例：

```
#pragma omp sections
#pragma omp section
  MPI_Send( x, /* to process 2 */ )
#pragma omp section
  MPI_Recv( x, /* from process 3 */ )
```

MPI 标准在这里将责任放在用户身上：此代码不合法，行为未定义 .

13.1 MPI 对线程的支持

在混合执行中，主要问题是是否允许所有线程进行 MPI 调用。为确定这一点，将 `MPI_Init` 调用替换为 `MPI_Init_thread` (图 13.1) 这里的 `required` 和 `provided` 参数可以取以下（单调递增的）值：

- `MPI_THREAD_SINGLE`: 只有单个线程会执行。
- `MPI_THREAD_FUNNELED`: 程序可能使用多个线程，但只有主线程会进行 MPI 调用。主线程通常是由 `master` 指令选择的，但从技术上讲，它是唯一执行 `MPI_Init_thread` 的线程。如果你在并行区域调用此例程，主线程可能与 master 不同。
- `MPI_THREAD_SERIALIZED`: 程序可能使用多个线程，所有线程都可以进行 MPI 调用，但不会有多个线程同时进行 MPI 调用。
- `MPI_THREAD_MULTIPLE`: 多个线程可以发出 MPI 调用，没有限制。

初始化调用之后，您可以通过 `MPI_Query_thread` (图 13.2) 查询支持级别。

如果多个线程执行通信， `MPI_Is_thread_main` (图 13.3) 可以确定某个线程是否为主线程。

Python 注释 *32*：线程级别。 线程级别可以通过 *mpi4py.rc* 对象（第 2.2.2 节）设置：

图 13.1 MPI_Init_thread

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Init_thread (				
argc	desired level of thread support	int*	INTEGER	IN
argv	provided level of thread support	char***	INTEGER	OUT
)				

图 13.2 MPI_Query_thread

名称 参数名	说明	C type	F ty pe	i nout
MPI_Query_thread (				
provided	provided level of thread support	int*	INTEGER	OUT
)				

```
mpi4py.rc.threads # default True
mpi4py.rc.thread_level # default "multiple"
```

可用级别为 multiple, serialized, funneled, single。

MPL 注释 *75*：线程支持。MPL 总是调用 MPI_Init_thread 请求最高级别 MPI_THREAD_MULTIPLE。

```
enum mpl::threading_modes {
mpl::threading_modes::single = MPI_THREAD_SINGLE,
mpl::threading_modes::funneled = MPI_THREAD_FUNNELED,
mpl::threading_modes::serialized = MPI_THREAD_SERIALIZED,
mpl::threading_modes::multiple = MPI_THREAD_MULTIPLE};
threading_modes mpl::environment::threading_mode ();
bool mpl::environment::is_thread_main ();
```

MPI 的 *mvapich* 实现确实具备所需的线程支持，但您需要设置此环境变量：

```
export MV2_ENABLE_AFFINITY=0
```

另一种解决方案是这样运行你的代码：

```
  ibrun tacc_affinity <my_multithreaded_mpi_executable
```

Intel MPI uses an environment variable to turn on thread support:

```
I_MPI_LIBRARY_KIND=<value>
where
release : multi-threaded with global lock
release_mt : multi-threaded with per-object lock for thread-split
```

程序 *mpiexec* 通常会传播环境变量，因此当你调用 mpiexec 时，OMP_NUM_THREADS 的值将被每个 MPI 进程看到。

图 13.3 MPI_Is_thread_main

名称 参数名 说明		C 类型	F 类型	输入输出
MPI_Is_thread_main (				
flag	true if calling thread is main thread, false otherwise	int*	LOGICAL	OUT
)				

- 可以在线程中使用阻塞发送，并让线程阻塞。这就消除了轮询的需求。
- 你不能发送到线程编号：使用 MPI *message tag* 发送到特定线程。

Exercise 13.1. 考虑二维热方程并探索 MPI/OpenMP 并行的混合：

- 给每个节点一个完全多线程的 MPI 进程。
- 给每个核心一个 MPI 进程且不使用多线程。

理论上讨论为什么前者可以提供更高的性能。实现两者方案作为通用混合方案的特例，并运行测试以找到最佳混合。

```
// thread.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&threading);
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm,&procno);
MPI_Comm_size(comm,&nprocs);

if (procno==0) {
  switch (threading) {
  case MPI_THREAD_MULTIPLE : printf("Glorious multithreaded MPI\n"); break;
  case MPI_THREAD_SERIALIZED : printf("No simultaneous MPI from threads\n"); break;
  case MPI_THREAD_FUNNELED : printf("MPI from main thread\n"); break;
  case MPI_THREAD_SINGLE : printf("no threading supported\n"); break;
  }
}
MPI_Finalize();
```

第 14 章

MPI 主题：工具接口

从 MPI-3.0 开始，并在 MPI-3.1 和 MPI-4.0 中扩展，最近版本的 MPI 有一种标准化的读取性能变量的方法： *tools interface*，它改进了第 15.6.2 节中描述的旧接口。

14.1 初始化 tools 接口

tools 接口需要不同的初始化例程 `MPI_T_init_thread`

```
int MPI_T_init_thread( int required,int *provided );
```

同样，也有 `MPI_T_finalize`

```
int MPI_T_finalize();
```

这些匹配调用可以多次进行，在 MPI 已经通过 `MPI_Init` 或 `MPI_Init_thread` 初始化之后。

详细级别是一个整数参数。
`MPI_T_VERBOSITY_{USER,TUNER,MPIDEV}_{BASIC,DETAIL,ALL}`

14.2 控制变量

控制变量 是依赖于实现的变量，可用于检查和 / 或控制 MPI 的内部工作。访问控制变量需要初始化工具接口；详见 14.1 节。

我们通过 查询有多少 控制变量可用，使用 `MPI_T_cvar_get_num` (图 14.1)。控制变量的描述可以从 `MPI_T_cvar_get_info` (图 14.2) 获得。

- 无效的索引会导致函数结果为 `MPI_T_ERR_INVALID_INDEX`。
- 任何输出参数都可以指定为 `NULL`，MPI 将不会设置它。
- *bind* 变量是对象类型或 `MPI_T_BIND_NO_OBJECT`。
- *enumtype* 变量是 `MPI_T_ENUM_NULL`，如果该变量不是枚举类型。

图 14.1 MPI_T_cvar_get_num

名称 参数名 说明	C 类型	F 类型	输入输出
MPI_T_cvar_get_num ( )			

Figure 14.2 MPI_T_cvar_get_info

名称 参数名 说明	C 类型	F 类型	输入输出
MPI_T_cvar_get_info ( )			

```
// cvar.c
MPI_T_cvar_get_num(&ncvar);
printf("#cvars: %d\n",ncvar);
for (int ivar=0; ivar<ncvar; ivar++) {
  char name[100]; int namelen = 100;
  char desc[256]; int desclen = 256;
  int verbosity,bind,scope;
  MPI_Datatype datatype;
  MPI_T_enum enumtype;
  MPI_T_cvar_get_info
        (ivar,
         name,&namelen,
         &verbosity,&datatype,&enumtype,desc,&desclen,&bind,&scope
         );
  printf("cvar %3d: %s\n  %s\n",ivar,name,desc);
```

备注 26 没有指示这些变量最大缓冲区长度的常量。但是，你可以执行以下操作：

1. 使用 NULL 值调用 *info* 例程以读取缓冲区长度；*2.* 分配足够长度的缓冲区，即包括一个额外的位置用于空终止符；*3.* 第二次调用 *info* 例程，填充字符串缓冲区。

反过来，给定一个变量名，可以通过 MPI_T_cvar_get_index 获取其索引：

```
int MPI_T_cvar_get_index(const char *name, int *cvar_index)
```

如果名称无法匹配，索引为 MPI_T_ERR_INVALID_NAME。

访问控制变量是通过 控制变量句柄 完成的。

```
int MPI_T_cvar_handle_alloc
(int cvar_index, void *obj_handle,
MPI_T_cvar_handle *handle, int *count)
```

该句柄通过 MPI_T_cvar_handle_free 释放：

```
int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
```

控制变量访问 是通过 `MPI_T_cvar_read` 和 `MPI_T_cvar_write` 完成的：

```
int MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf);
int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf);
```

14.2.1 回调接口以下内容适用于 最近发布的 *MPI-4* 标准，可能尚未被支持。

MPI_T_Source_.... MPI_T_Event_.... MPI_T_Category_get_num_events_.... MPI_T_Category_get_events_....
MPI-4 材料结束

14.3 性能变量

工具接口的实现依赖于安装，您首先需要查询提供了多少工具接口。

```
// mpitpvar.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_SINGLE,&tlevel);
MPI_T_init_thread(MPI_THREAD_SINGLE,&tlevel);
int npvar;
MPI_T_pvar_get_num(&npvar);
```

```
int name_len=256,desc_len=256,
  verbosity,var_class,binding,isreadonly,iscontiguous,isatomic;
char var_name[256],description[256];
MPI_Datatype datatype; MPI_T_enum enumtype;
for (int pvar=0; pvar<npvar; pvar++) {
  name_len = 256; desc_len=256;
  MPI_T_pvar_get_info(pvar,var_name,&name_len,
                      &verbosity,&var_class,
                      &datatype,&enumtype,
                      description,&desc_len,
                      &binding,&isreadonly,&iscontiguous,&isatomic);
  if (procid==0)
    printf("pvar %d: %d/%s = %s\n",pvar,var_class,var_name,description);
}
```

性能变量分为类： `MPI_T_PVAR_CLASS_STATE` `MPI_T_PVAR_CLASS_LEVEL`
`MPI_T_PVAR_CLASS_SIZE` `MPI_T_PVAR_CLASS_PERCENTAGE` `MPI_T_PVAR_CLASS_HIGHWATERMARK`
`MPI_T_PVAR_CLASS_LOWWATERMARK` `MPI_T_PVAR_CLASS_COUNTER` `MPI_T_PVAR_CLASS_AGGREGATE`
`MPI_T_PVAR_CLASS_TIMER` `MPI_T_PVAR_CLASS_GENERIC`

使用以下方式查询性能变量的数量 `MPI_T_pvar_get_num`：

```
int MPI_T_pvar_get_num(int *num_pvar);
```

通过索引获取每个变量的信息，使用 `MPI_T_pvar_get_info`：

```
int MPI_T_pvar_get_info
   (int pvar_index, char *name, int *name_len,
    int *verbosity, int *var_class, MPI_Datatype *datatype,
    MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind,
    int *readonly, int *continuous, int *atomic)
```

关于这些的一般说明见第 14.2 节。

- 变量表示该变量不能被写入。
- 该 *continuous* 变量需要使用 MPI_T_pvar_start 和 MPI_T_pvar_stop。

Given a name, the index can be retried with MPI_T_pvar_get_index:

```
int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)
```

Again, see section 14.2.

14.3.1 性能实验会话

为了防止测量结果混淆，必须在性能实验会话中进行测量，本章中称之为 “ 会话 ”。但请参见第 8.3 节。

Create a session with MPI_T_pvar_session_create

```
int MPI_T_pvar_session_create(MPI_T_pvar_session *session)
```

并用 MPI_T_pvar_session_free 发布：

```
int MPI_T_pvar_session_free(MPI_T_pvar_session *session)
```

这将会话变量设置为 MPI_T_PVAR_SESSION_NULL。

我们通过与某个会话关联的句柄访问变量。句柄是用 MPI_T_pvar_handle_alloc 创建的：

```
int MPI_T_pvar_handle_alloc
   (MPI_T_pvar_session session, int pvar_index,
    void *obj_handle, MPI_T_pvar_handle *handle, int *count)
```

(如果一个例程同时接受会话和句柄参数，且两者不关联，则返回错误 MPI_T_ERR_INVALID_HANDLE。)

用 MPI_T_pvar_handle_free 释放句柄：

```
int MPI_T_pvar_handle_free
   (MPI_T_pvar_session session,
    MPI_T_pvar_handle *handle)
```

将变量设置为 MPI_T_PVAR_HANDLE_NULL。

连续变量（见 MPI_T_pvar_get_info 上文，输出此内容）可以通过 MPI_T_pvar_start 和 MPI_T_pvar_stop 启动和停止：

```
int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle);
int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

传递 MPI_T_PVAR_ALL_HANDLES 给 stop 调用尝试停止会话中的所有变量。未能停止变量将返回 MPI_T_ERR_PVAR_NO_STARTSTOP。

变量可以通过 MPI_T_pvar_read 和 MPI_T_pvar_write 进行读写：

```
int MPI_T_pvar_read
   (MPI_T_pvar_session session, MPI_T_pvar_handle handle,
    void* buf)
int MPI_T_pvar_write
   (MPI_T_pvar_session session, MPI_T_pvar_handle handle,
    const void* buf)
```

I如果变量无法写入（参见 *readonly* 的参数 MPI_T_pvar_get_info），MPI_T_ERR_PVAR_NO_WRITE 被返回。

写入变量的一种特殊情况是用

```
int MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

允许的句柄值是 MPI_T_PVAR_ALL_HANDLES 。

对 MPI_T_pvar_readreset 的调用是读取和重置调用的原子组合：

```
int MPI_T_pvar_readreset
   (MPI_T_pvar_session session,MPI_T_pvar_handle handle,
    void* buf)
```

14.4 变量的类别

变量，无论是控制类还是性能类，都可以由 MPI 实现进行分类。

类别的数量通过以下方式查询 MPI_T_category_get_num：

```
int MPI_T_category_get_num(int *num_cat)
```

对于每个类别，信息通过以下方式获取 MPI_T_category_get_info：

```
int MPI_T_category_get_info(int cat_index,char *name,
 int *name_len, char *desc, int *desc_len,int *num_cvars,
 int *num_pvars, int *num_categories)
```

对于给定的类别名称，可以通过 MPI_T_category_get_index 找到索引：

```
int MPI_T_category_get_index(const char *name, int *cat_index)
```

类别的内容通过 MPI_T_category_get_cvars, MPI_T_category_get_pvars,MPI_T_category_get_categories 获取：

```
int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
int MPI_T_category_get_categories(int cat_index, int len, int indices[])
```

这些索引随后可以用于调用 MPI_T_cvar_get_info, MPI_T_pvar_get_info,MPI_T_category_get_info。

如果类别动态变化，可以通过 MPI_T_category_changed 检测到

```
int MPI_T_category_changed(int *stamp)
```

14.5 事件

```
// mpitevent.c
int nsource;
MPI_T_source_get_num(&nsource);
```

```
int name_len=256,desc_len=256;
char var_name[256],description[256];
MPI_T_source_order ordering;
MPI_Count ticks_per_second,max_ticks;
MPI_Info info;
MPI_Datatype datatype; MPI_T_enum enumtype;
for (int source=0; source<nsource; source++) {
  name_len = 256; desc_len=256;
  MPI_T_source_get_info(source,var_name,&name_len,
                        description,&desc_len,
                        &ordering,&ticks_per_second,&max_ticks,&info);
```

第 15 章

MPI 剩余主题

15.1 上下文信息、属性等

15.1.1 Info 对象

某些 MPI 例程可以接受 MPI_Info 对象。（有关文件，请参见第 15.1.1.3 节，关于窗口，请参见第 9.5.5 节。）这些包含键值对，可以提供系统或实现相关的信息。

Info 对象可以通过 MPI_Info_create (图 15.1) 创建，通过 MPI_Info_free (图 15.2) 删除；有一个名为 MPI_INFO_ENV 的 info 对象，由 MPI_Init 或 MPI_Init_thread 创建；详见第 15.1.1.1 节。

键随后通过 MPI_Info_set (图 15.3) 设置，可以通过 MPI_Info_get (图 15.4) 查询。注意，‘get’ 例程的输出不是分配的：它是一个传入的缓冲区。键的最大长度由参数 MPI_MAX_INFO_KEY 给出。你可以通过 MPI_Info_delete (图 15.5) 从 info 对象中删除一个键。

info 对象的复制非常直接： MPI_Info_dup (图 15.6)。

你也可以通过 MPI_Info_get_nkeys (图 15.7) 查询 info 对象中键的数量，之后可以通过 MPI_Info_get_nthkey 依次查询这些键

I被标记为 ‘In’ 或 ‘Inout’ 参数的 info 对象会在该例程返回之前被解析。这意味着
t在非阻塞例程中它们可以立即释放，不像例如发送缓冲区那样。
*T*以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

例程 MPI_Info_get 和 MPI_Info_get_valuelen 对于 C 语言的 *nullterminator* 不够健壮。因此，它们已被弃用，应替换为 MPI_Info_get_string，后者总是返回以 null 结尾的字符串。

```
int MPI_Info_get_string
    (MPI_Info info, const
     char *key,int *buflen, char *value, int *flag)
```

End of MPI-4 material

MPL note 76: Info objects. There is an `info` object in the `mpl` namespace:

```
mpl::info infoobject; // default constructor
```

图 15.1 MPI_Info_create

Name	Param name	Explanation	C type	F type	inout
MPI_Info_create (					
	info	info object created	MPI_Info*	TYPE (MPI_Info)	OUT
)					

Figure 15.2 MPI_Info_free

Name	Param name	Explanation	C type	F type	inout
MPI_Info_free (					
	info	info object	MPI_Info*	TYPE (MPI_Info)	INOUT
)					

图 15.3 MPI_Info_set

名称	Param name	说明	C 类型	F 类型	输入输出
MPI_Info_set (					
	info	info object	MPI_Info	TYPE (MPI_Info)	INOUT
	key	key	const char*	CHARACTER	IN
	value	value	const char*	CHARACTER	IN
)					

图 15.4 MPI_Info_get

名称	参数名	说明	C 类型	F 类型	输入输出
MPI_Info_get (					
	info	info object	MPI_Info	TYPE (MPI_Info)	IN
	key	key	const char*	CHARACTER	IN
	valuelen	length of value associated with key	int	INTEGER	IN
	value	value	char*	CHARACTER	OUT
	flag	true if key defined, false if not	int*	LOGICAL	OUT
)					

图 15.5 MPI_Info_delete

Name	参数名	说明	C 类型	F 类型	输入输出
MPI_Info_delete (					
	info	info object	MPI_Info	TYPE (MPI_Info)	INOUT
	key	key	const char*	CHARACTER	IN
)					

图 15.6 MPI_Info_dup

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Info_dup (				
info	info object	MPI_Info	TYPE (MPI_Info)	IN
newinfo	info object created	MPI_Info*	TYPE (MPI_Info)	OUT
)				

图 15.7 MPI_Info_get_nkeys

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Info_get_nkeys (				
info	info object	MPI_Info	TYPE (MPI_Info)	IN
nkeys	number of defined keys	int*	INTEGER	OUT
)				

Sample methods:

```
void set(const std::string &key, const std::string &value);
[[nodiscard]] std::optional<std::string> value(const std::string &key) const;
```

15.1.1.1 环境信息

该对象 MPI_INFO_ENV 是预定义的，包含：

- command 执行的程序名称。
- argv 以空格分隔的命令参数。
- maxprocs 启动的最大 MPI 进程数。
- soft 允许的处理器数量值。
- host 主机名。
- arch 架构名称。
- wdir MPI 进程的工作目录。
- file 值是指定附加信息的文件名。
- thread_level 如果在程序开始执行前请求，则为请求的线程支持级别。

注意，这些是请求的值；运行中的程序例如可能具有较低的线程支持。

15.1.1.2 通信器和窗口信息

MPI 内置了通过调用 *communicators* 和 *windows* 附加信息的功能 MPI_Comm_get_info MPI_Comm_set_info, MPI_Win_get_info, MPI_Win_set_info。

复制一个通信器时，MPI_Comm_dup 不会复制信息；要将信息传播到副本，需要使用 MPI_Comm_dup_with_info (第 7.2 节)。

15.1.1.3 文件信息

An `MPI_Info` 对象可以传递给以下文件例程：

- `MPI_File_open`
- `MPI_File_set_view`
- `MPI_File_set_info`；集合操作。相反的例程是 `MPI_File_get_info`。

以下键在 MPI-2 标准中定义：

- *access_style*：由一个或多个以下项组成的逗号分隔列表：`read_once`，`write_once`,`read_mostly`，`write_mostly`, `sequential`, `reverse_sequential`, `random`
- *collective_buffering*：true 或 false；启用或禁用集合 I/O 操作的缓冲
- *cb_block_size*：集合缓冲的整数块大小，单位为字节
- *cb_buffer_size*：集合缓冲的整数缓冲区大小，单位为字节
- *cb_nodes*：集合缓冲中使用的 MPI 进程数（整数）
- *chunked*：一个用逗号分隔的整数列表，描述要使用子数组访问的多维数组的维度，从最高维度开始（C 语言中为第 1 维，Fortran 中为最后一维）
- *chunked_item*：一个用逗号分隔的列表，指定每个数组元素的大小，单位为字节
- *chunked_size*：一个用逗号分隔的列表，指定用于分块的子数组大小
- *file_perm*：创建时的 UNIX 文件权限，八进制表示
- *io_node_list*：一个用逗号分隔的 I/O 节点列表

*The f*以下内容针对最近发布的 *MPI-4* 标准和 *ma* *y not be supported yet.*

- *mpi_minimum_memory_alignment*：已分配内存的对齐。*fMPI-4 material*

End o

- *nb_proc*: 预期同时访问文件的进程整数数量
- *num_io_nodes*: 要使用的 I/O 节点整数数量
- *striping_factor*: 文件应跨越的 I/O 节点 / 设备的整数数量
- *striping_unit*: 条带大小，单位为字节的整数

此外，还可以存在特定于文件系统的键。

15.1.2 属性

某些运行时（或安装依赖）值可以作为属性通过 `MPI_Comm_set_attr`(图 15.8) 和 `MPI_Comm_get_attr` （图 15.9) 对通信器可用，或 `MPI_Win_get_attr`,`MPI_Type_get_attr`。（MPI-2 例程 `MPI_Attr_get` 已被弃用）。flag 参数有两个功能：

- 它返回是否找到了该属性；
- 如果入口时设置为 false，则忽略 value 参数，例程仅测试键是否存在。

返回值参数较为微妙：虽然它声明为 `void*`，但实际上是一个 `void*` 指针的地址。

```
// tags.c
int tag_upperbound;
void *v; int flag=1;
ierr = MPI_Comm_get_attr(comm,MPI_TAG_UB,&v,&flag);
tag_upperbound = *(int*)v;
```

图 15.8 MPI_Comm_set_attr

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Comm_set_attr (				
comm	communicator to which attribute will be attached	MPI_Comm	TYPE (MPI_Comm)	INOUT
comm_keyval	key value	int	INTEGER	IN
attribute_val	attribute value	void*	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
)				

图 15.9 MPI_Comm_get_attr

Name	Param name	说明	C 类型	F ty pe	i nout
MPI_Comm_get_attr (					
	comm	communicator to which the attribute is attached	MPI_Comm	TYPE (MPI_Comm)	IN
	comm_keyval	key value	int	INTEGER	IN
	attribute_val	attribute value, unless flag = false	void*	INTEGER (KIND=MPI_ADDRESS_KIND)	OUT
	flag	false if no attribute is associated with the key	int*	LOGICAL	OUT
)					

Python:

```
MPI.Comm.Get_attr(self, int keyval)
```

```
## tags.py
tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}".format(tag_upperbound))
```

属性包括：

- MPI_TAG_UB *tagvalue* 的上界。（下界为零。）注意 MPI_TAG_UB 是键，而不是实际的上界！该值必须至少为 32767。
- MPI_HOST 主机进程的 rank（如果存在），否则为 MPI_PROC_NULL。标准未定义主机的含义，甚至未定义是否应该存在主机。从 MPI-4.1 开始，此项已弃用。
- MPI_IO 具有常规 I/O 设施的节点的 rank。相同通信器中的节点可能返回此参数的不同值。如果返回 MPI_ANY_SOURCE，则所有 rank 都可以执行 I/O。

- MPI_WTIME_IS_GLOBAL 布尔变量，指示时钟是否同步。另见：

- MPI_UNIVERSE_SIZE：可创建的进程总数。如果主机列表大于初始启动的进程数，则此数可能大于 MPI_COMM_WORLD。详见第 8.1 节。
- MPI_APPNUM：如果 MPI 以 MPMD 模式使用（第 15.9.4 节），或使用了 MPI_Comm_spawn_multiple（第 8.1 节），此属性报告当前是第几个程序。

图 15.10 MPI_Comm_create_keyval

Name	Param name	说明	C 类型	F 类型	i nout
MPI_Comm_create_keyval (					
	comm_copy_attr_fn	copy callback function for comm_keyval	MPI_Comm_copy_attr_function*	PROCEDURE (MPI_Comm_copy_attr_function)	IN
	comm_delete_attr_fn	delete callback function for comm_keyval	MPI_Comm_delete_attr_function*	PROCEDURE (MPI_Comm_delete_attr_function)	IN
	comm_keyval	key value for future access	int*	INTEGER	OUT
	extra_state	extra state for callback function	void*	INTEGER (KIND=MPI_ADDRESS_KIND)	IN
	)				

Fortran 注释 *15*：属性查询。Fortran 没有这种双重间接引用的东西。属性的值会立即返回，作为一种类型为 MPI_ADDRESS_KIND 的整数：

```
!! tags.F90
  logical :: flag
  integer(KIND=MPI_ADDRESS_KIND) :: attr_v,tag_upperbound
  call MPI_Comm_get_attr(comm,MPI_TAG_UB,attr_v,flag,ierr)
  tag_upperbound = attr_v
        print '("Determined tag upperbound: ",i9)', tag_upperbound
```

Python 注释 *33*：*Universe size*。 *mpi4py.MPI.UNIVERSE_SIZE.*

15.1.3 创建新的 keyval 属性

使用 MPI_Comm_create_keyval （图 15.10）创建一个键值，MPI_Type_create_keyval，MPI_Win_create_keyval。使用此键通过 MPI_Comm_set_attr，MPI_Type_set_attr,MPI_Win_set_attr 设置新属性。使用 MPI_Comm_delete_attr，MPI_Type_delete_attr,MPI_Win_delete_attr 释放属性。

Thisuses a function type MPI_Comm_attr_function。当复制通信器时，此函数会被复制；详见第 7.2 节。使用 MPI_Comm_free_keyval 释放。

15.1.4 处理器名称

您可以使用 MPI_Get_processor_name 查询处理器的 *hostname*。该名称在不同的处理器等级之间不必唯一。

您必须传入字符存储：字符数组长度至少为 MPI_MAX_PROCESSOR_NAME 个字符。名称的实际长度将通过 resultlen 参数返回。

15.1.5 版本信息

对于运行时确定，*MPI* 版本可以通过两个参数 MPI_VERSION 和 MPI_SUBVERSION 或函数 MPI_Get_version（图 15.11）获得。

图 15.11 MPI_Get_version

名称 参数名	说明	C 类型	F 类型	i nout
MPI_Get_version (				
version	version number	int*	INTEGER	OUT
subversion	subversion number	int*	INTEGER	OUT
)				

库版本可以通过查询 MPI_Get_library_version。结果字符串必须适合于 MPI_MAX_LIBRARY_VERSION_STRING。

Python 注释 *34*：*MPI* 版本。 提供了一个函数用于版本和子版本，以及显式参数：

Code:

```
## version.py
print(MPI.Get_version())
print(MPI.VERSION)
print(MPI.SUBVERSION)
```

Output:

```
(3, 1)
3
1
```

15.1.6 Python utilityfunctions

Python 注释 *35*：实用函数。

```
## util.py
print(f"Configuration:\n{mpi4py.get_config()}")
print(f"Include dir:\n{mpi4py.get_include()}")
```

通过 *macports* 安装了 Python 的 Mac OS X：

```
Configuration:
{'mpicc': '/opt/local/bin/mpicc-mpich-mp'}
Include dir:
/opt/local/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/mpi4py/include
```

Intel compiler and locally installed Python:

```
Configuration:
{'mpicc': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpicc',
'mpicxx': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpicxx',
'mpifort': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif90',
'mpif90': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif90',
'mpif77': '/opt/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpif77'}
Include dir:/opt/apps/intel19/impi19_0/python3/3.9.2/lib/python3.9/site-packages/mpi4py/include
```

15.2 错误处理

普通程序中的错误可能难以处理；并行程序中的错误则更难。这是因为除了单个可执行文件可能出现的所有问题（浮点错误、内存违规）之外，你还会遇到多个可执行文件之间错误交互引起的错误。

一些可能出错的例子：

- MPI 错误：MPI 例程可能因各种原因提前退出，例如接收到的数据远多于其缓冲区能容纳的量。这类错误，以及上述更常见的类型，通常会导致整个执行终止。也就是说，如果你的可执行文件的一个实例退出，MPI 运行时将终止所有其他实例。
- 死锁和其他挂起的执行：存在各种场景，进程个别不退出，但都在等待彼此。这可能发生在两个进程都在等待对方的消息时，这种情况可以通过使用非阻塞调用来缓解。在另一种场景中，由于程序逻辑错误，一个进程会等待比发送给它的消息（包括非阻塞消息）更多的消息。

虽然希望 MPI 实现能够返回错误，但这并不总是可能的。因此，某些场景，无论是提供某些过程参数，还是执行某种过程调用序列，都被简单地标记为 “ 错误 ”，且错误调用后的 MPI 状态是未定义的。

15.2.1 错误代码

有一堆错误代码。这些都是正的 `int` 值，而 `MPI_SUCCESS` 是零。任何内置错误代码的最大值是 `MPI_ERR_LASTCODE`。用户定义的错误代码都比这个大。

- `MPI_ERR_ARG`：一个无效的参数，且不属于其他错误代码覆盖的范围。
- `MPI_ERR_BUFFER` 缓冲区指针无效；这通常意味着您提供了一个空指针。
- `MPI_ERR_COMM`：无效的通信器。一个常见错误是在调用中使用了空通信器。
- `MPI_ERR_COUNT` 无效的计数参数，通常是由负计数值引起的；零通常是一个有效的计数。

- `MPI_ERR_INTERN` 检测到 MPI 内部错误。
- `MPI_ERR_IN_STATUS` 返回状态数组的函数至少有一个状态的 `MPI_ERROR` 字段被设置为非 `MPI_SUCCESS`。参见章节 4.3.3。

- `MPI_ERR_INFO`：无效的 info 对象。
- `MPI_ERR_NO_MEM` 由 `MPI_Alloc_mem` 返回，如果内存耗尽。
- `MPI_ERR_OTHER`：发生错误；使用 `MPI_Error_string` 检索有关此错误的更多信息；参见章节 15.2.2.3。

- `MPI_ERR_PORT`：无效端口；这适用于 `MPI_Comm_connect` 等。

以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

- `MPI_ERR_PROC_ABORTED` 当一个进程尝试与已中止的进程通信时返回 .

MPI-4 内容结束 • `MPI_ERR_RA`

`NK`：指定了无效的源或目标秩。有效秩为 $0 \ldots s-1$，其中 s 是通信器的大小，或 `MPI_PROC_NULL`，或接收操作的 `MPI_ANY_SOURCE`。

- `MPI_ERR_SERVICE`：无效的服务于 `MPI_Unpublish_name`； 章节 8.2.3。

15.2.2 错误处理

MPI 库有一个处理其检测到的错误的一般机制：可以为特定的 MPI 对象指定一个错误处理程序。

- 最常见的是，错误处理程序与通信器相关联： `MPI_Comm_set_errhandler`（同样也可以通过 `MPI_Comm_get_errhandler`）检索；

- 其他可能性有 `MPI_File_set_errhandler`, `MPI_File_call_errhandler`,

以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。`MPI_Session_set_errhandler`, `MPI_Session_call_errhandler`,

MPI-4 内容结束 `MPI_Win_se`

`t_errhandler`, `MPI_Win_call_errhandler`.

备注 27 例程 *MPI_Errhandler_set* 已被弃用，取而代之的是其 *MPI-2* 变体 *MPI_Comm_set_errhandler*。

某些类型为 MPI_Errhandler 的处理程序是预定义的（MPI_ERRORS_ARE_FATAL, MPI_ERRORS_ABORT,MPI_ERRORS_RETURN；见下文），但你可以使用 MPI_Errhandler_create 定义你自己的处理程序，稍后用 MPI_Errhandler_free 释放。

默认情况下，MPI 使用 MPI_ERRORS_ARE_FATAL，文件操作除外；参见第 10.5 节。

Python 注释 *36*：*Errorpolicy*。处理错误的策略可以通过 *mpi4py.rc* 对象（第 2.2.2 节）设置：

```
mpi4py.rc.errors # default: "exception"
```

可用的级别是 exception, default, fatal。

15.2.2.1 中止

默认行为是中止整个运行，这相当于您的代码中有以下调用

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_ARE_FATAL);
```

处理程序 MPI_ERRORS_ARE_FATAL，尽管它与一个通信器相关联，仍会导致整个应用程序中止。以下内容针对最近发布的 *MPI-4* 标准，可能尚未被支持。

处理程序 MPI_ERRORS_ABORT (MPI-4) 会中止其指定的通信器中的进程。*MPI-4* 内容结束

15.2.2.2 返回

另一种简单的可能性是指定 MPI_ERRORS_RETURN：

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
```

这会将错误代码返回给用户。这样你就有机会编写处理错误返回值的代码；参见下一节。

15.2.2.3 错误打印

如果 MPI_Errhandler 值 MPI_ERRORS_RETURN 被使用，你可以将返回代码与 MPI_SUCCESS 进行比较并打印调试信息：

```
int ierr;
ierr = MPI_Something();
if (ierr!=MPI_SUCCESS) {
    // print out information about what your programming is doing
    MPI_Abort();
}
```

例如，

```
Fatal error in MPI_Waitall:
See the MPI_ERROR field in MPI_Status for the error code
```

Figure 15.12 MPI_Comm_create_errhandler

Name	Param name	说明	C 类型	F 类型	输入输出
MPI_Comm_create_errhandler (					
	comm_errhandler_fn	user defined error handling procedure	MPI_Comm_errhandler_function*	PROCEDURE (MPI_Comm_errhandler_function)	IN
	errhandler	MPI error handler	MPI_Errhandler*	TYPE (MPI_Errhandler)	OUT
)					

然后你可以检索 MPI_ERROR 状态的字段，并使用 MPI_Error_string 或 maximalsize MPI_MAX_ERROR_STRING 打印出错误字符串：

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
ierr = MPI_Waitall(2*ntids-2,requests,status);
if (ierr!=0) {
   char errtxt[MPI_MAX_ERROR_STRING];
   for (int i=0; i<2*ntids-2; i++) {
     int err = status[i].MPI_ERROR;
     int len=MPI_MAX_ERROR_STRING;
     MPI_Error_string(err,errtxt,&len);
     printf("Waitall error: %d %s\n",err,errtxt);
   }
   MPI_Abort(MPI_COMM_WORLD,0);
}
```

一种可以处理错误的情况是 *MPI 文件 I/O*：如果输出文件的权限错误，代码可能会继续执行但不写入数据，或者写入到临时文件。

MPI 运算符（MPI_Op）不会返回错误代码。发生错误时，它们会调用 MPI_Abort；如果 MPI_ERRORS_RETURN 是错误处理程序，错误代码可能会被静默忽略。

你可以使用 MPI_Comm_create_errhandler （图 15.12）创建你自己的错误处理程序，然后用 MPI_Comm_set_errhandler 安装它。你可以用 MPI_Comm_get_errhandler 检索错误处理程序。

MPL 注 77：*Communicator errhandler*。MPL 没有设置错误处理程序的例程。相反，使用 *native_handle* 方法来检索嵌入的 communicator。

15.2.3 定义你自己的 MPI 错误

你可以定义自己的错误，使其行为类似于 MPI 错误。举例来说，我们编写一个发送例程，拒绝发送大小为零的数据。

定义新错误的第一步是使用 MPI_Add_error_class 定义一个错误类：

```
int nonzero_class;
MPI_Add_error_class(&nonzero_class);
```

此错误编号大于 MPI_ERR_LASTCODE，即内置错误代码的上限。属性 MPI_LASTUSEDCODE 记录了最后发出的值。

然后在此类中使用 MPI_Add_error_code 定义您的新错误代码，并且可以使用 MPI_Add_error_string 添加错误字符串：

```
int nonzero_code;
MPI_Add_error_code(nonzero_class,&nonzero_code);
MPI_Add_error_string(nonzero_code,"Attempting to send zero buffer");
```

然后你可以用这个代码调用错误处理程序。例如，创建一个包装的发送例程，该例程不会发送零大小的消息：

```
// errorclass.c
int MyPI_Send( void *buffer,int n,MPI_Datatype type, int target,int tag,MPI_Comm comm) {
  if (n==0)
    MPI_Comm_call_errhandler( comm,nonzero_code );
   MPI_Ssend(buffer,n,type,target,tag,comm);
  return MPI_SUCCESS;
};
```

这里我们使用了与通信器关联的默认错误处理程序，但可以用 `MPI_Comm_create_errhandler` 设置不同的错误处理程序。

我们测试我们的示例：

```
for (int msgsize=1; msgsize>=0; msgsize--) {
  double buffer;
  if (procno==0) {
    printf("Trying to send buffer of length %d\n",msgsize);
    MyPI_Send(&buffer,msgsize,MPI_DOUBLE, 1,0,comm);
    printf(".. success\n");
  } else if (procno==1) {
    MPI_Recv (&buffer,msgsize,MPI_DOUBLE, 0,0,comm,MPI_STATUS_IGNORE);
  }
}
```

结果为：

```
Trying to send buffer of length 1
.. success
Trying to send buffer of length 0
Abort(1073742081) on node 0 (rank 0 in comm 0):
Fatal error in MPI_Comm_call_errhandler: Attempting to send zero buffer
```

15.3 Fortran 问题

MPI 通常用 C 编写，如果你用 *Fortran* 编程怎么办?

参见章节 6.2.2.1 关于与 *Fortran90* 类型对应的 MPI 类型 .

15.3.1 假定形状数组

使用非连续数据，例如 `A(1:N:2)`，在 MPI 调用中尤其是非阻塞调用中是一个问题。在这种情况下，最好将数据复制到一个连续数组中。这在 MPI-3 中已被修复。

- Fortran 例程与 C 例程具有相同的签名，除了增加了一个整数错误参数。

- Fortran 中的调用 `MPI_Init` 没有命令行参数；它们需要单独处理。
- 例程 `MPI_Sizeof` 仅在 Fortran 中可用，它提供了 C/C++ 操作符 `sizeof` 的功能。

15.3.2 Preventcompiler optimizations

TFortran 编译器可以通过重新排列指令进行激进优化。这可能导致
MPI 代码中的不正确行为。在以下序列中：

```
call MPI_Isend( buf, ..., request )
call MPI_Wait(request)print *,buf(1)
```

wait 调用不涉及缓冲区，因此编译器可以将其转换为

```
call MPI_Isend( buf, ..., request )
register = buf(1)
call MPI_Wait(request)print *,register
```

防止这种情况可以使用 Fortran2018 机制。首先，缓冲区应声明为 `asynchronous`

```
<type>,Asynchronous :: buf
```

并引入

```
IF (.NOT. MPI_ASYNC_PROTECTS_NONBLOCKING) &
    CALL MPI_F_SYNC_REG( buf )
```

对 *MPI_F_sync_reg* 的调用将在编译时被移除，如果 `MPI_ASYNC_PROTECTS_NONBLOCKING` 为真。

15.4 进展

异步进展的概念描述了 *MPI* 消息在应用程序忙碌时仍然继续通过网络传输。

这里的问题是，与直接的 `MPI_Send` 和 `MPI_Recv` 调用不同，这种通信通常不能卸载到网卡，因此需要不同的机制。

这可以通过多种方式实现：

- 计算节点可能有专用的通信处理器。*Intel Paragon* 就是这种设计；现代多核处理器是这一理念的更高效实现。
- MPI 库可能会为通信处理保留一个核心或线程。这取决于具体实现；详见下文 *IntelMPI* 的信息。
- 保留一个核心，或在连续忙等待的 *spin loop* 中保留一个线程，会降低代码的潜在性能。因此，Ruhela 等人 [24] 提出使用 *pthreads* 信号来唤醒进程线程。
- 在没有此类专用资源的情况下，应用程序可以通过偶尔调用轮询例程（如 `MPI_Iprobe`）来强制 MPI 进展。

备注 28 该 *MPI_Probe* 调用在精神上与 *MPI_Test* 有些相似，尽管功能上不完全相同。然而，它们在进展方面的行为不同。引用标准内容：

> *MPI* 对 *MPI_Probe* 和 *MPI_Iprobe* 的实现需要保证进展：如果某个进程已经发出了对 *MPI_Probe* 的调用，并且某个进程已经启动了与探测匹配的发送，那么对 *MPI_Probe* 的调用将返回。

换句话说：探测会促使 *MPI* 取得进展。另一方面，

> 如果由请求标识的操作已完成，则对 *MPI_Test* 的调用返回 `flag = true`。

换句话说，如果已经取得进展，那么测试将报告完成，但测试本身并不会导致完成。

被动目标同步也会出现类似的问题：起始进程可能会挂起，直到目标进程调用 MPI。

以下命令强制进展： `MPI_Win_test`, `MPI_Request_get_status`。

Intel note. 仅适用于 Intel MPI 库的 `release_mt` 和 `debug_mt` 版本。将 `I_MPI_ASYNC_PROGRESS` 设置为 1 以启用异步进展线程，使用 `I_MPI_ASYNC_PROGRESS_THREADS` 设置进展线程的数量。

> 参见 https://software.intel.com/en-us/mpi-developer-guide-linux-asynchronou
>
> s-progress-control,https://software.intel.com/en-us/
>
> mpi-developer-reference-linux-environment-variables-for-asynchronous-progress-control

进展问题涉及： `MPI_Test`, `MPI_Request_get_status`, `MPI_Win_test`.

15.5 容错

处理器并非完全可靠，因此可能会出现某个处理器 “ 故障 ” 的情况：由于软件或硬件原因，它变得无响应。对于 MPI 程序来说，这意味着无法向其发送数据，且任何涉及它的集合操作都会挂起。我们能处理这种情况吗？可以，但这需要一些编程。

首先，可能的 MPI 错误返回码之一（第 15.2 节）是 `MPI_ERR_COMM`，当通信器中的某个处理器不可用时可能返回该错误。你可能想捕获此错误，并向程序中添加一个 “ 替代处理器 ”。为此，可以使用 `MPI_Comm_spawn` （详见 8.1 节）。但这需要改变程序设计：包含新进程的通信器不是旧的 `MPI_COMM_WORLD` 的一部分，因此最好一开始就将代码设置为一组互通信器。

15.6 性能、工具和分析

在本书的大部分内容中，我们讨论的是 MPI 库的功能。有些情况下，一个问题可以通过多种方式解决，然后我们会想知道哪种方式最有效。在本节中，我们将明确讨论性能。我们从两个关于性能测量本身的章节开始。

15.6.1 计时

MPI 有一个 实时时钟 计时器： `MPI_Wtime` (图 15.13)，它给出从过去某个时间点起的秒数。（注意 Fortran 调用中缺少误差参数。）

图 15.13 MPI_Wtime

名称 参数名	说明	C 类型	F 类型	输入输出
MPI_Wtime ( )				

Python:

```
MPI.Wtime()
```

图 15.14 MPI_Wtick

Name 参数名	解释	C 类型	F 类型	输入输出
MPI_Wtick ( )				

Python:

```
MPI.Wtick()
```

```
double t;
t = MPI_Wtime();
for (int n=0; n<NEXPERIMENTS; n++) {
  // do something;
}
t = MPI_Wtime()-t; t /= NEXPERIMENTS;
```

计时器的分辨率为 MPI_Wtick (图 15.14)。

MPL 注释 *78*：计时。 计时例程 wtime 和 wtick 以及 wtime_is_global 是环境方法：

```
double  mpl::environment::wtime ();
double  mpl::environment::wtick ();
bool mpl::environment::wtime_is_global ();
```

并行计时是一个棘手的问题。例如，大多数集群没有中央时钟，因此你无法将一个进程上的开始和停止时间与另一个进程上的时间对应起来。你可以如下测试全局时钟 MPI_WTIME_IS_GLOBAL：

```
int *v,flag;MPI_Attr_get( comm, MPI_WTIME_IS_GLOBAL, &v, &flag );
if (mytid==0) printf("Time synchronized? %d->%d\n",flag,*v);
```

通常你不需要担心这个计时器的起点：你在事件前后调用它，然后相减即可。

```
t = MPI_Wtime();
// something happens here
t = MPI_Wtime()-t;
```

如果你在单个处理器上执行这个操作，你会得到相当可靠的计时，除了你需要减去计时器的开销。这是测量计时器开销的常用方法：

```
t = MPI_Wtime();
// absolutely nothing here
t = MPI_Wtime()-t;
```

15.6.1.1 全局计时

然而，如果你尝试对一个并行应用进行计时，你很可能会得到每个进程不同的时间，因此你必须取平均值或最大值。另一种解决方案是通过使用 *barrier* 来同步处理器，方法是 `MPI_Barrier`：

```
MPI_Barrier(comm)
t = MPI_Wtime();
// something happens here
MPI_Barrier(comm)
t = MPI_Wtime()-t;
```

练习 15.1. 该方案也有一些相关的开销。你会如何测量它?

15.6.1.2 局部计时

现在假设你想测量一次单独发送的时间。无法在发送方启动时钟然后在接收方进行第二次测量，因为两个时钟不一定同步。通常会进行 *ping-pong* 操作：

```
if ( proc_source ) {
  MPI_Send( /* to target */ );
  MPI_Recv( /* from target */ );
else if ( proc_target ) {
  MPI_Recv( /* from source */ );
  MPI_Send( /* to source */ );
}
```

无论你进行何种计时，了解计时器的精度都是有益的。例程 `MPI_Wtick` 给出了最小可能的计时器增量。如果你发现计时结果过于接近这个 “ 滴答 ”，你需要找到更好的计时器（对于 CPU 测量，有周期精确的计时器），或者你需要增加运行时间，例如通过增加数据量。

15.6.2 简单分析

备注 29 本节描述了 *MPI* 工具接口引入之前的 *MPI* 分析。有关内容，请参见第 MPI tools interface 章。*14*。

MPI 允许您编写自己的性能分析接口。为了实现这一点，每个例程 `MPI_Something` 调用一个例程 `PMPI_Something` 来完成实际工作。您现在可以编写自己的 `MPI_...` 例程，该例程调用 *`PMPI_...`*，并插入您自己的性能分析调用。见图 15.1。

默认情况下，MPI 例程被定义为弱链接符号，作为 PMPI 例程的同义词。在 gcc 的情况下：

```
#pragma weak MPI_Send = PMPI_Send
```

如图 15.2 所示，通常只有 `PMPI` 例程出现在堆栈跟踪中。

15.6.3 编程以提升性能

我们概述了一些与性能相关的问题。

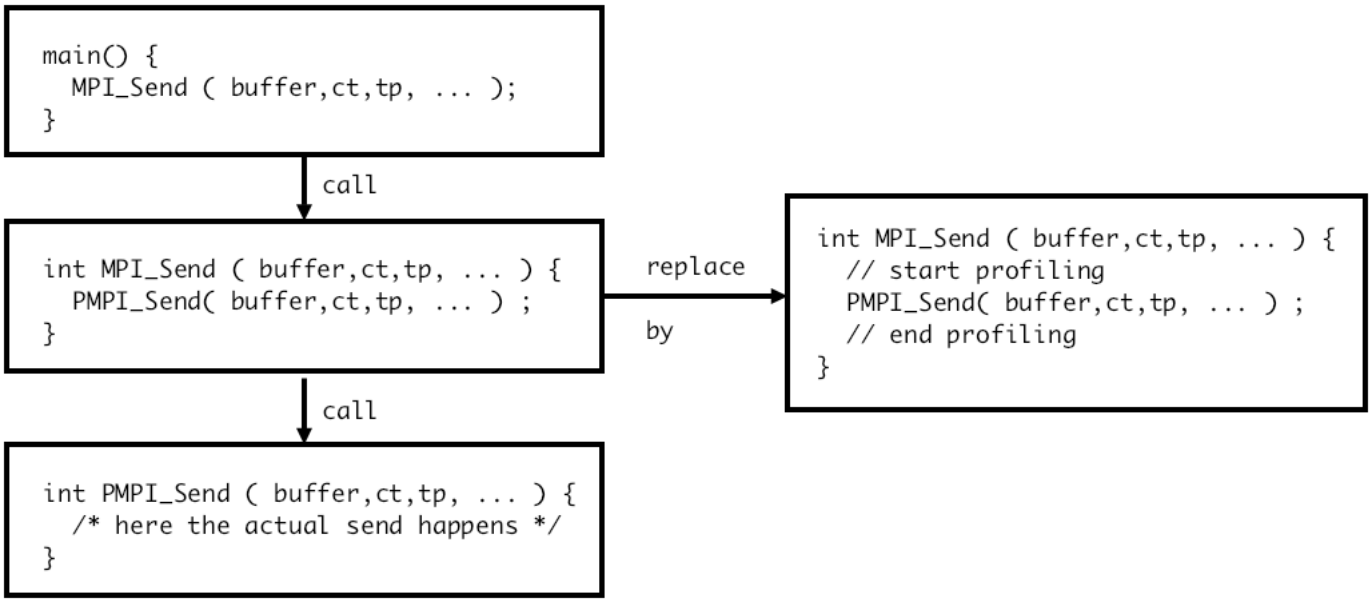


图 15.1：MPI 和 PMPI 例程的调用层次结构

Eager limit 短阻塞消息由比长消息更简单的机制处理。被视为 “ 短 ” 的限制称为 *eager limit* (章节 4.1.4.2)，你可以通过增加其值来调优你的代码。然而，请注意一个进程可能需要为每个其他进程准备一个缓冲区以容纳 eager 发送，这可能会占用你的可用内存。

阻塞与非阻塞 阻塞与非阻塞通信的问题有些误导。虽然非阻塞通信允许延迟隐藏，但我们不能将其视为阻塞发送的替代方案，因为用阻塞调用替换非阻塞调用通常会导致死锁。

即使你使用非阻塞通信仅仅是为了避免死锁或串行化（章节 4.1.4.3 ），也要记住通信与计算重叠的可能性。这也引出了我们的下一个要点。

换个角度看，在使用阻塞发送的代码中，即使结构上没有必要，使用非阻塞发送可能会获得更好的性能。

Progress MPI 并不会在后台神奇地自动激活，特别是当用户代码执行的是不涉及 MPI 的标量工作时。如第 15.4 节所示，有多种方法可以确保隐藏延迟真正发生。

Persistent sends 如果同一对进程之间涉及相同缓冲区的通信定期发生，可以设置一个 *persistent co mmunication*。参见第 5.1 节。

Buffering MPI 使用内部缓冲区，从用户数据复制到这些缓冲区可能会影响性能。例如，派生类型（第 6.3 节）通常不能直接通过网络流式传输（这需要

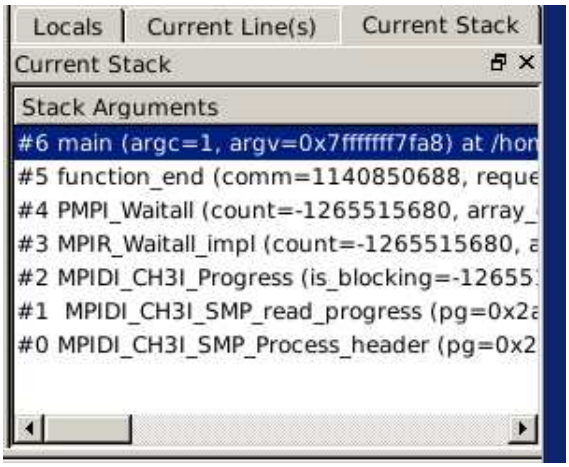


图 15.2：一个堆栈跟踪，显示了 `PMPI` 调用。

特殊硬件支持 [19]) 因此它们首先被复制。有些令人惊讶的是，我们发现 缓冲通信 (第 5.5 节) 并没有帮助。也许 MPI 实现者没有优化这种模式，因为它很少被使用。

This is issue is extensively investigated in [10].

图拓扑和邻域集合操作 负载均衡和通信最小化在不规则应用中非常重要。有专门的程序用于此（ *ParMetis*， *Zoltan* ），以及诸如 *PETSc* 这样的库可能提供方便的访问这些功能的途径。

在 图拓扑 (第 11.2 节) 的声明中，MPI 允许重新排序进程，这可以用来支持此类活动。当使用 邻域集合操作 时，它也可以用于更好的消息排序。

网络问题 到目前为止的讨论中，我们假设网络是数据的完美通道。然而，端口设计存在问题，特别是由 过度订阅 引起的，这会对性能产生不利影响。虽然在理想情况下可能可以设置例程来避免这种情况，但在超级计算机集群的实际操作中，网络争用 或来自不同用户作业的 消息冲突 是难以避免的。

卸载和加载有不同的网卡设计理念：作为网卡制造商的 *Mellanox*，相信将网络活动卸载到网络接口卡（ NIC ），而作为处理器制造商的 *Intel*，则相信将活动 “ 加载 ” 到处理器。两种观点各有争论。

无论哪种方式，都要调查你的网络能力。

15.6.4 MPIR

MPIR 是用于进程获取和消息队列提取的非正式指定调试接口。

15.7 确定性

MPI 进程只在一定程度上同步，因此您可能会想知道运行代码两次会得到相同结果的保证是什么。您需要考虑两种情况：首先，如果两次运行使用的处理器数量不同，已经存在数值问题；参见 HPC 书籍，第 3.6.5 节。

那么，我们将限制在同一组处理器上进行两次运行。在这种情况下，只要不使用通配符如 `MPI_ANY_SOURCE`，MPI 就是确定性的。形式上，MPI 消息是 “ 无超车 ” 的：同一发送者 - 接收者对之间的两条消息将按顺序到达。实际上，它们可能不会按顺序到达：它们在用户程序中是按顺序匹配的。如果第二条消息比第一条小得多，它实际上可能在较低的传输层更早到达。

非确定性的另一个来源来自混合计算；参见第 45.1 节。

15.8 处理器同步的微妙之处

阻塞通信涉及两个处理器之间复杂的对话。处理器一说 “ 我有这么多数据要发送；你有空间接收吗？ ”，处理器二回复 “ 有，我有；请发送 ”，随后处理器一执行实际的发送。这个来回过程（技术上称为握手）需要一定的通信开销。因此，网络硬件有时会放弃小消息的握手，直接发送，前提是知道另一进程为此类情况准备了一个小缓冲区。

这种策略的一个奇怪副作用是，根据 MPI 规范，代码本应死锁，但实际上并不会。实际上，你可能会被自己的编程错误所保护！当然，如果你运行一个更大的问题，而小消息变得超过阈值，死锁就会突然发生。因此，你会发现一个错误只在大问题上表现出来，而大问题通常更难调试。在这种情况下，将每个 `MPI_Send` 替换为 `MPI_Ssend` 将强制握手，即使是对于小消息。

相反，有时你可能希望避免在大消息上的握手。MPI 对此的解决方案是：（‘ready send’）例程立即发送其数据，但它需要接收方准备好。你如何保证接收进程已准备好？例如，你可以执行以下操作（这使用了非阻塞例程，详见下面第 4.2.1 节）：

```
if ( receiving ) {
  MPI_Irecv()   // post nonblocking receive
  MPI_Barrier() // synchronize
else if ( sending ) {
  MPI_Barrier() // synchronize
  MPI_Rsend()   // send data fast
```

当达到屏障时，接收操作已经发布，因此执行 ready send 是安全的。然而，全局屏障并不是一个好主意。相反，你只需同步涉及的两个进程。

练习 15.2. 给出一个伪代码方案，通过交换阻塞的零大小消息来同步两个进程。

15.9 Shell interaction

MPI 程序不是直接从 shell 运行的，而是通过 *ssh tunnel* 启动的。我们简要讨论其衍生影响。

15.9.1 标准输入

让 MPI 进程与环境交互并非完全简单。例如，*shell* 输入重定向可能不起作用。

```
mpiexec -n 2 mpiprogram < someinput
```

可能不起作用。

相反，使用一个带有一个参数的脚本 `programscript` eter:

```
#!/bin/bash
mpirunprogra
    m < $1
```

并行运行此命令：

```
mpiexec -n 2 programscript someinput
```

15.9.2 标准输出和错误

MPI 进程的 *stdout* 和 *stderr* 流通过 ssh 隧道返回。因此它们可以作为 `mpiexec` 的 *stdout/err* 被捕获。

```
// outerr.c
fprintf(stdout,"This goes to std out\n");
fprintf(stderr,"This goes to std err\n");
```

变量名依赖于实现，对于 `mpich` 及其派生版本如 *Intel MPI*，变量名是 `PMI_RANK`。（有一个类似的 `PMI_SIZE`。）

如果您只对显示 rank 感兴趣

- `srun` 有一个选项 `--label`。

15.9.3 进程状态

返回码 `MPI_Abort` 作为 *mpiexec* 的进程状态返回。运行中 g

```
// abort.c
if (procno==nprocs-1)
  MPI_Abort(comm,37);
```

as

```
mpiexec -n 4 ./abort ; \
echo "Return code from ${MPIRUN} is <<$$?>>"
```

给出 `TACC:  Starting up job 3760534`

```
TACC:  Starting parallel tasks...
application called MPI_Abort(MPI_COMM_WORLD, 37) - process 3
TACC:  MPI job exited with code: 37
TACC:  Shutdown complete. Exiting.
Return code from ibrun is <<37>>
```

15.9.4 多程序启动

如果 MPI 应用程序由子应用程序组成，也就是说，如果我们有一个真正的 *MPMD* 运行，通常有两种启动方式。（启动后，每个进程可以通过 `MPI_APPNUM` 来获取它属于哪个应用程序。）

第一种可能性是作业启动器， `mpiexec` 或 `mpirun` 或本地变体，接受多个可执行文件：

```
mpiexec spec0 [ : spec1 [ : spec2 : ... ] ]
```

如果没有这种机制，也可以使用第 15.9.1 节中的那种脚本来实现 *MPMD* 运行。我们让脚本启动多个程序中的一个，并利用 MPI 排名在环境中已知的事实；参见第 15.9.2 节。

Use a script `mpmdscript`:

```
#!/bin/bash

rank=$PMI_RANK
half=$(( ${PMI_SIZE} / 2 ))

if [ $rank -lt $half  ] ; then
    ./prog1
else
    ./prog2
fi
```

```
TACC:  Starting up job 4032931
TACC:  Starting parallel tasks...
Program 1 has process 1 out of 4
Program 2 has process 2 out of 4
Program 2 has process 3 out of 4
Program 1 has process 0 out of 4
TACC:  Shutdown complete. Exiting.
```

此脚本并行运行：

```
mpiexec -n 25 mpmdscript
```

15.10 剩余主题

15.10.1 MPI 常量

MPI 有许多内置的常量。这些常量的行为并不完全相同。

- 有些是编译时常量。例如 `MPI_VERSION` 和 `MPI_MAX_PROCESSOR_NAME`。因此，它们可以用于数组大小声明，甚至在 `MPI_Init` 之前。
- 有些链接时常量通过 MPI 初始化获得其值，例如 `MPI_COMM_WORLD`。这些符号，包括所有预定义的句柄，可以用于初始化表达式。
- 有些链接时符号不能用于初始化表达式，例如 `MPI_BOTTOM` 和 `MPI_STATUS_IGNORE`。

对于符号，二进制实现未定义。例如， `MPI_COMM_WORLD` 的类型是 `MPI_Comm`，但该类型的实现未指定。

完整列表请参见 MPI-3.1 标准的附录 A。

以下是编译时常量：

- `MPI_MAX_PROCESSOR_NAME`
- `MPI_MAX_LIBRARY_VERSION_STRING`
- `MPI_MAX_ERROR_STRING`

- MPI_MAX_DATAREP_STRING
- MPI_MAX_INFO_KEY
- MPI_MAX_INFO_VAL
- MPI_MAX_OBJECT_NAME
- MPI_MAX_PORT_NAME
- MPI_VERSION
- MPI_SUBVERSION

Fortran 注释 *16*：仅限 *Fortran* 的编译时常量。

- MPI_STATUS_SIZE. 随着 Fortran2008 的支持，不再需要；参见第 8 节。
- MPI_ADDRESS_KIND
- MPI_COUNT_KIND
- MPI_INTEGER_KIND
- MPI_OFFSET_KIND
- MPI_SUBARRAYS_SUPPORTED
- MPI_ASYNC_PROTECTS_NONBLOCKING

以下是链接时常量：

- MPI_BOTTOM
- MPI_STATUS_IGNORE
- MPI_STATUSES_IGNORE
- MPI_ERRCODES_IGNORE
- MPI_IN_PLACE
- MPI_ARGV_NULL
- MPI_ARGVS_NULL
- MPI_UNWEIGHTED
- MPI_WEIGHTS_EMPTY

各种常数：

- MPI_PROC_NULL 和其他 ..._NULL 常数。
- MPI_ANY_SOURCE
- MPI_ANY_TAG
- MPI_UNDEFINED
- MPI_BSEND_OVERHEAD
- MPI_KEYVAL_INVALID
- MPI_LOCK_EXCLUSIVE
- MPI_LOCK_SHARED
- MPI_ROOT

(本节内容灵感来源于 http://blogs.cisco.com/performance/mpi-outside-of-c-and-fortran.)

15.10.2 取消消息

在第 4.3.1 节中，我们展示了一个主 - 工作者示例，其中主进程以任意顺序接受来自工作者的消息。这里我们将展示一个稍微复杂一点的示例，只需要第一个完成任务的结果。因此，我们发出一个 MPI_Recv，源为 MPI_ANY_SOURCE。当结果到达时，我们将其源广播给所有进程。所有其他工作者随后使用此信息通过 MPI_Cancel 操作取消它们的消息。

```
// cancel.cfprintf(stderr,"get set, go!\n");if (procno==nprocs-1) {
MPI_Status status;MPI_Recv(dummy,0,MPI_INT, MPI_ANY_SOURCE,0,comm,
&status);first_tid = status.MPI_SOURCE;
MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm);
fprintf(stderr,"[%d] first msg came from %d\n",procno,first_tid);
} else {float randomfraction = (rand() / (double)RAND_MAX);
int randomwait = (int) ( nprocs * randomfraction );
MPI_Request request;fprintf(stderr,"[%d] waits for %e/%d=%d\n",
procno, randomfraction, nprocs, randomwait);sleep(randomwait);
MPI_Isend(dummy,0,MPI_INT, nprocs-1,0,comm,&request);
MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm);
if (procno!=first_tid) {MPI_Cancel(&request);
fprintf(stderr,"[%d] canceled\n",procno);}}
```

取消操作后仍然需要调用 `MPI_Request_free`， `MPI_Wait`，或 `MPI_Test` 以释放请求对象。

该 `MPI_Cancel` 操作是本地的，因此不能用于非阻塞集合操作或单边传输。

备注 30 从 *MPI-3.2* 开始，取消发送操作已被弃用 .

15.10.3 ShMem 中单边通信的起源

Cray T3E 有一个名为 *shmem* 的库，提供了一种共享内存。它不是通过真正的全局地址空间工作，而是通过支持保证在处理器之间相同且确实保证占据内存中相同位置的变量来实现。变量可以通过 “ 对称 ” pragma 或指令声明为共享；它们的值可以通过 `shmem_get` 和 `shmem_put` 调用来获取或设置。

15.11 文献

在线资源：

- MPI 1 完整参考：http://www.netlib.org/utk/papers/mpi-book/mpi-book.html
- 官方 MPI 文档：
  http://www.mpi-forum.org/docs/
- 所有 MPI 例程列表：
  http://www.mcs.anl.gov/research/projects/mpi/www/www3/

MPI 教程书籍：

- Using MPI [13] by some of the original authors.

第 16 章

MPI 示例

16.1 Bandwidth and halfbandwidth

带宽是衡量每秒可以通过连接的字节数的量。这个定义看似简单，但伴随着许多注释。

- 消息的大小很重要，因为仅仅启动消息就有一个延迟成本。通常，引用的带宽数字是一个渐近值，实际中很难达到。
- 如果在一对进程之间达到了某个带宽数值，那么两对进程同时发送时，会达到相同的数值吗?
- 带宽是否取决于选择测量的进程?
- 以及这些考虑的组合。

一个有用的度量是询问如果所有进程都在发送或接收，能够达到的带宽是多少。作为进一步的细化，我们询问通信对中最不利的选择是什么：

> 割半带宽定义为在所有可能将进程分成发送和接收两半的划分中，总带宽的最小值。

另见 HPC 书籍，第 2.7.1 节。

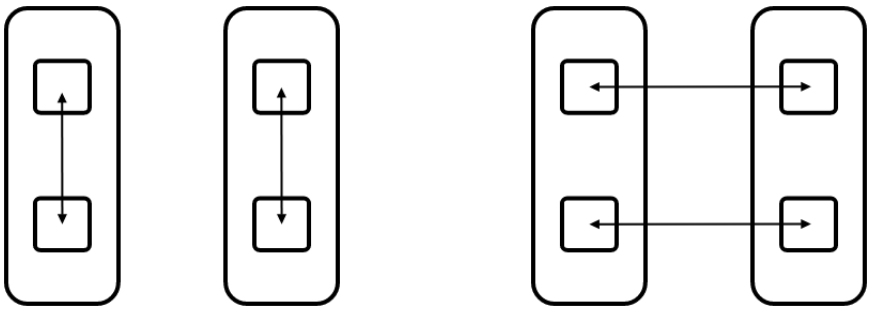


图 16.1：带宽的内部和外部方案

Figure16.1 展示了让所有进程成对通信的 ‘intra’ （左侧）和 ‘inter’ （右侧）方案。使用 intra 通信时，消息不依赖网络，因此我们预计测得较高的带宽。使用 inter 通信时，所有消息都通过网络传输，因此我们预计测得较低的数值。

However, there are more issues to explore, which we will now do.

首先我们需要找到进程对。连续的进程对：

```
// halfbandwidth.cxxint sender = procid - procid%2,
 receiver = sender+1;
```

相距 $P/2$ 的对：

```
int sender = procid<halfprocs ? procid : procid-halfprocs,
receiver = sender + halfprocs;
```

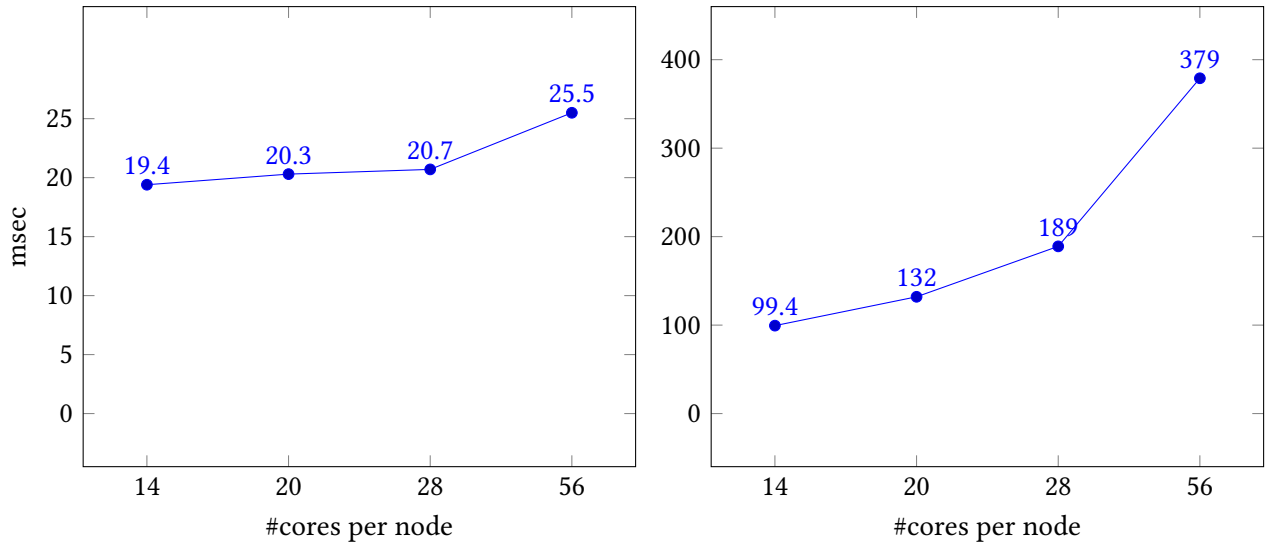


Figure 16.2: Time as a function of core count. Left: on node. Right: between nodes.

半带宽的测量方法是发送的总字节数除以总时间。两个数值均在一个重复循环之外测量，该循环执行每个事务 100 次。

```
auto duration = myclock::now()-start_time;
auto microsec_duration = std::chrono::duration_cast<std::chrono::microseconds>(duration);
int total_ping_count;MPI_Allreduce(&pingcount,&total_ping_count,1,MPI_INT,MPI_SUM,comm);
long bytes = buffersize * sizeof(double) * total_ping_count;
float fsec = microsec_duration.count() * 1.e-6,halfbandwidth = bytes / fsec;
```

在图 16.2 的左侧图中，我们看到 $P/2$ 同时进行的 pingpong 操作时间保持相当恒定。这反映了这样一个事实：在节点上，pingpong 操作是数据拷贝，且是同时进行的。因此，时间与移动数据的核心数量无关。唯一的例外是最后一个数据点：当所有核心都处于活动状态时，我们占用了节点上可用带宽的超过部分。

在右侧图中，每个 pingpong 都是跨节点的，通过网络进行。这里我们看到运行时间随着 pingpong 数量线性增加，甚至略微更糟。这反映了网络传输是顺序完成的。（实际上，消息可以被拆分成数据包，只要它们满足 MPI 消息语义。这并不改变我们的论点。）

接下来我们探讨缓冲区大小对性能的影响。图 16.3 右侧图显示跨节点带宽几乎不受缓冲区大小影响。这意味着即使是我们最小的缓冲区也足够大，可以克服任何 MPI 启动成本。

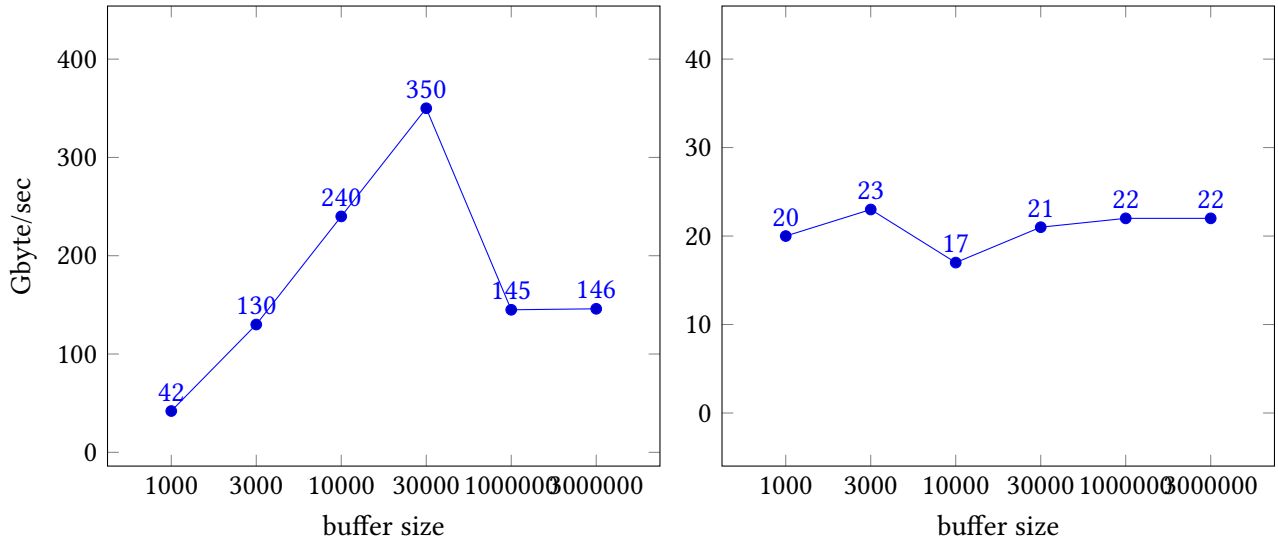


Figure 16.3: Bandwidth as a function of buffer size. Left: on node. Right: between nodes.

另一方面，左侧图显示了一个更复杂的模式。最初，带宽增加，可能反映了 MPI 启动开销的重要性降低。然而，对于最后的数据点，性能再次下降。这是因为数据大小超过了缓存大小，我们的性能受限于来自内存的带宽，而不是缓存。

第二部分

OPENMP

本书本节介绍 OpenMP （ “Open Multi Processing” ），这是科学与工程领域共享内存编程的主流模型。它将培养以下能力。

基础水平：

- 线程模型：学生将理解 OpenMP 的线程模型，以及线程与核心之间的关系（第 17 章）；并行区域的概念以及私有数据与共享数据的区别（第 18 章）。
- 循环并行性：学生将能够并行化循环，理解并行化的障碍，以及迭代调度（第 19 章；归约操作（第 20 章）。
- 学生将理解工作共享构造的概念及其对同步的影响（第 21 章）。

中级水平：

- 学生将理解同步的抽象概念、其在 OpenMP 中的实现及其对性能的影响（第 23 章）。
- 学生将理解任务模型作为线程模型的基础，能够编写生成任务的代码，并能区分何时需要任务与更简单的工作共享结构（第 24 章）。
- 学生将理解线程 / 代码亲和性、如何控制它，以及对性能的可能影响（第 25 章）。

高级水平：

- 学生将理解 OpenMP 内存模型和顺序一致性（第 28.8 章）。
- 学生将理解 SIMD 处理、编译器在 OpenMP 之外进行 SIMD 的程度，以及 OpenMP 如何指定进一步的 SIMD 化机会（第 26 章）。
- 学生将理解卸载到 GraphicsProcessing Units (GPUs) 的概念，以及实现此目的的 OpenMP 指令（章节 27 ）。

第 17 章

OpenMP 入门

本章解释了 OpenMP 的基本概念，并帮助您开始运行第一个 OpenMP 程序。

17.1 OpenMP 模型

我们首先建立一个关于 OpenMP 所针对的硬件和软件的心智图景。

17.1.1 Target hardware

现代计算机具有多层设计。也许你可以访问一个集群，也许你已经学会了如何使用 MPI 在集群节点之间通信。本章的主题 OpenMP 关注的是单个集群节点，以及如何充分利用那里可用的并行性。

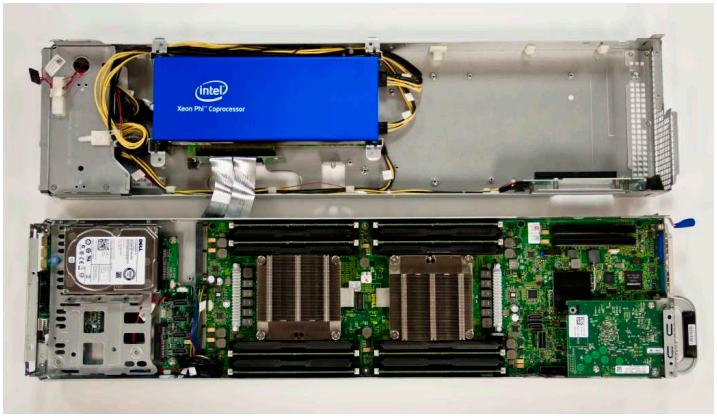


Figure 17.1: A node with two sockets and a co-processor

图 17.1 展示了一个节点的典型设计：在一个机箱内你会发现两个插槽，单个处理器芯片，加上一个加速器。（该图是 *TACC Stampede* 集群中一个已停用节点的图片，配备两个插槽和一个 *IntelXeon PHI* 协处理器。）

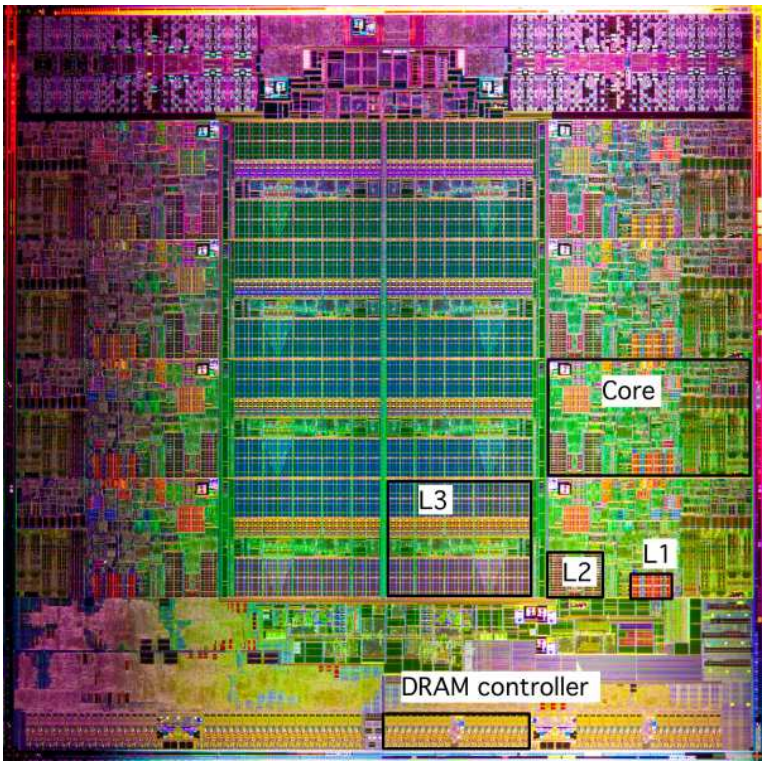


图 17.2：Intel Sandybridge 八核插槽结构

您的个人笔记本或台式电脑可能只有一个插槽；而大多数超级计算机的节点则有两个或四个插槽。在任何情况下，都可能有一个 GPU 作为协处理器；超级计算机集群还可能配备其他类型的加速器。OpenMP 版本 —— 从 OpenMP-4.0 开始 —— 支持针对这些可卸载设备。

要了解 OpenMP 关注该架构的哪些方面，我们需要深入了解插槽。图 17.2 展示了一个 *Intel Sandybridge* 插槽的图片。你会看到一个拥有八个 核心 的结构：独立的处理单元，所有核心都能访问相同的内存。（在图 17.1 中，你看到每个两个插槽上各连接了四个内存芯片或 DIMM；所有十六个核心都能访问所有这些内存。）OpenMP 使得在同一个程序中轻松利用所有这些核心成为可能。OpenMP-4.0 标准还增加了将计算卸载到 GPU 或其他加速器的功能。

总结 OpenMP 所针对的架构结构：

- 一个节点有若干个插槽，通常是 1 、 2 或 4 个；
- 每个插槽有若干个核心，截至 2022 年，这个数量可达 64 个；
- 每个核心是一个独立的处理单元，可以访问节点上的所有内存。
- 可能有一个加速器，可以用来卸载计算任务。

OpenMP 不针对的是 *cluster* 结构，其中节点通过可以访问网络的库进行通信，例如 Message Passing Interface (MPI)

17.1.2 Target software

OpenMP 基于两个概念：使用 *threads* 和并行的 *fork/join model* 。目前你可以将线程视为一种进程：处理单元执行一系列指令。fork/join 模型表示线程可以将自身 “fork” 成多个相同的线程副本。在某个时刻，这些副本消失，原始线程留下（ “join” ），但在由 fork 创建的 *team of threads* 存在期间，你可以利用并行性。fork 和 join 之间的执行部分称为 *parallel region*。

图 17.3 给出了一个简单的示意图：一个线程分叉成一个线程团队，这些线程本身也可以再次分叉。

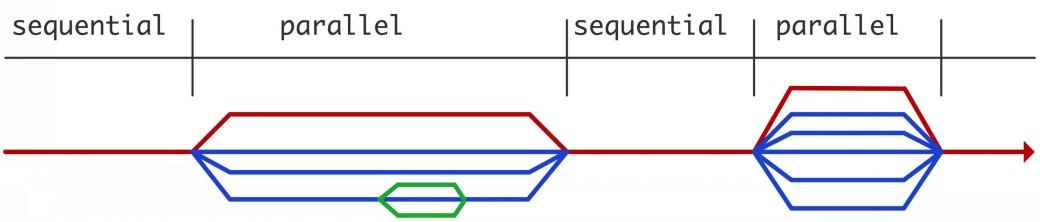


图 17.3：并行执行期间的线程创建和删除

被 fork 的线程都是 *master thread* 的副本：它们可以访问到迄今为止计算的所有内容；这就是它们的 *shared data*。当然，如果线程完全相同，并行性就毫无意义，因此它们也有私有数据，并且可以识别自己：它们知道自己的线程编号。这使你能够使用线程进行有意义的并行计算。

这将我们带到了第三个重要概念：*work sharing* 结构。在一个线程团队中，最初会有重复执行；work sharing 结构将可用的工作分配给各个线程。

> 所以你明白了：OpenMP 使用线程团队，在并行区域内，工作通过 work sharing 结构分配给各个线程。线程可以访问共享数据，并且它们有一些私有数据。

OpenMP 和 MPI 之间的一个重要区别是，OpenMP 中的并行性是由一个线程动态激活，生成一个线程团队。此外，使用的线程数可以在不同的并行区域之间变化，线程可以递归地创建线程。相比之下，在 MPI 程序中，运行的进程数在整个运行期间（大多数情况下）是固定的，并且由程序外部的因素决定。

17.1.3 About threadsand cores

OpenMP 编程通常是为了利用 *multicore* 处理器。因此，为了获得良好的加速，通常会让线程数等于核心数。然而，如果这符合你的算法的自然表达，也没有什么阻止你创建更多线程：操作系统会使用 *time slicing* 让它们都被执行。只是你不会获得超过实际可用核心数的加速。

在一些现代处理器上存在 *hardwarethreads*，这意味着一个核心实际上可以让多个线程执行，相较于单线程有一定的加速。为了高效利用这样的处理器，你可以让 OpenMP 线程数是核心数的 2 倍或 4 倍，具体取决于硬件。

17.2 OpenMP 程序运行的后勤

在我们开始研究 OpenMP 之前，需要先处理一些关于 OpenMP 程序的形式问题。

17.2.1 Compiling

一个 C 程序需要包含：

```
#include "omp.h"
```

而 Fortran 程序需要包含：

```
use omp_lib
```

or`#include "omp_lib.h"`

OpenMP 通过对常规编译器的扩展来处理，通常通过在命令行中添加一个选项：
`# gccgcc -o foo foo.c -fopenmp# Intel compiler`
`icc -o foo foo.c -qopenmp` 如果你有单独的编译和链接阶段，则需
要在两者中都添加该选项。

17.2.2 标准

OpenMP 系统经历了多个标准版本，您在本课程中阅读到的一些功能可能在您的编译器中不可用。

当您想要检查这一点时，可以按如下方式查询 *OpenMP* 标准。当您使用上述编译器选项时，*OpenMP* 宏（或 *cpp* 宏） `_OPENMP` 将被定义。因此，您可以通过编写条件编译来实现

```
#ifdef _OPENMP
   ...
#else
   ...
#endif
```

该宏的值是一个十进制值 `yyyymm`，表示该编译器支持的 OpenMP 标准版本；参见第 28.7 节。

Fortran 注释 *17*：*OpenMP* 版本。参数 `openmp_version` 包含 `yyyymm` 格式的版本号。

```
!! version.F90
  use omp_lib
  implicit none
  integer :: standard
  standard = openmp_version
```

17.2.3 运行一个 OpenMP 程序

你通过常规方式调用来运行一个 OpenMP 程序（例如 `./a.out`），但它的行为会受到一些 *OpenMP* 环境变量的影响。最重要的一个是 `OMP_NUM_THREADS`：

```
export OMP_NUM_THREADS=8
```

它设置程序将使用的线程数。你通常会将其设置为硬件中的核心数，并期望获得大致线性的加速比。

参见第 28.1 节，了解所有环境变量的列表。

17.3 你的第一个 OpenMP 程序

在本节中，你将看到足够的 OpenMP 内容来编写第一个程序并探索其行为。为此，我们需要介绍几个 OpenMP 语言构造。它们将在后续章节中更详细地讨论。

17.3.1 指令

OpenMP 不是魔法，因此你必须告诉它何时可以并行执行某些操作。这主要是通过 *directives* 完成的；额外的说明可以通过库调用来实现。

在 C/C++ 中使用 *pragma* 机制：这是为了编译器的注释，否则不属于语言的一部分。它看起来像这样：

```
#pragma omp somedirective clause(value,othervalue)
  statement;

#pragma omp somedirective clause(value,othervalue)
 {
  statement 1;
  statement 2;
 }
```

with

- the `#pragma omp` *sentinel* 用来指示即将出现一个 OpenMP 指令；
- 一个指令，例如 `parallel`；
- 并且可能带有带值的子句。
- 指令后面跟着单条语句或用花括号括起来的代码块。

C/C++ 中的指令区分大小写。指令可以通过转义行尾来分成多行。

Fortrannote 18: OpenMP sentinel. Fortran 中的哨兵看起来像一个 commen t:

```
!$omp directive clause(value)
  statements
!$omp end directive
```

与 C 指令的区别在于 Fortran 没有代码块，因此有一个显式的指令结束行。

如果你将一个指令分成多行，除最后一行外，所有行都需要有续行字符，并且每行都需要有哨兵：

```
!$omp parallel &
!$omp     num_threads(7)
  tp = omp_get_thread_num()
!$omp end parallel
```

这些指令不区分大小写。在 *Fortran* 固定格式源代码 文件中（这是 Fortran77 中唯一的可能），`c$omp` 和 `*$omp` 也被允许。

17.3.2 并行区域

在 OpenMP 中创建并行性的最简单方法是使用 `parallel` pragma。一个由 `parallel`pragma 前置的代码块称为 并行区域；它由新创建的线程团队执行。这是 单程序多数据（*SPMD*） 模型的一个实例：所有线程（冗余地）执行相同的代码段。

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

练习 17.1. 编写一个 “hello world” 程序，其中打印语句位于并行区域内。编译并运行。使用环境变量 `OMP_NUM_THREADS` 的不同值运行你的程序。如果你知道你的机器有多少个核心，能否将该值设置得更高?

让我们开始探索 OpenMP 如何处理并行性，使用以下函数：

- `omp_get_num_threads` 报告当前有多少线程处于活动状态，且
- `omp_get_thread_num` 报告调用该函数的线程编号。
- `omp_get_num_procs` 报告可用核心的数量。

练习 17.2. 取练习 17.1 中的 hello world 程序，并在并行区域的前、中、后插入上述函数。你的观察是什么?

练习 17.3. 扩展练习 17.2 中的程序。基于以下代码行编写一个完整的程序：

Code:

```
// reduct.c
int tsum=0;
#pragma omp parallel
{
  tsum += // expression
}
printf("Sum is %d\n",tsum);
```

Output:

```
With 4 threads, sum s/b 6
Sum is 6
Sum is 5
Sum is 1
Sum is 4
Sum is 6
Sum is 5
Sum is 6
Sum is 5
Sum is 3
Sum is 4
```

重新编译并运行。（实际上，多次运行你的程序。）你是否看到了一些意料之外的现象？你能想到解释吗?

如果上述内容让你感到困惑，请阅读 HPC 书中关于 *race condition* 的内容，章节 -2.6.1.5。

17.3.3 代码和执行结构

这里有几个重要的概念：

- 一个 OpenMP 指令后面跟着一个结构化块；在 C 语言中，这可以是单个语句、复合语句或用大括号括起来的块；在 Fortran 中，它由指令及其匹配的 ‘end’ 指令界定。结构化块不能被跳入，因此它不能以带标签的语句开始，也不能包含跳出该块的跳转语句。
- OpenMP 构造是以指令开始并跨越随后的结构化代码块的代码段，在 Fortran 中还包括结束指令。这是一个词法概念：它包含直接包含的语句，而不包括从中调用的任何子程序。
- 一个代码区域被定义为在执行代码时动态遇到的所有语句 e
  OpenMP 构造的。这是一个动态概念：与 “ 构造 ” 不同，它确实包括任何子程序 s
  这些子程序是从结构化块中的代码调用的。

17.4 线程数据

在大多数编程语言中，数据的可见性由变量作用域规则控制：变量在一个块中声明，然后在该块及其包含的具有词法作用域的子块中的任何语句中可见，但在外围块中不可见：

```
main () {
  // no variable `x' define here
  {
    int x = 5;
    if (somecondition) { x = 6; }
    printf("x=%e\n",x); // prints 5 or 6
  }
  printf("x=%e\n",x); // syntax error: `x' undefined
}
```

Fortran 的规则更简单，因为它没有块中嵌套块。

OpenMP 对并行区域和其他 OpenMP 结构中的数据有类似的规则。首先，数据在封闭的作用域中是可见的：

```
main() {
  int x;
#pragma omp parallel
  {
     // you can use and set `x' here
  }
  printf("x=%e\n",x); // value depends on what
                      // happened in the parallel region
}
```

在 C 语言中，你可以在嵌套作用域内重新声明一个变量：

```
{int x;if (something) {
double x; // same name, different entity}
```

```
  x = ... // this refers to the integer again
}
```

这样做会使外部变量无法访问。

OpenMP 有类似的机制：

```
{
  int x;
#pragma omp parallel
  {
    double x;
  }
}
```

有一个重要的区别：团队中的每个线程都会获得所包含变量的一个独立实例。

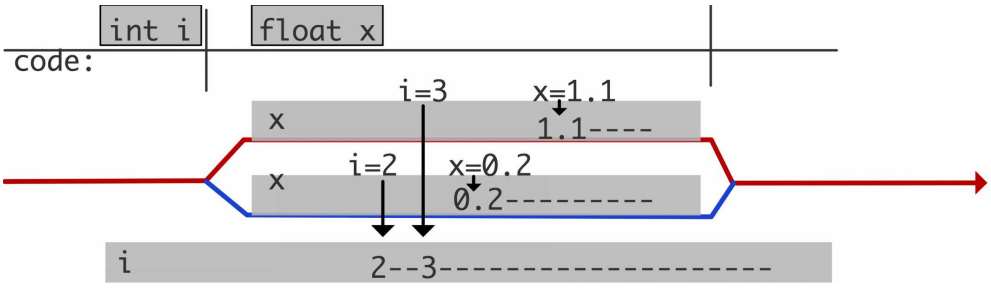


图 17.4：线程中变量的局部性

这在图 17.4 中有所说明。

除了存活在栈上的此类作用域变量外，还有存活在堆上的变量，通常由调用 `malloc`（在 C 中）或 `new`（在 C++ 中）创建。它们的规则更为复杂。

总结以上内容，有

- 共享变量，每个线程引用相同的数据项，　　d
- 私有变量，每个线程拥有自己的实例。

除了使用作用域外，OpenMP 还使用指令上的选项来控制数据是私有还是共享。

OpenMP 并行编程的许多困难源于共享变量的使用。例如，如果两个线程更新一个共享变量，更新的顺序是无法保证的。

我们将在第 22 节详细讨论这些内容。

17.5 创建并行性

OpenMP 的 *fork/join* 模型 意味着你需要某种方式来指示一个活动可以被 fork 以独立执行。有两种方法可以做到这一点：

1. 你可以声明一个并行区域，将一个线程拆分成一整个线程团队。我们将在第 18 章中讨论这个。线程间工作的划分由 工作共享结构 控制；详见第 21 章。

2. 另外，你可以使用任务并一次指定一个并行活动。你将在第 24 节看到这一点。

注意，OpenMP 仅指示存在多少并行性；是否独立活动实际上并行执行是运行时的决定。
执行是运行时的决定。

声明一个并行区域告诉 OpenMP 可以创建一个线程团队。团队的实际大小取决于各种因素（参见第 28.1 节，关于本节提到的变量和函数）。

- 环境变量 `OMP_NUM_THREADS` 限制了可以创建的线程数量。
- 如果你没有设置这个变量，也可以通过库例程 `omp_set_num_threads` 动态设置这个限制。如果两者都指定了，该例程优先于上述环境变量。
- 线程数的限制也可以作为并行区域的 *num_threads* 子句设置：

```
#pragma omp parallel num_threads(ndata)
```

要查询你的并行区域实际使用了多少并行度，使用 `omp_get_num_threads`。要查询这些硬件限制，使用 `omp_get_num_procs`。你可以使用 `omp_get_max_threads` 查询最大线程数。这等于 `OMP_NUM_THREADS` 的值，而不是并行区域中实际活跃线程的数量。

```
// proccount.c
void nested_report() {
#pragma omp parallel
#pragma omp master
  printf("Nested    : %2d cores and %2d
    ↪threads out of max %2d\n",
        omp_get_num_procs(),
        omp_get_num_threads(),
        omp_get_max_threads());
}
  int env_num_threads;
#pragma omp parallel
#pragma omp master
  {
    env_num_threads =
    ↪omp_get_num_threads();
    printf("Parallel  : %2d cores and %2d
    ↪threads out of max %2d\n",
           omp_get_num_procs(),
           omp_get_num_threads(),
           omp_get_max_threads());
  }

#pragma omp parallel \
  num_threads(2*env_num_threads)
#pragma omp master
  {
    printf("Double    : %2d cores and %2d
    ↪threads out of max %2d\n",
           omp_get_num_procs(),
           omp_get_num_threads(),
           omp_get_max_threads());
  }
```

```
#pragma omp parallel
#pragma omp master
  nested_report();
```

```
[c:48] for t in 1 2 4 8 16 ; do OMP_NUM_THREADS=$t ./proccount ; done
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  1
Parallel  : count  4 cores and  1 threads out of max  1
Parallel  : count  4 cores and  1 threads out of max  1
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  2
Parallel  : count  4 cores and  2 threads out of max  2
Parallel  : count  4 cores and  1 threads out of max  2
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  4
Parallel  : count  4 cores and  4 threads out of max  4
Parallel  : count  4 cores and  1 threads out of max  4
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  8
Parallel  : count  4 cores and  8 threads out of max  8
Parallel  : count  4 cores and  1 threads out of max  8
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max 16
Parallel  : count  4 cores and 16 threads out of max 16
Parallel  : count  4 cores and  1 threads out of max 16
```

当使用嵌套并行区域时，线程数量会受到另一个限制。如果你在一个子程序中有一个并行区域，而该子程序有时以顺序方式调用，有时以并行方式调用，就会出现这种情况。详情请参见第 18.2 节。

第 18 章

OpenMP 主题：并行区域

18.1 使用并行区域创建并行性

在 OpenMP 中，你需要明确指出哪些代码段是并行的。创建并行性，这里指的是：创建一个线程团队，是通过 `parallel` pragma 完成的。一个由 `omp parallel` pragma 前置的代码块称为 并行区域；它由新创建的线程团队执行。这是 *SPMD* 模型的一个实例：所有线程执行相同的代码段。

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

让所有线程完全相同地执行该代码块是没有意义的。一种获得有意义的并行代码的方法是使用函数 `omp_get_thread_num` 来确定你是哪个线程，并执行该线程特有的工作。该函数返回相对于当前团队的编号；回想图 17.3 中说明新团队可以递归创建。

还有一个函数 `omp_get_num_threads` 用来找出线程的总数 ds.

我们首先要做的是创建一个线程团队。这是通过一个 *parallel region* 来完成的。这里有一个非常简单的例子，每个线程输出它的编号：**Code:**

```
// hello.c
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n",t);
  }
```

输出：

```
Hello world from 1!
Hello world from 0!
Hello world from 2!
Hello world from 3!
```

或者在 Fortran 中 **Code:**

```
!! hello.F90!$omp parallel
print *,"Hello world!"
!$omp end parallel
```

Output:

```
Hello world from           1
Hello world from           2
Hello world from           3
Hello world from           0
```

C++ 注 2：并行中的输出流。使用 *cout* 可能会导致输出混乱：每个 << 处都可能断行。使用 `stringstream` 来形成单一流进行输出。

```
// hello.cxx
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    stringstream proctext;
    proctext << "Hello world from " << t << '\n';
    cerr << proctext.str();
  }
```

C++ 注 *3*：*lambda* 中的并行区域。OpenMP 并行区域可以出现在函数中，包括 lambda 表达式。

```
const int s = [] () {
  int s;
#   pragma omp parallel
#   pragma omp master
  s = 2 * omp_get_num_threads();
  return s; }();
```

（‘Immediately Invoked Function Expression’）

以下示例使用并行性进行实际计算：

```
result = f(x)+g(x)+h(x)
```

你可以将其并行化为

```
double result,fresult,gresult,hresult;
#pragma omp parallel
{ int num = omp_get_thread_num();
if (num==0)      fresult = f(x);
else if (num==1) gresult = g(x);
else if (num==2) hresult = h(x);}
result = fresult + gresult + hresult;
```

这段代码对应于我们刚才讨论的模型：

- 紧接在并行块之前，一个线程将执行代码。在主程序中，这就是 *initialthread*。
- 在该块开始时，会创建一个新的 *team of threads*，并且在该块之前处于活动状态的线程成为该团队的主线程。
- 块结束后，只有主线程处于活动状态。
- 在块内部有一个线程团队：团队中的每个线程执行该块的主体，并且可以访问周围环境的所有变量。线程数量可以通过多种方式确定；我们稍后会讲到。

备注 31 在未来版本的 *OpenMP* 中，主线程将被称为主线程（primary thread）。在 *5.1* 版本中，*master* 结构将被弃用，`masked`（带有新增功能）将取代它。在 *6.0* 版本中，*master* 将从规范中消失，包括 `proc_bind` *master* “变量”及组合 *master* 结构（*master taskloop* 等）。

练习 18.1. 如果你在并行区域外调用 `omp_get_thread_num` 和 `omp_get_num_threads` 会发生什么？

18.2 嵌套并行

如果你在一个并行区域内调用一个函数，而该函数本身又包含一个并行区域，会发生什么?

```
int main() {
  ...
#pragma omp parallel
  {
  ...
  func(...)
  ...
  }
} // end of main
void func(...) {
#pragma omp parallel
  {
  ...
  }
}
```

由于任何线程都可以创建一个团队，你可能会期望每个线程在调用 *func* 时都会创建它自己的新团队。这被称为嵌套并行，并且它按描述工作。

然而，默认情况下，嵌套的并行区域只有一个线程。你需要显式允许非平凡的嵌套并行。

要允许嵌套线程创建，使用环境变量 `OMP_MAX_ACTIVE_LEVELS`（默认值：1）来设置并行嵌套的层数。同样，也有函数 `omp_set_max_active_levels` 和 `omp_get_max_active_levels`：

```
OMP_MAX_ACTIVE_LEVELS=3
```

or

```
void omp_set_max_active_levels(int);
int omp_get_max_active_levels(void);
```

备注 32 一种已弃用的机制是设置环境变量 *OMP_NESTED*（默认值：*false*）或其对应的函数：

```
OMP_NESTED=true
 or
omp_set_nested(1)
```

嵌套并行可以发生在嵌套循环中，但也可以是 `sections` 构造和循环嵌套。示例：

Code:

```
// sectionnest.c
#pragma omp parallel sections reduction(+:s)
    {
    #pragma omp section
      {
        double s1=0;
        omp_set_num_threads(team);
        #pragma omp parallel for reduction(+:s1)
        for (int i=0; i<N; i++) {
```

Output:

```
Nesting: false
Threads:  2, speedup:   2.0
Threads:  4, speedup:   2.0
Threads:  8, speedup:   2.0
Threads: 12, speedup:   2.0
Nesting: true
Threads:  2, speedup:   1.8
Threads:  4, speedup:   3.7
Threads:  8, speedup:   6.9
Threads: 12, speedup:  10.4
```

嵌套并行的数量可以设置为：
`OMP_NUM_THREADS=4,2`
意味着最初一个并行区域将有四个线程，每个线程可以创建两个更多线程。仍然需要设置 activelevels 的数量。

同时活跃的线程总数（技术上讲：在一个 *contention group* 中）可以通过以下方式限制：
`OMP_THREAD_LIMIT=123`
Its value can be queried with `omp_get_thread_limit`.

More functions: `omp_get_level`, `omp_get_active_level`, `omp_get_ancestor_thread_num`, `omp_get_team_size(level)`.

18.2.1 带有并行区域的子程序

嵌套并行性的一个常见应用是你有一个包含并行区域的子程序，而该子程序本身又被从一个并行区域中调用。

Exercise18.2. 通过编写如下的 OpenMP 程序来测试嵌套并行性：
1. 编写一个包含并行区域的子程序。 2. 编写一个带有并行区域的主程序；在并行区域内外都调用该子程序。
3. 插入打印语句 (a) 在主程序的并行区域外，(b) 在主程序的并行区域中，(c) 在子程序的并行区域外，(d) 在子程序内部的并行区域中。

运行你的程序并统计每种类型的打印语句出现了多少次。

编写指令是相对于调用的动态作用域进行评估的，而不仅仅是词法作用域。在下面的例子中：

```
#pragma omp parallel
{f();}void f() {
#pragma omp for
for ( .... ) {...}}
```

函数 `f` 的主体位于并行区域的动态作用域内，因此 for 循环将被并行化。

如果函数可能既从并行区域内调用，也从并行区域外调用，可以使用 `omp_in_parallel` 来测试是哪种情况。

C++ 注释 *4*：类方法的动态作用域。类方法的动态作用域与其他函数相同：**代码：**

```
// nested.cxx
class withnest {
public:
  void f() {
    stringstream ss;
    ss
      << omp_get_num_threads()
      << '\n';
    cout << ss.str();
  };
};
int main() {
  withnest my_object;
#pragma omp parallel
  my_object.f();
```

输出：

```
executing: OMP_MAX_ACTIVE_LEVELS=2
    ↪OMP_PROC_BIND=true
    ↪OMP_NUM_THREADS=2 ./nested
2
2
```

18.3 取消并行构造

可以使用 *cancel* 指令提前终止并行构造 e:

```
!$omp cancel construct [if (expr)]
```

其中 construct 是 `parallel`, `sections`, `do` 或 `taskgroup`。

参见第 30.3 节的示例。

出于性能原因，取消功能默认是禁用的。要启用它，请将 `OMP_CANCELLATION` 变量设置为 true。

取消状态可以通过 `omp_get_cancellation` 查询，但没有设置该状态的函数。

取消最多可能发生在 OpenMP 活跃的最明显位置，但可以通过以下方式设置额外的取消点

```
#pragma omp cancellation point <construct>
```

其中构造是 `parallel`, `sections`, `for`, `do`, `task` `group`.

18.4 复习问题

练习 18.3. 对 / 错? 函数 `omp_get_num_threads()` 返回的数字等于核心数。

Exercise 18.4. T/F? 函数 `omp_set_num_threads()` 不能设置为比核心数更高的数字。

Exercise18.5. 可以使用什么函数来检测核心数?

第 19 章

OpenMP 主题：循环并行

循环并行是科学代码中非常常见的一种并行类型，因此 OpenMP 为此提供了一个简单的机制。OpenMP 并行循环是 OpenMP “ 工作共享 ” 构造的第一个示例（完整列表见第 21.1 节）：这些构造将一定量的工作分配给并行区域中可用的线程，该并行区域由 `parallel` pragma 创建。

循环的并行执行可以通过多种不同方式处理。例如，你可以在循环周围创建一个并行区域，并调整循环边界：

```
#pragma omp parallel
{
  int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();
  int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;
  for (int i=low; i<high; i++)
    // do something with i
}
```

实际上，这就是你在 MPI 中并行化循环的方式：`parallel` pragma 创建一个线程团队，每个线程执行代码块，并根据其线程编号找到一个唯一的工作块来完成。

Exercise 19.1. OpenMP 和 MPI 代码之间的一些重要区别是什么？

19.1 通过指令实现循环并行

在 OpenMP 中并行化循环的自然方式是使用 `for` pragma，OpenMP 会为你完成上述循环的划分：

```
#pragma omp parallel
#pragma omp for
for (int i=0; i<N; i++) {
  // do something with i
}
```

该代码片段结合了两种 OpenMP 习惯用法：

1. 首先，`parallel` 指令创建了一个线程组；接着 2. `for` 指令是一个工作共享构造：它将可用的工作分配给可用的线程。

备注 33 在此示例中，循环变量在循环头中声明，这是推荐的做法，但如果你不这样做，循环变量会自动对线程私有。

将工作分配留给 OpenMP 有几个优点。首先，你不必自己计算线程的循环段，还可以告诉 OpenMP 根据不同的调度方式分配循环迭代（见第 19.3 节）。

Fortran 注释 19：*OMP do pragma*。 该 `for` pragma 仅存在于 C 中；Fortran 中有一个对应名称的 `do` pragma。

```
$!omp parallel
$!omp dodo i=1,N
! something with i
end do$!omp end do
$!omp end parallel
```

19.1.1 关于并行性和工作共享

重要的是要认识到 `parallel` <style id='4'} 指令并不会立即分配任何工作：所有线程一开始都执行相同的代码。作为说明，图 19.1 显示了四个线程执行在 `parallel` 和 `for` 指令之间的代码：

```
#pragma omp parallel
{
  code1();
#pragma omp for
  for (int i=1; i<=4*N; i++) {
    code2();
  }
  code3();
}
```

循环前后的代码在每个线程中执行相同；循环迭代被分配到四个线程上。

`do` 和 `for` 指示本身并不创建并行性：它们接管当前活动的线程组，并将循环迭代分配给它们。这意味着 `omp for` 或 `omp do` 指令需要位于并行区域内。在并行区域外，它们将顺序执行。

作为说明：

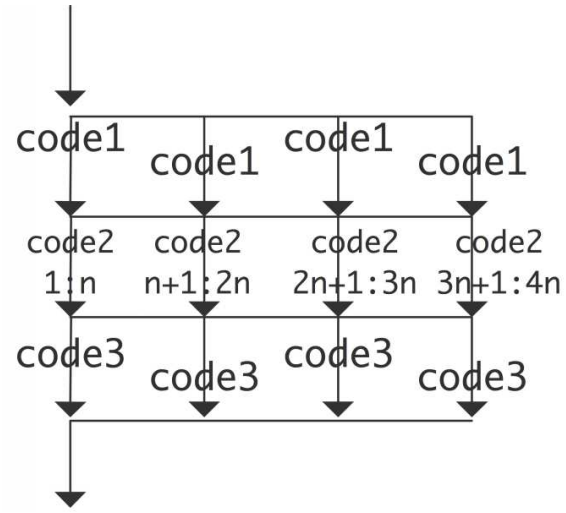


图 19.1：循环内外并行代码的执行

Code:

```
// parfor.c
#pragma omp parallel
  {
    int
      nthreads = omp_get_num_threads(),
      thread_num = omp_get_thread_num();
    printf("Threads entering parallel region:
    ↪%d\n",
           nthreads);
    #pragma omp for
    for (int iter=0; iter<nthreads; iter++)
      printf("thread %d executing iter %d\n",
             thread_num,iter);
  }
```

Output:

```
%%%% equal thread/core counts %%%%
Threads entering parallel region: 4
thread 3 executing iter 3
Threads entering parallel region: 4
thread 0 executing iter 0
Threads entering parallel region: 4
thread 2 executing iter 2
Threads entering parallel region: 4
thread 1 executing iter 1
```

练习 19.2. 如果你将线程数增加到超过核心数，上述示例会发生什么？

也可以有组合的 `omp parallel for` 或 `omp parallel do` 指令。

```
#pragma omp parallel for
  for (int i=0; .....
```

C++ 注 5：自定义迭代器。 OpenMP 可以并行化任何带有随机访问迭代器的基于范围的循环。

```
// iterator.cxx
template<typename T>
class NewVector {
protected:
  T *storage;
  int s;
public:
```

```
// iterator stuff
  class iter;
  iter begin();
  iter end();
};
```

The following methods are needed for the contained *iter* class:

```
NewVector<T>::iter& operator++();
T& operator*();
bool operator==( const NewVector::iter &other ) const;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-
    ( const NewVector::iter& other ) const;
NewVector<T>::iter& operator+=( int add );
```

And then a range-based loop is allowed:

```
NewVector<float> v(s);
#pragma omp parallel for
for ( auto e : v )
  cout << e << " ";
```

19.1.2 循环是 static 的

循环有一些限制：基本上，OpenMP 需要能够预先确定将有多少次迭代。

- 循环不能包含 `break`、 `return`、 `exit` 语句，或跳转到循环外部标签的 `goto`。
- 允许使用 `continue`（针对 C/C++）或 `cycle`（针对 Fortran）语句。
- 索引更新必须是按固定量递增（或递减）。
- 循环索引变量自动为私有（见第 22.2 节），且不允许在循环内修改。以下循环在 OpenMP 中不可并行化：

```
for (int i=0; i<N; ) {
  // something
  if (something)
  i++;
  else
  i += 2;
}
```

备注 34 循环索引需要是整数值，循环才能并行化。自 *OpenMP-3* 起允许使用无符号值。

19.1.3 何时进行循环并行?

OpenMP 并行并非魔法。你不能仅仅对一个满足上述限制的顺序循环，加上一个 `omp parallel for`，然后希望得到相同的结果，只是更快。速度问题我们稍后会详细讨论；现在让我们先考虑并行代码是否首先计算出正确的结果。

一个循环能够正确并行执行的简单情况是，迭代 `i` 写入某个数组的 `i` 位置：

```
for (int i=low; i<hi; i++)
   x[i] = // expression
```

该循环的迭代是独立的，因此如果右侧表达式不包含对 `x` 的任何引用，或者最多只包含对 `x[i]` 的引用，则可以以任意顺序并行计算。

撇开右侧表达式的考虑，我们可以更一般地说，如果在

```
for (int i=low; i<hi; i++)
   x[ f(i) ] = // expression
```

函数 `f` 满足：

$$i \neq j \Rightarrow f(i) \neq f(j).$$

Exercise 19.3. 考虑以下代码

```
for (int i=0; i<n; i++)
  x[i/2] += f(i)
```

论证这不满足上述条件。你能重写这个循环使其可并行化吗？

19.1.4 练习

练习 19.4. 通过数值积分计算 π。我们利用 π 是单位圆面积的事实，并通过计算四分之一圆的面积来近似它，使用黎曼和。

- 设 $f(x) = \sqrt{1-x^2}$ 是描述 $x = 0 \ldots 1$ 四分之一圆的函数；
- 然后我们计算

$$\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \qquad \text{where } x_i = i\Delta x \text{ and } \Delta x = 1/N$$

为此编写一个程序，并使用 OpenMP parallel for 指令进行并行化。

1. 在你的循环周围放置一个 `parallel` <style id='4'} 指令。它仍然计算出正确的结果吗？随着线程数的增加，时间会减少吗？（答案应该是否定的。） 2. 将 `parallel` 改为 `parallel for` <style id='10'}（或 `parallel do`<style id='13'}）。现在结果正确吗？执行速度加快了吗？（答案现在应该是否定和肯定。） 3. 在更新前放置一个 `critical` <style id='16'} 指令。（是的，非常不。） 4. 移除 `critical` <style id='19'} 并在 `for` <style id='25'} 指令中添加一个子句 `reduction(+:quarterpi)` <style id='22'}。现在它应该是正确且高效的。

使用不同数量的核心，计算相对于顺序计算所获得的加速比。OpenMP 代码使用 1 个线程与顺序代码之间有性能差异吗？

备注 35 在本练习中，你可能会看到运行时间在意料之外的情况下增加了几次。这里的问题是 伪共享；详见 *HPC* 书籍，第 *3.6.5* 节以获得更多解释。

19.2 一个例子

为了说明完美并行计算的加速效果，我们考虑一个简单的代码，对数组的每个元素应用相同的计算。

所有测试均在 *TACC Frontera* 集群上完成，该集群配备双插槽 *Intel Cascade Lake* 节点，总共有 56 个核心。我们通过设置 `OMP_PROC_BIND=true` 来控制亲和性。

以下是关键代码片段：

```
// speedup.c
#pragma omp parallel for
      for (int ip=0; ip<N; ip++) {
        for (int jp=0; jp<M; jp++) {
          double f = sin( values[ip] );
          values[ip] = f;
        }
      }
```

练习 19.5. 验证外层循环是并行的，但内层循环不是。

练习 19.6. 比较顺序代码和单线程 OpenMP 代码的时间。如果有不同的编译器，尝试不同的优化级别和不同的编译器。

- 你有时会得到显著的差异吗？这可能的解释是什么？
- 你的编译器是否有生成优化报告的功能？例如 `-qoptreport=5` 对于 *Intel compiler*。

现在我们研究两个参数的影响：

1. OpenMP 线程数：虽然我们有 56 个核心，但允许使用比这更大的值；以及 2. 问题规模：问题越小，创建和同步线程组的相对开销越大。

我们多次执行上述计算以平衡缓存加载的影响 ng.

结果见图 19.2：

- 虽然问题规模始终大于线程数，但只有对于最大的问题（每个线程至少有 400 个点），加速比才基本呈线性。
- OpenMP 允许线程数超过核心数，但这样做并不会带来性能提升。

上述测试未使用 超线程，因为在 Frontera 上该功能被禁用。然而，*Intel Knights Landing* 节点的 *TACCStampede2* 集群每个核心有四个超线程。表 19.3 显示这确实会带来适度的加速。

作为参考，执行的命令行如下：

```
# frontera
  make localclean run_speedup EXTRA_OPTIONS=-DN=200 NDIV=8 NP=112
  make localclean run_speedup EXTRA_OPTIONS=-DN=2000 NDIV=8 NP=112
  make localclean run_speedup EXTRA_OPTIONS=-DN=20000 NDIV=8 NP=112
# stampede2
  make localclean run_speedup NDIV=8 EXTRA_OPTIONS="-DN=200000 -DM=1000" NP=272
```

C++note 6: Range syntax. C++ 中的并行循环自 OpenMP-5.0 起可以使用基于范围的语法：

```
// vecdata.cxx
vector<float> values(100);
```

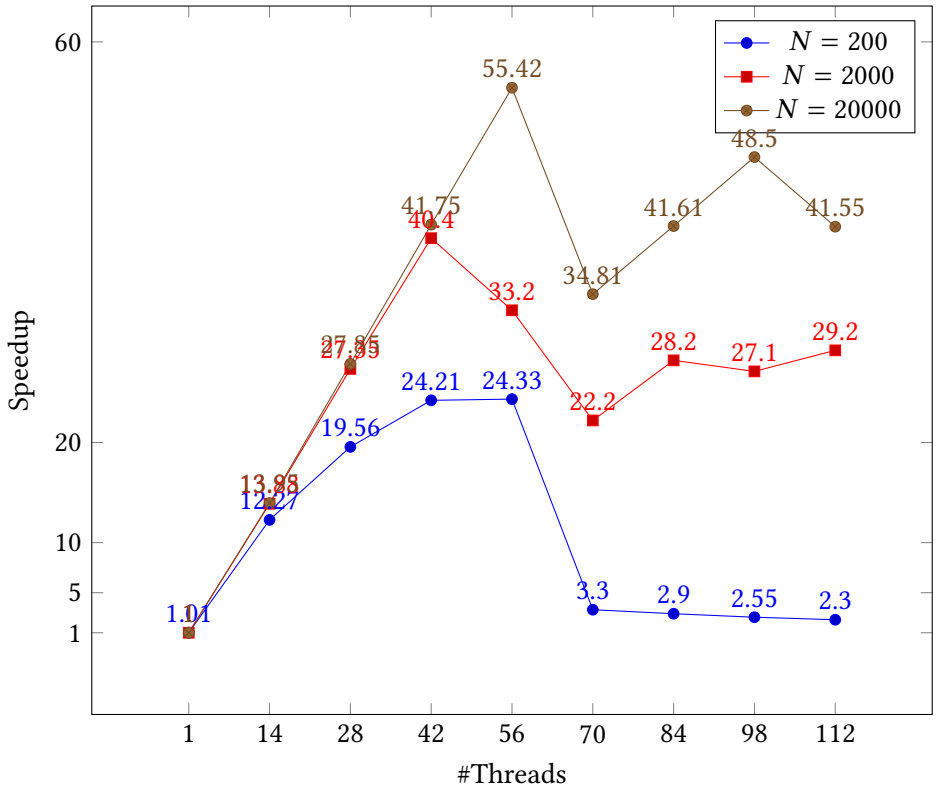


Figure 19.2: Speedup as function of problem size

```
#pragma omp parallel for
for ( auto& elt : values ) {
  elt = 5.f;
}

float sum{0.f};
#pragma omp parallel for reduction(+:sum)
for ( auto elt : values ) {
  sum += elt;
}
```

测试显示与 C 代码完全相同的加速效果。

C++ note 7: C++20 ranges header. The C++20 `ranges` library is supported:

```
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data
              | std::ranges::views::drop(1) )
        count += e;
#     pragma omp parallel for reduction(+:count)
      for ( auto e : data
              | std::ranges::views::transform
```

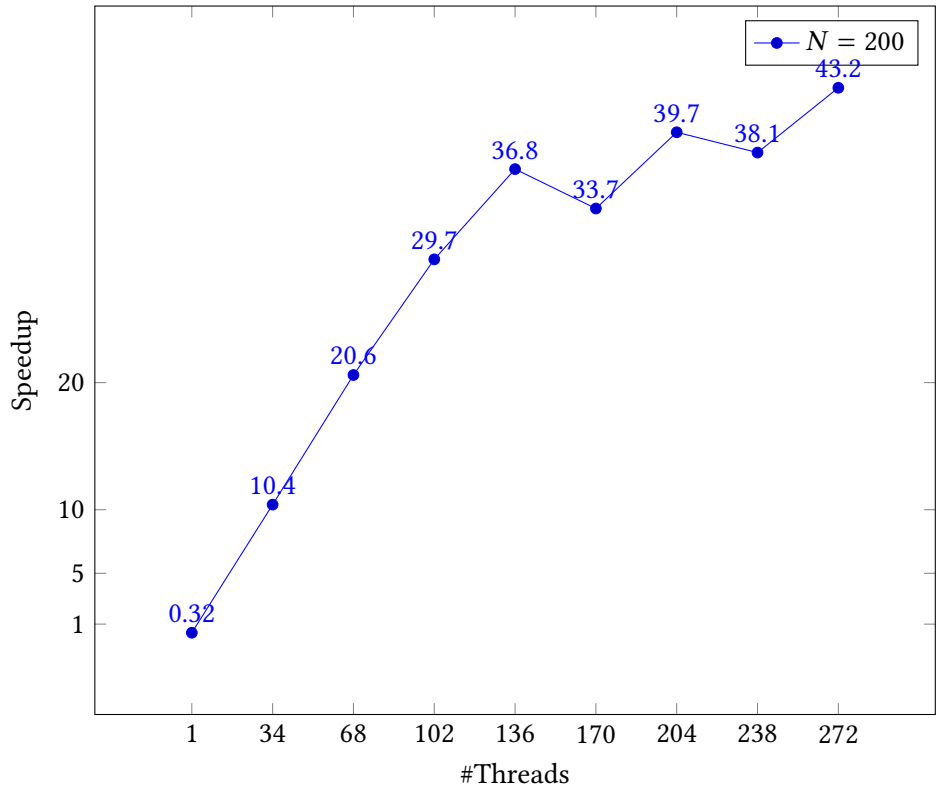


图 19.3：超线程架构上的加速比 re

```
            ( []( auto e ) { return 2*e; } ) )
        count += e;
```

C++ 注 *8*：*C++20* 范围的加速比。

```
==== Run range on 1 threads ====
sum of vector: 50000005000000 in 6.148
sum w/ drop 1: 50000004999999 in 6.017
sum times 2  : 100000010000000 in 6.012
==== Run range on 25 threads ====
sum of vector: 50000005000000 in 0.494
sum w/ drop 1: 50000004999999 in 0.477
sum times 2  : 100000010000000 in 0.489
==== Run range on 51 threads ====
sum of vector: 50000005000000 in 0.257
sum w/ drop 1: 50000004999999 in 0.248
sum times 2  : 100000010000000 in 0.245
==== Run range on 76 threads ====
sum of vector: 50000005000000 in 0.182
sum w/ drop 1: 50000004999999 in 0.184
sum times 2  : 100000010000000 in 0.185
==== Run range on 102 threads ====
sum of vector: 50000005000000 in 0.143
```

```
sum w/ drop 1: 50000004999999 in 0.139
sum times 2  : 100000010000000 in 0.134
==== Run range on 128 threads ====
sum of vector: 50000005000000 in 0.122
sum w/ drop 1: 50000004999999 in 0.11
sum times 2  : 100000010000000 in 0.106
 scaling results in: range-scaling-ls6.out
```

C++ 注释 *9*：范围和索引。 使用 *iota_view* 来获取索引：

```
// iota.cxx
vector<long> data(N);
# pragma omp parallel for
for ( auto i : std::ranges::iota_view( 0UZ,data.size() ) )
  data[i] = f(i);
```

注意这使用了 C++23 后缀用于 `unsigned size_t`。对于旧版本：

```
iota_view( static_cast<size_t>(0),data.size() )
```

19.3 循环调度

通常情况下，循环中的迭代次数远多于线程数。因此，有多种方式可以将循环迭代分配给线程。OpenMP 允许你通过 *schedule* 子句来指定这一点。

```
#pragma omp for schedule(....)
```

我们现在首先要区分的是静态调度和动态调度。静态调度中，迭代的分配纯粹基于迭代次数和线程数（以及 *chunk* 参数；详见后文）。而在动态调度中，迭代分配给空闲的线程。如果迭代所需时间不可预测，动态调度是一个好主意，因为这时需要负载均衡。

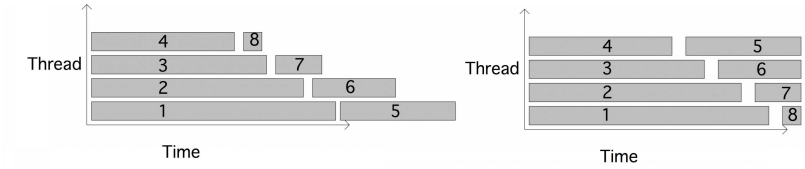


图 19.4：静态轮询调度与动态调度示意

图 19.4 说明了这一点：假设每个核心分配两个（块）迭代，这些块所需时间逐渐减少。你可以从左图看到，线程 1 得到两个相当长的块，而线程 4 得到两个较短的块，因此线程 4 先完成。（线程工作量不均的现象称为 负载不均衡。）另一方面，在右图中，线程 4 在早早完成第一组块后，获得了第 5 块。效果是完美的负载均衡。

T默认的 static 调度是将一段连续的迭代分配给每个线程。如果你想要不同大小的
b块，你可以定义一个 chunk size：

```
#pragma omp for schedule(static[,chunk])
```

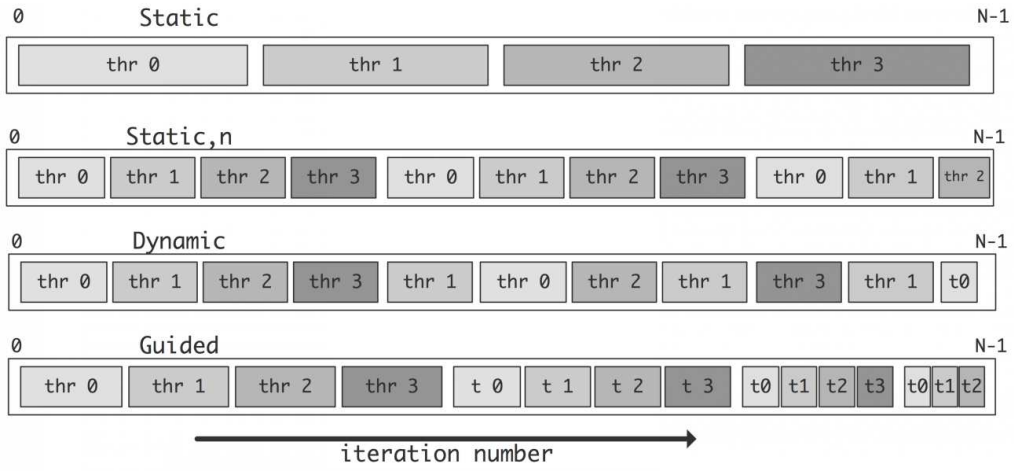


图 19.5：循环迭代调度策略示意图

(其中方括号表示可选参数）。使用静态调度时，编译器将在
t编译时确定循环迭代分配给线程的方式，因此，只要迭代所需时间大致相同，
a这在运行时是最高效的。

块大小的选择通常是在只有少数块时的低开销与较小块带来的负载均衡效果之间的权衡。

练习 19.7. 为什么块大小为 1 通常是个坏主意？（提示：考虑缓存行，并阅读 HPC 书籍，第 1.4.2 节。）

在动态调度中，OpenMP 会将迭代块（默认块大小为 1）放入任务队列，线程在完成前一个任务后会取出其中一个任务。

```
#pragma omp for schedule(static[,chunk])
```

虽然这种调度在迭代执行时间差异很大时可能提供良好的负载均衡，但它确实带来了管理迭代任务队列的运行时开销。

最后，还有 `guided` 调度，它逐渐减小块大小。这里的想法是大块带来的开销最小，但较小的块更有利于负载均衡。各种调度方式在图 19.5 中有所说明。

如果你不想在代码中决定调度方式，可以指定 `runtime` 调度。实际的调度将在运行时从 `OMP_SCHEDULE` 环境变量中读取。你甚至可以通过指定 `auto` 将其留给运行时库处理。

练习 19.8. 编写一个简单的素数测试器和一个计数素数的循环 es:

```
// primesched.c
for (int n = 0; n < N; n++) {
  if (is_prime(n))
    j++;
}
```

你是否期望动态调度比静态调度更好？完成程序并使用不同的调度进行测试。

练习 19.9. 我们继续练习 19.4。我们添加 “ 自适应积分 ”： 在需要时，程序会细化步长 [1]。这意味着迭代不再花费可预测的时间。

```
for (int i=0; i<nsteps; i++) {
    double
      x = i*h,x2 = (i+1)*h,
      y = sqrt(1-x*x),
      y2 = sqrt(1-x2*x2),
      slope = (y-y2)/h;
    if (slope>15) slope = 15;
    int
    samples = 1+(int)slope, is;
    for (int is=0; is<samples;
    ↪is++) {
        double
        hs = h/samples,
        xs = x+ is*hs,
        ys = sqrt(1-xs*xs);
        quarterpi += hs*ys;
        nsamples++;
    }
}
  pi = 4*quarterpi;
```

1. 使用 *omp parallel for* 构造来并行化循环。和之前的实验一样，你一开始可能会看到错误的结果。使用 *reduction* 子句来修正这个问题。2. 你的代码现在应该能看到不错的加速，但可能不是所有核心都能加速。通过调整调度，有可能获得完全线性的加速。先使用 *schedule*(static,*n*)。尝试不同的 *n* 值。什么时候能获得更好的加速？请解释。3

。由于这段代码有些动态，尝试 *schedule*(*dynamic*)。这实际上会得到相当差的结果。为什么？改用 *schedule*(dynamic,n)，并尝试不同的 *n* 值。

4。最后，使用 *schedule*(*guided*)，OpenMP 会使用启发式方法。结果如何给出？

练习 19.10. 编写无选主元的 *LU* 分解 算法。

```
for k=1,n:A[k,k] = 1./A[k,k]for i=k+1,
n:A[i,k] = A[i,k]/A[k,k]for j=k+1,n:
A[i,j] = A[i,j] - A[i,k]*A[k,j]
```

1. 论证外层循环无法并行化。 2. 论证 i 和 j 循环都可以并行化。 3. 通过关注 i 循环来并行化算法。为什么这里给出的算法最适合按行存储的矩阵？如果矩阵是按列存储的，你会怎么做？ 4. 论证在默认调度下，如果一行在一次迭代中被一个线程更新，那么它很可能在另一迭代中被另一个线程更新。你能找到一种调度循环迭代的方法以避免这种情况发生吗？这样做有什么实际原因？

调度可以显式声明，通过 OMP_SCHEDULE 环境变量在运行时设置，或者通过指定 auto 由运行时系统决定。特别是在后两种情况下，您可能想使用 omp_get_schedule (自 OpenMP-5.1 起)：

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

1. It doesn't actually do this in a mathematically sophisticated way, so this code is more for the sake of the example.

其对应的调用是 `omp_set_schedule`，它设置在使用 schedule 值 *runtime* 时所用的值。实际上，这等同于设置环境变量 `OMP_SCHEDULE`。

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

Type	环境变量 OMP_SCHEDULE=	子句 schedule(...)	*omp_sched_t* name	*omp_sched_t* value	modifier defau	lt
static	static[,n]	static[,n]	*omp_sched_static*	1	*N/nthrea*	*ds*
dynamic	dynamic[,n]	dynamic[,n]	*omp_sched_dynamic*	2		1
guided	guided[,n]	guided[,n]	*omp_sched_guided*	3		
auto	auto	auto	*omp_sched_auto*	4		

以下是您可以使用 *schedule* 子句设置的各种调度方式：

affinity 通过使用值 *omp_sched_affinity* 设置 **auto** 调度由实现决定。通过使用值 *omp_sched_auto* 设置 **static** 值：1。修饰符参数是 *chunk* 大小。也可以通过使用值 *omp_sched_static* 设置 **dynamic** 值：2。修饰符参数是 *chunk* 大小；默认值为 1。也可以通过使用值 *omp_sched_dynamic* 设置

guided 值：3。修饰符参数是 chunk 大小。通过使用值 *omp_sched_guided***runtime** 使用 `OMP_SCHEDULE` 环境变量的值。通过使用值 *omp_sched_runt* 设置 *ime*

19.4 定时实验

19.4.1 索引方案

对于二维循环，有几种可能的索引处理方式。

1. 我们可以使用嵌套循环，并将 `i,j` 索引转换为线性索引；

```
// collapse.cxx
#       pragma omp parallel for
        for ( int i=0; i<nsize; i++ )
          for ( int j=0; j<nsize; j++ )
            data[ i*nsize+j] += sqrt(x*y);
#       pragma omp parallel for reduction(+:s)
        for ( int i=0; i<nsize; i++ )
          for ( int j=0; j<nsize; j++ )
            s + sqrt(data[ i*nsize+j]);
```

2. 同样，但使用 *collapse(2)* 指令；

```
#       pragma omp parallel for collapse(2)
        for ( int i=0; i<nsize; i++ )
          for ( int j=0; j<nsize; j++ )
            data[ i*nsize+j] += sqrt(x*y);
#       pragma omp parallel for reduction(+:s) collapse(2)
        for ( int i=0; i<nsize; i++ )
          for ( int j=0; j<nsize; j++ )
            s + sqrt(data[ i*nsize+j]);
```

3. 使用 C++32 *mdspan* 我们可以实现真正的二维索引；

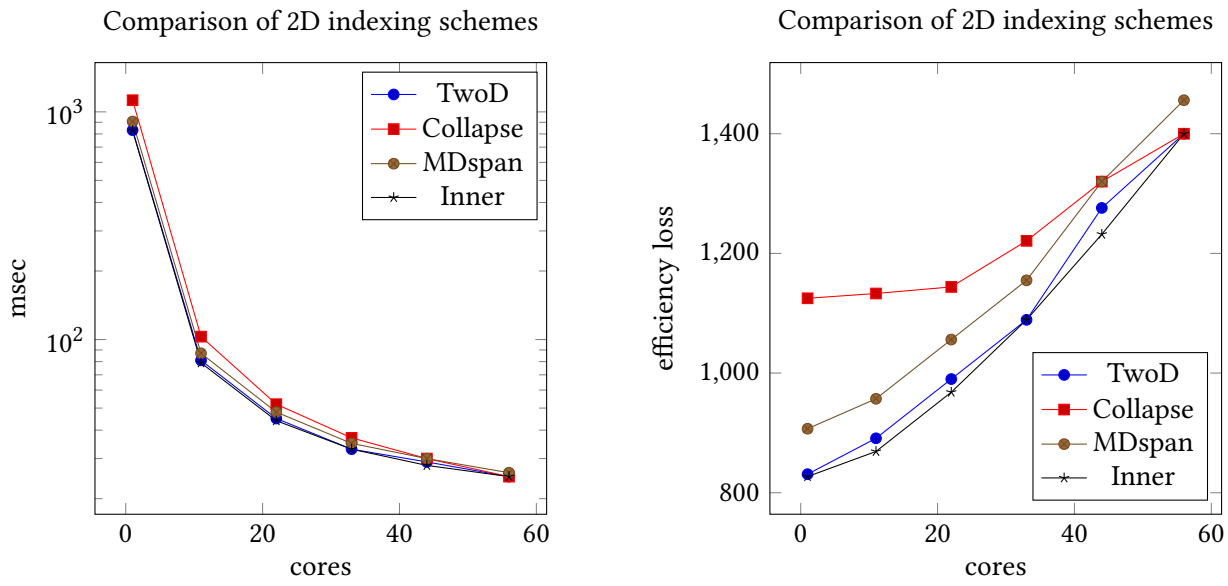


Figure 19.6: Different ways of indexing a 2D parallel loop. Left: absolute times; Right: normalized to t_1.

```
#       pragma omp parallel for
        for ( int i=0; i<nsize; i++ )
          for ( int j=0; j<nsize; j++ )
            mdata[ i,j] += sqrt(x*y);
#       pragma omp parallel for reduction(+:s)
        for ( int i=0; i<nsize; i++ )
          for ( int j=0; j<nsize; j++ )
            s + sqrt(mdata[ i,j]);
```

4. finally, to deal with Partial Diffential Equations (PDEs) with a boundary condition, we use a nested loop that only traverses the interior of the array.

```
#       pragma omp parallel for
        for ( int i=1; i<nsize-1; i++ )
          for ( int j=1; j<nsize-1; j++ )
            data[ i*nsize+j] += sqrt(x*y);
#       pragma omp parallel for reduction(+:s)
        for ( int i=1; i<nsize-1; i++ )
          for ( int j=1; j<nsize-1; j++ )
            s + sqrt(data[ i*nsize+j]);
```

我们在图 19.6 中报告了结果。这些结果是在 *TACCFrontera* 集群上运行的，配备了 56 核双路 *IntelCascadeLake* 处理器。

我们得出结论，所有方案的性能大致相当。令人惊讶的是， `collapse(2)` 降低了性能。来自 “内层” 方案的索引间隙似乎无关紧要。 `mdspan` 索引的性能略逊于最佳方案。

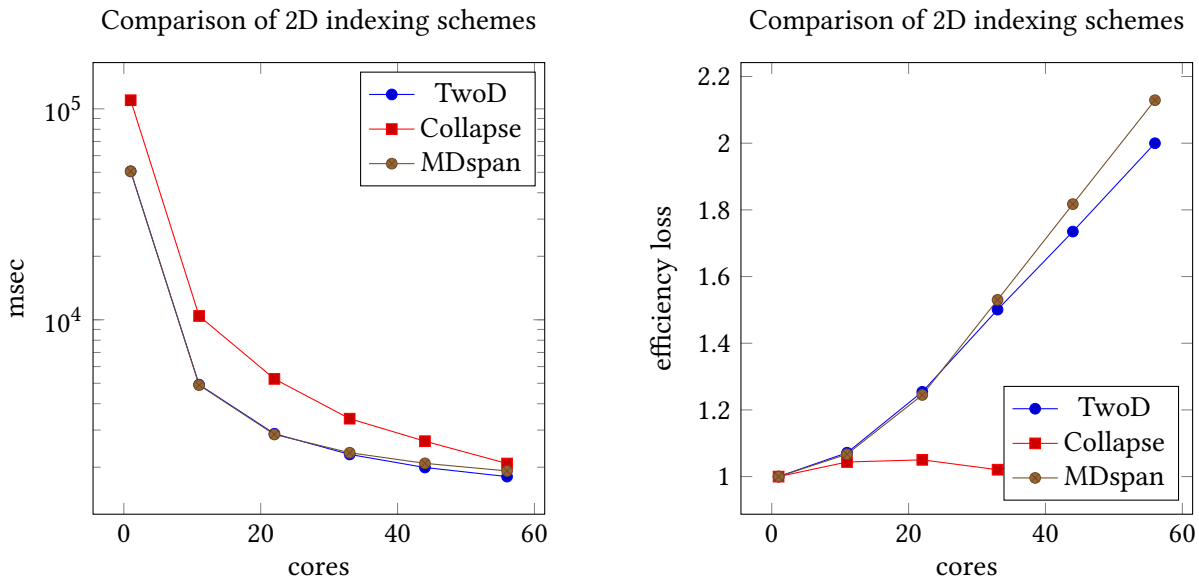


图 19.7：五点拉普拉斯循环的不同索引方式。左图：绝对时间；右图：归一化到 t_1。

19.4.2 OpenMP 循环 vs C++ standardalgorithms

C++ 注 10：并行标准算法。 C++17/C++20 标准引入了对标准算法的执行策略的概念，指的是对位于 `algorithm` 库中的容器进行的操作。

这种并行化通常通过 Thread Building Blocks (TBB) 实现。

举个例子，考虑素数标记：创建一个数组，其中 `p[i]` 为 1 表示 `i` 是素数，否则为 0。

```
#pragma omp parallel for \
  schedule(static)
 for ( int i=0; i<nsize; i++) {
   results[i] =
     one_if_prime( number(i) );
 }
```

```
// primepolicy.cxx
transform
  ( std::execution::par,
    numbers.begin(),numbers.end(),
    results.begin(),
    [] (int n ) -> int {
      return one_if_prime(n); }
    );
```

结果我们发现（图 19.8 ）并行算法在低线程数时与 OpenMP 循环并行化具有竞争力，但在线程数较高时则不然。

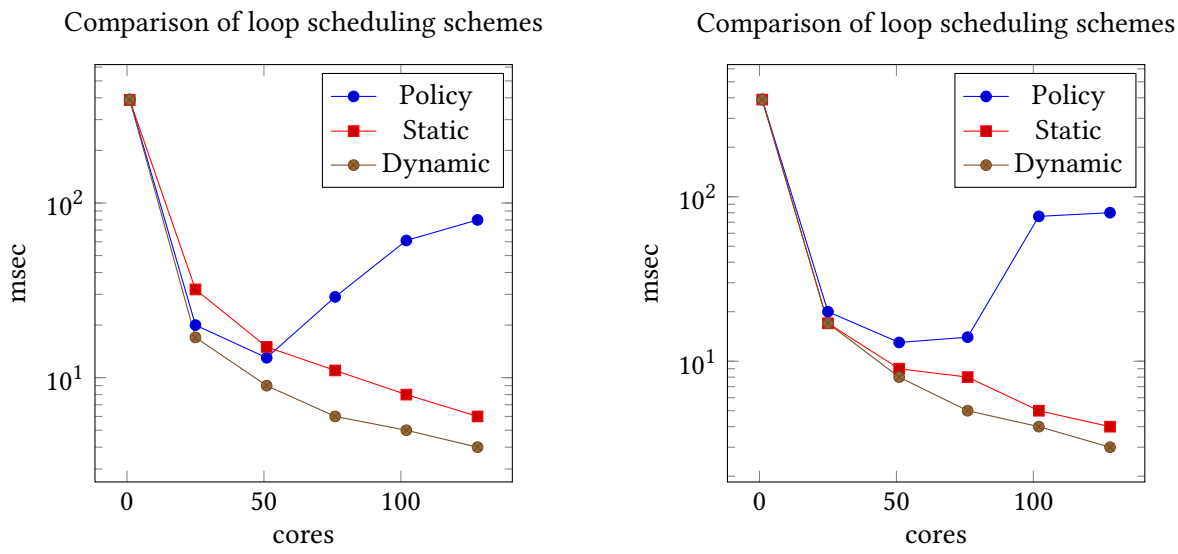


图 19.8：负载不平衡算法（左：并行循环，右：归约）与 1. 执行策略 2. OpenMP 静态调度，3. OpenMP 动态调度

19.5 归约

到目前为止，我们主要关注具有独立迭代的循环。归约是一种常见的具有依赖关系的循环类型。第 20 章中对归约有详细讨论。

C++ 注 11：性能比较。图 19.8 给出了 OpenMP 归约与 C++ 执行策略之间的性能比较。

19.6 嵌套循环

19.6.1 折叠嵌套循环

一般来说，分配给多个线程的工作越多， 并行化的效率就越高。在并行循环的上下文中，可以通过并行化所有层级的循环而不仅仅是最外层循环来增加工作量。

示例：在

```
for ( int i=0; i<N; i++ )
  for ( int j=0; j<N; j++ )
    A[i][j] = B[i][j] + C[i][j]
```

所有 N^2 迭代都是独立的，但常规的 `omp for` 指令只会并行化一层。 `collapse` 子句将并行化多于一层：

```
#pragma omp for collapse(2)
for ( int i=0; i<N; i++ )
  for ( int j=0; j<N; j++ )
    A[i][j] = B[i][j] + C[i][j]
```

只有完全嵌套的循环才能被合并，也就是说，外层循环的循环体只能包含内层循环；在外层循环的循环体中，内层循环之前或之后不能有其他语句。也就是说，两个循环在

```
for ( int i=0; i<N; i++ ) {
  y[i] = 0.;
  for ( int j=0; j<N; j++)
    y[i] += A[i][j] * x[j]
}
```

不能被合并。

Exercise 19.11. 你可以将上述代码重写为

```
for (int i=0; i<N; i++)
  y[i] = 0.;
for (int i=0; i<N; i++) {
  for (int j=0; j<N; j++)
    y[i] += A[i][j] * x[j]
}
```

现在在嵌套循环上使用 `collapse` 指令是否正确?

练习 19.12. 考虑这个矩阵转置的代码:

```
void transposer(int n, int m, double *dst, const double *src) {
   int blocksize;
   for (int i = 0; i < n; i += blocksize) {
      for (int j = 0; j < m; j += blocksize) {
          // transpose the block beginning at [i,j]
          for (int k = i; k < i + blocksize; ++k) {
              for (int l = j; l < j + blocksize; ++l) {
                  dst[k + l*n] = src[l + k*m];
               }
          }
      }
   }
}
```

假设 `src` 和 `dst` 数组是不相交的，哪些循环是并行的，你可以折叠多少层?

19.6.2 Array traversal

考虑数组

```
float Amat[N][N];
float xvec[N],yvec[N];
```

以及操作 $s \leftarrow y^t Ax$。

1. 将此代码编写为 OpenMP 并行双重循环。2. 论证矩阵 A 可以通过两种方式遍历

: 按行和列，或按列和行，两者在精确算术中都给出相同结果。3. 论证循环可以用 `collapse` 直接合并。4. 现在你有了除顺序代码外的 4 个变体。对它们进行计时。

Yo你应该发现行 / 列（或 *row-major*）变体更快。你能找到原因吗?

19.7 有序迭代

并行循环中并行执行的迭代并不是同步进行的。这意味着在

```
#pragma omp parallel for
for ( ... i ... ) {
  ... f(i) ...
  printf("something with %d\n",i);
}
```

中，并非所有函数求值都大致同时发生，随后才是所有打印语句。打印语句实际上可以以任何顺序发生。*ordered* 子句与 *ordered* 指令结合可以强制按正确顺序执行：

```
#pragma omp parallel for ordered
for ( ... i ... ) {
  ... f(i) ...
#pragma omp ordered
  printf("something with %d\n",i);
}
```

Example code structure:

```
#pragma omp parallel for shared(y) ordered
for ( ... i ... ) {
  int x = f(i)
#pragma omp ordered
  y[i] += f(x)
  z[i] = g(y[i])
}
```

有一个限制：每次迭代只能遇到一个 *ordered* 指令。

19.8 nowait

OpenMP 循环是一个工作共享构造，之后并行区域的执行回到复制执行。为了同步这一点，OpenMP 插入了一个 *barrier*，意味着线程等待彼此到达这一点。详见第 23.1.1 节。

工作共享构造末尾的隐式屏障可以通过 *nowait* 子句取消。这使得已完成的线程可以继续执行并行区域中的下一段代码：

```
#pragma omp parallel{
#pragma omp for nowait
for (int i=0; i<N; i++) {...
}// more parallel code}
```

在下面的示例中，完成第一个循环的线程可以开始执行第二个循环。请注意，这要求两个循环具有相同的调度。我们在这里指定静态调度，以便在线程之间对迭代进行相同的调度：

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for schedule(static) nowait
  for (int i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for schedule(static)
  for (int i=0; i<N; i++) {
    y[i] = ... x[i] ...
  }
}
```

19.9 While 循环

OpenMP 只能处理 'for' 循环：*while loops* 不能并行化。所以你必须找到解决方法。例如，while 循环用于搜索数据：

```
while ( a[i]!=0 && i<imax ) {
 i++; }
// now i is the first index for which a[i] is zero.
```

我们用一个检查所有位置的 for 循环来替代 while 循环：

```
result = -1;
#pragma omp parallel for
for (int i=0; i<imax; i++) {
  if (a[i]!=0 && result<0) result = i;
}
```

练习 19.13. 证明这段代码存在竞态条件。

你可以通过将条件变成临界区来修复竞态条件；参见章节 23.2.2。在这个特定的例子中，每次迭代的工作量非常小，这种方法可能效率不高（为什么？）。一个更高效的解决方案是使用 `lastprivate` pragma:

```
result = -1;
#pragma omp parallel for lastprivate(result)
for (int i=0; i<imax; i++) {
  if (a[i]!=0) result = i;
}
```

你现在解决了一个稍有不同的问题：result 变量包含了 `a[i]` 为零的最后一个位置。

19.10 复习问题

Exercise 19.14. 以下循环可以用 `parallel for` 并行化。添加指令 `collapse(2)` 是否正确?

```
for (int i=0; i<N; i++) {
  y[i] = 0.;
  for (int j=0; j<N; j++)
  y[i] += A[i][j] * x[j]
}
```

Exercise19.15. 这里嵌套循环的同样问题：

```
for (int i=0; i<N; i++)
y[i] = 0.;
for (int i=0; i<N; i++) {
  for (int j=0; j<N; j++)
  y[i] += A[i][j] * x[j]
}
```

Exercise19.16. 在这个三重循环中：

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    for (int k=0; k<kmax; k++)
      x[i][j] += f(i,j,k)
```

你使用哪些 OpenMP 指令？你能折叠所有层级吗？循环边界是否重要？

第 20 章

OpenMP 主题：归约

20.1 归约：为什么，是什么，如何做?

并行任务通常会产生一些需要求和或以其他方式合并的量。如果你写：

```
int sum=0;
#pragma omp parallel for
for (int i=0; i<N; i++)
  sum += f(i);
```

你会发现 `sum` 的值依赖于线程数，并且很可能与顺序执行代码时不同。这里的问题是涉及 `sum` 变量的竞争条件，因为该变量在线程之间是共享的。

我们将讨论几种处理此问题的策略。

20.1.1 Reduction clause

实现归约的最简单方法当然是使用 `reduction` clause。将其添加到 `omp parallel`region 中具有以下效果：

- OpenMP 会为每个线程复制一份归约变量，初始化为归约操作符的单位元，例如乘法的单位元为 1。
- 然后每个线程将归约到其本地变量中；
- 在并行区域结束时，本地结果将再次使用归约操作符合并到全局变量中。

最简单的情况是在并行循环上进行归约。这里我们将 $\pi/4$ 计算为一个 *Riemann sum*：

```
// pi.c
#pragma omp parallel for reduction(+:pi4)
  for (int isample=0; isample<N; isample++) {
    float xsample = isample * h;
    float y = sqrt(1-xsample*xsample);
    pi4 += h*y;
  }
```

你也可以在 `sections` 上进行归约：

```
// sectionreduct.c
float y=0;
#pragma omp parallel reduction(+:y)
#pragma omp sections
{
#pragma omp section
  y += f();
#pragma omp section
  y += g();
}
```

另一个归约，这次是在一个并行区域上，没有任何工作共享：

```
// reductpar.c
m = INT_MIN;
#pragma omp parallel reduction(max:m) num_threads(ndata)
{
  int t = omp_get_thread_num();
  int d = data[t];
  m = d>m ? d : m;
};
```

如果你想用相同的操作符归约多个变量，使用

```
reduction(+:x,y,z)
```

对于使用不同操作符的多重归约，使用多个子句。

备注 36 归约是并行执行可能与顺序计算结果略有不同的情况之一，因为浮点运算不是结合律的，所以 舍入误差 会导致结果不同。详见 *HPC* 书籍，第 3.6.5 节的更多解释。

OpenMP 标准甚至没有规定使用相同线程数和相同调度的两次运行必须给出相同结果。一些运行时可能有设置来强制执行这一点；例如 KMP_DETERMINISTIC_REDUCTION 针对 *Intel* 运行时。

20.1.2 编写你自己的解决方案

虽然使用 *reduction* 子句是处理如第 20.1 节中代码的首选方式，但查看其他机制也是有启发性的。

最直接的方法是通过声明一个 临界区 来消除竞争条件：

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  int num = omp_get_thread_num();
  if (num==0)      local_result = f(x);
  else if (num==1) local_result = g(x);
  else if (num==2) local_result = h(x);
# pragma omp critical
  result += local_result;
}
```

如果临界区中的串行化量相比计算函数 f, g, h 来说很小，这是一个很好的解决方案。另一方面，你可能不想在循环中这样做：

```
double result = 0;
#pragma omp parallel
{
  double local_result;
# pragma omp for
  for (i=0; i<N; i++) {
    local_result = f(x,i);
#   pragma omp critical
    result += local_result;
  } // end of for loop
}
```

练习 20.1. 你能想到对这段代码做一个小修改，仍然使用临界区，但更高效吗？对两段代码进行计时。

20.1.2.1 伪共享

如果你的代码不能轻易地结构化为归约，你可以通过 “ 复制 ” 全局变量并稍后收集贡献，手动实现上述方案。这个例子假设有三个线程，并给每个线程分配一个位置来存储该线程上计算的结果：

```
double result,local_results[3];#pragma omp parallel{
int num = omp_get_thread_num();
if (num==0)      local_results[num] = f(x)
else if (num==1) local_results[num] = g(x)
else if (num==2) local_results[num] = h(x)}
result = local_results[0]+local_results[1]+local_results[2]
```

虽然这段代码是正确的，但由于一种称为伪共享的现象，可能效率不高。即使线程写入的是不同的变量，这些变量很可能位于同一个缓存行 (参见 HPC 书籍，第 -1.4.2 节的解释)。这意味着各个核心会浪费大量时间和带宽来更新彼此的该缓存行副本。

可以通过为每个线程分配自己的缓存行来防止伪共享：

```
double result,local_results[3][8];
#pragma omp parallel{
int num = omp_get_thread_num();
if (num==0)      local_results[num][1] = f(x)
// et cetera}
```

一个更优雅的解决方案是给每个线程一个真正的本地变量，并在最后使用临界区来汇总这些变量：

```
double result = 0;
#pragma omp parallel
```

```
{
  double local_result;
  local_result = .....
#pragam omp critical
  result += local_result;
}
```

20.2 内置归约

20.2.1 运算符

算术归约：`+,*,-,max,min`。减号运算符自 OpenMP-5.2 起已弃用。

C 语言中的逻辑运算符归约： `& && | || ^`

Fortran 中的逻辑运算符归约： `.and. .or. .eqv. .neqv. .iand. .ior. .ieor.`

归约可以应用于定义了该运算符的任何类型。`max/min` 适用的类型是有限的。

练习 20.2. 最大值和最小值归约直到 OpenMP-3.1 才被加入 OpenMP。编写一个并行循环，在不使用 `reduction` 指令的情况下计算数组中的最大值和最小值。讨论各种选项。进行计时以评估所获得的加速比并找到最佳选项。

20.2.2 数组上的归约

从 OpenMP-4.5 标准开始，您可以对静态和动态分配的数组进行归约：

```
// reductarray.c
#pragma omp parallel for schedule(static,1) \
  reduction(+:data[:nthreads])
for (int it=0; it<nthreads; it++) {
  for (int i=0; i<nthreads; i++)
    data[i]++;
}
```

```
int *alloced = (int*)malloc( nthreads*sizeof(int) );
for (int i=0; i<nthreads; i++)
  alloced[i] = 0;
#pragma omp parallel for schedule(static,1) \
  reduction(+:alloced[:nthreads])
for (int it=0; it<nthreads; it++) {
  for (int i=0; i<nthreads; i++)
        alloced[i]++;
}
```

C++ 注 12：向量上的归约。使用 `data` 方法提取要归约的数组。然而，这种方法不起作用：

```
vector<float> x;
#pragma omp parallel reduction(+:x.data())
```

因为归约子句需要一个变量，而不是数组的表达式，所以你需要一个额外的裸指针：

```
// reductarray.cxx
vector<int> data(nthreads,0);
int *datadata = data.data();
#pragma omp parallel for schedule(static,1) \
  reduction(+:datadata[:nthreads])
```

在归约过程中，OpenMP 可能需要分配临时数组。这可能会遇到栈大小的限制。你可能需要增加 `OMP_STACKSIZE` 环境变量的值。

20.3 归约的初始值

归约中初始值的处理稍显复杂。

```
  x = init_x
#pragma omp parallel for reduction(min:x)
  for (int i=0; i<N; i++)
    x = min(x,data[i]);
```

每个线程执行部分归约，但其初始值不是用户提供的 `init_x` 值，而是依赖于操作符的值。最终，部分结果将与用户初始值合并。初始化值大多是不言自明的，例如加法为零，乘法为一。对于 min 和 max，它们分别是结果类型的最大和最小可表示值。

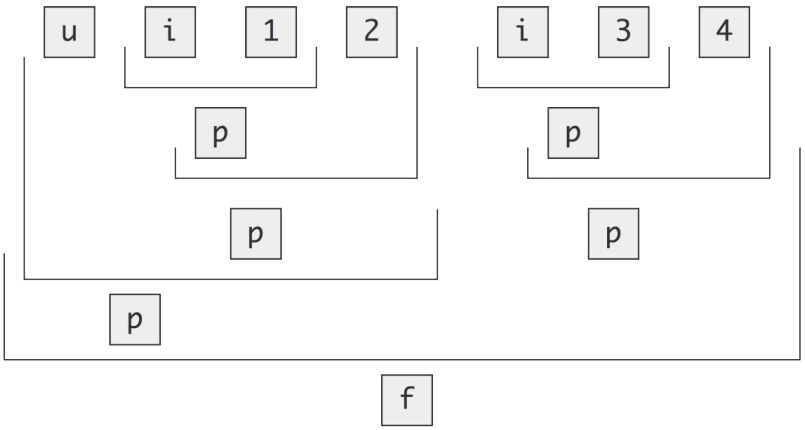


图 20.1：在两个线程上对四个项进行归约，考虑初始值。

Figure 20.1 说明了这一点，其中 `1,2,3,4` 是四个数据项， `i` 是 OpenMP 初始化，且 `u` 是用户初始化；每个 `p` 代表一个部分归约值。该图基于使用两个线程的执行。

Exercise 20.3. 编写一个程序以测试部分结果被初始化为归约操作符单位元的事实。

20.4 用户自定义归约

在执行归约的循环中，大部分元素逐个元素的归约操作是在用户代码中完成的。然而，在该循环的并行版本中，OpenMP 需要对线程的部分结果执行相同的归约。因此，如果你想执行自己的归约操作，需要向 OpenMP 声明该归约。

使用用户自定义归约时，程序员需要指定执行元素级比较的函数。我们讨论两种策略：

1. 在非面向对象（ OO ）语言中，你可以定义一个函数，并使用 *declare reduction* 构造将其声明为归约操作符。2. 在面向对象语言（ C++ 和 Fortran2003 ）中，你可以为类型（包括类对象）重载普通操作符。

20.4.1 归约函数

这需要两个步骤。

1. 你需要一个带有两个参数的函数，该函数返回比较的结果。你可以自己实现，但特别是在使用 C++ 标准库时，你可以使用诸如 `std::vector::insert` 之类的函数。 2. 指定该函数如何作用于两个变量 *omp_out* 和 *omp_in*，分别对应部分归约结果和新的操作数。新的部分结果应保留在 `omp_out` 中。 3. 可选地，你可以指定归约应初始化到的值。

这是归约定义的语法，然后可以在多个 *reduction* 子句中使用。

```
#pragma omp declare reduction
    ( identifier : typelist : combiner )
    [initializer(initializer-expression)]
```

where:

identifier 是一个名称；它可以针对不同类型重载，并在内部作用域中重新定义。**typelist** 是一个类型列表。

combiner is an expression that updates the internal variable *omp_out* as function of itself and *omp_in*.

initializer sets *omp_priv* to the identity of the reduction; this can be an expression or a brace initializer.

Fortran 注释 20：归约声明。声明语句必须位于子程序的声明部分。

20.4.1.1 显式表达式

对于非常简单的情况：

```
for (i=0; i<N; i++) {
  if (abs(data[i]) < result) {
    result = abs(data[i]);
  }
}
```

你可以通过一个表达式声明归约：

```
// reductexpr.c
#pragma omp declare reduction\
  (minabs : int :                                                    \
   omp_out = abs(omp_in) > omp_out ? omp_out : abs(omp_in) )         \
  initializer (omp_priv=LARGENUM)
```

并在 *reduction* 子句中使用它：

```
#pragma omp parallel for reduction(minabs:result)
```

针对上述循环

```
for (i=0; i<N; i++) {
  if (abs(data[i]) < result) {
    result = abs(data[i]);
  }
}
```

C++ note 13: Lambda expressionsin declared reductions. 你可以在声明的归约的显式表达式中使用 lambda 表达式：

```
// reductexpr.cxx
#pragma omp declare reduction\
  (minabs : int : \
   omp_out = \
       [] (int x,int y) -> int { \
          return abs(x) > abs(y) ? abs(y) : abs(x); } \
       (omp_in,omp_out) ) \
  initializer (omp_priv=limit::max())
```

> 你不能将 lambda 表达式赋值给变量并使用它，因为 *omp_in/out* 是唯一允许的在显式表达式中允许的变量。

20.4.1.2 归约函数

例如，重新创建最大归约看起来像这样：

```
// ireduct.c
int mymax(int r,int n) {
// r is the already reduced value
// n is the new value
  int m;
  if (n>r) {
    m = n;
  } else {
    m = r;
  }
  return m;
}
#pragma omp declare reduction \
  (rwz:int:omp_out=mymax(omp_out,omp_in)) \
  initializer(omp_priv=INT_MIN)
  m = INT_MIN;
#pragma omp parallel for reduction(rwz:m)
  for (int idata=0; idata<ndata; idata++)
    m = mymax(m,data[idata]);
```

练习 20.4. 编写一个归约例程，操作一个非负整数数组，找到最小的非零元素。如果数组大小为零，或完全由零组成，则返回 -1。

C++ 注释 14：迭代器上的归约。 支持 *C++* 迭代器

```
#pragma omp declare reduction \
    (merge             // identifier
    : std::vector<int> // typelist
    : omp_out.insert(omp_out.end(), omp_in.begin(),
                     omp_in.end()) // combiner
    )
```

C++ 注释 *15*：模板归约。如果将声明和归约都放在同一个模板函数中，就可以使用模板函数进行归约：

```
template<typename T>T generic_reduction( vector<T> tdata ) {
#pragma omp declare reduction                                    \
(rwzt:T:omp_out=reduce_without_zero<T>(omp_out,omp_in))       \
initializer(omp_priv=-1.f)T tmin = -1;
#pragma omp parallel for reduction(rwzt:tmin)
for (int id=0; id<tdata.size(); id++)
tmin = reduce_without_zero<T>(tmin,tdata[id]);return tmin;};
```

然后用特定数据调用它：`auto tmin = generic_reduction<float>(fdata);`

C++ 注 *16*：示例：在 *map* 上的归约。通过合并线程本地 map 对 `std::map` 进行归约：

```
// charcount.cxxtemplate<typename key>
class bincounter : public map<key,int> {
public:// merge this with other map
void operator+=
( const bincounter<key>& other ) {
for ( auto [k,v] : other )
if ( map<key,int>::contains(k) )
this->at(k) += v;else
this->insert( {k,v} );};
// insert one char in this map
void inc(char k) {
if ( map<key,int>::contains(k) )
this->at(k) += 1;else
this->insert( {k,1} );};};
```

```
/*
 * Reduction loop in main program
 */
bincounter<char> charcount;
#pragma omp parallel for reduction(+ :
    ↪charcount)
for ( int i=0; i<text.size(); i++ )
  charcount.inc( text[i] );
```

20.4.2 重载运算符

Fortran 注 *21*：派生类型上的归约。归约可以应用于任何定义了归约运算符的派生类型。

```
!! reducttype.F90
  Type inttype
     integer :: value = 0
  end type inttype
  Interface operator(+)
     module procedure addints
  end Interface operator(+)
```

```
Type(inttype),dimension(nsize) :: intarray
Type(inttype) :: intsum = inttype(0)
!$OMP declare
    ↪reduction(+:inttype:omp_out=omp_out+omp_in)
!$OMP parallel do reduction(+:intsum)
do i=1,nsize
   intsum = intsum + intarray(i)
end do
!$OMP end parallel do
```

但请注意额外的 *declare* 子句。

C++ 注 *17*：对类对象的归约。归约可以应用于任何定义了归约操作符的类， `operator+` 或视情况而定的其他操作符。

```
// reductclass.cxx
class Thing {
private:
  float x{0.f};
public:
  Thing() = default;
  Thing( float x ) : x(x) {};
  Thing operator+
      ( const Thing& other ) {
    return Thing( x + other.x );
  };
};
```

```
vector< Thing >
  things(500,Thing(1.f) );
Thing result(0.f);
#pragma omp parallel for \
    reduction( +:result )
for ( const auto& t : things )
  result = result + t;
```

内部使用的初始化值需要一个默认构造函数；见图 20.1。

20.5 扫描 / 前缀操作

“ 扫描 ” 或前缀操作类似于归约，只不过你关注的是部分结果。对于此，OpenMP 从 OpenMP-5.0 开始提供了 *scan* 指令。它需要以下内容：

- 归约子句获得一个修饰符 *inscan*：

```
#pragma omp parallel for reduction(inscan,+:sumvar)
```

- 在并行循环体中有一个 *scan* 指令允许你存储部分结果。对于包含扫描，归约变量在 *scan* pragma 之前更新：

```
sumvar // update#pragma omp scan inclusive(sumvar)
partials[i] = sumvar
```

对于排他扫描，归约变量在 *scan* pragma 之后更新：

```
partials[i] = sumvar#pragma omp scan inclusive(sumvar)
sumvar // update
```

Code:

```
// scanintsum.c
partial_sum=0;
#pragma omp parallel for \
reduction(inscan,+:partial_sum)
for (int i=0; i<nthreads; i++) {
  partial_sum += amounts[i];
#   pragma omp scan inclusive(partial_sum)
  inc_partials[i] = partial_sum;
}
partial_sum=0;
#pragma omp parallel for \
reduction(inscan,+:partial_sum)
for (int i=0; i<nthreads; i++) {
  exc_partials[i] = partial_sum;
#   pragma omp scan exclusive(partial_sum)
  partial_sum += amounts[i];
}
```

Output:

```
Summing   :  1  2  3  4  5  6  7  8
Inclusive:  1  3  6 10 15 21 28 36
Exclusive:  0  1  3  6 10 15 21 28
```

20.6 Reductionsand floating-point math

OpenMP 用于实现归约并行的机制违反了 C 语言中浮点表达式求值的严格规则；参见 HPC 书籍，第 3.7.7 节。OpenMP 忽略了这个问题：确保正确的舍入行为是程序员的责任。

20.7 Reductions in C++ standard algorithms

C++ 注 18：并行标准算法上的归约。在第 19.4.2 节中，你看到了如何通过 C++ 中的 执行策略 参数实现某些循环结构。归约同样适用。

```
#pragma omp parallel for \
schedule(guided,8) \
reduction(+:prime_count)
for ( auto n : numbers ) {
prime_count += one_if_prime( n );}
```

```
// reducepolicy.cxx
prime_count = transform_reduce
( std::execution::par,
numbers.begin(),numbers.end(), 0,
std::plus<>{},[] ( int n ) -> int {
return one_if_prime(n); });
```

```
Threads:  1
TBB:  Time:       391 msec
Stat: Time:       390 msec
Dyn:  Time:       389 msec

Threads: 25
TBB:  Time:        20 msec
Stat: Time:        17 msec
Dyn:  Time:        17 msec

Threads: 51
TBB:  Time:        13 msec
Stat: Time:         9 msec
Dyn:  Time:         8 msec

Threads: 76
TBB:  Time:        14 msec
Stat: Time:         8 msec
Dyn:  Time:         5 msec

Threads: 102
TBB:  Time:        76 msec
Stat: Time:         5 msec
Dyn:  Time:         4 msec

Threads: 128
TBB:  Time:        80 msec
Stat: Time:         4 msec
Dyn:  Time:         3 msec
```

第 21 章

OpenMP 主题：工作共享

parallel region 的声明建立了一个线程团队。这提供了并行的可能性，但要真正获得有意义的并行活动，你需要更多东西。OpenMP 使用了 *worksharingconstruct* 的概念：一种将可并行工作分配给线程团队的方法。

你已经在第 19 章中看到过循环并行作为分配并行工作的方式。现在我们将讨论其他工作共享构造。

21.1 工作共享构造

工作共享构造包括：

- `for` (for C) or `do` (for Fortran): 线程在它们之间划分循环迭代；参见 19.1.
- `sections`: 线程在它们之间划分固定数量的 sections ；参见章节 21.2.
- `single` 该 section 由单个线程执行；章节 21.3.
- `task`: 参见第 24 章。
- `workshare`. 这可以并行化 Fortran 数组语法；章节 21.4.

21.2 Sections

并行循环是编号的独立工作单元的一个例子。如果你有预先确定数量的独立工作单元， `sections` 更合适。在一个 `sections` 结构中可以有任意数量的 `section` 结构。这些需要是独立的，并且可以由当前团队中的任何可用线程执行，包括同一线程完成多个部分。

```
#pragma omp sections
{
#pragma omp section
  // one calculation
#pragma omp section
  // another calculation
}
```

该结构可用于划分大型独立工作块。假设在以下行中， `f(x)` 和 `g(x)` 都是大型计算：

```
y = f(x) + g(x)
```

你可以这样写

```
double y1,y2;
#pragma omp sections
{#pragma omp section
y1 = f(x)
#pragma omp section
y2 = g(x)}
y = y1+y2;
```

你也可以不用两个临时变量，而使用一个 critical section；参见章节 23.2.2。不过，最好的解决方案是在 *reduction* 指令上使用 parallel sections <style id='11'} 子句。对于求和

```
  y = f(x) + g(x)
```

你可以这样写

```
// sectionreduct.c
float y=0;
#pragma omp parallel reduction(+:y)
#pragma omp sections
{
#pragma omp section
  y += f();
#pragma omp section
  y += g();
}
```

21.3 Single/master

single pragma 限制一个代码块只能由单个线程执行。例如，这可以用来打印跟踪信息或执行 *I/O* 操作。

```
#pragma omp parallel
{
#pragma omp single
  printf("We are starting this section!\n");
  // parallel stuff
}
```

另一种 single 的用法是在并行区域中执行初始化：

```
int a;
#pragma omp parallel
{
  #pragma omp single
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

这个最后示例中 single 指令的重点是计算只需要执行一次，因为是共享内存。由于它是一个工作共享结构，其后有一个隐式屏障，这保证了所有线程在它们的本地内存中都有正确的值（参见第 23.4 节）。

练习 21.1. 这种方法与在 MPI 中并行化相同计算的方法有什么区别?

该 `master` 指令还强制在单个线程上执行，具体是团队的主线程。这不是一个工作共享结构，因此没有通过隐式屏障进行同步。

Exercise 21.2. Modify the above code to read:

```
int a;
#pragma omp parallel
{
  #pragma omp master
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

这段代码不再正确。请解释。

上文我们将 `single` 指令作为初始化共享变量的一种方式。也可以使用 `single` 来初始化私有变量。在这种情况下，需要添加 *copyprivate* 子句。如果设置变量涉及 I/O，这是一个很好的解决方案。

练习 21.3. 给出另外两种初始化私有变量的方法，使所有线程接收相同的值。你能给出三种策略中每种更优的场景吗?

21.4 Fortran 数组语法并行化

该 `parallel do` 指令用于并行化循环，这适用于 C 和 Fortran。然而，Fortran 在其 数组语法中也有隐含循环。要并行化数组语法，可以使用 `workshare` 指令。

`workshare` 指令仅存在于 Fortran 中。它可用于并行化 数组语法中的隐含循环，以及 *forall* 循环。

我们比较了两种 $C \leftarrow C + A \times B$ 版本（所有操作均为元素级），在 *TACC Frontera* 上最多使用 56 核心运行。

基于工作分配：

```
!! workshare2d.F90
          !$omp parallel workshare
          C = A*B + C
          !$omp end parallel workshare
```

SIMD'ized loop

```
!$omp parallel do simd
do i=1,dim
   do j=1,dim
      C(i,j) = C(i,j) + A(i,j) * B(i,j)
   end do
end do
!$omp end parallel do simd
```

结果：

SIMD times :0.071150.04053 0.02498 0.01609 0.01210 0.01247 0.01765 0.02689Speedup:1 1.75549 2.84828 4.422 5.88017 5.70569 4.03116 2.64597

Workshare times:0.061880.03186 0.01625 0.00867 0.00619 0.00379 0.00354 0.00373Speedup:1 1.94225 3.808 7.13725 9.99677 16.3272 17.4802 16.5898

第 22 章

OpenMP 主题：控制线程数据

在并行区域中，有两种类型的数据：私有和共享。在本节中，我们将看到控制数据所属类别的各种方法；对于私有数据项，我们还将讨论它们的值如何与共享数据相关。

22.1 共享数据

在并行区域中，任何在其外部声明的数据都是共享的：任何线程使用变量 `x` 时，都会访问与该变量关联的相同内存位置。

示例：

```
  int x = 5;
#pragma omp parallel
  {
    x = x+1;
    printf("shared: x is %d\n",x);
  }
```

所有线程都增加同一个变量，因此循环结束后它的值将是五加上线程数；或者可能更少，因为涉及数据竞争。这个问题在 HPC 书中讨论，章节 -2.6.1.5 ；参见 23.2.2 ，了解 OpenMP 中数据竞争的解决方案。

22.2 私有数据

在 C/C++ 语言中，可以在词法作用域内声明变量；大致来说：在花括号内。这个概念扩展到 OpenMP 并行区域和指令：任何在 OpenMP 指令后面的代码块中声明的变量都将是执行线程的局部变量。

在下面的示例中，每个线程创建一个私有变量 `x` 并将其设置为唯一值：

Code:

```
// private.cint x=5;
#pragma omp parallel num_threads(4){
int t = omp_get_thread_num(),x = t+1;
printf("Thread %d sets x to %d\n",t,x);}
printf("Outer x is still %d\n",x);
```

Output:

```
Thread 3 sets x to 4
Thread 2 sets x to 3
Thread 0 sets x to 1
Thread 1 sets x to 2
Outer x is still 5
```

并行区域结束后，外部变量 x 仍然保持值 5：私有变量和全局变量之间没有存储关联。

Fortran 注释 *22*：并行区域中的私有变量。Fortran 语言没有作用域的概念，因此你必须使用 private 子句：

代码：

```
!! private.F90x=5
!$omp parallel private(x,t) num_threads(4)
t = omp_get_thread_num()x = t+1
print '("Thread ",i2," sets x to ",i2)',t,x
!$omp end parallel
print '("Outer x is still ",i2)',x
```

输出：

```
Thread  0 sets x to  1
Thread  2 sets x to  3
Thread  3 sets x to  4
Thread  1 sets x to  2
Outer x is still  5
```

C++ 注 *19*：私有化类成员。 类成员只能从（非静态）类方法中私有化。

In this example *f* can not be static:

```
// private.cxx
class foo {
private:
  int x;
public:
  void f() {
#pragma omp parallel private(x)
    somefunction(x);
  };
};
```

You can not privatize just a member:

```
// privateno.cxx
class foo { public: int x; };
int main() {
  foo thing;
#pragma omp parallel private(thing.x) // NOPE
```

private 指令声明数据在每个线程的内存中有一个独立的副本。这样的私有变量会像在主程序中那样初始化。任何计算得到的值在并行区域结束时都会消失。（但是，见下面的 *lastprivate* 。）因此，你不应依赖任何初始值，或依赖区域外变量的值。

```
  int x = 5;
#pragma omp parallel private(x)
  {
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
  }
  printf("after: x is %d\n",x);
```

使用 *private* 指令声明为私有的数据会放在每个线程的独立栈上。OpenMP 标准没有规定这些栈的大小，但要注意栈溢出。典型的默认值是几兆字节；你可以通过环境变量 OMP_STACKSIZE 来控制它。（你可以通过设置 OMP_DISPLAY_ENV 来查看当前值。）其值可以是字面量或带后缀：

```
123 456k 567K 678m 789M 246g 357G
```

Remark 37 *TOpenMP* 栈大小在数组的归约操作中也起作用 *s; section 20.2.2.*

一个普通的 *Unix* 进程也有栈，但这与用于私有数据的 OpenMP 栈是独立的。你可以使用 ulimit 查询或设置 Unix 栈：

```
[] ulimit -s
64000
[] ulimit -s 8192
[] ulimit -s8192
```

Unix 栈可以根据需要动态增长。OpenMP 栈则不然：它们会立即分配请求的大小。因此，重要的是不要让它们太大。

22.3 动态作用域中的数据

从并行区域调用的函数属于该并行区域的动态作用域。该函数中变量的规则如下：

- 函数中局部定义的任何变量都是私有的。
- static C 语言中的变量和 save Fortran 中的变量是共享的。
- 函数参数继承调用环境的状态。

Fortran 注释 23：保存的变量。子程序中的变量是私有的，和 C 语言一样，除非它们具有 Save 属性。任何具有值初始化的变量都会隐式地获得此属性。

在以下示例中，我们有两个几乎相同的例程，唯一的区别是第一个对局部变量进行了值初始化，从而实际上使其变为共享变量。第二个例程则没有这个问题。

代码： `subroutine savehello`

```
use omp_libimplicit none
integer :: thread = -1
thread = omp_get_thread_num()
print *,"Hello from",thread
end subroutine savehello
subroutine finehello
use omp_libimplicit none
integer :: thread
thread = omp_get_thread_num()
print *,"World from",thread
end subroutine finehello
```

Output:

```
Hello from           3
Hello from           3
Hello from           3
Hello from           3
World from           0
World from           1
World from           2
World from           3
```

22.4 循环中的临时变量

在每次循环迭代中设置并使用一个变量是很常见的 :

```
#pragma omp parallel for
for ( ... i ... ) {
  x = i*h;
  s = sin(x); c = cos(x);
  a[i] = s+c;
  b[i] = s-c;
}
```

根据上述规则，变量 `x`、`s`、`c` 都是共享变量。然而，它们在一次迭代中接收的值不会在下一次迭代中使用，因此它们实际上表现得像每次迭代的私有变量。

- 在 C 和 Fortran 中，你都可以在并行 for 指令中将这些变量声明为 private。
- 在 C 中，你也可以在循环内部局部定义这些变量。

有时，即使你忘记将这些临时变量声明为 private，代码仍然可能给出正确的输出。这是因为编译器有时可以将它们从循环体中消除，因为它检测到它们的值没有被其他地方使用。

22.5 Default

OpenMP 结构中数据是 private 还是 shared 有默认规则，你也可以显式控制这一点。

首先是默认行为：

- 在并行区域外声明的变量如上所述是共享的；
- `omp for` 中的循环变量是私有的；
- 并行区域内的局部变量是私有的。

您可以使用 `default` 子句更改此默认行为：

```
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

如果你想保险起见：

```
#pragma omp parallel default(none) private(x) shared(matrix)
{ ... }
```

- `shared` 子句意味着所有来自外部作用域的变量在并行区域中是共享的；任何私有变量需要显式声明。这是默认行为。
- `private` 子句意味着所有外部变量在并行区域中变为私有。它们未被初始化；请参见下一个选项。并行区域中的任何共享变量需要显式声明。此值在 C 语言中不可用。
- `firstprivate` 子句意味着所有外部变量在并行区域中都是私有的，并用它们的外部值初始化。任何共享变量都需要显式声明。此值在 C 语言中不可用。
- `none` 选项适合调试，因为它强制你为并行区域中的每个变量指定它是私有的还是共享的。此外，如果你的代码在并行执行时与顺序执行表现不同，可能存在数据竞争。指定每个变量的状态是调试此问题的好方法。

22.6 First and last private

上面你已经看到，私有变量与周围作用域中同名的任何变量完全分离。然而，有两种情况你可能希望私有变量与全局对应变量之间存在某种存储关联。

首先，私有变量是用未定义的值创建的。你可以用 *firstprivate* 强制它们初始化。

```
  int t=2;
#pragma omp parallel firstprivate(t)
  {
    t += f( omp_get_thread_num() );
    g(t);
  }
```

变量 `t` 的行为类似于私有变量，除了它被初始化为外部值 e.

备注 38 变量默认在任务中是 *firstprivate* ；参见第 *24* 章。

其次，你可能希望将私有值保留到并行区域之外的环境中。这实际上只有在一种情况下才有意义，即保留并行循环的最后一次迭代的私有变量，或 `sections` 构造中的最后一个部分。这是通过 *lastprivate* 实现的：

```
#pragma omp parallel for \
        lastprivate(tmp)
for (int i=0; i<N; i+) {
  tmp = ......
  x[i] = .... tmp ....
}
..... tmp ....
```

22.7 数组数据

数组的规则与标量数据略有不同：

1. Statically allocated data, that is with a syntax like

```
int array[100];
integer,dimension(:) :: array(100}
```

   可以是共享的或私有的，取决于你使用的子句 se.

2. 动态分配的数据，即使用 `malloc` 或 `allocate` 创建的数据，只能是共享的。

第一种类型的示例：每个线程获得数组的私有副本，并正确初始化。**代码：**

输出：

```
int array[nthreads];
for (int i=0; i<nthreads; i++)
  array[i] = 0;

#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
  array[t] = t+1;
}
```

```
Executing: OMP_PROC_BIND=true
    ↪OMP_NUM_THREADS=4 ./alloc
Array result:
0:0, 1:0, 2:0, 3:0,
```

当然，由于只有私有副本被修改，原始数组不受影响。

另一方面，在下面的示例中，每个线程获得一个私有指针，但所有指针都指向同一个对象：**代码：**

输出：

```
// alloc.c
int *array =
  (int*) malloc(nthreads*sizeof(int));
for (int i=0; i<nthreads; i++)
  array[i] = 0;

#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
// ptr arith: needs private array
  array += t;
  array[0] = t;
}
// ... print the array
```

```
Array result:
0:0, 1:1, 2:2, 3:3,
```

C++ 注 20：向量是被复制的，不同于数组。Compare

Code:

```
// alloc.c
int *array =
  (int*) malloc(nthreads*sizeof(int));
for (int i=0; i<nthreads; i++)
  array[i] = 0;

#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
// ptr arith: needs private array
  array += t;
  array[0] = t;
}
// ... print the array
```

和**代码**：`// alloc.cxx`

```
vector<int> array(nthreads);
#pragma omp parallel firstprivate(array)
{int t = omp_get_thread_num();
array[t] = t+1;}// ... print the array
```

Output:

```
Array result:
0:0, 1:1, 2:2, 3:3,
```

Output:

privvector 缺少输出

22.8 通过 `threadprivate` 实现持久数据

OpenMP 并行区域中的大多数数据要么是从主线程继承而来，因此是共享的，要么是在区域范围内的临时数据，完全是私有的。还有一种机制用于线程私有数据，其生命周期不限于一个并行区域。 *threadprivate* pragma 用于声明每个线程拥有一个变量的私有副本：

```
#pragma omp threadprivate(var)
```

变量需要是：

- C 语言中的文件或静态变量，
- C++ 语言中的静态类成员，或者
- Fortran 语言中的程序变量或公共块。

22.8.1 Threadprivate 初始化

如果每个线程在其 threadprivate 变量中需要不同的值，则初始化需要在线程并行区域中进行。

在以下示例中，创建了一个由 7 个线程组成的团队，所有线程都设置了它们的线程私有变量。稍后，这个变量被一个更大的团队读取：那些未被设置的变量是未定义的，尽管通常只是零：

```
// threadprivate.c
static int tp;
#pragma omp threadprivate(tp)

int main(int argc,char **argv) {

#pragma omp parallel num_threads(7)
  tp = omp_get_thread_num();

#pragma omp parallel num_threads(9)
  printf("Thread %d has %d\n",omp_get_thread_num(),tp);
```

Fortran 注释 *24*：私有公共块。命名公共块可以通过以下语法设为线程私有

```
$!OMP threadprivate( /blockname/ )
```

示例：

Code:

```
!! threadprivate.F90
  common /threaddata/tp
  integer :: tp
!$omp threadprivate(/threaddata/)
```

Output:

```
Thread  0 sets x to  1
Thread  2 sets x to  3
Thread  3 sets x to  4
Thread  1 sets x to  2
Outer x is still  5
```

另一方面，如果线程私有数据在所有线程中初始相同，可以使用 *copyin* 子句：

```
#pragma omp threadprivate(private_var)

private_var = 1;
#pragma omp parallel copyin(private_var)
  private_var += omp_get_thread_num()
```

如果一个线程需要将所有线程私有数据设置为其值，可以使用 *copyprivate* 子句：

```
#pragma omp parallel{...
#pragma omp single copyprivate(private_var)
private_var = read_data();...}
```

Threadprivate 变量需要 OMP_DYNAMIC 被关闭。

22.8.2 Threadprivate 示例

线程私有变量的典型应用是在随机数生成器中。随机数生成器需要保存状态，因为它每次计算下一个值都是基于当前值。为了拥有一个并行生成器，每个线程将创建并初始化一个私有的 “ 当前值 ” 变量。即使执行不在并行区域中，该变量也会持续存在；它只在并行区域中被更新。

Exercise 22.1. 通过随机采样计算 *Mandelbrot set* 的面积。为每个线程单独初始化随机数生成器；然后使用并行循环来评估这些点。探索不同循环调度策略的性能影响。

C++ 注 *21*：*Threadprivate* 随机数生成器。新的 C++ *random* 头文件具有线程安全的生成器，这是因为标准中声明没有 STL 对象可以依赖全局状态。通常的惯用法由于初始化问题无法实现线程安全：

```
static random_device rd;
static mt19937 rng(rd);
```

但是，以下代码是可行的：

```
// privaterandom.cxx
static random_device rd;
static mt19937 rng;
#pragma omp threadprivate(rd)
#pragma omp threadprivate(rng)
int main() {
#pragma omp parallel
rng = mt19937(rd());
```

C++ 注 *22*：*Threadprivate* 随机使用。基于前面的注释，你可以安全且独立地使用生成器：

```
#pragma omp parallel
  {
    stringstream res;
    uniform_int_distribution<int> percent(1, 100);
    res << "Thread " << omp_get_thread_num() << ": " << percent(rng) << "\n";
    cout << res.str();
  }
```

22.9 分配器

OpenMP 最初是为共享内存设计的。随着加速器的出现（参见第 27 章），非一致性内存被添加进来。在 OpenMP-5 标准中，情况进一步复杂化，以适应诸如 高带宽内存 和 非易失性内存 等新型内存。

使用 OpenMP 内存分配器有几种方式。

- 首先，在静态数组的目录中：`float A[N], B[N];`

```
#pragma omp allocate(A) \
allocator(omp_large_cap_mem_alloc)
```

- 作为 private 变量的一个子句：

```
#pragma omp task private(B) allocate(omp_const_mem_alloc: B)
```

- 使用 *omp_alloc*，采用（可能是用户定义的）分配器。

接下来是内存空间。OpenMP 标识符与硬件之间的绑定由实现定义。

22.9.1 预定义类型

分配器： *omp_default_mem_alloc*， *omp_large_cap_mem_alloc*， *omp_const_mem_alloc*， *omp_high_bw_mem_alloc*， *omp_low_lat_mem_alloc*， *omp_cgroup_mem_alloc*， *omp_pteam_mem_alloc*， *omp_thread_mem_alloc*。

Memory spaces:*omp_default_mem_space*， *omp_large_cap_mem_space*， *omp_const_mem_space*， *omp_high_bw_mem_space*， *omp_low_lat_mem_space*.

第 23 章

OpenMP 主题：同步

在上面声明并行区域的构造中，你几乎无法控制线程执行分配给它们的工作的顺序。本节将讨论同步构造：告诉线程以某种顺序执行操作的方式。

- `critical`：一段代码一次只能被一个线程执行；参见 23.2.2。
- `atomic` 单个内存位置的原子更新。仅支持某些指定的语法模式。添加此功能是为了能够利用硬件对原子更新的支持。

- `barrier`：章节 23.1。
- 锁：章节 23.3.
- `flush`: section 23.4.

之前讨论过与循环相关的同步构造：

- `ordered`: 章节 19.7.
- `nowait`: 章节 19.8.

23.1 Barrier

一个 *barrier* 定义了代码中的一个点，所有活动线程将在此处停止，直到所有线程都到达该点。通过这种方式，你可以保证某些计算已经完成。例如，在这段代码片段中，`y` 的计算不能继续，直到另一个线程计算出它的 `x` 值。

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
  y[mytid] = x[mytid]+x[mytid+1];
}
```

这可以通过一个 *barrier* pragma 来保证：

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
#pragma omp barrier
  y[mytid] = x[mytid]+x[mytid+1];
}
```

23.1.1 隐式屏障

除了插入显式屏障的 barrier 指令外，OpenMP 在工作共享构造之后有 21.3 节中所述的隐式屏障。因此，以下代码是定义良好的：

```
#pragma omp parallel
{
#pragma omp for
  for (int mytid=0; mytid<number_of_threads; mytid++)
    x[mytid] = some_calculation();
#pragma omp for
  for (int mytid=0; mytid<number_of_threads-1; mytid++)
    y[mytid] = x[mytid]+x[mytid+1];
}
```

你也可以将每个并行循环放在它自己的并行区域中，但在区域之间创建和销毁线程团队会有一些开销。

在并行区域结束时，线程团队解散，只有主线程继续执行。因此，并行区域结束时存在隐式屏障。这种屏障行为可以通过 *nowait* 子句取消。

你经常会看到这样的惯用语

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++)
    a[i] = // some expression
#pragma omp for
  for (i=0; i<N; i++)
    b[i] = ..... a[i] .....
```

这里的 `nowait` 子句意味着线程可以在第二个循环开始时启动，而其他线程仍在处理第一个循环。由于这两个循环在这里使用相同的调度，使用 `a[i]` 的迭代确实可以依赖于该值已经被计算出来。

23.2 互斥

有时需要限制一段代码，使其一次只能被一个线程执行。这段代码称为 临界区，OpenMP 有几种机制来实现这一点。

23.2.1 竞争条件

OpenMP，基于共享内存，存在竞态条件的潜在风险。当两个线程访问同一数据项，且至少有一次访问是写操作时，就会发生这种情况。竞态条件的问题在于程序员的便利性与高效执行相冲突。

举一个简单的例子：

Code:

```
// race.c
#pragma omp parallel for shared(counter)
  for (int i=0; i<count; i++)
    counter += f(counter,i);
  printf("Counter should be %d, is %d\n",
         count,counter);
```

Output:

```
On 1 threads:
Counter should be 100000, is 100000
On 2 threads:
Counter should be 100000, is 100000
On 4 threads:
Counter should be 100000, is 75000
On 8 threads:
Counter should be 100000, is 87500
On 12 threads:
Counter should be 100000, is 100000
```

关于多线程访问单个数据项的基本规则是：

> 任何由一个线程写入的内存位置，如果没有进行同步，另一个线程在同一并行区域内不能读取该位置。

先从最后一句话说起：任何 workshare 构造都以一个隐式屏障结束，因此在该屏障之前写入的数据可以在屏障之后安全读取。

23.2.2 `critical` 和 `atomic`

有两种用于临界区的 pragma：*critical* 和 *atomic*。两者在技术意义上都表示原子操作。第一种是通用的，可以包含任意指令序列；第二种限制更多，但具有性能优势。

初学者常常倾向于在循环中使用 *critical* 进行更新：

```
#pragma omp parallel{
int mytid = omp_get_thread_num();
double tmp = some_function(mytid);
// This Works, but Not Best Solution:
#pragma omp criticalsum += tmp;}
```

但这实际上应该使用 *reduction* 子句来完成，这样效率会高得多。

临界区的一个良好用途是进行文件写入或数据库更新。

练习 23.1. 考虑一个循环，其中每次迭代都会更新一个变量。

```
#pragma omp parallel for shared(result)
  for ( i ) {
      result += some_function_of(i);
  }
```

定性讨论以下内容之间的区别：

将更新语句转换为临界区，与此同时

让线程像上面那样累积到一个私有变量 • 中，并在循环结束后将这些变量求和 `tmp`。

对第一个案例进行阿姆达尔风格的定量分析，假设你在 p 个线程上执行 n 次迭代，每次迭代都有一个占用 f 比例的临界区。假设迭代次数 n 是线程数 p 的倍数。还假设循环迭代在各线程间采用默认的静态分配方式。

临界区是一种将现有代码转换为正确并行代码的简便方法。然而，临界区存在性能上的缺点，有时需要更彻底的重写。

临界区通过获取锁来工作，这会带来相当大的开销。此外，如果代码中有多个临界区，它们是相互排斥的：如果一个线程处于某个临界区，其他临界区都会被阻塞。

`critical` 区段相互排斥的问题可以通过为它们命名来缓解：

```
#pragma omp critical (optional_name_in_parens)
```

另一方面， `atomic` 区段的语法仅限于更新单个内存位置，但这类区段不是排他的，并且它们可能更高效，因为它们假设存在硬件机制使其成为临界区。详见下一节。

23.2.3 `atomic` 构造

虽然 *critical* 构造可以包含任意代码块， *atomic* 子句只有有限的几种形式，这些形式很可能得到硬件支持。这些形式包括对变量的赋值：

```
x++;
// or:
x += y;
```

可能与读取该变量结合使用：

```
v = x; x++;
```

关于 atomic 规范还有各种进一步的细化 cation:

1. *omp atomic write* 后跟对共享变量的单个赋值语句。 2. *omp atomic read* 后跟从共享变量的单个赋值语句。 3. *omp atomic* 等同于 *omp atomic update* ；它适用于诸如

```
x++; x += 1.5;
```

4. *omp atomic capture* 可以容纳类似于 *omp atomic update* 的单条语句，或本质上结合了 *read* 和 *update* 形式的代码块。

23.3 锁

OpenMP 也有传统的 *lock* 机制。锁有点类似于临界区：它保证某些指令一次只能被一个进程执行。然而，临界区确实是关于代码的；锁是关于数据的。通过锁，你确保某些数据元素一次只能被一个进程访问。

23.3.1 例程

创建 / 销毁：

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

设置和发布：

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

由于 set 调用是阻塞的，也有

```
int omp_test_lock();
```

如果锁成功设置则返回 true；否则返回 false 并继续执行下一条语句。

解除锁定需要由设置锁的线程完成。

锁操作隐式具有一个 *flush*；参见第 23.4 节。

Exercise 23.2. 在以下代码中，一个进程设置数组 A，然后使用它来更新 B；另一个进程设置数组 B，然后使用它来更新 A。论证该代码可能会死锁。你如何解决这个问题?

```
#pragma omp parallel shared(a, b, nthreads, locka, lockb)
  #pragma omp sections nowait
    {
    #pragma omp section
      {
      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        a[i] = ..

      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        b[i] = .. a[i] ..
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);
      }

    #pragma omp section
      {
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        b[i] = ...

      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        a[i] = .. b[i] ..
      omp_unset_lock(&locka);
      omp_unset_lock(&lockb);
      }
    }  /* end of sections */
  }  /* end of parallel region */
```

23.3.2 Example: Mandelbrotset

S<section>49.2.2</section> 有一种基于 Mandelbrot 集的锁定 FIFO 方法，该 FIFO 包含待处理的坐标。

23.3.3 示例：带有 atomicupdate 的对象

面向对象语言如 C++ 允许语法简化，例如将加锁和解锁操作构建到更新操作符中。

C++ 注 23：重载操作符内部的锁。

```
// lockobject.cxxclass atomic_int {private:
omp_lock_t the_lock;int _value{0};public:
atomic_int() {omp_init_lock(&the_lock);};
atomic_int( const atomic_int& )= delete;
atomic_int& operator=( const atomic_int& )
= delete;~atomic_int() {
omp_destroy_lock(&the_lock);}; 运行此代码:
atomic_int my_object;
vector<std::thread> threads;
for (int ithread=0;ithread<NTHREADS;
ithread++) {threads.push_back( std::thread(
[=,&my_object] () {
for (int iop=0; iop<nops; iop++)
my_object += 1; } ) );}
for ( auto &t : threads )t.join();
```

23.3.4 示例：histogram/binning

使用锁的一个简单示例是生成 *histogram*（也称为 *binning* 问题）。一个 histogram 由若干个 bin 组成，这些 bin 会根据某些数据进行更新。下面是此类代码的基本结构：

```
int count[100];
float x = some_function();
int ix = (int)x;
if (ix>=100)
  error();
else
  count[ix]++;
```

可以对最后一行进行保护：

```
#pragma omp critical
  count[ix]++;
```

但那是不必要的限制。如果直方图中有足够多的箱子，并且 `some_function` 花费的时间足够长，那么冲突写入的可能性很小。解决方案是创建一个锁数组，每个 `count` 位置对应一个锁。

另一种解决方案是给每个线程一个结果数组的本地副本，并对这些副本执行归约。参见章节 20.2.2。

23.3.5 Nested locks

如上所述，如果一个锁已经被锁定，则不能再次锁定。嵌套锁允许同一个线程在解锁之前多次锁定。

- `omp_init_nest_lock`
- `omp_destroy_nest_lock`
- `omp_set_nest_lock`
- `omp_unset_nest_lock`
- `omp_test_nest_lock`

23.4 Relaxed memory model

flush

- 在并行区域的开始和结束时，所有变量都会隐式刷新。
- 每个屏障处都会有一次 flush，无论是显式的还是隐式的，比如在 *worksharing* 结束时。
- *critical section* 的入口和出口处
- 当 *lock* 被设置或解除时。

23.5 Example:Fibonacci computation

Fibonacci sequence 递归定义为

$$F(0) = 1, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2.$$

我们先简要介绍基本的单线程解决方案。简单的代码如下：

```
int main() {
value = new int[nmax+1];
value[0] = 1;value[1] = 1;
fib(10);}int fib(int n) {
int i, j, result;
if (n>=2) {
i=fib(n-1); j=fib(n-2);
value[n] = i+j;}
return value[n];}
```

然而，这种方法效率低下，因为大多数中间值会被计算多次。我们通过跟踪哪些结果是已知的来解决这个问题：

```
  ...
  done = new int[nmax+1];
  for (i=0; i<=nmax; i++)
    done[i] = 0;
  done[0] = 1;
  done[1] = 1;
  ...
int fib(int n) {
  int i, j;
  if (!done[n]) {
    i = fib(n-1); j = fib(n-2);
    value[n] = i+j; done[n] = 1;
  }
  return value[n];
}
```

OpenMP 并行解决方案采用了两种不同的思路。首先，我们通过使用任务（章节 24：

```
int fib(int n) {int i, j;if (n>=2) {
#pragma omp task shared(i) firstprivate(n)
i=fib(n-1);
#pragma omp task shared(j) firstprivate(n)
j=fib(n-2);#pragma omp taskwait
value[n] = i+j;}return value[n];}
```

这计算出了值正确的解，但正如在简单的单线程解法中一样，它重新计算了许多中间值，

对 done 数组的简单相加会导致数据竞争，并且很可能得到错误的解：

```
int fib(int n) {int i, j, result;
if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
i=fib(n-1);
#pragma omp task shared(i) firstprivate(n)
j=fib(n-2);#pragma omp taskwait
value[n] = i+j;done[n] = 1;}
return value[n];}
```

例如，不能保证 done 数组的更新晚于 value 数组，因此一个线程可能认为 done[n-1] 为真，但 value[n-1] 还没有正确的值。

解决此问题的一种方法是使用锁，并确保对于给定索引 n，值 done[n] 和 value[n] 不会被多个线程同时访问：

```
int fib(int n){int i, j;
omp_set_lock( &(dolock[n]) );
if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
i = fib(n-1);
#pragma omp task shared(j) firstprivate(n)
j = fib(n-2);#pragma omp taskwait
value[n] = i+j;done[n] = 1;}
omp_unset_lock( &(dolock[n]) );
return value[n];}
```

该解决方案是正确的，在不重复计算任何内容的意义上是最优高效的，并且它使用任务来实现并行执行。

然而，该解决方案的效率仅是一个常数级别。即使一个值已经计算完成并且只会被读取，锁仍然会被设置。这个问题可以通过复杂地使用 critical sections 来解决，但我们将放弃这样做。

第 24 章

OpenMP 主题：Tasks

Tasks 是 OpenMP 在幕后使用的一种机制：如果你将某些内容指定为 task，OpenMP 会创建一个 “ 工作块 ”：一段代码加上其发生时的数据环境。这个工作块会被留待稍后执行。因此，基于 task 的代码通常看起来像这样：

```
#pragma omp parallel
{
  // generate a bunch of tasks
# pragma omp taskwait
  // the result from the tasks is now available
}
```

例如，一个并行循环总是隐式地被转换成类似于：

顺序循环：

```
for (int i=0; i<N; i++)
  f(i);
```

并行循环：

```
for (int ib=0; ib<nblocks; ib++) {
  int first=... last=... ;
# pragma omp task
  for (int i=first; i<last; i++)
    f(i)
}
#pragma omp taskwait
// the results from the loop are available
```

24.1 任务生成

如果我们坚持使用通过任务实现并行循环的这个例子，下一个问题是：到底是谁生成任务？以下代码存在一个严重的问题：

```
// WRONG. DO NOT WRITE THIS
#pragma omp parallel
for (int ib=0; ib<nblocks; ib++) {
  int first=... last=... ;
# pragma omp task
  for (int i=first; i<last; i++)
    f(i)
}
```

因为并行区域创建了一个团队，团队中的每个线程都会执行任务生成代码。相反，我们使用以下惯用法：

```
#pragma omp parallel
#pragma omp single
for (int ib=0; ib<nblocks; ib++) {
  // setup stuff
# pragma omp task
  // task stuff
}
```

1. 一个并行区域创建一个线程团队；2. 然后单个线程创建任务，将它们添加到属于该团队的队列中，3. 并且该团队中的所有线程（可能包括生成任务的那个线程）

Btw，实际的任务队列对程序员不可见。程序员无法控制的另一个方面是任务执行的确切时机：这由一个 <style id='1'> 任务调度器 </style> 负责，该调度器对 trol 是任务执行的精确时机：这取决于一个任务调度器，它在后台运行，用户不可见。 programmer.

任务机制允许你做一些用循环和 section 结构难以或不可能完成的事情。例如，可以用任务实现一个 *whileloop* 遍历 *linked list*：

Code	执行
`p = head_of_list();`	一个线程遍历列表
`while (!end_of_list(p)) {`	
`#pragma omp task`	创建了一个任务，
`process( p );`	每个元素一个
`p = next_element(p);`	生成线程继续执行而不等待
`}`	任务在执行的同时
	更多任务正在生成。

另一个以前难以并行化的概念是 “while 循环”。这不符合 OpenMP 并行循环的要求，即循环边界需要在循环执行前已知。

Exercise 24.1. 使用任务来寻找一个大数的最小因子（以 $2999 \cdot 3001$ 作为测试用例）：为每个试探因子生成一个任务。开始使用以下代码：

```
  int factor=0;
#pragma omp parallel
#pragma omp single
  for (int f=2; f<4000; f++) {
    { // see if `f' is a factor
      if (N%f==0) { // found factor!
        factor = f;
      }
    }
    if (factor>0)
      break;
  }
  if (factor>0)
    printf("Found a factor: %d\n",factor);
```

- 将因子查找块转换为任务。
- 运行你的程序若干次：

```
for i in `seq 1 1000` ; do ./taskfactor ; done | grep -v 2999
```

它是否找到了错误的因子？为什么？尝试修复它。
- 一旦找到一个因子，你应该停止生成任务。让那些本不该生成的任务，即测试比已找到因子更大的候选因子的任务，打印出一条消息。

24.2 任务数据

任务中数据的处理有些微妙。基本问题是任务在某一时间被创建，并在稍后的某个时间执行。因此，如果访问共享数据，任务看到的是创建时的值还是执行时的值？事实上，根据应用的不同，这两种可能性都有意义，所以我们需要讨论何时适用哪种规则。

第一个规则是共享数据在任务中是共享的，但私有数据变成了 *firstprivate*。为了看清区别，考虑两个代码片段。

```
int count = 100;
#pragma omp parallel
#pragma omp single
{
  while (count>0) {
#   pragma omp task
    {
      int countcopy = count;
      if (count==50) {
        sleep(1);
        printf("%d,%d\n",
            count,countcopy);
      } // end if
    }   // end task
    count--;
  }     // end while
}       // end single
```

```
#pragma omp parallel
#pragma omp single
{
  int count = 100;
  while (count>0) {
#   pragma omp task
    {
      int countcopy = count;
      if (count==50) {
        sleep(1);
        printf("%d,%d\n",
            count,countcopy);
      } // end if
    }   // end task
    count--;
  }     // end while
}       // end single
```

在第一个例子中，变量 `count` 在并行区域外声明，因此是共享的。当执行打印语句时，所有任务都已生成，因此 `count` 将为零。因此，输出很可能是 0,50。

在第二个例子中，`count` 变量对创建任务的线程是私有的，因此它在任务中将是 *firstprivate*，保持任务创建时的当前值。

24.3 任务同步

尽管上述代码段看起来像一组线性语句，但无法确定 `task` 指令之后的代码何时执行。这意味着以下代码是不正确的：

```
  x = f();
#pragma omp task
  { y = g(x); }
  z = h(y);
```

说明：当执行计算 `z` 的语句时，计算 `y` 的任务仅已被调度；它不一定已经执行。

为了保证任务完成，你需要使用 `taskwait` 指令。以下代码创建了两个可以并行执行的任务，然后等待结果：

Code	执行
`x = f();`	变量 `x` 获得一个值

`#pragma omp task`
`{ y1 = g1(x); }`
`#pragma omp task`
`{ y2 = g2(x); }`
	使用当前值 `x` 创建了两个任务

`#pragma omp taskwait`	线程等待直到任务完成
`z = h(y1)+h(y2);`	变量 `z` 是使用任务重新计算的 sults

gma is followed by a structured block. Each time the structured block is encoun 会创建一个新任务。另一方面，是一个独立指令。后续的代码只是代码，它不 `task` 被打印后 是属于该指令的结构化块。

你可以通过多种方式指示任务依赖：

1. 使用 ‘task wait’ 指令，你可以显式指示分支任务的合并。因此，wait 指令之后的指令将依赖于生成的任务。2. `taskgroup` 指令在第 24.3.1 节中讨论。3. *`taskloop`* 指令在第 24.3.2 节中讨论。
4. 每个 OpenMP 任务可以有一个 `depend` 子句，指示任务的数据依赖；见第 24.4 节。通过指示任务产生或吸收的数据，调度器可以为你构建依赖图。

24.3.1 任务组

The `taskgroup` 指令，后跟一个结构化块，确保块中创建的所有任务完成，即使是递归创建的任务。

任务组有点类似于在块后使用一个 `taskwait` 指令。最大的区别是 `taskwait` 指令不等待递归生成的任务，而 `taskgroup` 指令会等待。

24.3.2 任务循环

The *`taskloop`* 指令置于 for/do 循环之前，就像一个 `for` pragma。不同之处在于现在每次迭代都被转换成一个任务，而不是像 `for` 情况中那样将迭代分组。循环结束是一个同步点：循环后的语句只有在循环中的所有任务完成后才执行。

Thereis a *`mast taskloop`* 指令是 `master` 的简写，只包含一个 *`taskloop`*。

24.4 任务依赖

It is possible to 通过使用 `depend` c 对任务进行部分排序 lause. For example, in

```
#pragma omp task
  x = f()
#pragma omp task
  y = g(x)
```

可以想象第二个任务在第一个任务之前执行，这可能导致错误的结果。通过指定以下内容可以解决这个问题：

```
#pragma omp task depend(out:x)
  x = f()
#pragma omp task depend(in:x)
  y = g(x)
```

- 这些依赖关系仅在同级任务之间成立。
- 各个任务所依赖的数据项要么是相同的，要么是互不相交的。特别地，不允许依赖数组的不同部分，尽管编译器可能并不总能检测到这一点。

练习 24.2. 考虑以下代码：

```
for i in [1:N]:
    x[0,i] = some_function_of(i)
    x[i,0] = some_function_of(i)

for i in [1:N]:
    for j in [1:N]:
        x[i,j] = x[i-1,j]+x[i,j-1]
```

- 注意第二个循环嵌套不适合 OpenMP 循环并行。
- 你能想到用 OpenMP 循环并行实现该计算的方法吗？提示：你需要重写代码，使得相同的操作以不同的顺序完成 t 顺序。
- 使用带依赖关系的任务使该代码并行，无需重写：唯一的改变是添加 OpenMP 指令。

任务依赖用于指示同一数据项的两次使用之间的关系。由于任一使用都可以是读或写，因此存在四种类型的依赖。

RaW(Read after Write) 第二个任务读取第一个任务写入的项。第二个任务必须在第一个任务之后执行：

```
... omp task depend(OUT:x)
  foo(x)
... omp task depend( IN:x)
  foo(x)
```

WaR(Write after Read) 第一个任务读取一个项，第二个任务覆盖它。第二个任务必须第二个执行，以防止覆盖初始值：

```
... omp task depend( IN:x)
  foo(x)
... omp task depend(OUT:x)
  foo(x)
```

WaW(Writeafter Write) 两个任务都设置相同的变量。由于该变量可能被中间任务使用，这两个写操作必须按此顺序执行。

```
... omp task depend(OUT:x)
  foo(x)
... omp task depend(OUT:x)
  foo(x)
```

RaR(Read after Read) 两个任务都读取一个变量。由于两个任务都没有 “out” 声明，它们可以按任意顺序运行。

```
... omp task depend(IN:x)
  foo(x)
... omp task depend(IN:x)
  foo(x)
```

24.5 任务归约

reduction 子句仅适用于普通的并行循环，而不适用于任务的 taskgroup 循环。要对任务中的计算进行归约，需要使用 task_reduction 子句（OpenMP-5.0 特性）：

```
#pragma omp taskgroup task_reduction(+:sum)
```

任务组可以包含既参与归约又不参与归约的任务。前者类型需要一个子句 in_reduction：

```
#pragma omp task in_reduction(+:sum)
```

作为一个例子，这里使用任务计算了和 $\sum_{i=1}^{100} i$：

```
// taskreduct.c
#pragma omp parallel
#pragma omp single
  {
  #pragma omp taskgroup task_reduction(+:sum)
  for (int itask=1; itask<=bound; itask++) {
    #pragma omp task in_reduction(+:sum)
    sum += itask;
    }
  }
```

24.6 更多

24.6.1 调度点

通常，任务会绑定到最初执行它的线程上。然而，在 任务调度点，线程可能会切换到执行由同一团队创建的另一个任务。

- 在显式任务创建之后存在一个调度点。这意味着，在上述示例中，创建任务的线程也可以参与执行这些任务。

在 taskwait 和 taskyield 处有一个调度点。

另一方面，使用任务指令中的 *untied* 子句创建的任务并不绑定到某个线程。这意味着在调度点挂起后，任何线程都可以恢复执行该任务。如果这样做，请注意线程 ID 的值不会保持固定。同时，锁也会成为一个问题。

Example: if a thread is w等待锁时，通过调度点它可以挂起 th e task and work on another task.

```
while (!omp_test_lock(lock))
#pragma omp taskyield
  ;
```

24.6.2 性能提升提示

如果一个任务只涉及少量工作，调度开销可能会抵消任何性能提升。有两种方式直接执行任务代码：

- 该 `if` 子句仅在条件为真时创建任务：

```
#pragma omp task if (n>100)
  f(n)
```

- 该 `if` 子句仍可能导致递归生成任务。另一方面， `final` 将执行代码，并且会跳过任何递归创建的任务：

```
#pragma omp task final(level<3)
```

如果您想表示某些任务比其他任务更重要，请使用 `priority` 子句：

```
#pragma omp task priority(5)
```

其中 priority 是任何小于 `OMP_MAX_TASK_PRIORITY` 的非负标量。

24.6.3 任务取消

在 OpenMP-4.0 中可以取消任务。这在使用任务执行搜索时非常有用：第一个找到结果的任务可以取消任何未完成的搜索任务。详情见第 18.3 节。

练习 24.3. 修改质数查找示例以使用 *cancel*。

24.7 示例

24.7.1 递归矩阵乘法

可以使用以下公式递归地乘以大矩阵

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

with

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

等等。你可以通过创建四个任务来实现，每个任务又可以创建另外四个。

C++ 注 *24*：对子矩阵使用 *mdspan*。对于数据结构，使用 *mdspan*。

24.7.2 Fibonacci

作为使用任务的一个例子，考虑计算一个斐波那契值数组：

```
// taskgroup0.c
for (int i=2; i<N; i++)
  {
    fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
  }
```

如果你只是简单地将每个计算变成一个任务，结果将是不可预测的（请确认这一点！），因为任务可以以任何顺序执行。为了解决这个问题，我们对任务设置了依赖关系：

```
// taskgroup2.c
  for (int i=2; i<N; i++)
#pragma omp task \
depend(out:fibo_values[i]) \
depend(in:fibo_values[i-1],fibo_values[i-2])
    {
      fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
    }
```

24.7.3 Binomial coefficients

Exercise24.4. 二项式系数数组可以按如下方式计算：

```
// binomial1.c
for (int row=1; row<=n; row++)
  for (int col=1; col<=row; col++)
    if (row==1 || col==1 || col==row)
        array[row][col] = 1;
    else
        array[row][col] = array[row-1][col-1] + array[row-1][col];
```

将一个单独的任务组包裹在双重循环周围，并使用 `depend` 子句来使执行满足适当的依赖关系。

第 25 章

OpenMP 主题：亲和性

25.1 OpenMP 线程亲和性控制

线程亲和性的问题在多插槽节点上变得重要；参见第 25.2 节中的示例。

线程放置可以通过两个环境变量来控制：

- 环境变量 `OMP_PROC_BIND` 描述了线程如何绑定到 *OpenMP* 位置；而
- 变量 `OMP_PLACES` 描述了这些位置，基于可用的硬件。
- 当你在试验这些变量时，最好将 `OMP_DISPLAY_ENV` 设置为 true，这样 OpenMP 将在运行时打印出它如何解释你的规范。以下章节中的示例将显示此输出。

25.1.1 线程绑定

变量 `OMP_PLACES` 定义了一系列线程被分配到的位置， `OMP_PROC_BIND` 描述了线程如何绑定到这些位置。

`OMP_PLACES` 的典型值为

- *socket*：线程绑定到一个 *socket*，但可以在该 socket 内的核心之间移动；
- *core*: 线程绑定到一个核心，但可以在该核心的超线程之间移动；
- *thread*: 线程绑定到特定的超线程。

`OMP_PROC_BIND` 的取值由实现定义，但通常为：

- `master`: 线程绑定到与主线程相同的位置；
- *close*: 后续线程编号被放置在定义位置的相近处；
- *spread*: 后续线程编号在位置上最大程度地分散；
- *true*：线程绑定到它们的初始位置；
- *false*：线程不绑定到它们的初始位置；

其中值 `master`,*close*,*spread* 由标准规定，其他值取决于实现。

没有用于设置绑定的运行时函数，但可以通过 *omp_get_proc_bind* 获取内部控制变量（ ICV ） *bind-var* 。

绑定也可以通过 `parallel` 指令上的 *proc_bind* 子句设置，取值为 `master`,*close*,*spread*。

示例：如果你有两个插槽并且你定义

```
OMP_PLACES=sockets
```

then

线程 0 进入插槽 0，
线程 1 进入插槽 1，
线程 2 再次进入插槽 0，
• and so on.

另一方面，如果两个插槽共有十六个核心，并且你定义

```
OMP_PLACES=cores
OMP_PROC_BIND=close
```

then

- 线程 0 绑定到核心 0，该核心位于插槽 0，
- 线程 1 绑定到核心 1，该核心位于插槽 0，
- 线程 2 绑定到核心 2，该核心位于插槽 0，
- 依此类推，直到线程 7 绑定到插槽 0 上的核心 7，且
- 线程 8 绑定到核心 8，该核心位于插槽 1，
- 等等。

值 `OMP_PROC_BIND=close` 意味着分配依次通过可用的位置。变量 `OMP_PROC_BIND` 也可以设置为 `spread`，这会将线程分散到各个位置。使用

```
OMP_PLACES=cores
OMP_PROC_BIND=spread
```

你会发现

- 线程 0 绑定到核心 0，该核心位于插槽 0 上，
- 线程 1 绑定到核心 8，该核心位于插槽 1 上，
- 线程 2 绑定到 socket 0 上的核心 1，
- 线程 3 绑定到 socket 1 上的核心 9，
- 依此类推，直到线程 14 绑定到 socket 0 上的核心 7，且
- 线程 15 绑定到 socket 1 上的核心 15。

所以你会看到 `OMP_PLACES=cores` 和 `OMP_PROC_BIND=spread` 与 `OMP_PLACES=sockets` 非常相似。区别在于后者的选择不会将线程绑定到特定核心，因此操作系统可以移动线程，并且可以在同一核心上放置多个线程，即使还有其他核心未被使用。

值 `OMP_PROC_BIND=master` 将线程放置在与团队主线程相同的位置。如果你递归地创建团队，这很方便。在这种情况下，你会使用 *dproc* 子句，而不是环境变量，初始团队设置为 `spread`，递归创建的团队设置为 `master`。

25.1.2 线程绑定的影响

让我们考虑两个示例程序。首先我们考虑用于计算 π 的程序，该程序纯粹是计算密集型的。

#threads	close/cores	spread/sockets	spread/cores
1	0.359	0.354	0.353
2	0.177	0.177	0.177
4	0.088	0.088	0.088
6	0.059	0.059	0.059
8	0.044	0.044	0.044
12	0.029	0.045	0.029
16	0.022	0.050	0.022

我们看到 OMP_PLACES=cores 策略几乎实现了完美的加速；而使用 OMP_PLACES=sockets 时，可能会偶尔发生两个线程落在同一核心上的冲突。

Next we take a program for computing the time evolution of the *heat equation*:

$$t = 0, 1, 2, \ldots : \forall_i : x_i^{(t+1)} = 2x_i^{(t)} - x_{i-1}^{(t)} - x_{i+1}^{(t)}$$

这是一个带宽受限的操作，因为每个数据项的计算量较低。

#threads	close/cores	spread/sockets	spread/cores
1	2.88	2.89	2.88
2	1.71	1.41	1.42
4	1.11	0.74	0.74
6	1.09	0.57	0.57
8	1.12	0.57	0.53
12	0.72	0.53	0.52
16	0.52	0.61	0.53

A我们看到 OMP_PLACES=sockets 在高核心数时表现更差，可能是因为
of 线程最终集中在同一个核心上。这个例子中需要注意的是，当核心数为 6 或 8 时，
OMP_PROC_BIND=spread 策略的性能是 OMP_PROC_BIND=close 的两倍。

原因是单个插槽没有足够的带宽支持插槽上的所有八个核心。因此，将八个线程分布在两个插槽上，比将所有线程放在一个插槽上，每个线程可用的带宽更高。

25.1.3 Place definition

有三个预定义值用于 OMP_PLACES 变量： sockets，cores，threads。你已经见过前两个； threads 该值在具有硬件线程的处理器上变得相关。在这种情况下，OMP_PLACES=cores 不会将线程绑定到特定的硬件线程，这又可能导致如上例所示的冲突。设置 OMP_PLACES=threads 将每个 OpenMP 线程绑定到特定的硬件线程。

还有一种非常通用的语法用于定义 places，它使用

```
location:number:stride
```

语法。示例：

- OMP_PLACES="{0:8:1},{8:8:1}" 在一个两插槽设计中，每个插槽有八个核心时等价于 sockets：它定义了两个 places，每个 place 包含连续的八个核心。线程随后交替分布在这两个 places 之间，但在 place 内部没有进一步指定。
- 设置 cores 等价于 OMP_PLACES="{0},{1},{2},...,{15}"
- 在一个四插槽设计中，规范 OMP_PLACES="{0:4:8}:4:1" 指出位置 0,8,16,24 需要重复四次，步长为 1。换句话说，thread0 最终位于某个插槽的 core0，thread1 位于某个插槽的 core1，依此类推。

25.1.4 绑定可能性

OMP_PROC_BIND 的值为： false，true，master，close，spread。

- false: 不设置绑定
- true: 将线程锁定到一个核心
- master: 将线程与主线程放置在一起
- close: 将线程放置在位置列表中靠近主线程的位置
- spread: 尽可能分散线程

这个效果可以通过在 `dproc` 指令中的 `parallel` 子句来实现局部化。

A safe default setting is

```
export OMP_PROC_BIND=true
```

which prevents the operating system from *migrating a thread*. This prevents many scaling problems.

在 *Intel Knight's Landing* 上的线程放置的好例子： https://software.intel.com/en-us/articles/process-and-thread-affinity-for-intel-xeon-phi-processors-x200

举个例子，考虑一个有两个线程写入共享位置的代码。

```
// sharing.c
#pragma omp parallel
  { // not a parallel for: just a bunch of reps
    for (int j = 0; j < reps; j++) {
#pragma omp for schedule(static,1)
      for (int i = 0; i < N; i++){
#pragma omp atomic
        a++;
      }

    }
  }
```

现在运行时间有很大差异，取决于线程之间的距离。我们在一个同时具有核心和超线程的处理器上测试这个。首先我们将 OpenMP 线程绑定到核心：

```
OMP_NUM_THREADS=2 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 4752.231836usec
sum = 80000000.0
```

Next we force the OpenMP threads to bind to hyperthreads inside one core:

```
OMP_PLACES=threads OMP_PROC_BIND=close ./sharing
run time = 941.970110usec
sum = 80000000.0
```

当然，在这个例子中，内层循环几乎没有意义，且并行并没有加速任何操作：

```
OMP_NUM_THREADS=1 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 806.669950usec
sum = 80000000.0
```

然而，我们看到两线程的结果几乎一样快，这意味着并行化开销非常小。

25.2 First-touch

亲和性问题出现在 *first-touch* 现象中。

一些背景知识。内存是以内存页组织的，我们所认为的 “ 地址 ” 实际上是虚拟地址，通过页表映射到物理地址。

这意味着程序中的数据可以位于物理内存的任何位置。特别是在双插槽节点上，内存可以映射到任一插槽。

接下来需要知道的是，使用 `malloc` 等例程分配的内存不会立即映射；只有在写入数据时才会发生映射。基于此，考虑以下 OpenMP 代码：

```
double *x = (double*) malloc(N*sizeof(double));

for (i=0; i<N; i++)
  x[i] = 0;

#pragma omp parallel for
for (i=0; i<N; i++)
  .... something with x[i] ...
```

由于初始化循环不是并行执行的，它由主线程执行，使得所有内存都与该线程所在的插槽相关联。随后另一个插槽的访问将访问不属于该插槽的内存中的数据，这会导致相当大的延迟和性能下降。

25.2.1 示例

让我们考虑一个示例。我们使初始化并行，受一个选项控制：

```
// heat.c
#pragma omp parallel if (init>0)
  {
#pragma omp for
    for (int i=0; i<N; i++)
      y[i] = x[i] = 0.;
    x[0] = 0; x[N-1] = 1.;
  }
```

如果初始化不是并行的，数组将映射到主线程的插槽；如果是并行的，则可能映射到不同的插槽，取决于线程运行的位置。

作为一个简单的应用，我们运行一个热方程，它是并行的，尽管不是完全无依赖的：

```
for (int it=0; it<1000; it++) {
#pragma omp parallel for
  for (int i=1; i<N-1; i++)
    y[i] = ( x[i-1]+x[i]+x[i+1] )/3.;
#pragma omp parallel for
  for (int i=1; i<N-1; i++)
    x[i] = y[i];
}
```

On the *TACC Frontera* machine, with dual 28-core *Intel Cascade Lake* processors, we use the following settings:

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
# no parallel initialization
make heat && OMP_NUM_THREADS=56 ./heat
```

```
# yes parallel initialization
make heat && OMP_NUM_THREADS=56 ./heat 1
```

这给我们带来了显著的运行时间差异：

- 顺序初始化：平均值 =2.089，标准差 =0.1083
- 并行初始化：平均值 =1.006，标准差 =0.0216

This large difference will be mitigated for algorithms with higher arithmetic intensity.

Exercise 25.1. How do the OpenMP dynamic schedules relate to this issue?

25.2.2 C++ 中的解决方案

在 *C++* 中实现 first-touch 的问题是 *std::vector* 用默认值填充其分配。这被称为 “ 值初始化 ”，它使得

```
vector<double> x(N);
```

等同于上述非并行的分配和初始化。

这里有一个解决方案。

C++ 注 *25*：未初始化的容器。默认初始化是一个问题。我们为未初始化类型制作一个模板：

```
// heatalloc.cxx
template<typename T>
struct uninitialized {
  uninitialized() {};
  T val;
  constexpr operator T() const {return val;};
  T operator=( const T&& v ) { val = v; return val; };
};
```

这样我们就可以创建表现正常的向量：

```
vector<uninitialized<double>> x(N),y(N);
#pragma omp parallel for
for (int i=0; i<N; i++)y[i] = x[i] = 0.;
x[0] = 0; x[N-1] = 1.;
```

使用常规定义的向量和上述修改运行代码，可以重现上面 C 变体的运行时间。

另一种选择是将用 *new* 分配的内存包装在 *unique_ptr* 中：

```
// heatptr.cxx
unique_ptr<double[]> x( new double[N] );
unique_ptr<double[]> y( new double[N] );

#pragma omp parallel for
for (int i=0; i<N; i++) {
  y[i] = x[i] = 0.;
}
x[0] = 0; x[N-1] = 1.;
```

注意这会产生相当优雅的代码，因为方括号索引被重载用于 *unique_ptr*。唯一的缺点是我们无法查询这些数组的 size 。或者用 at 进行边界检查，但在高性能环境中这通常不合适。

25.2.3 备注

你可以用 move_pages 移动页面。

通过考虑亲和性，实际上你是在采用一种 SPMD 风格的编程。你可以通过让每个线程分别分配其部分数组，并将私有指针存储为 *threadprivate* [20] 来明确这一点。然而，这使得线程无法访问分布式数组中彼此的部分，因此这只适用于完全的数据并行或极易并行应用。

25.3 OpenMP 之外的亲和性控制

有各种工具用于控制进程和线程的放置。

进程放置可以在操作系统级别通过 numactl 进行控制（TACC 工具 tacc_affinity 是其封装器）在 Linux 上（也有 taskset ）；Windows 上是 start/affinity。

对应的系统调用：Solaris 上的 pbing，Linux 上的 sched_setaffinity，Windows 上的 SetThreadAffinityMask。

对应的环境变量：Solaris 上的 SUNW_MP_PROCBIND，Intel 上的 KMP_AFFINITY。

Intel compiler 有一个用于亲和性控制的环境变量：

```
export KMP_AFFINITY=verbose,scatter
```

取值： none,scatter,compact

对于 *gcc*:

```
export GOMP_CPU_AFFINITY=0,8,1,9
```

对于 *Sun* 编译器：

```
SUNW_MP_PROCBIND
```

25.4 测试

我们取一个简单的循环并考虑绑定参数的影响。

```
// speedup.c
#pragma omp parallel for
      for (int ip=0; ip<N; ip++) {
        for (int jp=0; jp<M; jp++) {
          double f = sin( values[ip] );
          values[ip] = f;
        }
      }
```

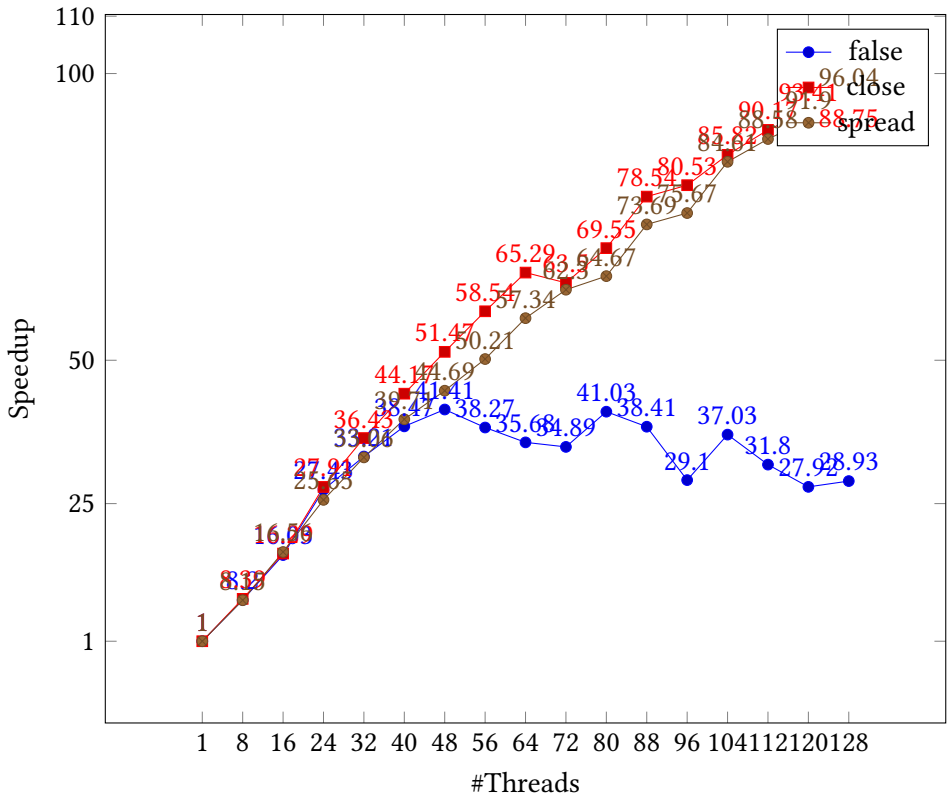


图 25.1：线程数与加速比的关系，Lonestar 6 集群，不同绑定参数

25.4.1 Lonestar 6

Lonestar 6，双插槽 *AMD Milan*，共计 112 核：图 25.1。

25.4.2 Frontera

Intel Cascade Lake，双插槽，共 56 核；图 25.2。

对于所有核心数达到总数一半，所有绑定策略的性能似乎相等。之后，`close` 和 `spread` 表现相同，但 `false` 值的加速比给出了不稳定的数字。

25.4.3 Stampede2 skylake

双 24 核 *Intel Skylake*; 图 25.3.

我们看到 `close` 绑定的性能比 `spread` 差。将绑定设置为 `false` 仅在大核心数时表现不佳。

25.4.4 Stampede2 KnightsLanding

我们在单插槽 68 核处理器上测试：*Intel Knights Landing*。

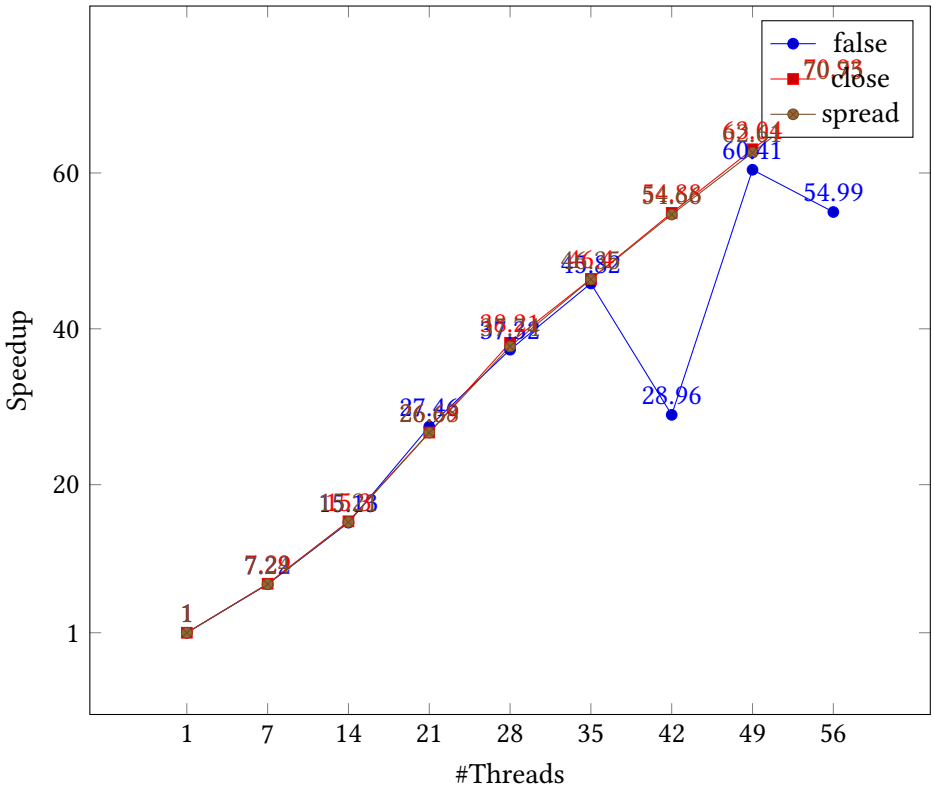


图 25.2：Frontera 集群中不同绑定参数下，线程数与加速比的关系

由于这是单插槽设计，我们不区分 `close` 和 `spread` 绑定。然而，`true` 的绑定值显示出良好的加速效果 —— 实际上超过了核心数 —— 而 `false` 的性能比其他架构更差。

25.4.5 Longhorn

双 20 核 *IBMPower9*，4 个超线程；25.5

与 Intel 处理器不同，这里我们使用超线程。图 25.5 显示在 40 线程时加速比出现下降。对于更高的线程数，加速比增加，远超物理核心数 40。

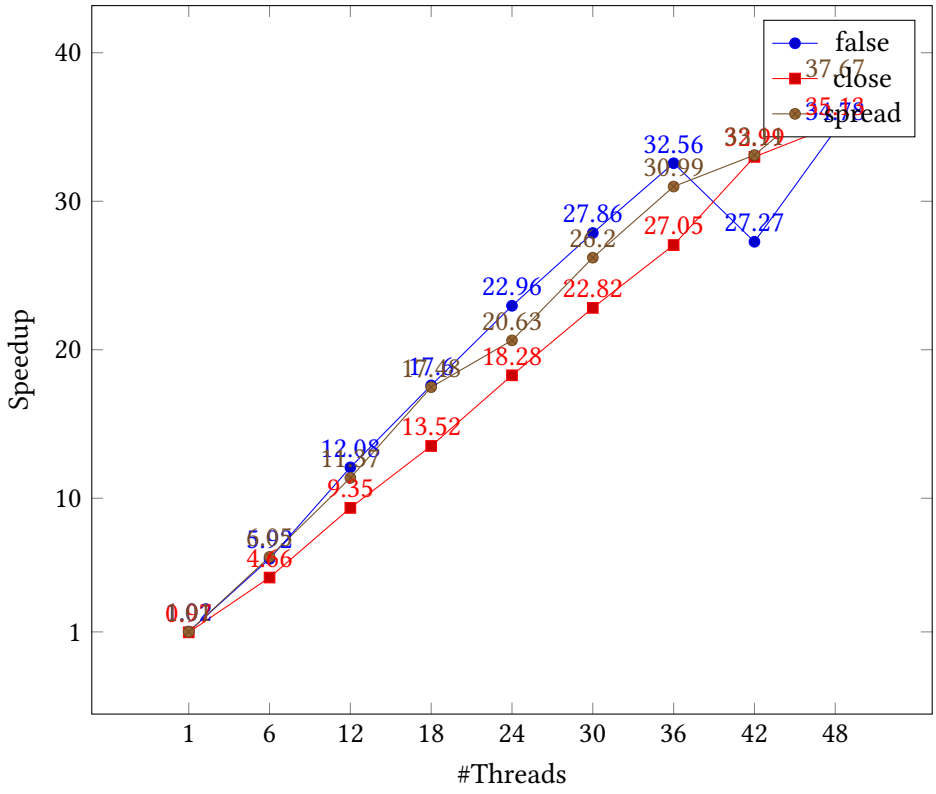


图 25.3：线程数的加速比，Stampede2 skylake 集群，不同绑定参数

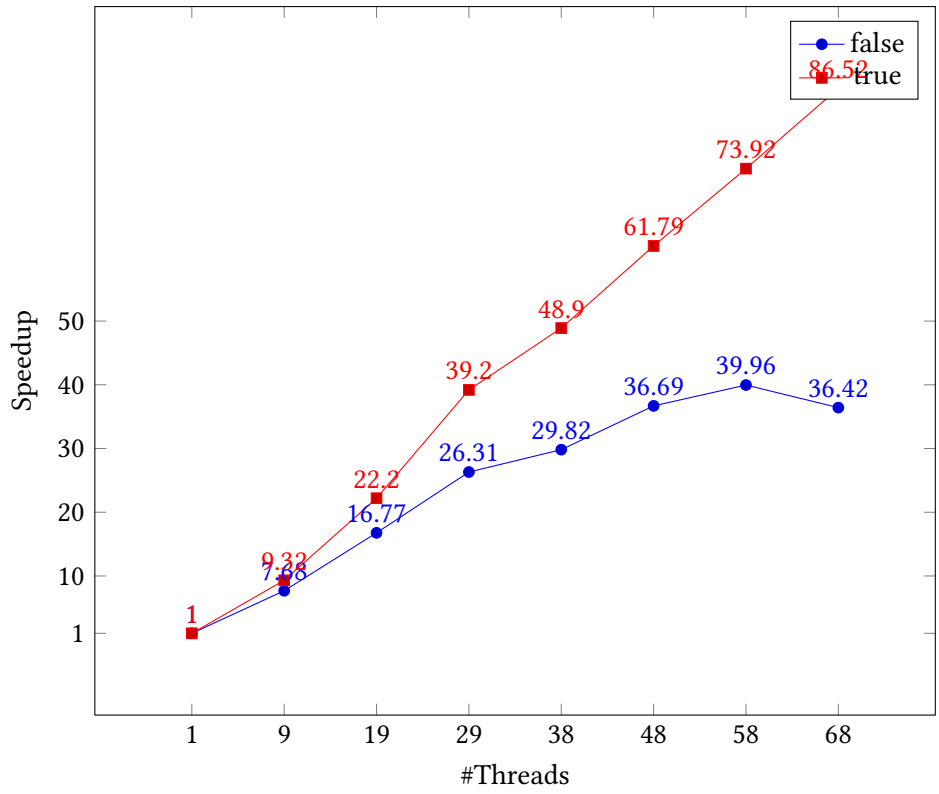


图 25.4：线程数的加速比，Stampede2 Knights Landing 集群，不同绑定参数

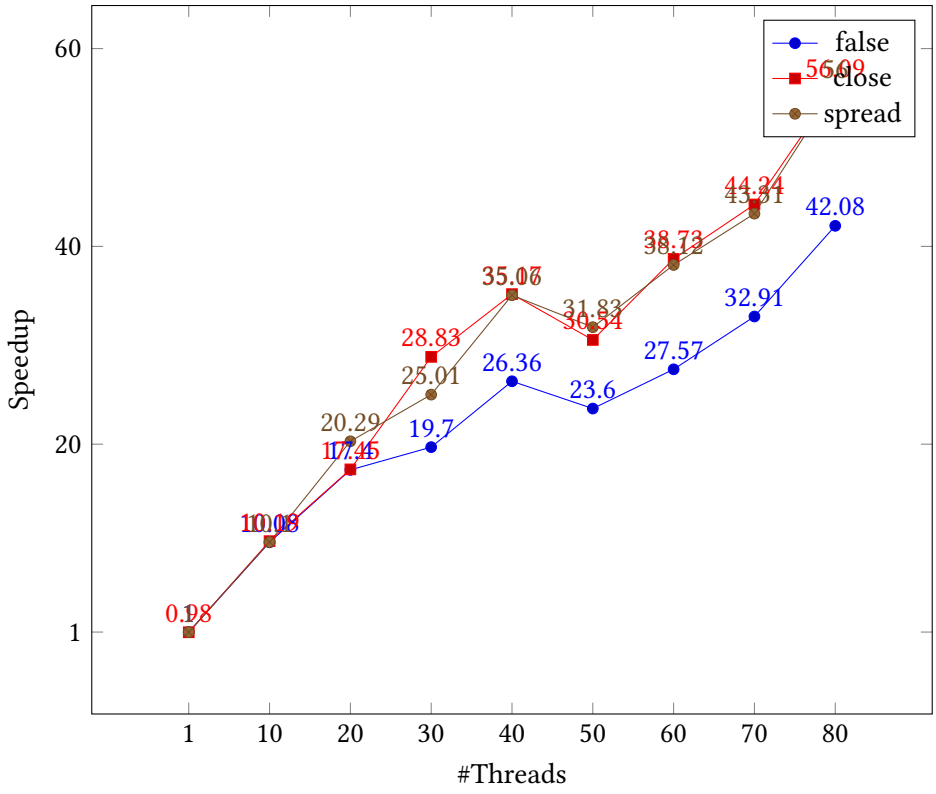


图 25.5：线程数的加速比，Longhorn 集群，不同绑定参数

第 26 章

OpenMP 主题：SIMD 处理

您可以声明一个循环使用 *simd* 以向量指令执行。

备注 39 根据您的编译器，可能需要额外的选项来启用 *SIMD*：

- `-fopenmp-simd` 对于 GCC / Clang，以及
- `-qopenmp-simd` 对于 ICC。

simd pragma 有以下子句：

- *safelen*(*n*)：限制 SIMD 块中的迭代次数。如果你结合使用 `parallel for simd`，这可能很有用。
- *linear*：列出与迭代参数线性相关的变量。
- *aligned*：指定变量的对齐方式。

I如果你的 SIMD 循环包含函数调用，你可以声明该函数可以被转换为向量指令
with *declare simd*

I如果一个循环既支持多线程又支持向量化，你可以将指令组合为 `pragma omp parallel for simd`.

编译器可以被设置为报告一个循环是否被向量化 torized:

```
LOOP BEGIN at simdf.c(61,15)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END
```

对于 Intel 编译器，可以使用诸如 `-Qvec-report=3` 的选项。

P这些指令的性能提升不一定立即显现。在操作
b带宽受限的情况下，使用 `simd` 并行可能会得到与线程并行相同或更差的性能。

以下函数可以向量化：

```
// simdfunctions.c#pragma omp declare simd
double cs(double x1,double x2,double y1,double y2) {
doubleinprod = x1*x2+y1*y2,
xnorm = sqrt(x1*x1 + x2*x2),
ynorm = sqrt(y1*y1 + y2*y2);
return inprod / (xnorm*ynorm);}
```

```
#pragma omp declare simd uniform(x1,x2,y1,y2) linear(i)
double csa(double *x1,double *x2,double *y1,double *y2, int i) {
doubleinprod = x1[i]*x2[i]+y1[i]*y2[i],
xnorm = sqrt(x1[i]*x1[i] + x2[i]*x2[i]),
ynorm = sqrt(y1[i]*y1[i] + y2[i]*y2[i]);
return inprod / (xnorm*ynorm);}
```

以常规方式编译

```
# parameter 1(x1): %xmm0
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

movaps    %xmm0, %xmm5    5 <- x1movaps    %xmm2, %xmm4    4 <- y1
mulsd     %xmm1, %xmm5    5 <- 5 * x2 = x1 * x2mulsd     %xmm3,
 %xmm4    4 <- 4 * y2 = y1 * y2mulsd     %xmm0,
 %xmm0    0 <- 0 * 0 = x1 * x1mulsd     %xmm1,
 %xmm1    1 <- 1 * 1 = x2 * x2addsd     %xmm4,
 %xmm5    5 <- 5 + 4 = x1*x2 + y1*y2mulsd     %xmm2,
 %xmm2    2 <- 2 * 2 = y1 * y1mulsd     %xmm3,
 %xmm3    3 <- 3 * 3 = y2 * y2addsd     %xmm1,
 %xmm0    0 <- 0 + 1 = x1*x1 + x2*x2addsd     %xmm3,
 %xmm2    2 <- 2 + 3 = y1*y1 + y2*y2sqrtsd    %xmm0,
 %xmm0    0 <- sqrt(0) = sqrt( x1*x1 + x2*x2 )sqrtsd    %xmm2,
 %xmm2    2 <- sqrt(2) = sqrt( y1*y1 + y2*y2 )
```

使用标量指令 `mulsd`：标量双精度乘法。

使用 `declare simd` 指令：

```
movaps    %xmm0, %xmm7
movaps    %xmm2, %xmm4
mulpd     %xmm1, %xmm7
mulpd     %xmm3, %xmm4
```

使用矢量指令 `mulpd`：打包双精度乘法，操作 128 位 *SSE2* 寄存器。

为 *Intel Knight's Landing* 编译会生成更复杂的代码：

```
# parameter 1(x1): %xmm0
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

vmulpd     %xmm3, %xmm2, %xmm4                              4 <- y1*y2
vmulpd     %xmm1, %xmm1, %xmm5                              5 <- x1*x2
vbroadcastsd .L_2il0floatpacket.0(%rip), %zmm21movl      $3, %eax

                                                            set accumulator EAX
vbroadcastsd .L_2il0floatpacket.5(%rip), %zmm24
kmovw     %eax, %k3                                         set mask k3
```

```
vmulpd    %xmm3, %xmm3, %xmm6                          6 <-y1*y1 (stall)vfmadd231pd %xmm0,
 %xmm1, %xmm4                        4 <- 4 + x1*x2 (no reuse!)vfmadd213pd %xmm5, %xmm0,
 %xmm0                          0 <- 0 + 0*5 = x1 + x1*(x1*x2vmovaps   %zmm21,                 )
 %zmm18                                 #25.26 c7vmovapd   %zmm0,
 %zmm3{%k3}{z}                           #25.26 c11vfmadd213pd %xmm6, %xmm2,
 %xmm2                          #24.29 c13vpcmpgtq  %zmm0, %zmm21,
 %k1{%k3}                        #25.26 c13vscalefpd .L_2il0floatpacket.1(%rip){1to8}, %zmm0,
 %zmm3{%k1} #25.26 c15vmovaps   %zmm4, %zmm26                                  #25.26 c15
vmovapd   %zmm2, %zmm7{%k3}{z}                          #25.26 c17vpcmpgtq  %zmm2, %zmm21,
 %k2{%k3}                        #25.26 c17vscalefpd .L_2il0floatpacket.1(%rip){1to8}, %zmm2,
 %zmm7{%k2} #25.26 c19vrsqrt28pd %zmm3, %zmm16{%k3}{z}                         #25.26 c19
vpxorq    %zmm4, %zmm4, %zmm26{%k3}                    #25.26 c19vrsqrt28pd %zmm7,
 %zmm20{%k3}{z}                         #25.26 c21vmulpd    {rn-sae}, %zmm3, %zmm16,
 %zmm19{%k3}{z}       #25.26 c27 stall 2vscalefpd .L_2il0floatpacket.2(%rip){1to8}, %zmm16,
 %zmm17{%k3}{z} #25.26 c27vmulpd    {rn-sae}, %zmm7, %zmm20, %zmm23{%k3}{z}       #25.26 c29
vscalefpd .L_2il0floatpacket.2(%rip){1to8}, %zmm20, %zmm22{%k3}{z} #25.26 c29
vfnmadd231pd {rn-sae}, %zmm17, %zmm19, %zmm18{%k3}      #25.26 c33 stall 1vfnmadd231pd {rn-sae},
 %zmm22, %zmm23, %zmm21{%k3}      #25.26 c35vfmadd231pd {rn-sae}, %zmm19, %zmm18,
 %zmm19{%k3}       #25.26 c39 stall 1vfmadd231pd {rn-sae}, %zmm23, %zmm21,
 %zmm23{%k3}       #25.26 c41vfmadd213pd {rn-sae}, %zmm17, %zmm17,
 %zmm18{%k3}       #25.26 c45 stall 1vfnmadd231pd {rn-sae}, %zmm19, %zmm19,
 %zmm3{%k3}       #25.26 c47vfmadd213pd {rn-sae}, %zmm22, %zmm22,
 %zmm21{%k3}       #25.26 c51 stall 1vfnmadd231pd {rn-sae}, %zmm23, %zmm23,
 %zmm7{%k3}       #25.26 c53vfmadd213pd %zmm19, %zmm18,
 %zmm3{%k3}                  #25.26 c57 stall 1vfmadd213pd %zmm23, %zmm21,
 %zmm7{%k3}                  #25.26 c59vscalefpd .L_2il0floatpacket.3(%rip){1to8}, %zmm3,
 %zmm3{%k1} #25.26 c63 stall 1vscalefpd .L_2il0floatpacket.3(%rip){1to8}, %zmm7,
 %zmm7{%k2} #25.26 c65vfixupimmpd $112, .L_2il0floatpacket.4(%rip){1to8}, %zmm0,
 %zmm3{%k3} #25.26 c65vfixupimmpd $112, .L_2il0floatpacket.4(%rip){1to8}, %zmm2,
 %zmm7{%k3} #25.26 c67vmulpd    %xmm7, %xmm3, %xmm0                            #25.26 c71
vmovaps   %zmm0, %zmm27                                 #25.26 c79vmovaps   %zmm0,
 %zmm25                                #25.26 c79vrcp28pd  {sae}, %zmm0,
 %zmm27{%k3}                     #25.26 c81vfnmadd213pd {rn-sae}, %zmm24, %zmm27,
 %zmm25{%k3}      #25.26 c89 stall 3vfmadd213pd {rn-sae}, %zmm27, %zmm25,
 %zmm27{%k3}       #25.26 c95 stall 2vcmppd    $8, %zmm26, %zmm27,
 %k1{%k3}                  #25.26 c101 stall 2vmulpd    %zmm27, %zmm4,
 %zmm1{%k3}{z}                  #25.26 c101kortestw  %k1,
 %k1                                    #25.26 c103
je        ..B1.3        # Prob 25%                      #25.26 c105vdivpd    %zmm0, %zmm4,
 %zmm1{%k1}                      #25.26 c3 stall 1vmovaps   %xmm1,
 %xmm0                                  #25.26 c77
ret                                                     #25.26 c79
```

```
#pragma omp declare simd uniform(op1) linear(k) notinbranch
  double SqrtMul(double *op1, double op2, int k) {
    return (sqrt(op1[k]) * sqrt(op2));
```

```
|| }
```

第 27 章

OpenMP 主题：Offloading

本章解释了在 OpenMP-4.0 中引入的将工作卸载到图形处理单元（ GPU ）的机制。

处理器的内存和附加 GPU 的内存不是 *coherent* 的：它们有各自独立的内存空间，在一个中写入数据不会自动反映到另一个中。

当你进入一个 *target* 构造时，OpenMP 会传输数据（或映射数据）。

```
#pragma omp target
{
  // do stuff on the GPU
}
```

你可以用 *omp_is_initial_device* 测试目标区域是否确实在设备上执行：

```
#pragma omp target
  if (omp_is_initial_device()) printf("Offloading failed\n");
```

27.0.1 目标和任务

target 子句使 OpenMP 创建一个目标任务。这是一个在主机上运行的任务，专门用于管理卸载区域。

target 区域由一个新的初始任务执行。这与执行主程序的初始任务不同。

创建目标任务的任务称为生成任务。

默认情况下，生成任务在设备上的任务运行时会被阻塞，但添加 *target*nowait 子句后，任务变为异步。这需要 `taskwait` 指令来同步主机和设备。

27.1 Dataon thedevice

- 标量被视为 *firstprivate*，也就是说，它们被复制进来但不复制出去。
- 栈数组 *tofrom*。
- 堆数组默认不映射。

对于使用 `map` 的显式映射：

```
#pragma omp target map(...)
{
  // do stuff on the GPU
}
```

存在以下 map 选项：

- *map*(*to*: *x*,*y*,*z*) 进入 target 区域时从主机复制到设备。
- *map*(*from*: *x*,*y*,*z*) 退出 target 区域时从设备复制到主机。
- *map*(*tofrom*: *x*,*y*,*z*) 等同于结合前两个。
- *map*(*allo*: *x*,*y*,*z*) 在设备上分配数据。

Fortran 注释 25：*map* 子句中的数组大小。如果编译器能够推断数组的边界和大小，则无需在 'map' 子句中指定它们。

向设备传输数据可能很慢，因此在卸载代码段开始时映射数据可能不是最佳选择。此外，在许多情况下，数据会在设备上驻留多个迭代周期，例如时间步进 PDE 求解器。基于这些原因，可以使用 `enter data` 和 `exit data` 指令显式地将数据移动到设备上或从设备移出。

```
#pragma omp target enter data map(to: x,y)
#pragma omp target
{
  // do something
}
#pragma omp target enter data map(from: x,y)
```

还有 `update to` （从主机同步数据到设备），`update from` （从设备同步数据到主机）。

27.2 在设备上的执行

在设备上并行执行循环时使用 *teams* 子句：

```
#pragma omp target teams distribute parallel do
```

在 GPU 设备及类似设备上，线程有如下结构：

- 线程被分组在 *team* 中，并且只能在这些 team 内进行同步；
- team 是 *league* 中的组，在 *target* 区域内 league 之间不可能进行同步。

T组合 `teams distribute` 将迭代空间分割到各个 team。默认使用静态调度，但
t选项 *dist_schedule* 可用于指定不同的一个。然而，这种组合仅提供了该块的
s将任务分配给每个团队中的主线程。接下来我们需要 `parallel for` 或 `parallel do` 来分配这部分任务。
t团队中的线程。

创建团队时，通常有必要用 *thread_limit* 限制每个团队中的线程数。这也可以通过 `OMP_THREAD_LIMIT` 环境变量设置。该值可以通过 `omp_get_thread_limit` 查询。

第 28 章

OpenMP 剩余主题

28.1 运行时函数，环境变量，内部控制变量

OpenMP 有许多可以通过环境变量设置的设置项，并且可以通过库例程查询和设置。这些设置称为内部控制变量 *(ICVs)*：OpenMP 实现的行为就像有一个内部变量存储该设置一样。

运行时函数包括：

- 计数线程和核心： `omp_set_num_threads`，`omp_get_num_threads`，`omp_get_max_threads`,`omp_get_num_procs`；参见第 17.5 节。
- 查询当前线程： `omp_get_thread_num`, `omp_in_parallel`
- `omp_set_dynamic`
- `omp_get_dynamic`
- `omp_set_nested`
- `omp_get_nested`
- `omp_get_wtime`
- `omp_get_wtick`
- `omp_set_schedule`
- `omp_get_schedule`
- `omp_set_max_active_levels`
- `omp_get_max_active_levels`
- `omp_get_thread_limit`
- `omp_get_level`
- `omp_get_active_level`
- `omp_get_ancestor_thread_num`
- `omp_get_team_size`

以下是 OpenMP 环境变量：

- `OMP_CANCELLATION` 设置是否启用取消；参见第 18.3 节。
- `OMP_DISPLAY_ENV` 显示 OpenMP 版本（章节 28.7 ）和环境变量。
- `OMP_DEFAULT_DEVICE` 设置 target 区域中使用的设备
- `OMP_DYNAMIC` 线程的动态调整
- `OMP_MAX_ACTIVE_LEVELS` 设置嵌套并行区域的最大数量；章节 18.2。
- `OMP_MAX_TASK_PRIORITY` 设置最大任务优先级值；章节 24.6.2。
- `OMP_NESTED` 嵌套并行区域
- `OMP_NUM_THREADS` 指定要使用的线程数
- `OMP_PROC_BIND` 线程是否可以在 CPU 之间移动；参见章节 25.1。

- OMP_PLACES 指定线程应放置在哪些 CPU 上；参见章节 25.1。
- OMP_STACKSIZE 设置默认线程栈大小；参见章节 22.2。
- OMP_SCHEDULE 线程的调度方式；参见章节 19.3。
- OMP_THREAD_LIMIT 设置最大线程数；参见章节 27.2。
- OMP_WAIT_POLICY 等待线程的处理方式；ICV *wait-policy-var*。取值： ACTIVE 表示保持线程自旋， PASSIVE 表示线程等待时可能让出处理器。没有运行时函数用于设置此项。

有 4 个 ICV 表现得好像每个线程都有自己的副本。默认值是实现定义的，除非另有说明。

- 可能可以动态调整并行区域的线程数。变量：OMP_DYNAMIC ； 例程： omp_set_dynamic, omp_get_dynamic。
- 如果代码包含嵌套并行区域，内部区域可能创建新的团队，或者可能由遇到它们的单个线程执行。变量：OMP_NESTED ； 例程 omp_set_nested,omp_get_nested。允许的值是 TRUE 和 FALSE ； 默认值为 false。
- 可以控制遇到的并行区域使用的线程数。变量：OMP_NUM_THREADS ； 例程 omp_set_num_threads, omp_get_max_threads。
- 可以设置并行循环的调度。变量： OMP_SCHEDULE ； 例程 omp_set_schedule,omp_get_schedule。

不明显的语法：

```
export OMP_SCHEDULE="static,100"
```

其他设置：

- omp_get_num_threads：查询当前代码位置活跃的线程数；这可能低于通过 omp_set_num_threads 设置的数量。为了获得有意义的答案，应在并行区域内进行查询。
- omp_get_thread_num
- omp_in_parallel：测试是否处于并行区域。
- omp_get_num_procs：查询可用的物理核心数。

其他环境变量：

- OMP_STACKSIZE 控制为每个线程分配的空间大小栈。这用作私有变量的空间，参见第 22.2 节，或归约，参见第 20.2.2 节。
- OMP_WAIT_POLICY 决定等待的线程的行为，例如等待 临界区 :- ACTIVE 使线程进入 自旋锁，线程会主动检查是否可以继续；- PASSIVE 使线程进入睡眠状态，直到操作系统 (OS) 唤醒它。

  “ 主动 ” 策略在线程等待时使用 CPU ； 另一方面，等待结束后激活线程是瞬时的。采用 “ 被动 ” 策略时，线程在等待时不使用任何 CPU，但再次激活线程的开销较大。因此，只有当线程等待时间（相对）较长时，被动策略才有意义。
- OMP_PROC_BIND 取值为 TRUE 和 FALSE 时，可以将线程绑定到处理器。一方面，这样做可以最小化数据移动；另一方面，可能会增加负载不平衡。

28.2 Timing

OpenMP 有一个墙钟计时例程 omp_get_wtime

```
double omp_get_wtime(void);
```

起点是任意的，并且每次程序运行时都不同；然而，在一次运行中，它对所有线程是相同的。该计时器的分辨率由 `omp_get_wtick` 给出。

练习 28.1. 使用计时例程来演示使用多线程带来的加速。

- 编写一段耗时可测量的代码，即它应耗费时钟周期的整数倍。
- 编写一个并行循环并测量加速比。例如，你可以这样做

```
for (int use_threads=1; use_threads<=nthreads; use_threads++) {
#pragma omp parallel for num_threads(use_threads)
    for (int i=0; i<nthreads; i++) {
        .....
    }
    if (use_threads==1)
      time1 = tend-tstart;
    else // compute speedup
```

- 为了防止编译器优化掉你的循环，让循环体计算一个结果，并使用归约来保留这些结果。

28.3 线程安全

使用 OpenMP，通过引入并行区段，将现有代码并行化相对容易。如果你小心地声明适当的变量为共享和私有，这可能运行良好。然而，你的代码可能包含调用带有竞态条件的库例程；这样的代码被称为不是线程安全的。

例如一个例程

```
static int isave;
int next_one() {
int i = isave;isave += 1;
return i;}...for ( .... ) {
int ivalue = next_one();}
```

存在明显的竞态条件，因为循环的迭代可能获得不同的 `next_one` 值，正如它们应该的那样，或者不然。这可以通过对 `next_one` 调用使用 *critical* 指令来解决；另一种解决方案是对 `isave` 使用 *threadprivate* 声明。例如，如果 `next_one` 例程实现了一个随机数生成器，这就是正确的解决方案。

28.4 性能与调优

OpenMP 代码的性能可能受到以下因素的影响 .

阿姆达尔效应您的代码需要有足够多的并行部分（参见 HPC 书籍，第 2.2.3 节）。顺序部分可以通过在每个线程上冗余执行来加速，因为这样可以保持数据的本地性。

Dynamism 创建线程团队需要时间。实际上，不会为每个并行区域创建和删除团队，但创建不同大小的团队或递归线程创建可能会引入开销。

负载不平衡 即使你的程序是并行的，你也需要关注负载平衡。在并行的情况下
循环中，你可以将 `schedule` 子句设置为 `dynamic`，这可以平衡工作量，但可能会导致通信增加。

通信 缓存一致性会导致通信。线程应尽可能地访问自己的数据。

- 线程很可能会读取彼此的数据。这在很大程度上是不可避免的。
- 应避免线程写入彼此的数据：这可能需要同步，并且会导致一致性流量。

- 如果线程可以迁移，曾经是本地的数据在迁移后将不再是本地数据。
- 从一个插槽读取分配在另一个插槽上的数据效率低下；参见第 25.2 节。

亲和性 数据和执行线程都可以在一定程度上绑定到特定的本地。使用本地数据比远程数据更高效，因此你希望使用本地数据，并尽量减少数据或执行的移动范围。

- 参见上述关于导致通信现象的要点。
- 第 25.1.1 节描述了如何指定线程与位置的绑定。亲和性可以有影响，但不一定必须有影响。例如，如果一个 OpenMP 线程可以在硬件线程之间迁移，缓存数据将保持本地。完全允许 OpenMP 线程自由迁移对于负载均衡可能有利，但只有在数据亲和性不那么重要时才应这样做。

- 静态循环调度更有可能使用与执行位置相关的数据，但它们在负载均衡方面表现较差。另一方面，`nowait` 子句可以缓解静态循环调度的一些问题。**绑定** 你可以选择将 OpenMP 线程放得很近或分散开。将它们放得很近

如果它们使用大量共享数据，这是合理的。将它们分散开可能会增加带宽。（参见第 25.1.2 节中的示例。）

同步 屏障是一种同步形式。它们本身代价高昂，并且暴露出负载
不平衡。隐式屏障发生在工作共享构造的末尾；它们可以通过 `nowait` 移除。临界区意味着并行性的损失，但它们也很慢，因为它们是通过操作系统 函数实现的。这些通常相当昂贵，耗费数千个周期。只有当并行工作远远超过其开销时，才应使用临界区。

28.5 Accelerators

在 OpenMP-4.0 中，支持将工作卸载到 *accelerator* 或 *co-processor*：

```
#pragma omp target [clauses]
```

带有如下子句

- `data`: place data
- `update`：使主机和设备之间的数据保持一致

28.6 Tools interface

OpenMP-5.0 定义了一个 tools interface。这意味着可以定义由 OpenMP 运行时调用的例程。例如，下面的示例定义了在 OpenMP 初始化和终止时被调用的回调，从而为应用程序提供运行时。

```
int ompt_initialize(ompt_function_lookup_t lookup, int initial_device_num,
ompt_data_t *tool_data) {printf("libomp init time: %f\n",
omp_get_wtime() - *(double *)(tool_data->ptr));
*(double *)(tool_data->ptr) = omp_get_wtime();
return 1; // success: activates tool}
void ompt_finalize(ompt_data_t *tool_data) {
printf("application runtime: %f\n",
omp_get_wtime() - *(double *)(tool_data->ptr));}
ompt_start_tool_result_t *ompt_start_tool(unsigned int omp_version,
const char *runtime_version) {
static double time = 0; // static defintion needs constant assigment
time = omp_get_wtime();
static ompt_start_tool_result_t ompt_start_tool_result = {
&ompt_initialize, &ompt_finalize, {.ptr = &time}};
return &ompt_start_tool_result; // success: registers tool}
```

(Example courtesy of https://git.rwth-aachen.de/OpenMPTools/OMPT-Examples.)

28.7 OpenMP 标准

以下是 OpenMP 版本值（由 _OPENMP 宏给出）与标准版本之间的对应关系：

- OpenMP-3.1– proc bind 环境变量 – 任务的扩展
- OpenMP-4.0– procbind 子句，places 环境变量 – simd 指令 – GPU 的 device 指令 – taskgroups– 任务的 depend 子句 – cancel– 用户定义的归约

- 201511 OpenMP-4.5，现有构造的许多扩展。
- 201611 技术报告 4：关于 OpenMP-5.0 的信息，但尚未强制执行。
- 201811OpenMP-5.0– 更好地支持 C11 、 C++11/14/18 、 Fortran2008– 非矩形循环嵌套。– scan 扩展为具有 in/exclusive 版本 – 任务、任务循环上的归约。– 内存空间。
- 202011OpenMP-5.1, OpenMP-5.2。

```
// version.c
int standard = _OPENMP;
printf("Supported OpenMP standard:
    ↪%d\n",standard);
switch (standard) {
case  201511: printf("4.5\n");
  break;
case 201611: printf("Technical report 4:
    ↪information about 5.0 but not yet
    ↪mandated.\n");
  break;
case 201811: printf("5.0\n");
  break;
case 202011:
  printf("5.1\n");
  break;
case 202111: printf("5.2\n");
  break;
default:
  printf("Unrecognized version\n");
  break;
}
```

The openmp.org 网站维护了一个记录，说明哪些编译器支持哪些标准： https://www.openmp.org/resources/openmp-compilers-tools/.

28.8 内存模型

28.8.1 Dekker 算法

弱内存模型的一个标准示例是 *Dekker* 算法。我们在 OpenMP 中将其建模如下；

```
// weak1.cint a=0,b=0,r1,r2;
#pragma omp parallel sections shared(a, b, r1, r2)
{#pragma omp section{a = 1;r1 = b;tasks++;}
#pragma omp section{b = 1;r2 = a;tasks++;}}
```

在任何合理的并行执行解释下， `r1,r2` 的可能值是 1,1 0,1 或 1,0。这被称为顺序一致性：并行结果与一个顺序执行相一致，该顺序执行交错了并行计算，尊重它们的局部语句顺序。（另见 HPC 书，章节 -2.6.1.6。）

然而，运行此程序时，我们得到少数情况下出现 $r_1 = r_2 = 0$。有两种可能的解释：

1. 编译器被允许交换第一条和第二条语句，因为它们之间没有依赖关系；或者 2. 线程被允许拥有一个与内存中值不一致的变量本地副本。

我们通过刷新 `a,b` 来解决这个问题：

```
// weak2.cint a=0,b=0,r1,r2;
#pragma omp parallel sections shared(a, b, r1, r2)
{#pragma omp section{a = 1;#pragma omp flush (a,b)
r1 = b;tasks++;}#pragma omp section{b = 1;
#pragma omp flush (a,b)r2 = a;tasks++;}}
```

.

第 29 章

OpenMP 回顾

29.1 Concepts review

29.1.1 基本概念

- 进程 / 线程 / 线程组
- 线程 / 核心 / 任务
- 指令 / 库函数 / 环境变量

29.1.2 并行区域

由一个团队执行

29.1.3 Work sharing

循环 / sections / single / workshare

- implied barrier
- loop scheduling, reduction
- sections
- single vs master
- (F) workshare

29.1.4 数据范围

- 共享与私有，C 与 F
- loop variables and reduction variables
- default declaration
- firstprivate, lastprivate

29.1.5 Synchronization

- barriers, implied and explicit
- nowait
- critical sections
- locks, difference with critical

29.1.6 任务

- 生成与执行
- 依赖关系

29.2 复习问题

29.2.1 指令

以下程序输出什么?

```
int main() {
  printf("procs %d\n",
    omp_get_num_procs());
  printf("threads %d\n",
    omp_get_num_threads());
  printf("num %d\n",
    omp_get_thread_num());
  return 0;
}
```

```
int main() {
#pragma omp parallel
  {
  printf("procs %d\n",
    omp_get_num_procs());
  printf("threads %d\n",
    omp_get_num_threads());
  printf("num %d\n",
    omp_get_thread_num());
  }
  return 0;
}
```

```
Program main
  use omp_lib
  print *,"Procs:",&
    omp_get_num_procs()
  print *,"Threads:",&
    omp_get_num_threads()
  print *,"Num:",&
    omp_get_thread_num()
End Program
```

```
Program main
  use omp_lib
!$OMP parallel
  print *,"Procs:",&
    omp_get_num_procs()
  print *,"Threads:",&
    omp_get_num_threads()
  print *,"Num:",&
    omp_get_thread_num()
!$OMP end parallel
End Program
```

29.2.2 并行性

以下循环可以并行化吗？如果可以，如何并行化？（假设所有数组已填充完毕，且不存在越界错误。）

```
// variant #1
for (i=0; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i] = 2*x[i] + c[i+1];
}
```

```
// variant #2
for (i=0; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i] = 2*x[i+1] + c[i+1];
}
```

```
// variant #3
for (i=1; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i] = 2*x[i-1] + c[i+1];
}
```

```
// variant #4
for (i=1; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i+1] = 2*x[i-1] + c[i+1];
}
```

```
! variant #1
do i=1,N
  x(i) = a(i)+b(i+1)
  a(i) = 2*x(i) + c(i+1)
end do
```

```
! variant #2
do i=1,N
  x(i) = a(i)+b(i+1)
  a(i) = 2*x(i+1) + c(i+1)
end do
```

```
! variant #3
do i=2,N
  x(i) = a(i)+b(i+1)
  a(i) = 2*x(i-1) + c(i+1)
end do
```

```
! variant #3
do i=2,N
  x(i) = a(i)+b(i+1)
  a(i+1) = 2*x(i-1) + c(i+1)
end do
```

29.2.3 数据与同步

29.2.3.1

以下代码片段的输出是什么？假设有四个线程 .

```
// variant #1
int nt;
#pragma omp parallel
  {
  nt = omp_get_thread_num();
  printf("thread number: %d\n",nt);
  }
```

```
// variant #2
int nt;
#pragma omp parallel private(nt)
  {
  nt = omp_get_thread_num();
  printf("thread number: %d\n",nt);
  }
```

```
// variant #3
int nt;
#pragma omp parallel
  {
#pragma omp single
    {
    nt = omp_get_thread_num();
    printf("thread number: %d\n",nt);
    }
  }
```

```
! variant #1
  integer nt
!$OMP parallel
  nt = omp_get_thread_num()
  print *,"thread number:",nt
!$OMP end parallel
```

```
! variant #2
  integer nt
!$OMP parallel private(nt)
  nt = omp_get_thread_num()
  print *,"thread number:",nt
!$OMP end parallel
```

```
! variant #3
  integer nt
!$OMP parallel
!$OMP single
```

```
// variant #4
int nt;
#pragma omp parallel
  {
#pragma omp master
    {
    nt = omp_get_thread_num();
    printf("thread number: %d\n",nt);
    }
  }
```

```
// variant #5
int nt;
#pragma omp parallel
  {
#pragma omp critical
    {
    nt = omp_get_thread_num();
    printf("thread number: %d\n",nt);
    }
  }
```

```
    nt = omp_get_thread_num()
    print *,"thread number:",nt
!$OMP end single
!$OMP end parallel
```

```
! variant #4
  integer nt
!$OMP parallel
!$OMP master
    nt = omp_get_thread_num()
    print *,"thread number:",nt
!$OMP end master
!$OMP end parallel
```

```
! variant #5
  integer nt
!$OMP parallel
!$OMP critical
    nt = omp_get_thread_num()
    print *,"thread number:",nt
!$OMP end critical
!$OMP end parallel
```

29.2.3.2

以下是对串行代码进行并行化的尝试。假设所有变量和数组都已定义。你在这段代码中看到了哪些错误和潜在问题？你会如何修复它们？

```
#pragma omp parallel
{
  x = f();
  #pragma omp for
  for (i=0; i<N; i++)
    y[i] = g(x,i);
  z = h(y);
}
```

```
!$OMP parallel
x = f()!$OMP do
do i=1,N
y(i) = g(x,i)
end do!$OMP end do
z = h(y)
!$OMP end parallel
```

29.2.3.3

假设有两个线程。以下程序输出什么?

```
int a;
#pragma omp parallel private(a) {
  ...
  a = 0;
  #pragma omp for
  for (int i = 0; i < 10; i++)
  {
    #pragma omp atomic
    a++; }
  #pragma omp single
    printf("a=%e\n",a);
}
```

29.2.4 归约

29.2.4.1 以下代码正确吗? 它高效吗? 如果不是, 你能改进它吗?

```
#pragma omp parallel shared(r){int x;
x = f(omp_get_thread_num());#pragma omp critical
r += f(x);}
```

29.2.4.2

比较两个片段:

```
// variant 1
#pragma omp parallel reduction(+:s)
#pragma omp for
  for (i=0; i<N; i++)
    s += f(i);
```

```
// variant 2
#pragma omp parallel
#pragma omp for reduction(+:s)
  for (i=0; i<N; i++)
    s += f(i);
```

```
! variant 1
!$OMP parallel reduction(+:s)
!$OMP do
  do i=1,N
    s += f(i);
  end do
!$OMP end do
!$OMP end parallel
```

```
! variant 2
!$OMP parallel
!$OMP do reduction(+:s)
  do i=1,N
    s += f(i);
  end do
!$OMP end do
!$OMP end parallel
```

它们计算的是同一个东西吗?

29.2.5 屏障

以下两个代码片段定义良好吗?

```
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads; mytid++)
  x[mytid] = some_calculation();
#pragma omp for
for (mytid=0; mytid<nthreads-1; mytid++)
  y[mytid] = x[mytid]+x[mytid+1];
}
```

```
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads; mytid++)
  x[mytid] = some_calculation();
#pragma omp for nowait
for (mytid=0; mytid<nthreads-1; mytid++)
  y[mytid] = x[mytid]+x[mytid+1];
}
```

29.2.6 数据作用域

下面的程序应该初始化与线程数相同的数组行数。

```
int main() {
  int i,icount,iarray[100][100];
  icount = -1;
#pragma omp parallel private(i)
  {
#pragma omp critical
    { icount++; }
    for (i=0; i<100; i++)
      iarray[icount][i] = 1;
  }
  return 0;
}
```

```
Program main
  integer :: i,icount,iarray(100,100)
  icount = 0
!$OMP parallel private(i)
!$OMP critical
    icount = icount + 1
!$OMP end critical
    do i=1,100
      iarray(icount,i) = 1
    end do
!$OMP end parallel
End program
```

描述该程序的行为，并给出论证，

- 如给出；
- 如果你添加一个子句 `private(icount)` 到 `parallel` 指令;
- 如果你添加一个子句 *firstprivate*(icount)。

你怎么看这个解决方案:

```
#pragma omp parallel private(i)
    ↪shared(icount)
  {
#pragma omp critical
    { icount++;
      for (i=0; i<100; i++)
        iarray[icount][i] = 1;
    }
  }
  return 0;
}
```

```
!$OMP parallel private(i) shared(icount)
!$OMP critical
    icount = icount+1
    do i=1,100
      iarray(icount,i) = 1
    end do
!$OMP critical
!$OMP end parallel
```

29.2.7 任务

修正以下示例中的两处内容:

```
#pragma omp parallel
#pragma omp single
{
  int x,y,z;
#pragma omp task
  x = f();
#pragma omp task
  y = g();
#pragma omp task
  z = h();
  printf("sum=%d\n",x+y+z);
}
```

```
  integer :: x,y,z
!$OMP parallel
!$OMP single

!$OMP task
  x = f()
!$OMP end task

!$OMP task
  y = g()
!$OMP end task

!$OMP task
  z = h()
!$OMP end task

  print *,"sum=",x+y+z
!$OMP end single
!$OMP end parallel
```

29.2.8 调度

比较这两个代码片段。它们计算的结果相同吗？你如何评价它们的效率?

```
#pragma omp parallel
#pragma omp single
  {
    for (i=0; i<N; i++) {
    #pragma omp task
      x[i] = f(i)
    }
    #pragma omp taskwait
  }
```

```
#pragma omp parallel
#pragma omp for schedule(dynamic)
  {
    for (i=0; i<N; i++) {
      x[i] = f(i)
    }
  }
```

你如何使第二个循环更高效？你能对第一个循环做类似的优化吗?

第 30 章

OpenMP 示例

30.1 N-body problems

所谓的 *N-body problem* 是指我们描述在诸如引力等力作用下大量实体之间相互作用的问题。例子包括分子动力学和星团。

虽然存在考虑力随距离衰减的巧妙算法，但这里我们考虑显式计算所有相互作用的简单算法。

一个粒子有 x, y 坐标和质量 c。对于两个粒子 $(x_1, y_1, c_1), (x_2, y_2, c_2)$，粒子 1 受到粒子 2 的力为：

$$\vec{F}_{12} = \frac{c_1 \cdot c_2}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \cdot \vec{r}_{12}$$

其中 $\vec{r}_{12}$ 是从粒子 2 指向粒子 1 的单位向量。对于 n 个粒子，每个粒子 i 受到一个力

$$\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij}.$$

让我们从几个构建模块开始。

```
// molecularstruct.c
struct point{ double x,y; double c; };
struct force{ double x,y; double f; };

/* Force on p1 from p2 */
struct force force_calc( struct point p1,struct point p2 ) {
  double dx = p2.x - p1.x, dy = p2.y - p1.y;
  double  f = p1.c * p2.c / sqrt( dx*dx + dy*dy );
  struct force exert = {dx,dy,f};
  return exert;
}
```

力的累积：

```
void add_force( struct force *f,struct force g ) {
  f->x += g.x; f->y += g.y; f->f += g.f;
}
void sub_force( struct force *f,struct force g ) {
  f->x -= g.x; f->y -= g.y; f->f += g.f;
}
```

在 C++ 中，我们可以有一个带有加法运算符的类，如下所示：

```
// molecular.cxx
class force {
private:
  double _x{0.},_y{0.}; double _f{0.};
public:
  force() {};
  force(double x,double y,double f)
    : _x(x),_y(y),_f(f) {};
```

```
force operator+( const force& g ) {
  return { _x+g._x, _y+g._y, _f+g._f };
}
```

供参考，以下是顺序代码：

```
for (int ip=0; ip<N; ip++) {
  for (int jp=ip+1; jp<N; jp++) {
    struct force f = force_calc(points[ip],points[jp]);
    add_force( forces+ip,f );
    sub_force( forces+jp,f );
  }
}
```

这里 $\vec{F}_{ij}$ 仅为 $j > i$ 计算，然后加到 $\vec{F}_i$ 和 $\vec{F}_j$ 上。

In C++ we use the overloaded operators:

```
for (int ip=0; ip<N; ip++) {
  for (int jp=ip+1; jp<N; jp++) {
    force f = points[ip].force_calc(points[jp]);
    forces[ip] += f;
    forces[jp] -= f;
  }
}
```

Exercise 30.1. 论证外层循环和内层循环都不能直接并行化。

我们现在将探讨多种不同的并行化策略。所有测试均在 *TACCFrontera* 集群上进行，该集群配备双插槽 *Intel Cascade Lake* 节点，总共有 56 个核心。我们的代码使用了一万个粒子，每次相互作用评估重复 10 次，以消除缓存加载效应。

30.1.1 方案 1：无冲突写入

在我们首次尝试高效并行代码时，我们计算了完整的 N^2 相互作用。一种解决方案是计算所有 i, j 的 $\vec{F}_{ij}$ 相互作用，以避免冲突写入。

```
for (int ip=0; ip<N; ip++) {
        struct force sumforce;
        sumforce.x=0.; sumforce.y=0.; sumforce.f=0.;
#pragma omp parallel for reduction(+:sumforce)
  for (int jp=0; jp<N; jp++) {
```

```
    if (ip==jp) continue;
    struct force f = force_calc(points[ip],points[jp]);
    sumforce.x += f.x; sumforce.y += f.y; sumforce.f += f.f;
  } // end parallel jp loop
  add_force( forces+ip, sumforce );
} // end ip loop
```

在 C++ 中，我们利用了可以对任何具有加法运算符的类进行归约的事实：

```
for (int ip=0; ip<N; ip++) {
  force sumforce;
  #pragma omp parallel for reduction(+:sumforce)
  for (int jp=0; jp<N; jp++) {
    if (ip==jp) continue;
    force f = points[ip].force_calc(points[jp]);
    sumforce += f;
  } // end parallel jp loop
  forces[ip] += sumforce;
} // end ip loop
```

这使得标量工作量增加了两倍，但令人惊讶的是，在单线程上运行时间有所改善：我们测得相对于所谓的 “ 最优 ” 代码，速度提升了 6.51 倍。

Exercise 30.2. 这会有什么解释?

然而，增加线程数对该策略的收益有限。图 30.1 显示加速比不仅是次线性的：实际上随着核心数的增加而下降。

Exercise 30.3. 这会有什么解释?

30.1.2 解决方案 2：使用原子操作

接下来我们尝试并行化外层循环。

```
#pragma omp parallel for schedule(guided,4)
      for (int ip=0; ip<N; ip++) {
        for (int jp=ip+1; jp<N; jp++) {
          struct force f = force_calc(points[ip],points[jp]);
          add_force( forces+ip,f );
          sub_force( forces+jp,f );
        }
      }
```

为了解决冲突的 jp 写入，我们使写入操作变为原子操作：

```
void sub_force( struct force *f,struct force g ) {
#pragma omp atomic
  f->x -= g.x;
#pragma omp atomic
  f->y -= g.y;
#pragma omp atomic
  f->f += g.f;
}
```

这效果相当不错，如图 30.2 所示。

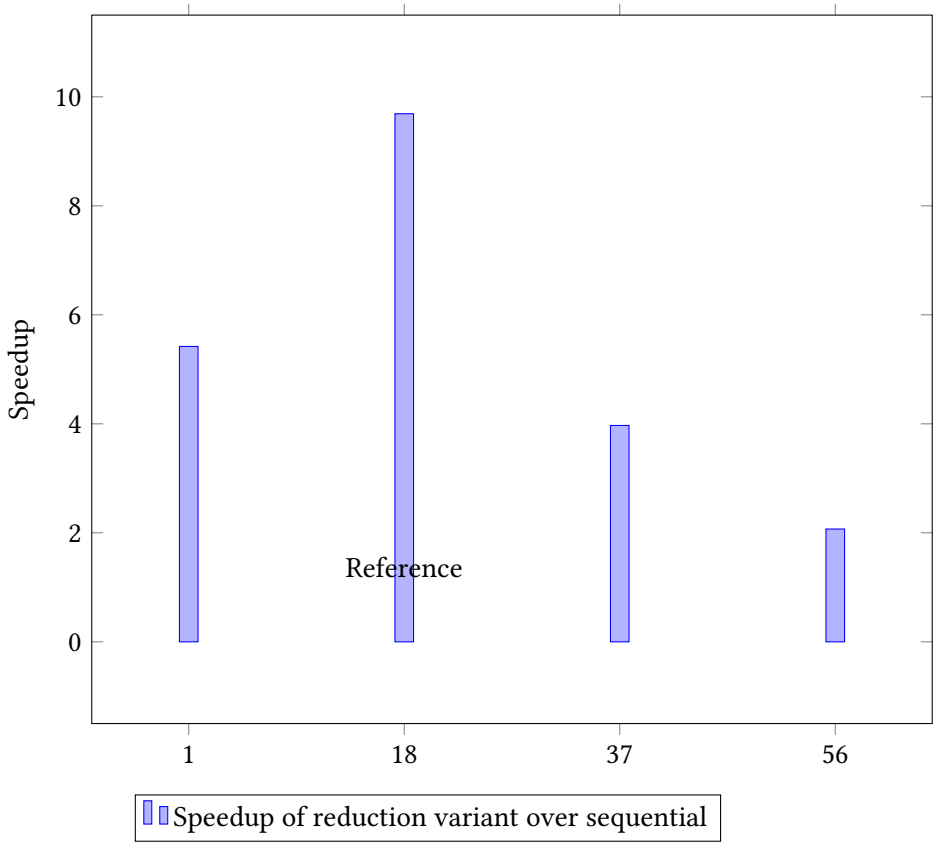


Figure 30.1: reduction 变体相对于顺序的加速比

30.1.3 Solution 3: all interactions atomic

但如果我们决定使用 atomic 更新，我们可以将完整的平方循环，折叠两个循环，并使每次写入都是 atomic 的。

```
#pragma omp parallel for collapse(2)
      for (int ip=0; ip<N; ip++) {
        for (int jp=0; jp<N; jp++) {
          if (ip==jp) continue;
          struct force f = force_calc(points[ip],points[jp]);
          add_force( forces+ip, f );
        } // end parallel jp loop
      } // end ip loop
```

Figure 30.3 显示这非常接近完美。

Everything in one plot in figure 30.4.

30.2 Tree traversal

OpenMP tasks 是处理树结构的绝佳方式。

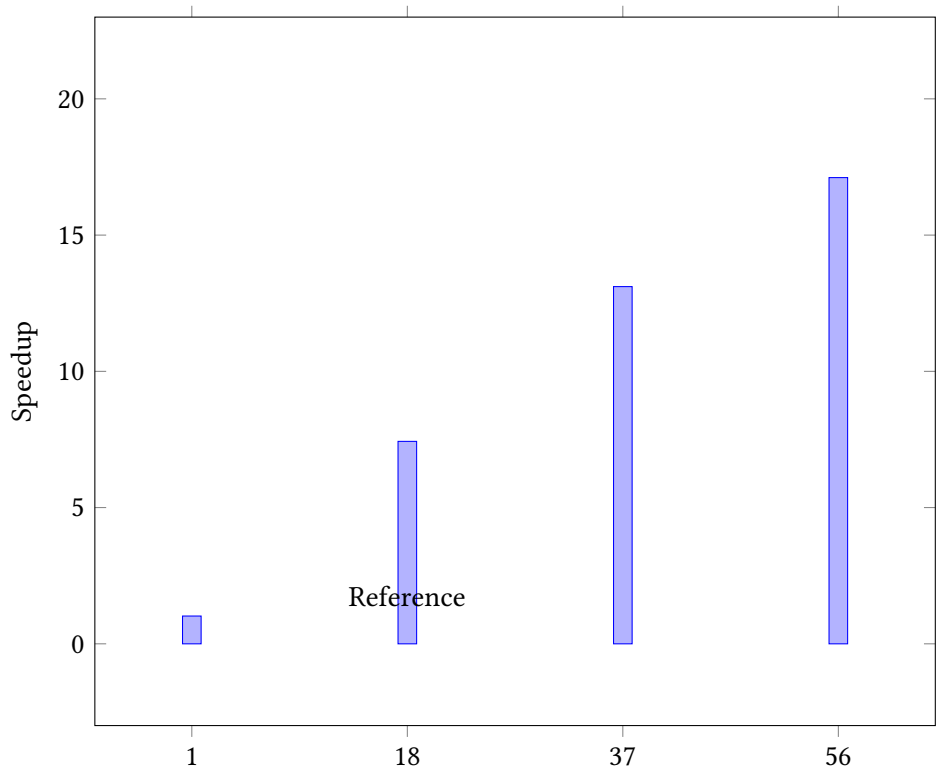


图 30.2：带原子更新的三角循环加速比

在后序树遍历中，你会先访问子树，然后再访问根节点。这种遍历用于查找树的汇总信息，例如所有节点的和，以及所有子树节点的和：

for 所有子节点 c **do** 计算和 s_c

$s \leftarrow \sum_c s_c$

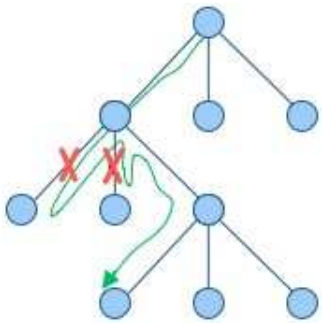
另一个例子是矩阵分解：

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

其中两个逆 A_{11}^{-1}, A_{22}^{-1}可以独立且递归地计算。

30.3 深度优先搜索

在本节中，我们以 “ 八皇后 ” 问题为例，介绍 *Depth First Search(DFS)*：是否有可能将八个皇后放置在棋盘上，使得它们互不威胁？使用 DFS，搜索的可能空间是



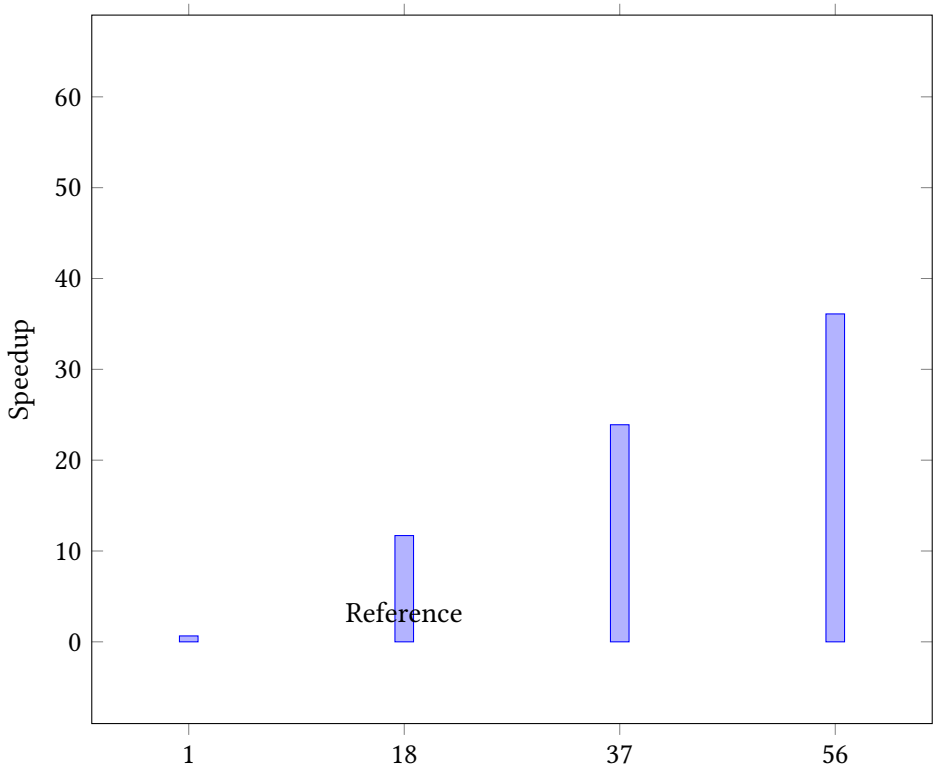


图 30.3：原子全交互计算的加速比

组织成一棵树 —— 每个部分解都会导致下一步的多个可能性 —— 以特定方式遍历：一条可能性的链条尽可能延伸，之后搜索回溯到下一条链。

顺序实现相当简单。主程序启动：

```
placement initial; initial.fill(empty);
auto solution = place_queen(0,initial);
```

我希望你能信任细节部分。

The recursive call then has this structure:

```
optional<placement> place_queen(int iqueen,const
↪placement& current) {
  for (int col=0; col<N; col++) {
    placement next = current;
    next.at(iqueen) = col;
    if (feasible(next)) {
      if (iqueen==N-1)
        return next;
      auto attempt = place_queen(iqueen+1,next);
```

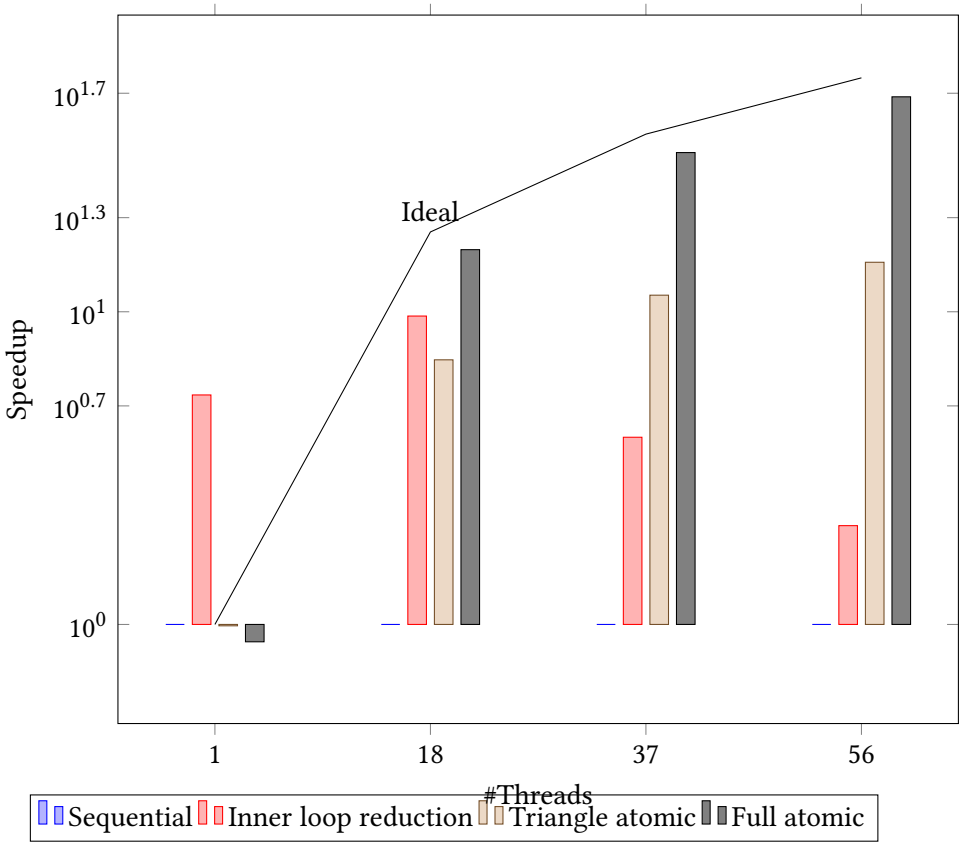


图 30.4：所有策略汇总

```
      if (attempt.has_value())
        return attempt;
    } // end if(feasible)
  }
  return {};
};
```

(This uses the C++17 *optional* header.) At each *iqueen* level we

- 遍历所有列位置的循环；
- 过滤掉不可行的位置；
- 如果这是最后一层，则报告成功；或者
- 否则递归继续下一层。

这个问题似乎是 OpenMP 任务的理想候选，因此我们从主程序的常用惯用法开始：

```
placement initial; initial.fill(empty);
optional<placement> eightqueens;
#pragma omp parallel
#pragma omp single
eightqueens = place_queen(0,initial);
```

我们为每一列创建一个任务，由于它们在循环中，我们使用 `taskgroup` 而不是 `taskwait`。

```
#pragma omp taskgroup
  for (int col=0; col<N; col++) {
    placement next = current;
    next.at(iqueen) = col;
#pragma omp task firstprivate(next)
    if (feasible(next)) {
    // stuff
    } // end if(feasible)
  }
```

然而，顺序程序在循环中有 `return` 和 `break` 语句，这在诸如 `taskgroup` 之类的工作共享结构中是不允许的。因此我们引入一个返回变量，声明为共享：

```
// queens0.cxx
optional<placement> result = {};
#pragma omp taskgroup
for (int col=0; col<N; col++) {
  placement next = current;
  next.at(iqueen) = col;
  #pragma omp task firstprivate(next) shared(result)
  if (feasible(next)) {
    if (iqueen==N-1) {
        result = next;
    } else { // do next level
        auto attempt = place_queen(iqueen+1,next);
        if (attempt.has_value()) {
          result = attempt;
        }
    } // end if(iqueen==N-1)
  } // end if(feasible)
}
return result;
```

这很简单，这计算了正确的解，并且使用了 OpenMP 任务。完成了吗?

A实际上这运行得很慢，因为现在我们已经取消了循环中的所有提前跳出，实际上遍历整个搜索树。（不过这并不完全是广度优先。）图 30.5 展示了 $N = 12$，左侧面板为 Intel 编译器（版本 2019），中间为 GNU 编译器（版本 9.1）。在两种情况下，蓝色条表示仅使用 `taskgroup` 指令的代码结果，时间以核心数为函数绘制。

我们看到，对于 Intel 编译器，运行时间确实随着核心数的增加而下降。因此，虽然我们计算过多（整个搜索空间），但至少并行化是有帮助的。当线程数超过问题规模时，并行化的好处消失了，这在某种程度上是合理的。

我们还看到 GCC 编译器在 OpenMP 任务上表现非常差：运行时间实际上随着线程数的增加而增加。

Fortunately, with 在 OpenMP-4 中，我们可以用一个 *cancel* 从循环中跳出 the task group:

```
// queenfinal.cxx
```

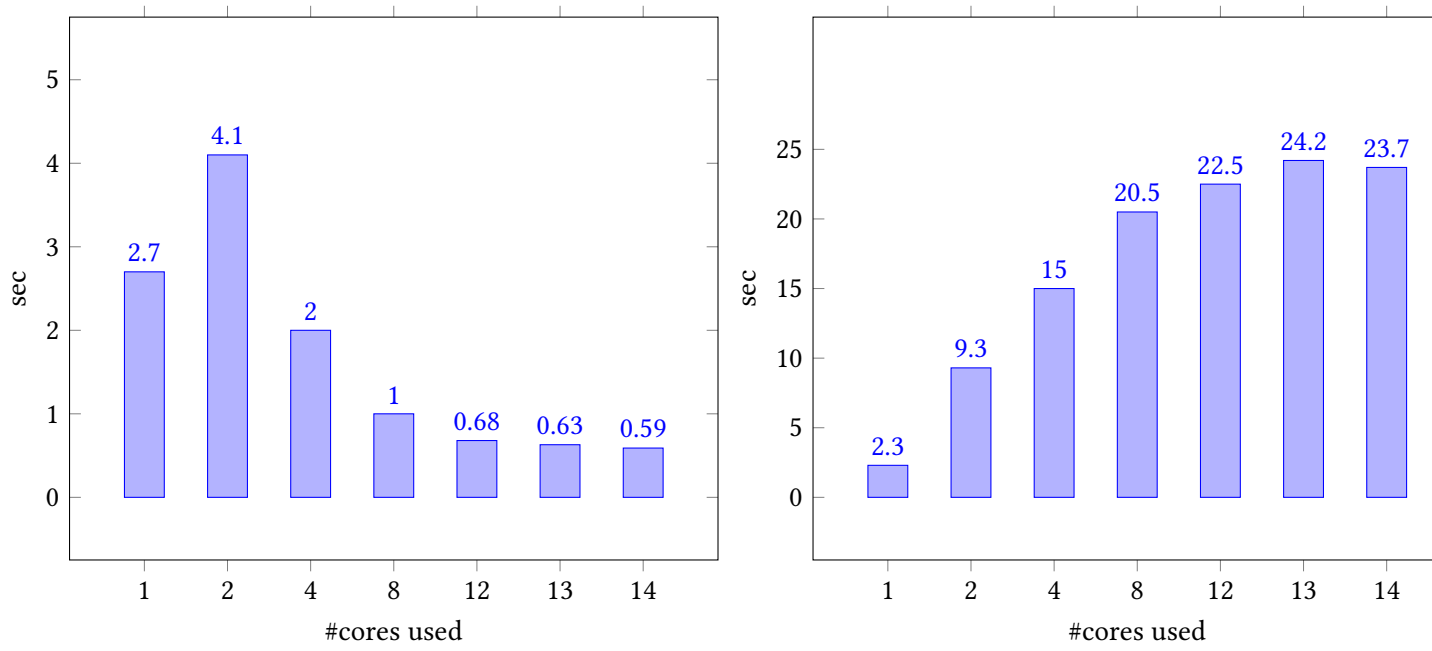


图 30.5：使用 taskgroups 进行 $N = 12$；左侧为 Intel 编译器，右侧为 GCC

```
if (feasible(next)) {
  if (iqueen==N-1) {
        result = next;
    #pragma omp cancel taskgroup
  } else { // do next level
        auto attempt = place_queen(iqueen+1,next);
        if (attempt.has_value()) {
          result = attempt;
      #pragma omp cancel taskgroup
        }
  } // end if (iqueen==N-1)
} // end if (feasible)
```

令人惊讶的是，这并不会立即带来性能提升。原因是取消操作默认被禁用，我们必须设置环境变量

```
OMP_CANCELLATION=true
```

通过设置后，我们获得了非常好的性能，如图 30.6 所示，该图列出了带有 `cancel` 指令的代码的顺序执行时间和多核运行时间。运行时间现在大致与顺序执行时间相同。仍有一些问题：

- 为什么随着核心数增加，时间反而变长?
- 为什么多核代码比顺序代码慢，如果标量工作量（例如在 `feasible` 函数中）更大，是否并行代码会比顺序代码更快?

这里没有报告的一个观察是，GNU 编译器在有无取消的情况下运行时间基本相同。这再次表明 GNU 编译器在处理 OpenMP 任务时表现非常差。

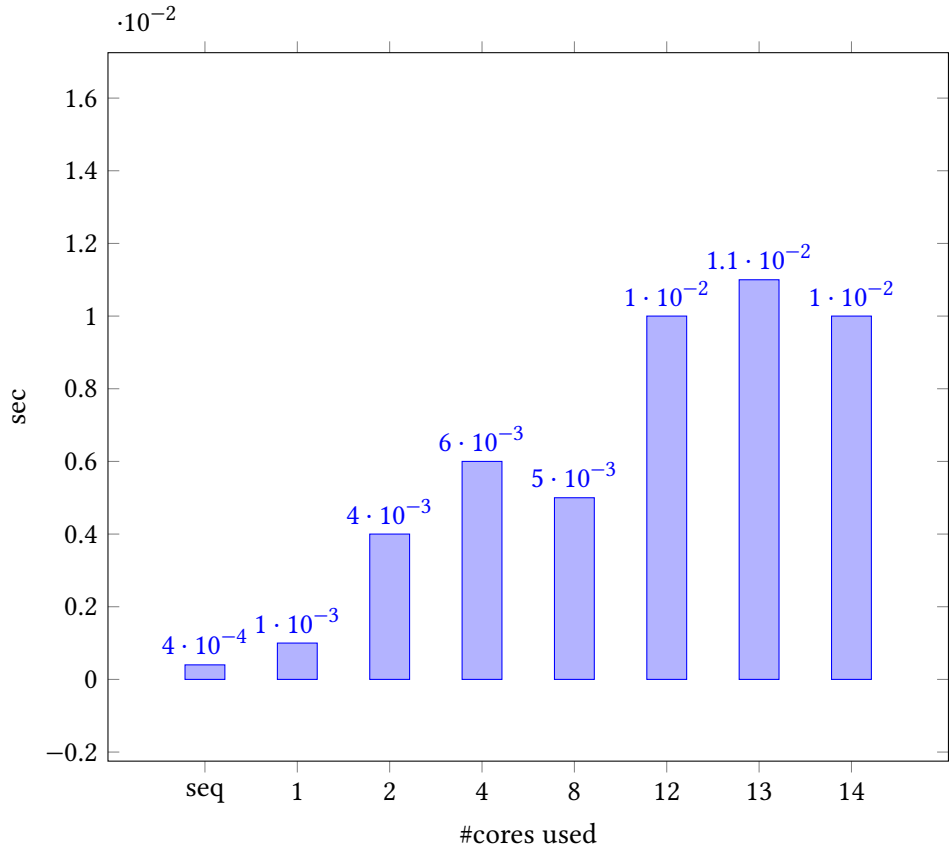


图 30.6：使用 taskgroup 取消，Intel 编译器

30.4 过滤数组元素

假设我们有一个元素数组（为了论证方便，假设是整数），我们想构造一个仅包含满足 sometest 的元素的子数组

```
bool f(int);
```

C++ 注 26：列表过滤示例。我们将仅在 C++ 中做这个示例，因为它处理起来更方便 `std::vector`。

顺序代码如下：

```
vector<int> data(100);
// fil the data
vector<int> filtered;
for ( auto e : data ) {
  if ( f(e) )
    filtered.push_back(e);
}
```

这里有两个问题。第一个是过滤数组上的竞争条件。即使我们用临界区解决了这个问题，插入元素的顺序仍然缺失。

解决的关键是让每个线程拥有一个本地数组，然后将它们连接起来：

```
#pragma omp parallel
{
  vector<int> local;
# pragma omp for
  for ( auto e : data ) {
    if ( f(e) ) {
      local.push_back(e);
    }
  }
  filtered += local;
}
```

其中我们对向量使用了追加操作：

```
// filterreduct.cxx
template<typename T>
vector<T>& operator+=( vector<T>& me, const vector<T>& other ) {
  me.insert( me.end(),other.begin(),other.end() );
  return me;
};
```

30.4.1 尝试 1：reduction

我们可以使用 plus-is 操作来声明一个 reduction：

```
#pragma omp declare reduction\
  (     \
        +:vector<int>:omp_out += omp_in \
        ) \
  initializer( omp_priv = vector<int>{} )
```

这里的问题是 OpenMP 归约不能声明为非交换的，因此来自
线程的输出可能不会按顺序出现。**代码：输出：**

```
#pragma omp parallel \
  reduction(+ : filtered)
  {
    vector<int> local;
#   pragma omp for
    for ( auto e : data )
      if ( f(e) )
        local.push_back(e);
    filtered += local;
  }
```

```
Mod 5: 80 85 90 95 100 5 10 15 20 25
    ↪30 35 40 45 50 55 60 65 70 75
```

30.4.2 尝试 2: sequentialappending

如果我们不使用归约，而是显式地进行追加，首先必须将此操作置于临界区。其次，我们必须强制正确的顺序。

这里尝试通过保持一个共享计数器来实现：

Code:

```
// filteratomic.cxx
#   pragma omp critical
    if (threadnum==ithread) {
      filtered += local;
      ithread++;
```

Output:

```
Mod 5: 5 10 15 20 25 30 35 40 45 50
```

这里的问题是，决定不是自己顺序的线程会直接跳过追加操作：没有办法让它们等待轮到自己。（你可以尝试用一个 while 循环。试试看。）

最好的解决方案是使用完全不同的机制。

30.4.3 尝试 3：使用 tasks

使用 task 后，可以实现自旋等待循环 :

代码:

```
#   pragma omp task \
      shared(filtered,ithread)
    {
// wait your turn
      while (threadnum>ithread) {
#       pragma omp taskyield
      }
// merge
      filtered += local;
      ithread++;
    }
```

输出 :

```
Mod 5: 5 10 15 20 25 30 35 40 45 50 55
    ↪60 65 70 75 80 85 90 95 100
```

30.5 线程同步

让我们做一个 生产者 - 消费者 模型 [1]。这可以通过 sections 实现，其中一个 section，生产者，在数据可用时设置一个标志，另一个，消费者，等待直到标志被设置。

```
#pragma omp parallel sections
{
  // the producer
  #pragma omp section
  {
    ... do some producing work ...
    flag = 1;
  }
  // the consumer
```

1. 这个例子来自 Intel 的优秀 OMP 课程，由 Tim Mattson 讲授

```
  #pragma omp section
  {
    while (flag==0) { }
    ... do some consuming work ...
  }
}
```

这不起作用的一个原因是，编译器会看到标志在生产部分从未被使用，并且在消费部分也从未被更改，因此它可能会对这些语句进行优化，甚至优化到将其完全去除的程度。

然后生产者需要执行：

```
... do some producing work ...
#pragma omp flush
#pragma atomic write
  flag = 1;
#pragma omp flush(flag)
```

而消费者执行：

```
#pragma omp flush(flag)
while (flag==0) {
  #pragma omp flush(flag)
}
#pragma omp flush
```

严格来说，这段代码在 `flag` 变量上存在竞态条件。

T解决方案是将其变成一个原子操作，并在这里使用一个 *atomic* pragma：生产者有

```
#pragma atomic write
  flag = 1;
```

消费者则是：

```
while (1) {
  #pragma omp flush(flag)
  #pragma omp atomic read
    flag_read = flag
  if (flag_read==1) break;
}
```

30. OpenMP 示例

第三部分

PETSC

第 31 章

PETSc 基础

31.1 What is PETSc and why?

PETSc 是一个用途广泛的库，但目前我们先说它主要是一个处理来自离散化 PDE 的线性代数的库。在单个处理器上，这类计算的基础内容可以由研究生在数值分析的一个学期课程中编写出来，但在大规模问题上情况变得更加复杂，库变得不可或缺。

PETSc 的主要理由是它帮助你实现大规模的科学计算，也就是说，在大量处理器上处理大规模问题。

这里有两点需要强调：

- 使用稠密矩阵的线性代数相对容易表述。对于稀疏矩阵，处理非零模式的后勤工作大大增加。PETSc 为你完成了大部分这类工作。
- 单处理器上的线性代数，即使是多核处理器，也比较容易处理；分布式内存并行则要困难得多，而分布式内存稀疏线性代数操作则更加困难。使用 PETSc 将为你节省大量、非常多、非常非常多！的编码时间，而不必从零开始开发所有内容。

Remark 40 *PETSc* 库包含数百个例程。在本章及接下来的几章中，我们只会涉及其中的基本子集。完整的 *man* 页面列表可以在 *https://petsc.org/release/docs/manualpages/singleindex.html* 找到。每个 *man* 页面都附带相关例程的链接，以及（通常）该例程的示例代码。

31.1.1 什么是 PETSc?

PETSc 中的例程（数以百计）大致可以分为以下几类：

- 基本线性代数工具：稠密和稀疏矩阵，既有顺序的也有并行的，及其构造和简单操作。
- 线性系统求解器，以及在较小程度上非线性系统求解器；还有时间步进方法。
- 性能分析和追踪：成功运行后，可以给出各种例程的时间。在失败的情况下，有回溯和内存追踪功能。

31.1.2 编程模型

PETSc 基于 MPI，使用 SPMD 编程模型（章节 2.1 ），其中所有进程执行相同的可执行文件。这里比常规 MPI 代码更有意义，因为大多数 PETSc 对象都是在某个通信器上集体创建的，通常是 `MPI_COMM_WORLD`。采用面向对象设计（章节 31.1.3 ）意味着 PETSc 程序几乎看起来像一个顺序程序。

```
MatMult(A,x,y);      // y <- Ax
VecCopy(y,res);      // r <- y
VecAXPY(res,-1.,b);  // r <- r - b
```

This is sometimes called *sequential semantics*.

31.1.3 设计理念

PETSc 采用面向对象设计，尽管它是用 C 语言编写的。存在对象的类，例如 `Mat` 用于矩阵的类和 `Vec` 用于向量的类，但也有 `KSP`（表示 “Krylov 空间求解器”）的线性系统求解器类，和 `PetscViewer` 用于将矩阵和向量输出到屏幕或文件的类。

面向对象设计的一部分是对象的多态性：在你创建了一个 `Mat` 矩阵作为稀疏或稠密矩阵后，所有方法如 MatMult （用于矩阵 - 向量乘法）都接受相同的参数：矩阵，以及输入和输出向量。

这种设计中程序员操作的是一个 “句柄”，这也意味着对象的内部，即元素的实际存储，对程序员是隐藏的。这种隐藏甚至延伸到元素的填充也不是直接完成的，而是通过函数调用来实现：

```
VecSetValue(i,j,v,mode)
MatSetValue(i,j,v,mode)
MatSetValues(ni,is,nj,js,v,mode)
```

31.1.4 语言支持

31.1.4.1 C/C++

PETSc 是用 C 实现的，因此有一个自然的 C 接口。没有单独的 C++ 接口。

31.1.4.2 Fortran

A *Fortran90* 接口存在。*Fortran77* 接口仅限于 i 兴趣 for historical reasons.

要使用 Fortran，需要同时包含一个模块和一个 cpp 头文件：

```
#include "petsc/finclude/petscXXX.h"
use petscXXX
```

（这里 `XXX` 代表 PETSc 类型之一，但包含 `petsc.h` 并使用 `use petsc` 会包含整个库。）

变量可以用它们的类型声明（`Vec`, `Mat`, `KSP` 等等），但它们内部是 Fortran `Type` 对象，因此也可以这样声明。

示例：

```
#include "petsc/finclude/petscvec.h"
use petscvec
Vec b
type(tVec) x
```

PETSc 中许多查询例程的输出参数是可选的。虽然在 C 中可以传递通用的 `NULL`，Fortran 有类型特定的空值，例如 `PETSC_NULL_INTEGER`, `PETSC_NULL_OBJECT`。

31.1.4.3 Python

A *python* 接口由 *Lisandro Dalcin* 编写。它可以在安装时添加到 PETSc 中；详见第 31.3 节。

This book discusses the Python interface in short remarks in the appropriate sections.

31.1.5 Documentation

PETSc 附带有 PDF 格式的手册和包含每个例程文档的网页。起点是网页
https://petsc.org/release/documentation/。

There is also a mailing list with excellent support for questions and bug reports.

TACC note. 关于在 TACC 资源上使用 PETSc 的具体问题，请向 *TACC* 或 *XSEDEportal* 提交工单。

31.2 运行 PETSc 程序的基础

31.2.1 编译

PETSc 的编译需要多个包含路径和库路径，可能多到无法交互式指定。最简单的解决方案是创建一个 makefile 并加载标准变量和编译规则。（你可以使用 `$PETSC_DIR/share/petsc/Makefile.user` 作为参考。）

在整个过程中，我们假设变量 `PETSC_DIR` 和 `PETSC_ARCH` 已经被设置。这些取决于你的本地安装；参见第 31.3 节。

在最简单的设置中，你将编译工作交给 PETSc，make 规则只做链接步骤，分别使用 `CLINKER` 或 `FLINKER` 进行 C/Fortran:

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
program : program.o
        ${CLINKER} -o $@ $^ ${PETSC_LIB}
```

这两个 include 行提供了编译规则和库变量。

您可以使用这些规则：`% : %.F90`

`$(LINK.F) -o $@ $^ $(LDLIBS)%.o: %.F90`

`$(COMPILE.F) $(OUTPUT_OPTION) $<% : %.cxx`

`$(LINK.cc) -o $@ $^ $(LDLIBS)%.o: %.cxx`

`$(COMPILE.cc) $(OUTPUT_OPTION) $<`

```
## example link rule:
# app : a.o b.o c.o
#       $(LINK.F) -o $@ $^ $(LDLIBS)
```

(该 `PETSC_CC_INCLUDES` 变量包含所有 C 程序的编译路径；相应地，对于 Fortran 源代码有 `PETSC_FC_INCLUDES` 。)

如果不想包含那些配置文件，可以通过以下方式找到包含选项：

```
cd $PETSC_DIR
make getincludedirs
make getlinklibs
```

并将结果复制到您的编译脚本中。

这里有一个示例 makefile `$PETSC_DIR/share/petsc/Makefile.user`，您可以参考。无参数调用时，它会打印出相关变量：

```
[c:246] make -f ! $PETSC_DIR/share/petsc/Makefile.user
CC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicc
CXX=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicxx
FC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpif90
CFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -Qunused-argum
CXXFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -fvisibility
FFLAGS=-m64 -g
CPPFLAGS=-I/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/include -I/Users/eijkhout/Ins
LDFLAGS=-L/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/lib -Wl,-rpath,/Users/eijkhout
LDLIBS=-lpetsc -lm
```

TACC 注释。在 TACC 集群上，petsc 安装通过如下命令加载

`module load petsc/3.16` 使用 `module avail petsc` 查看存在哪些配置。基本版本有

```
# developmentmodule load petsc/3.11-debug# productionmodule load petsc/3.11
```

其他安装版本是真实数与复数，或者是 64 位整数而非默认的 32 位。命令

```
module spider petsc
```

告诉你所有可用的 petsc 版本。列出的模块有命名约定，例如 `petsc/3.11-i64debug`，其中 3.11 是 PETSc 发布版本（此版本不包括小补丁；TACC 旨在只安装最新补丁，但通常有多个版本可用），而 `i64debug` 描述了使用 64 位整数的调试版本安装。

31.2.2 运行

PETSc 程序使用 MPI 进行并行，因此它们的启动方式与其他 MPI 程序相同：

```
mpiexec -n 5 -machinefile mf \
    your_petsc_program option1 option2 option3
```

TACCnote. 在 TACC 集群上，使用 `ibrun`。

31.2.3 初始化和终结

PETSc 有一个调用可以初始化 PETSc 和 MPI，因此通常你会用 `PetscInitialize`(图 31.1) 替换 `MPI_Init`。与 MPI 不同，你不想为 `argc,argv` 参数使用 `NULL` 值，因为 PETSc 广泛使用命令行选项；参见第 38.3 节。

```
// init.c
PetscCall( PetscInitialize
  (&argc,&argv,(char*)0,help) );
```

Figure 31.1 **PetscInitialize**

```
C:
PetscErrorCode  PetscInitialize
   (int *argc,char ***args,const char file[],const char help[])

Input Parameters:
argc - count of number of command line arguments
args - the command line argumentsfile - [optional] PETSc database file.
help - [optional] Help message to print, use NULL for no message


Fortran:
call PetscInitialize(file,ierr)

Input parameters:ierr - error return code
file - [optional] PETSc database file,
use PETSC_NULL_CHARACTER to not check for code specific file.
```

```c
int flag;
MPI_Initialized(&flag);
if (flag)
  printf("MPI was initialized by PETSc\n");
else
  printf("MPI not yet initialized\n");
```

There are two further arguments to `PetscInitialize`:

1. 一个选项数据库文件的名称；以及 2. 一个帮助字符串，当你使用 `-h` 选项运行程序时会显示该字符串。

Fortran note 26: Petsc Initialization. Fortran 版本没有命令行选项参数；但是，你可以传递一个数据库选项文件：

```
PetscInitialize(filename,ierr)
```

如果未指定，则给出 `PETSC_NULL_CHARACTER` 作为参数。

对于传递帮助信息，有一个变体接受帮助字符串：

Code:

```
!! mainhelp.F90
  Character(len=50) :: help = "This program
    ↪demonstrates help info"
  help = trim(help) // NEW_LINE('A')
  call PetscInitialize(PETSC_NULL_CHARACTER,help,ierr)
  CHKERRA(ierr)
```

输出：

```
This program demonstrates help info
```

如果你的主程序是用 C 写的，但你的一些 PETSc 调用是在 Fortran 文件中，则有必要在 `PetscInitialize` 之后调用 `PetscInitializeFortran`。

```
!! init.F90
  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
```

```
  CHKERRA(ierr)
  call MPI_Initialized(flag,ierr)
  CHKERRA(ierr)
  if (flag) then
  print *,"MPI      was initialized by PETSc"
```

Python note 37: Init, and with commandline options. 如果不需要命令行选项，以下方法可行。

```
from petsc4py import PETSc
```

要向 PETSc 传递命令行参数，请执行：

```
import sys
from petsc4py import init
init(sys.argv)
from petsc4py import PETSc
```

初始化后，可以使用 `MPI_COMM_WORLD` 或 `PETSC_COMM_WORLD`（由 `MPI_Comm_dup` 创建并由 PETSc 内部使用）：

```
MPI_Comm comm = PETSC_COMM_WORLD;
MPI_Comm_rank(comm,&mytid);
MPI_Comm_size(comm,&ntids);
```

Python 注释 *38*：*Communicator* 对象。

```
comm = PETSc.COMM_WORLD
nprocs = comm.getSize(self)
procno = comm.getRank(self)
```

对应替换 `MPI_Finalize` 的调用是 `PetscFinalize`。你可以优雅地捕获并返回错误 r
代码通过惯用法

```
return PetscFinalize();
```

在你的主程序末尾。

31.3 PETSc 安装

PETSc has a la大量的安装选项。这些大致可以分为 vided into:

1. 用于描述安装 PETSc 所在环境的选项，例如编译器名称或 MPI 库的位置；2. 用于指定 PETSc 安装类型的选项：实数与复数，32 位与 64 位整数，等等；3. 用于指定额外下载包的选项。

对于已有的安装，你可以在 `configure.log`/ `make.log` 文件中找到使用的选项及构建历史的其他方面：

```
$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/configure.log
$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/make.log
```

31.3.1 版本

PETSc 截至本文撰写时的版本为 `3.18.x`。较旧的版本可能缺少某些例程，或存在某些错误。
H然而，较旧的版本也可能包含后来被移除的例程和关键字。PETSc
v版本不向后兼容！

版本存储在宏 *PETSC_VERSION*、 *PETSC_VERSION_MAJOR*、 *PETSC_VERSION_MINOR*、 *PETSC_VERSION_SUBMINOR* 中。

用于测试，定义了以下宏： *PETSC_VERSION_EQ/LT/LE/GT/GE* Exampl e:

```
// cudainit316.c
#include <petsc.h>
#if PETSC_VERSION_LT(3,17,0)
#else
#error This program uses APIs abandoned in 3.17
#endif
```

31.3.2 调试

对于任何一组选项，您通常需要进行两次安装：一次使用 `-with-debugging=yes`，另一次使用 `no`。有关调试模式和非调试模式之间差异的更多细节，请参见第 38.1.1 节。

31.3.3 环境选项

编译器，编译选项，MPI。

虽然可以指定 *-download_mpich*，但这应仅在您确定机器上尚未安装 MPI 库的情况下进行，例如您的个人笔记本电脑。超级计算机集群很可能已经有优化的 MPI 库，让 PETSc 自行下载会导致性能下降。

31.3.4 变体

- 标量：选项 *-with-scalar-type* 有值 `real`, `complex`； *-with-precision* 有值 `single`,`double`, `__float128`, `__fp16`。

31.4 外部包

PETSc 可以通过外部包如 *mumps*、 *Hypre*、 *fftw* 来扩展其功能。这些可以通过两种方式指定：

1. 引用系统中已安装的版本：
`--with-hdf5-include=${TACC_HDF5_INC}`
`--with-hf5_lib=${TACC_HDF5_LIB}`2. 让 petsc 自行下载
并安装：`--with-parmetis=1 --download-parmetis=1`

Python 注 *39*：*petsc4py* 接口。Python 接口（章节 31.1.4.3）可以通过选项安装
`--download-petsc4py=<no,yes,filename,url>`
如果你的 python 已经包含 `mpi4py`，这将是最简单的；参见第 1.5.4 节。

Remark 41 *PETSc* 能够下载和安装两个包，但你可能想要避免使用它们：

- `fblaslapack`: 这为你提供了通过 *Fortran* “ 参考实现 ” 的 *BLAS/LAPACK*。如果你有优化版本，比如 *Intel* 的 mkl，这将带来更高的性能。

- `mpich`: 这会安装一个 *MPI* 实现，可能是你笔记本所需的。然而，超级计算机集群已经有了使用高速网络的 *MPI* 实现。*PETSc* 下载的版本不具备这一点。同样，找到并使用已安装的软件可能大大提升你的性能。

31.4.1 Slepc

大多数外部包为 Petsc 的底层增加功能。例如，*Hypre* 包为 Petsc 的工具库添加了一些预处理器（章节 35.1.7.3 ），而 *Mumps* (章节 35.2) 使得在并行环境中使用 LU 预处理器成为可能。

另一方面，有些包使用 Petsc 作为底层工具。特别是，特征值求解器包 *Slepc* [28] 可以通过选项安装

```
--download-slepc=<no,yes,filename,url>Download and install slepc  current: no
--download-slepc-commit=commitid
The commit id from a git repository to use for the build of slepc  current: 0
--download-slepc-configure-arguments=string
Additional configure arguments for the build of SLEPc
```

slepc 头文件最终会放在与 petsc 头文件相同的目录中，因此不需要更改您的编译规则。但是，您需要在链接行中添加 `-lslepc` 。

第 32 章

PETSc 对象

32.1 分布式对象

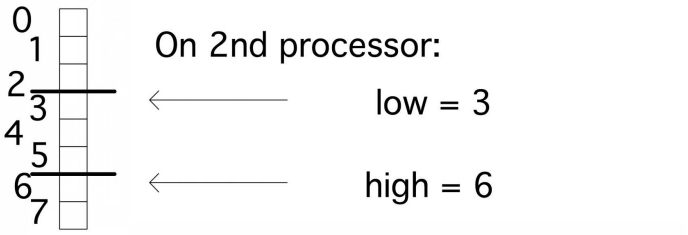
PETSc 基于 SPMD 模型，其所有对象表现得像是并行存在，分布在所有进程中。因此，在详细讨论具体对象之前，我们简要讨论 PETSc 如何处理分布式对象。

对于矩阵或向量，您需要指定大小。这可以通过两种方式完成：

- 您指定全局大小，PETSc 将对象分布到各个进程，或者
- 你在每个进程上指定本地大小

If you specify both the global size and the local sizes, PETSc will check for consistency.

例如，如果你有一个包含 N 组件的向量，或者一个有 N 行的矩阵，并且你有 P 个进程，如果 P 能被 N 整除，每个进程将接收 N/P 个组件或行。如果 P 不能被整除，多余的部分会分散到各个进程。



分布的方式是通过连续块实现的：对于 10 个进程和一个包含 1000 个组件的向量，进程 0 获得范围 $0 \cdots 99$，进程 1 获得 $1 \cdots 199$，依此类推。这种简单的方案适用于许多情况，但 PETSc 提供了更复杂的负载均衡功能。

32.1.1 对分布的支持

一旦对象被创建并分布，你不需要自己记住大小或分布：你可以通过调用诸如 `VecGetSize`, `VecGetLocalSize` 的查询来获取这些信息。

对应的矩阵例程 `MatGetSize`, `MatGetLocalSize` 提供了 i 和 j 方向上分布的相关信息，这两个方向可以是独立的。由于矩阵是按行分布的， `MatGetOwnershipRange` 只给出行范围。

Figure 32.1 **PetscSplitOwnership**

```
PetscSplitOwnership

Synopsis
#include "petscsys.h"
PetscErrorCode  PetscSplitOwnership
   (MPI_Comm comm,PetscInt *n,PetscInt *N)

Collective (if n or N is PETSC_DECIDE)

Input Parameters
comm - MPI communicator that shares the object being divided
n - local length (or PETSC_DECIDE to have it set)
N - global length (or PETSC_DECIDE)
```

```
// split.c
N = 100; n = PETSC_DECIDE;
PetscSplitOwnership(comm,&n,&N);
PetscPrintf(comm,"Global %d, local %d\n",N,n);

N = PETSC_DECIDE; n = 10;
PetscSplitOwnership(comm,&n,&N);
PetscPrintf(comm,"Global %d, local %d\n",N,n);
```

虽然 PETSc 对象是使用每个进程上的本地内存实现的，但从概念上它们表现得像全局对象，具有全局索引方案。因此，每个进程可以查询全局对象中哪些元素是本地存储的。对于向量，相关的例程是 `VecGetOwnershipRange`，它返回两个参数，*low* 和 *high*，分别是存储的第一个元素索引和最后一个元素索引加一。

这给出了惯用法：

```
VecGetOwnershipRange(myvector,&low,&high);
for (int myidx=low; myidx<high; myidx++)
  // do something at index myidx
```

这些局部和全局大小之间的转换也可以显式完成，使用 `PetscSplitOwnership`（图 32.1）例程。该例程接受两个参数，分别是局部和全局大小，初始化为 `PETSC_DECIDE` <style id='11'} 的那个参数将从另一个参数计算得出。

32.2 标量

与明确区分单精度和双精度数的编程语言不同，PETSc 只有一种标量类型：`PetscScalar`。其精度在安装时确定。事实上，如果安装时指定标量类型为复数，`PetscScalar` 甚至可以是复数。

即使在使用复数的应用中，也可能存在实数的量：例如，复向量的范数是一个实数。出于这个原因，PETSc 也有类型 `PetscReal`。还有一个显式的 `PetscComplex`。

此外，还有

```
#define PETSC_BINARY_INT_SIZE    (32/8)
#define PETSC_BINARY_FLOAT_SIZE  (32/8)
#define PETSC_BINARY_CHAR_SIZE   (8/8)
#define PETSC_BINARY_SHORT_SIZE  (16/8)
#define PETSC_BINARY_DOUBLE_SIZE (64/8)
#define PETSC_BINARY_SCALAR_SIZE sizeof(PetscScalar)
```

32.2.1 整数

PETSc 中的整数大小同样是在安装时确定的： `PetscInt` 可以是 32 位或 64 位。后者对于索引大型向量和矩阵非常有用。此外，还有一种 `PetscErrorCode` 类型用于捕获 PETSc 例程的返回码；参见第 38.1.2 节。

为了与其他包兼容，还有另外两种整数类型：

- `PetscBLASInt` 是基本线性代数子程序（*BLAS*）/ 线性代数包（*LAPACK*）库使用的整数类型。如果使用 *-download-blas-lapack* 选项，则为 32 位，但如果使用 *MKL*，则可以是 64 位。该例程 `PetscBLASIntCast` 将 `PetscInt` 转换为 `PetscBLASInt`，如果过大则返回 `PETSC_ERR_ARG_OUTOFRANGE`。
- `PetscMPIInt` 是 MPI 库的整数类型，始终为 32 位。该例程 *PetscMPIIntCast* 将 `PetscInt` 转换为 `PetscMPIInt`，如果过大则返回 `PETSC_ERR_ARG_OUTOFRANGE`。

许多外部包不支持 64 位整数。

32.2.2 复数

类型为 `PetscComplex` 的数字具有与 `PetscReal` 相匹配的精度。

使用 `PETSC_i` 构造复数：

```
PetscComplex x = 1.0 + 2.0 * PETSC_i;
```

实部和虚部可以用函数 `PetscRealPart` 和 `PetscImaginaryPart` 提取，这些函数返回一个 `PetscReal`。

还有例程 `VecRealPart` 和 `VecImaginaryPart`，分别用向量的实部或虚部替换向量。同样地 `MatRealPart` 和 `MatImaginaryPart`。

32.2.3 MPI 标量

对于 MPI 调用， `MPIU_REAL` 是对应当前 `PetscReal` 的 MPI 类型。

For MPI calls, `MPIU_SCALAR` is the MPI type corresponding to the current `PetscScalar`.

对于 MPI 调用， `MPIU_COMPLEX` 是对应当前 `PetscComplex` .

32.2.4 布尔值

有一种 `PetscBool` 数据类型，其值为 `PETSC_TRUE` 和 `PETSC_FALSE`。

图 32.2 VecCreateC:

```
PetscErrorCode VecCreate(MPI_Comm comm,Vec *v);

F:
VecCreate( comm,v,ierr )
MPI_Comm :: comm
Vec      :: v
PetscErrorCode :: ierr

Python:
vec = PETSc.Vec()
vec.create()
# or:
vec = PETSc.Vec().create()
```

图 32.3 VecDestroySynopsis

```
#include "petscvec.h"
PetscErrorCode  VecDestroy(Vec *v)

Collective on Vec

Input Parameters:
v -the vector
```

32.3 Vec：向量

向量是具有线性索引的对象。向量的元素是浮点数或复数（参见章节 32.2），但不是整数：有关整数，请参见章节 32.5.1。

32.3.1 向量构造

构造一个向量需要多个步骤。首先，需要在一个通信器上创建向量对象，使用 VecCreate（图 32.2）

Python 注释 *40*：向量创建。在 python 中，PETSc.Vec() 创建了一个空句柄的对象，因此需要随后调用 *create()*。在 C 和 Fortran 中，向量类型是一个关键字；在 Python 中，它是 PETSc.Vec.Type 的一个成员。

```
## setvalues.py
comm = PETSc.COMM_WORLD
x = PETSc.Vec().create(comm=comm)
x.setType(PETSc.Vec.Type.MPI)
```

对应的例程 VecDestroy（图 32.3）释放数据并将指针置零。（由于底层 MPI 技术细节，此例程及所有其他 Destroy 例程都是集合操作。）

向量类型需要用 VecSetType（图 32.4）.

最常见的向量类型有：

图 32.4 **VecSetType**Synopsis:#include "petscvec.h"

```
PetscErrorCode VecSetType(Vec vec, VecType method)


Collective on Vec

Input Parameters:
vec- The vector object
method- The name of the vector type

Options Database Key
-vec_type <type> -Sets the vector type; use -help for a list of available types
```

- VECSEQ 对于顺序向量，也就是存在于单个进程上的向量；这通常是在 MPI_COMM_SELF 或 PETSC_COMM_SELF communicator 上创建的。
- VECMPI 对于分布在 communicator 上的向量。这通常是在 MPI_COMM_WORLD 或 PETSC_COMM_WORLD communicator，或其派生的 communicator 上创建的。
- VECSTANDARD 在单个进程上使用时是 VECSEQ，在多个进程上使用时是 VECMPI。

你可能会想为什么会存在这些类型：你本可以只有一种类型，它会尽可能并行。原因是，在并行运行中，你可能偶尔在每个进程上有一个独立的线性系统，这将需要每个进程上有一个顺序向量（和矩阵），而不是更大线性系统的一部分。

一旦你创建了一个向量，你可以通过 VecDuplicate， 来创建更多类似的向量。

```
VecDuplicate(Vec old,Vec *new);
```

或 VecDuplicateVecs

```
VecDuplicateVecs(Vec old,PetscInt n,Vec **new);
```

用于多个向量。对于后者，有一个联合销毁调用 VecDestroyVecs:

```
VecDestroyVecs(PetscInt n,Vec **vecs);
```

(这在 Fortran 中是不同的)。

32.3.2 向量布局

接下来在创建过程中，向量大小通过 VecSetSizes (图 32.5) 进行设置。由于向量通常是分布式的，这涉及全局大小和各处理器上的大小。设置两者是多余的，因此可以指定其中一个，让库计算另一个。这通过将其设置为 PETSC_DECIDE 来表示。

Python note 41: 向量大小。使用 PETSc.DECIDE 作为未指定的参数 :

```
x.setSizes([2,PETSc.DECIDE])
```

大小通过 VecGetSize (图 32.6) 查询全局大小，通过 VecGetLocalSize (图 32.6) 查询局部大小。

每个处理器获得向量的连续部分。使用 VecGetOwnershipRange (图 32.7) 查询第一个索引

图 32.5 **VecSetSizes**C:#include "petscvec.h"

```
PetscErrorCode  VecSetSizes(Vec v, PetscInt n, PetscInt N)
Collective on Vec


Input Parameters
v :the vector
n : the local size (or PETSC_DECIDE to have it set)
N : the global size (or PETSC_DECIDE)

Python:
PETSc.Vec.setSizes(self, size, bsize=None)
size is a tuple of local/global
```

图 32.6 **VecGetSize**

```
VecGetSize / VecGetLocalSize

C:
#include "petscvec.h"
PetscErrorCode  VecGetSize(Vec x,PetscInt *gsize)
PetscErrorCode  VecGetLocalSize(Vec x,PetscInt *lsize)

Input Parameter
x -the vector

Output Parameters
gsize - the global length of the vector
lsize - the local length of the vector

Python:
PETSc.Vec.getLocalSize(self)
PETSc.Vec.getSize(self)
PETSc.Vec.getSizes(self)
```

图 32.7 **VecGetOwnershipRange**#include "petscvec.h"

```
PetscErrorCode  VecGetOwnershipRange(Vec x,PetscInt *low,PetscInt *high)

Input parameter:
x - the vector

Output parameters:low  - the first local element, pass in NULL if not interested
high - one more than the last local element, pass in NULL if not interested


Fortran note:
use PETSC_NULL_INTEGER for NULL.
```

图 32.8 **VecAXPY**Synopsis:#include "petscvec.h"

```
PetscErrorCode  VecAXPY(Vec y,PetscScalar alpha,Vec x)


Not collective on Vec

Input Parameters:
alpha - the scalarx,
 y  - the vectors

Output Parameter:
y - output vector
```

图 32.9 **VecView**C:

```
#include "petscvec.h"
PetscErrorCode  VecView(Vec vec,PetscViewer viewer)

for ascii output use:
PETSC_VIEWER_STDOUT_WORLD

Python:
PETSc.Vec.view(self, Viewer viewer=None)

ascii output is default or use:
PETSc.Viewer.STDOUT(type cls, comm=None)
```

在该进程上，以及下一个进程的第一个。

通常最好让 PETSc 负责矩阵和向量对象的内存管理，包括分配和释放内存。然而，在 PETSc 与其他应用程序接口的情况下，可能希望从已分配的数组创建一个 `Vec` 对象： `VecCreateSeqWithArray` 和 `VecCreateMPIWithArray`。

```
VecCreateSeqWithArray
   (MPI_Comm comm,PetscInt bs,
      PetscInt n,PetscScalar *array,Vec *V);
VecCreateMPIWithArray
   (MPI_Comm comm,PetscInt bs,
    PetscInt n,PetscInt N,PetscScalar *array,Vec *vv);
```

正如您将在第 32.4.1 节中看到的，您还可以基于矩阵的布局创建向量，使用 `MatCreateVecs`。

32.3.3 向量操作

有许多操作向量的例程是您编写科学应用程序所需要的。示例包括：范数、向量加法（包括 BLAS 类型的 ‘AXPY’ 例程： `VecAXPY` (图 32.8)), 点对点缩放，内积。通过对 VecXYZ 例程的单次函数调用，PETSc 中提供了大量此类操作。

For debugging purpoases, the `VecView` (figure 32.9) routine can be used to display vectors on screen as ascii output,

```
// fftsine.c
```

图 32.10 **VecDot**Synopsis:#include "petscvec.h"

```
PetscErrorCode VecDot(Vec x,Vec y,PetscScalar *val)


Collective on Vec

Input Parameters:
x, y - the vectors

Output Parameter:
val - the dot product
```

图 32.11 **VecScale**Synopsis:#include "petscvec.h"

```
PetscErrorCode VecScale(Vec x, PetscScalar alpha)


Not collective on Vec

Input Parameters:
x     - the vector
alpha - the scalar

Output Parameter:
x - the scaled vector
```

```
PetscCall( VecView(signal,PETSC_VIEWER_STDOUT_WORLD) );
PetscCall( MatMult(transform,signal,frequencies) );
PetscCall( VecScale(frequencies,1./Nglobal) );
PetscCall( VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD) );
```

但例程调用也使用更通用的 `PetscViewer` 对象，例如将向量转储到文件。

这里有几个具有代表性的向量例程：

```
PetscReal lambda;
ierr = VecNorm(y,NORM_2,&lambda); CHKERRQ(ierr);
ierr = VecScale(y,1./lambda); CHKERRQ(ierr);
```

Exercise 32.1. Create a vector where the values are a single sine wave. using `VecGetSize`, `VecGetLocalSize`, `VecGetOwnershipRange`. Quick visual inspection:
`ibrun vec -n 12 -vec_view`
(There is a skeleton for this exercise under the name vec.)

练习 32.2. 使用例程 `VecDot`（图 32.10），`VecScale`（图 32.11）和 `VecNorm`（图 32.12）来计算向量 `x,y` 的内积，缩放向量 `x`，并检查其范数：

$$
\begin{aligned}
p &\leftarrow x^t y \\
x &\leftarrow x/p \\
n &\leftarrow \|x\|_2
\end{aligned}
$$

Python 注释 *42*：向量操作。加号运算符被重载，使得

图 32.12 **VecNorm**C:#include "petscvec.h"

```
PetscErrorCode  VecNorm(Vec x,NormType type,PetscReal *val)
where type isNORM_1, NORM_2, NORM_FROBENIUS, NORM_INFINITY



Python:
PETSc.Vec.norm(self, norm_type=None)

where norm is variable in PETSc.NormType:NORM_1, NORM_2,
 NORM_FROBENIUS, NORM_INFINITY orN1, N2, FRB, INF
```

```
x+y
```

is defined.

```
x.sum() # max,min,....x.dot(y)

x.norm(PETSc.NormType.NORM_INFINITY)
```

32.3.3.1 分割集合操作

MPI 原则上能够实现 “ 计算与通信重叠 ”，或延迟隐藏。PETSc 通过将范数和内积分成两个阶段来支持这一点。

- 开始内积 / 范数，使用 `VecDotBegin` / `VecNormBegin`;
- 结束内积 / 范数，使用 `VecDotEnd` / `VecNormEnd`;

即使你没有实现重叠，也可以使用这些调用来组合多个 “ 集合操作 ”：先执行一个内积和一个范数的 `Begin` 调用；然后（按相同顺序）执行 `End` 调用。这意味着只对一个两字包进行一次归约，而不是对单字包进行两次独立归约。

32.3.4 向量元素

设置传统数组的元素很简单。设置分布式数组的元素则更困难。首先， `VecSet` 将向量设置为一个常数值：

```
ierr = VecSet(x,1.); CHKERRQ(ierr);
```

在一般情况下，通过函数 `VecSetValue` （图 32.13 ）使用全局编号来设置 PETSc 向量中的元素；任何进程都可以设置向量中的任意元素。还有一个用于设置多个元素的例程 `VecSetValues` （图 32.14 ）。这主要用于设置块矩阵的密集子块。

我们通过设置单个元素为 `VecSetValue` 和两个元素为 `VecSetValues` 来说明这两种例程。在后一种情况下，索引和值都需要长度为二的数组。索引不必是连续的。

图 32.13 **VecSetValue**Synopsis#include <petscvec.h>

```
PetscErrorCode VecSetValue
(Vec v,PetscInt row,PetscScalar value,InsertMode mode);


Not Collective

Input Parameters
v- the vector
row- the row location of the entry
value- the value to insert
mode- either INSERT_VALUES or ADD_VALUES
```

图 32.14 **VecSetValues**Synopsis

```
#include "petscvec.h"
PetscErrorCode  VecSetValues
(Vec x,PetscInt ni,const PetscIntix[],
const PetscScalar y[],InsertMode iora)

Not Collective

Input Parameters:x - vector to insert in
ni - number of elements to addix - indices where to add
y - array of values
iora - either INSERT_VALUES or ADD_VALUES, where
ADD_VALUES adds values to any existing entries, and
INSERT_VALUES replaces existing entries with new values
```

```
i = 1; v = 3.14;
VecSetValue(x,i,v,INSERT_VALUES);
ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;
VecSetValues(x,2,ii,vv,INSERT_VALUES);
```

Fortran note 27: 设置值。 在 Fortran 中，value/values 例程的工作方式相同。请注意，尽管有类型检查，使用‘values’ 例程并传递标量是允许的：

Python 注释 *43*：设置向量值。 单个元素：

```
x.setValue(0,1.)
```

多个元素：

```
x.setValues( [2*procno,2*procno+1], [2.,3.] )
```

使用 `VecSetValue` 指定本地向量元素对应于在本地数组中的简单插入。然而，属于另一个进程的元素需要被传输。这通过两次调用完成： `VecAssemblyBegin`(图 32.15) 和 `VecAssemblyEnd`。

图 32.15 **VecAssemblyBegin**

```
#include "petscvec.h"
PetscErrorCode  VecAssemblyBegin(Vec vec)
PetscErrorCode  VecAssemblyEnd(Vec vec)

Collective on Vec

Input Parameter
vec -the vector
```

图 32.16 **VecGetArray**// vecarray.c

```
PetscScalar const *in_array;
PetscScalar *out_array;
VecGetArrayRead(x,&in_array);
VecGetArray(y,&out_array);
PetscInt localsize;
VecGetLocalSize(x,&localsize);
for (int i=0; i<localsize; i++)
out_array[i] = 2*in_array[i];
VecRestoreArrayRead(x,&in_array);
VecRestoreArray(y,&out_array);
```

```
if (myrank==0) then
   do vecidx=0,globalsize-1
   vecelt = vecidx
      call VecSetValue(vector,vecidx,vecelt,INSERT_VALUES,ierr)
   end do
end if
call VecAssemblyBegin(vector,ierr)
call VecAssemblyEnd(vector,ierr)
```

(如果你了解 MPI 库，你会认识到第一个调用对应于发布非阻塞发送和接收调用；第二个调用则包含等待调用。因此，这些独立调用的存在使得延迟隐藏成为可能。)

```
VecAssemblyBegin(myvec);
// do work that does not need the vector myvec
VecAssemblyEnd(myvec);
```

Elements can either be inserted with `INSERT_VALUES`，或使用 `ADD_VALUES` 在 `VecSetValue` / `VecSetValues` 添加 call. 你不能立即混合这些模式；要做到这一点，你需要在 `VecAssemblyBegin` / `VecAssemblyEnd` 中调用 between add/insert phases.

32.3.4.1 显式元素访问

由于向量例程涵盖了大量的操作，您几乎不需要访问实际元素。如果您仍然需要这些元素，可以使用 `VecGetArray` (图 32.16) 进行一般访问，或使用 `VecGetArrayRead`(图 32.16) 进行只读访问。

图 32.17 **VecRestoreArray**C:#include "petscvec.h"

```
PetscErrorCode VecRestoreArray(Vec x,PetscScalar **a)


Logically Collective on Vec

Input Parameters:
x- the vector
a- location of pointer to array obtained from VecGetArray()

Fortran90:
#include <petsc/finclude/petscvec.h>
use petscvec
VecRestoreArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)

Input Parameters:
x- vector
xx_v- the Fortran90 pointer to the array
```

PETSc 坚持要求你使用 `VecRestoreArray`（图 32.17）正确发布这个指针，或者 `VecRestoreArrayRead`（图 32.17）。

在下面的示例中，通过直接数组访问对向量进行缩放。注意源向量和目标向量的不同调用，并注意源数组上的 `const` 限定符：

```
// vecarray.c
PetscScalar const *in_array;
PetscScalar *out_array;
VecGetArrayRead(x,&in_array);
VecGetArray(y,&out_array);
PetscInt localsize;
VecGetLocalSize(x,&localsize);
for (int i=0; i<localsize; i++)
  out_array[i] = 2*in_array[i];
VecRestoreArrayRead(x,&in_array);
VecRestoreArray(y,&out_array);
```

这个例子也使用了 `VecGetLocalSize` 来确定访问数据的大小。即使在分布式环境中运行，你也只能获得本地元素的数组。从另一个进程访问元素需要显式通信；参见章节 32.5.2。

有一些 `VecGetArray` 操作的变体：

- `VecReplaceArray`（图 32.18）释放了 `Vec` 对象的内存，并用另一个不同的数组替换它。后者数组需要用 `PetscMalloc` 分配。
- `VecPlaceArray`（图 32.18）也在向量中安装了一个新数组，但保留了原始数组；这可以通过 `VecResetArray` 恢复。

将一个向量的数组放入另一个向量有一个常见的应用场景，即你有一个分布式向量，但想将 PETSc 操作应用于其本地部分，就像它是一个顺序向量。在这种情况下，你会创建一个顺序向量，并将分布式向量的内容 `VecPlaceArray` 到其中。

Figure 32.18 **VecPlaceArray**

```
Replace the storage of a vector by another array
Synopsis

#include "petscvec.h"
PetscErrorCode  VecPlaceArray(Vec vec,const PetscScalar array[])
PetscErrorCode  VecReplaceArray(Vec vec,const PetscScalar array[])

Input Parameters
vec   - the vector
array - the array
```

Fortran 注释 *28*：通过指针访问 *F90* 数组。有一些例程如 `VecGetArrayF90`（对应的 `VecRestoreArrayF90`）返回一个指向一维数组的（Fortran）指针。

```
!! vecset.F90
  Vec           :: vectorPetscScalar,dimension(:),
  pointer :: elements
  call VecGetArrayF90(vector,elements,ierr)
  write (msg,10) myrank,elements(1)
10 format("First element on process",i3,":",f7.4,"\n")
  call PetscSynchronizedPrintf(comm,msg,ierr)
  call PetscSynchronizedFlush(comm,PETSC_STDOUT,ierr)
  call VecRestoreArrayF90(vector,elements,ierr)
```

```
!! vecarray.F90
  PetscScalar,dimension(:),Pointer :: &
       in_array,out_array
  call VecGetArrayReadF90( x,in_array,ierr )
  call VecGetArrayF90( y,out_array,ierr )
  call VecGetLocalSize( x,localsize,ierr )
  do index=1,localsize
     out_array(index) = 2*in_array(index)
  end do
  call VecRestoreArrayReadF90( x,in_array,ierr )
  call VecRestoreArrayF90( y,out_array,ierr )
```

Python 注释 *44*：向量访问。

```
   x.getArray()
   x.getValues(3)
   x.getValues([1, 2])
```

32.3.5 文件 I/O

如上所述，`VecView` 可用于在终端屏幕上显示向量。然而，查看器实际上用途更广。如第 38.2.2 节所述，它们也可以用于导出向量数据，例如导出到文件。

相反的操作，即加载以这种方式导出的向量，是 *VecLoad*。

图 32.19 **MatCreate**C:

```
PetscErrorCode MatCreate(MPI_Comm comm,Mat *v);

Python:
mat = PETSc.Mat()
mat.create()
# or:
mat = PETSc.Mat().create()
```

图 32.20 **MatSetType**#include "petscmat.h"

```
PetscErrorCode  MatSetType(Mat mat, MatType matype)

Collective on Mat

Input Parameters:
mat- the matrix object
matype- matrix type

Options Database Key
-mat_type <method> -Sets the type; use -help for a list of available methods (for instance, seqaij)
```

由于这些操作是彼此的逆操作，通常你不需要了解文件格式。但以防万一：

```
PetscInt    VEC_FILE_CLASSID
PetscInt    number of rows
PetscScalar *values of all entries
```

也就是说，文件以一个魔数开始，然后是向量元素的数量，随后是所有标量值。

32.4 Mat: 矩阵

PETSc matrices come in a number of types, sparse and dense being the most important ones. Another possibility is to have the matrix in operation form, where only the action $y \leftarrow Ax$ is defined.

32.4.1 矩阵创建

创建矩阵也从指定一个通信器开始，该通信器上矩阵以集体方式存在： `MatCreate` (图 32.19)

用 `MatSetType` (图 32.20) 设置矩阵类型。主要选择是在顺序与分布式以及稠密与稀疏之间，得到的类型有：
`MATMPIDENSE`, `MATMPIAIJ`, `MATSEQDENSE`, `MATSEQAIJ`。

分布式矩阵按块行划分：每个进程存储一个块行，即一组连续的矩阵行。它存储该块行中的所有元素。为了使矩阵 - 向量乘法可执行，输入和输出向量都需要按照矩阵的划分方式进行划分。

虽然对于稠密矩阵，块行方案不可扩展，但对于来自偏微分方程的矩阵来说，这种方案是合理的。在那里，按矩阵块细分会导致许多空块。

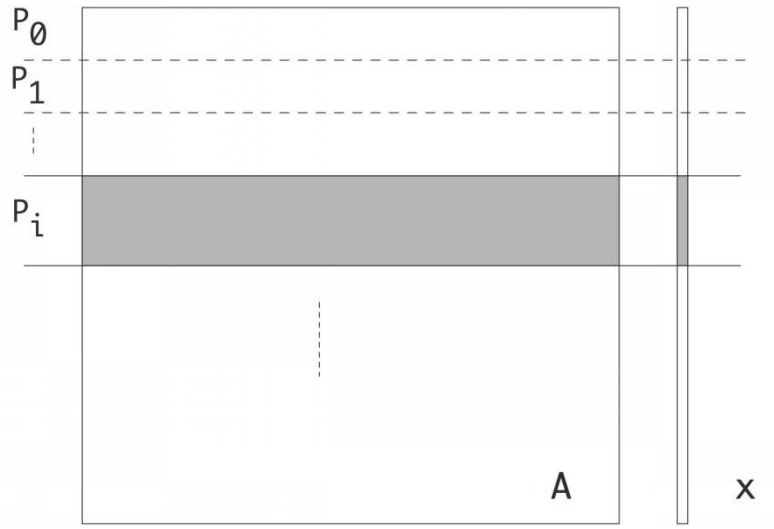


图 32.1：按块行划分矩阵

图 32.21 MatSetSizesC：

```
#include "petscmat.h"PetscErrorCode MatSetSizes(Mat A,
PetscInt m, PetscInt n, PetscInt M, PetscInt N)


Input Parameters
A : the matrix
m : number of local rows (or PETSC_DECIDE)
n : number of local columns (or PETSC_DECIDE)
M : number of global rows (or PETSC_DETERMINE)
N : number of global columns (or PETSC_DETERMINE)

Python:
PETSc.Mat.setSizes(self, size, bsize=None)
where 'size' is a tuple of 2 global sizes
or a tuple of 2 local/global pairs
```

就像向量一样，也有局部和全局大小；只不过现在适用于行和列。使用 MatSetSizes（图 32.21）设置大小，随后用 MatSizes（图 32.22）查询它们。局部列大小的概念比较棘手：由于一个进程存储一个完整的块行，你可能会期望局部列大小是整个矩阵的大小，但事实并非如此。确切的定义将在后面讨论，但对于方阵来说，让局部行和列大小相等是一个安全的策略。

与其查询矩阵大小并相应地创建向量，不如使用例程 MatCreateVecs（图 32.23）。（有时这甚至是必须的；见第 32.4.9 节。）

32.4.2 非零结构

对于稠密矩阵，一旦您指定了大小和 MPI 进程数，就很容易确定 PETSc 需要为矩阵分配多少空间。对于稀疏矩阵，这就更复杂了，因为

图 32.22 **MatSizes**C:#include "petscmat.h"

```
PetscErrorCode MatGetSize(Mat mat,PetscInt *m,PetscInt *n)
PetscErrorCode MatGetLocalSize(Mat mat,PetscInt *m,PetscInt *n)


Python:
PETSc.Mat.getSize(self) # tuple of global sizes
PETSc.Mat.getLocalSize(self) # tuple of local sizes
PETSc.Mat.getSizes(self) # tuple of local/global size tuples
```

图 32.23 **MatCreateVecs**SynopsisGet vector(s) compatible with the matrix,

```
 i.e. with the same parallel layout

#include "petscmat.h"
PetscErrorCode MatCreateVecs(Mat mat,Vec *right,Vec *left)

Collective on Mat

Input Parameter
mat - the matrix

Output Parameter;
right  - (optional) vector that the matrix can be multiplied against
left  - (optional) vector that the matrix vector product can be stored in
```

矩阵可以介于完全空和完全填满之间的任何状态。可以采用一种动态方法，随着元素的指定，空间逐渐增长；然而，反复的分配和重新分配效率低下。基于此，PETSc 对程序员提出了一个小要求：你需要指定矩阵将包含的元素数量的上限。

我们通过观察一些案例来解释这一点。首先考虑一个只存在于单个进程上的矩阵。此时你会使用 `MatSeqAIJSetPreallocation`（图 32.24）。对于三对角矩阵的情况，你需要指定每行有三个元素：

```
MatSeqAIJSetPreallocation(A,3, PETSC_NULLPTR);
```

如果矩阵不那么规则，你可以使用第三个参数来提供一个显式的行长度数组：

```
int *rowlengths;
// allocate, and   then:
for (int row=0; row<nrows; row++)
  rowlengths[row] = // calculation of row length
MatSeqAIJSetPreallocation(A,PETSC_NULLPTR,rowlengths);
```

对于分布式矩阵，您需要根据矩阵的块结构来指定此界限。如图 32.2 所示，矩阵具有对角部分和非对角部分。对角部分描述了耦合输入和输出向量中属于该进程的元素的矩阵元素。非对角部分包含与不属于该进程的元素相乘的矩阵元素，以计算属于该进程的元素。

图 32.24 MatSeqAIJSetPreallocation

```
#include "petscmat.h"
PetscErrorCode  MatSeqAIJSetPreallocation
(Mat B,PetscInt nz,const PetscInt nnz[])
PetscErrorCode  MatMPIAIJSetPreallocation
(Mat B,PetscInt d_nz,const PetscInt d_nnz[],
PetscInt o_nz, const PetscInt o_nnz[])

Input Parameters

B - the matrix
nz/d_nz/o_nz - number of nonzeros per row in matrix or
    diagonal/off-diagonal portion of local submatrix
nnz/d_nnz/o_nnz - array containing the number of nonzeros in the various rows of
    the sequential matrix / diagonal / offdiagonal part of the local submatrix
    or NULL (PETSC_NULL_INTEGER in Fortran) if nz/d_nz/o_nz is used.

Python:
PETSc.Mat.setPreallocationNNZ(self, [nnz_d,nnz_o] )
PETSc.Mat.setPreallocationCSR(self, csr)
PETSc.Mat.setPreallocationDense(self, array)
```

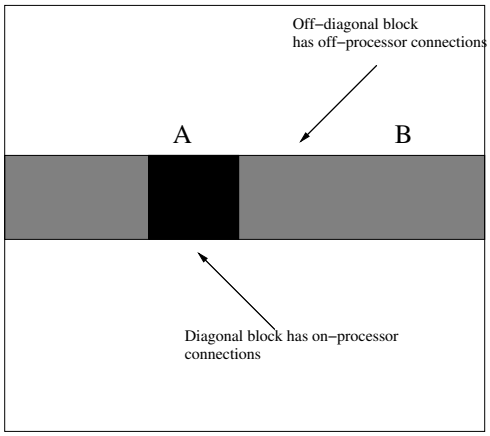


图 32.2：矩阵的对角线部分和非对角线部分

预分配规范现在对这些对角线和非对角线部分有单独的参数：使用 MatMPIAIJSetPreallocation（图 32.24）。你可以为两者指定非零元素数量的全局上限，或者详细列出行长度。对于 *Laplace* 方程的矩阵，这个规范看起来应该是：

```
MatMPIAIJSetPreallocation(A, 3, PETSC_NULLPTR, 2, PETSC_NULLPTR);
```

H然而，只有当并行划分的块结构等于域中线的结构时，这才是正确的。
通常可能需要使用一个高估值。然后可以通过复制矩阵来收缩存储。

指定非零元素数量的上限通常已经足够，且不会太浪费。然而，如果许多行有

图 32.25 **MatSetValue**C:#include <petscmat.h>

```
PetscErrorCode MatSetValue(Mat m,PetscInt row,PetscInt col,
PetscScalar value,InsertMode mode)


Input Parameters
m : the matrix
row : the row location of the entry
col : the column location of the entry
value : the value to insert
mode : either INSERT_VALUES or ADD_VALUES

Python:
PETSc.Mat.setValue(self, row, col, value, addv=None)
also supported:
A[row,col] = value
```

图 32.26 **MatAssemblyBegin**C:#include "petscmat.h"

```
PetscErrorCode MatAssemblyBegin(Mat mat,MatAssemblyType type)
PetscErrorCode MatAssemblyEnd(Mat mat,MatAssemblyType type)


Input Parametersmat- the matrix
type- type of assembly, either MAT_FLUSH_ASSEMBLY
or MAT_FINAL_ASSEMBLY

Python:
assemble(self, assembly=None)
assemblyBegin(self, assembly=None)
assemblyEnd(self, assembly=None)

there is a class PETSc.Mat.AssemblyType:
FINAL = FINAL_ASSEMBLY = 0
FLUSH = FLUSH_ASSEMBLY = 1
```

如果非零元素比这些界限更少，会浪费大量空间。在这种情况下，你可以用一个数组替换 *PETSC_NULLPTR* 参数，该数组列出每一行中非零元素的数量。

32.4.3 Matrix elements

你可以用 MatSetValue（图 32.25）设置单个矩阵元素，或者设置一块矩阵元素，其中你提供一组 i 和 j 索引，使用 MatSetValues。

设置矩阵元素后，矩阵需要被组装。这时如果矩阵元素被指定在其他地方，PETSc 会将它们移动到正确的处理器。与向量类似，这需要两次调用： MatAssemblyBegin（图 32.26）和 MatAssemblyEnd（图 32.26），这可以用来实现 延迟隐藏。

元素可以被插入（INSERT_VALUES）或添加（ADD_VALUES）。你不能立即混合这两种模式；要做到这一点，你需要调用 MatAssemblyBegin / MatAssemblyEnd，其值为 MAT_FLUSH_ASSEMBLY。

图 32.27 MatGetRowSynopsis:

```
#include "petscmat.h"PetscErrorCode MatGetRow(Mat mat,PetscInt row,
PetscInt *ncols,const PetscInt *cols[],const PetscScalar *vals[])
PetscErrorCode MatRestoreRow(Mat mat,PetscInt row,PetscInt *ncols,
const PetscInt *cols[],const PetscScalar *vals[])


Input Parameters:
mat - the matrix
row - the row to get

Output Parametersncols - if not NULL,
 the number of nonzeros in the rowcols - if not NULL,
 the column numbersvals - if not NULL, the values
```

PETSc 稀疏矩阵非常灵活：您可以先创建空矩阵，然后开始添加元素。然而，这在执行时非常低效，因为操作系统每次矩阵稍微增长时都需要重新分配矩阵。因此，PETSc 提供了调用接口，允许用户指明矩阵最终将包含多少元素。

```
MatSetOption(A, MAT_NEW_NONZERO_ALLOCATION_ERR, PETSC_FALSE)
```

32.4.3.1 元素访问

如果您确实需要访问矩阵元素，有诸如 MatGetRow (图 32.27) 的例程。通过这个，任何进程都可以使用全局行编号请求它所拥有的某一行的内容。（请求非本地元素需要采用不同的机制，即提取子矩阵；参见第 32.4.6 节。）

由于 PETSc 面向的是稀疏矩阵，这不仅返回元素值，还返回列号以及存储列的重新编号。如果这三个返回值中的任何一个不需要，可以通过设置传递给 *PETSC_NULLPTR* 的参数来取消请求。

PETSc 要求您使用 MatRestoreRow (图 32.27) 正确释放该行。

I也可以使用以下方法检索本地矩阵的完整压缩行存储（CRS）内容 *MatDenseGetArray*, *MatDenseRestoreArray*, *MatSeqAIJGetArray*, *MatSeqAIJRestoreArray*。（例程 MatGetArray / MatRestoreArray 已弃用。）

32.4.4 矩阵查看器

矩阵可以以多种方式“查看”（参见第 38.2.2 节关于 PetscViewer 机制的讨论），从 MatView 调用开始。然而，通常使用在线选项更为方便，例如

```
yourprogram -mat_view
yourprogram -mat_view draw
yourprogram -ksp_mat_view draw
```

图 32.28 **MatMult**Synopsis#include "petscmat.h"

```
PetscErrorCode MatMult(Mat mat,Vec x,Vec y)
PetscErrorCode MatMultTranspose(Mat mat,Vec x,Vec y)


Neighbor-wise Collective on Mat

Input Parameters
mat - the matrix
x - the vector to be multiplied

Output Parameters
y - the result
```

图 32.29 **MatMultAdd**Synopsis#include "petscmat.h"

```
PetscErrorCode MatMultAdd(Mat mat,Vec x,Vec y,Vec z)


Neighbor-wise Collective on Mat

Input Parameters
mat  - the matrixx,
 y  - the vectors

Output Parameters
z -the result

Notes
The vectors x and z cannot be the same.
```

其中 *-mat_view* 由组装例程激活，而 *-ksp_mat_view* 仅显示作为 KSP 对象的算子所用的矩阵。若无进一步的选项细化， 这将显示稀疏模式内的矩阵元素。使用子选项 draw 将导致稀疏模式在 *X11* 窗口中显示。

32.4.5 矩阵操作

32.4.5.1 矩阵 - 向量操作

在 PETSc 的典型应用中，使用迭代方法求解大型稀疏线性方程组时，矩阵 - 向量操作最为重要。首先是矩阵 - 向量乘积 MatMult (图 32.28) 和转置乘积 MatMultTranspose (图 32.28)。 （在复数情况下，转置乘积不是厄米矩阵乘积；若需使用厄米矩阵乘积，请使用 MatMultHermitianTranspose。）

对于 BLAS gemv 语义 $y \leftarrow \alpha Ax + \beta y$， MatMultAdd (图 32.29) 计算 $z \leftarrow Ax + y$。

32.4.5.2 矩阵 - 矩阵运算

有许多矩阵 - 矩阵运算例程，例如 MatMatMult。

图 32.30 **MatCreateShell**#include "petscmat.h"

```
PetscErrorCode  MatCreateShell(MPI_Comm comm,
PetscInt m,PetscInt n,PetscInt M,PetscInt N,
void *ctx,Mat *A)


Collective

Input Parameters:
comm- MPI communicator
m- number of local rows (must be given)
n- number of local columns (must be given)
M- number of global rows (may be PETSC_DETERMINE)
N- number of global columns (may be PETSC_DETERMINE)
ctx- pointer to data needed by the shell matrix routines

Output Parameter:
A -the matrix
```

32.4.6 子矩阵

给定一个并行矩阵，有两种提取子矩阵的例程：

- `MatCreateSubMatrix` 创建一个单一的并行子矩阵。
- `MatCreateSubMatrices` 在每个进程上创建一个顺序子矩阵。

32.4.7 Shell 矩阵

在许多科学应用中，矩阵代表某种算子，我们本质上并不关心矩阵元素，而只关心矩阵对向量的作用。事实上，在某些情况下，实现一个计算矩阵作用的例程比显式构造矩阵更为方便。

也许令人惊讶的是，线性方程组的求解可以通过这种方式处理。原因是 PETSc 的迭代求解器（章节 35.1 ）只需要矩阵乘向量（以及可能的矩阵转置乘向量）操作。

PETSc 支持这种工作模式。例程 `MatCreateShell` (图 32.30) 声明参数为以算子形式给出的矩阵。

32.4.7.1 Shell operations

下一步是添加自定义乘法例程，该例程将由 `MatMult` 调用：`MatShellSetOperation`(图 32.31)

实现实际乘积的例程应具有与 `MatMult` 相同的签名，接受一个矩阵和两个向量。实现自定义乘积例程的关键在于创建例程的 ‘context’ 参数。通过 `MatShellSetContext` (图 32.32) 你传递一个指向包含所有所需上下文信息的结构体的指针。在你的乘法例程中，你可以通过 `MatShellGetContext` (图 32.33) 来检索它。

指定的操作由关键字决定 *MATOP_<OP>* 其中 *OP* 是矩阵例程的名称，去掉 `Mat` 部分，全部大写。

图 32.31 MatShellSetOperation

```
#include "petscmat.h"
PetscErrorCode MatShellSetOperation
   (Mat mat,MatOperation op,void (*g)(void))

Logically Collective on Mat

Input Parameters:
mat- the shell matrix
op- the name of the operation
g- the function that provides the operation.
```

图 32.32 MatShellSetContextSynopsis

```
#include "petscmat.h"
PetscErrorCode  MatShellSetContext(Mat mat,void *ctx)

Input Parameters
mat - the shell matrix
ctx - the context
```

```
MatCreate(comm,&A);
MatSetSizes(A,localsize,localsize,matrix_size,matrix_size);
MatSetType(A,MATSHELL);
MatSetFromOptions(A);
MatShellSetOperation(A,MATOP_MULT,(void*)&mymatmult);
MatShellSetContext(A,(void*)Diag);
MatSetUp(A);
```

(调用 MatSetSizes 需要在 MatSetType 之前进行。)

32.4.7.2 Shell context

Setting the context means passing a pointer (really: an address) to some allocated structure

```
struct matrix_data mystruct;
MatShellSetContext( A, &mystruct );
```

该例程签名中此参数为 void* 但不必将其强制转换为该类型。获取上下文意味着需要设置指向 yourstructure 的指针

```
struct matrix_data *mystruct;
MatShellGetContext( A, &mystruct );
```

有些令人困惑的是，Get 例程也有一个 void* 参数，尽管它实际上是一个指针变量。

32.4.8 多组件矩阵

对于多组件物理问题，存储线性系统基本上有两种方式

图 32.33 MatShellGetContext

```
#include "petscmat.h"
PetscErrorCode  MatShellGetContext(Mat mat,void *ctx)

Not Collective

Input Parameter:
mat -the matrix,   should have been created with MatCreateShell()

Output Parameter:
ctx -the user provided context
```

1. 将物理方程组合在一起，或 2. 将域节点组合在一起。

在这两种情况下，这对应于一个块矩阵，但对于一个有 N 个节点和 3 个方程的问题，相应的结构是：

1. 3×3 个大小为 N 的块，对比
2. $N \times N$ 个大小为 3 的块。

第一种情况可以被形象化为

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

a虽然看起来很自然，但它存在一个计算问题。此类问题的预处理器通常看起来像 like

$$\begin{pmatrix} A_{00} & & \\ & A_{11} & \\ & & A_{22} \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} A_{00} & & \\ A_{10} & A_{11} & \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

对于 PETSc 矩阵的块行划分，这意味着预处理器求解的效率最多为 50%。

最好使用第二种方案，它需要 `MATMPIBIJ` 格式，并使用所谓的 *field-split precondi-tioner*；参见章节 35.1.7.3.5。

32.4.9 傅里叶变换

快速傅里叶变换（FFT）可以被视为矩阵 - 向量乘法。*PETSc* 通过允许您创建一个带有 `MatCreateFFT` 的矩阵来支持这一点。这要求您在配置时添加一个 FFT 库，例如 *fftw*；参见章节 31.4。

FFT 库可能使用填充，因此向量应使用 `MatCreateVecsFFTW` 创建，而不是使用独立的 `VecSetSizes`。

Tfftw 库不会缩放输出向量，因此先进行正向变换再进行反向变换会得到一个结果，该结果过于l大，按向量大小放大。

```
// fftsine.c
PetscCall( VecView(signal,PETSC_VIEWER_STDOUT_WORLD) );
PetscCall( MatMult(transform,signal,frequencies) );
PetscCall( VecScale(frequencies,1./Nglobal) );
PetscCall( VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD) );
```

一个完整的余弦波：

```
1.0.809017 + 0.587785 i
0.309017 + 0.951057 i
-0.309017 + 0.951057 i
-0.809017 + 0.587785 i
-1. + 1.22465e-16 i
-0.809017 - 0.587785 i
-0.309017 - 0.951057 i
0.309017 - 0.951057 i
0.809017 - 0.587785 i
```

Frequency $n = 1$ amplitude $\equiv 1$:

```
-2.22045e-17 + 2.33487e-17 i
1. - 9.23587e-17 i
2.85226e-17 + 1.56772e-17 i
-4.44089e-17 + 1.75641e-17 i
-3.35828e-19 + 3.26458e-18 i
0. - 1.22465e-17 i
-1.33873e-17 + 3.26458e-18 i
-4.44089e-17 + 7.59366e-18 i
7.40494e-18 + 1.56772e-17 i
0. + 1.8215e-17 i
```

奇怪的是，反向传播不需要缩放：

```
Vec confirm;
PetscCall( VecDuplicate(signal,&confirm) );
PetscCall( MatMultTranspose(transform,frequencies,confirm) );
PetscCall( VecAXPY(confirm,-1,signal) );
PetscReal nrm;
PetscCall( VecNorm(confirm,NORM_2,&nrm) );
PetscPrintf(MPI_COMM_WORLD,"FFT accuracy %e\n",nrm);
PetscCall( VecDestroy(&confirm) );
```

32.5 索引集和向量散射

在 PETSc 最初设计用于的 PDE 类型应用中，向量数据只能是实数或复数：不存在整数向量。另一方面，整数用于向量的索引，例如用于将边界元素收集到 *halo region*，或用于执行 *datatranspose* 的 *FFT* 操作。

为支持这一点，PETSc 具有以下对象类型：

- 一个 `IS` 对象描述了一组整数索引；
- 一个 `VecScatter` 对象描述了输入向量中一组索引与输出向量中一组索引之间的对应关系。

32.5.1 IS: indexsets

一个 `IS` 对象包含一组 `PetscInt` 值。它可以通过以下方式创建

- `ISCreate` 用于创建一个空集合；
- `ISCreateStride` 用于创建一个步进集合；
- `ISCreateBlock` 对于一组连续的块，放置在明确给出的起始索引列表中。
- `ISCreateGeneral` 对于明确给出的索引列表。

例如，描述奇数和偶数索引（在两个进程上）：

```
// oddeven.c
IS oddeven;
if (procid==0) {
  PetscCall( ISCreateStride(comm,Nglobal/2,0,2,&oddeven) );
} else {
```

图 32.34 VecScatterCreateSynopsis

```
Creates a vector scatter context. Collective on Vec

#include "petscvec.h"
PetscErrorCode VecScatterCreate(Vec xin,IS ix,Vec yin,IS iy,VecScatter *newctx)

Input Parameters:
xin : a vector that defines the layout of vectors from which we scatter
yin : a vector that defines the layout of vectors to which we scatter
ix : the indices of xin to scatter (if NULL scatters all values)
iy : the indices of yin to hold results (if NULL fills entire vector yin)

Output Parameter
newctx : location to store the new scatter context
```

```
  PetscCall( ISCreateStride(comm,Nglobal/2,1,2,&oddeven) );
}
```

之后，会对索引集进行各种查询和设置操作。

你可以通过 `ISGetIndices` 和 `ISRestoreIndices` 读取集合的索引。

32.5.2 VecScatter：全对全操作

A `VecScatter` 对象是 all-to-all 操作的一个泛化。然而，与 MPI `MPI_Alltoall` 不同，MPI 将所有内容都表述为本地缓冲区，`VecScatter` 则更隐式，仅描述输入和输出向量中的索引。

The `VecScatterCreate`(图 32.34) 调用的参数有：

- 一个输入向量。由此使用并行布局；任何被散布的向量都应具有相同的布局。
- 一个 `IS` 对象，描述哪些索引被散布；如果整个向量被重新排列，*PETSC_NULLPTR*(Fortran: `PETSC_NULL_IS`) 可以被提供。
- 一个输出向量。由此使用并行布局；任何被散布到的向量都应具有相同的布局。
- 一个 `IS` 对象，描述哪些索引被散布到；如果整个向量是目标，*PETSC_NULLPTR* 可以被提供。

A 作为一个简单的例子，上面定义的奇数 / 偶数集合可以用来将所有偶数索引的组件移动到进程零，而奇数索引的p组件移动到进程一：

```
VecScatter separate;
PetscCall( VecScatterCreate
  (in,oddeven,out,NULL,&separate) );
PetscCall( VecScatterBegin
  (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
PetscCall( VecScatterEnd
  (separate,in,out,INSERT_VALUES,SCATTER_FORWARD) );
```

注意索引集合应用于输入向量，因为它描述了要移动的组件。输出向量使用 *PETSC_NULLPTR*，因为这些组件是按顺序放置的。

练习 32.3. 修改此示例，使组件仍然按奇偶分开，但现在在每个进程上按降序排列。

练习 32.4. 你能扩展此示例，使进程 p 接收所有是 p 的倍数的索引吗？如果 `Nglobal` 不是 `nprocs` 的倍数，你的解决方案还正确吗？

32.5.2.1 MoreVecScatter 模式

有一个额外的复杂情况，即 `VecScatter` 可以同时具有顺序和并行的输入或输出向量。散布到进程零也是一个常用选项。

32.6 AO: 应用排序

PETSc 决定按连续块行划分矩阵，这在某种意义上可能是一个限制，因为应用程序可能有不同的自然排序。对于这种情况，`AO` 类型可以在这两种方案之间进行转换。

32.7 划分

默认情况下，PETSc 使用基于连续变量块的矩阵和向量划分。在常规情况下，这不是一个坏策略。然而，对于某些矩阵，置换和重新划分可能是有利的。例如，可以查看邻接图，并最小化边切割的数量或边权重的总和。

此功能未内置于 PETSc 中，但可以由图划分包如 *ParMetis* 或 *Zoltan* 提供。基本对象是 `MatPartitioning`，带有用于

- 创建和销毁：`MatPartitioningCreate`, `MatPartitioningDestroy`;
- 将类型 `MatPartitioningSetType` 设置为显式分区器，或作为当前矩阵的对偶或细化生成的某种分区器；
- 应用于 `MatPartitioningApply`，生成一个分布式 `IS` 对象，然后可以在 `MatCreateSubMatrix` 中用于重新分区。

示例说明：

```
MatPartitioning part;
MatPartitioningCreate(comm,&part);
MatPartitioningSetType(part,MATPARTITIONINGPARMETIS);
MatPartitioningApply(part,&is);
/* get new global number of each old global number */
ISPartitioningToNumbering(is,&isn);
ISBuildTwoSided(is,PETSC_NULLPTR,&isrows);
MatCreateSubMatrix(A,isrows,isrows,MAT_INITIAL_MATRIX,&perA);
```

其他场景：

```
MatPartitioningSetAdjacency(part,A);
MatPartitioningSetType(part,MATPARTITIONINGHIERARCH);
MatPartitioningHierarchicalSetNcoarseparts(part,2);
MatPartitioningHierarchicalSetNfineparts(part,2);
```

第 33 章

网格支持

PETSc's DM 对象将抽象层次从线性代数问题提升到物理问题：它们允许更直接地用算子的定义域来表达算子。在本节中，我们将关注 DMDA “ 分布式数组 ” 对象，它们对应于定义在笛卡尔网格上的问题。分布式数组使得构造定义为模板 的 1/2/3 维笛卡尔网格上的算子系数矩阵变得更容易。

主要的创建例程有三种变体，主要区别在于它们的参数数量。例如，DMDACreate2d <style id='4'> 具有沿着 x,y <style id='6'> 轴的参数。然而， DMDACreate1d <style id='7'> 没有模板类型的参数，因为在一维中它们都是相同的，也没有进程分布的参数。

33.1 网格定义

二维网格通过 DMDACreate2d 创建（图 33.1 ）

```
DMDACreate2d( communicator,x_boundary,
y_boundary,
   stenciltype,gridx,gridy,
       procx,procy, dof,width,partitionx,
   partitiony,grid);
```

- 边界类型是类型为 DMBoundaryType 的值。取值为：
  – DM_BOUNDARY_NONE–
  DM_BOUNDARY_GHOSTED, –
  DM_BOUNDARY_PERIODIC,
- 模板类型为 DMStencilType，取值为 –
  DMDA_STENCIL_BOX, – DMDA_STENCIL_STAR。
  (参见图 33.1.)
- gridx,gridy 这些值是全局网格大小。可以通过命令行选项设置 -da_grid_x/y/z。
- 变量 procx,procy 是处理器网格的显式指定。如果没有这个指定 ，PETSc 将尝试找到一个类似于域网格的分布。
- dof 表示 “ 自由度 ” 的数量，其中 1 对应于标量问题。
- width 表示模板的范围：1 表示 5 点模板，或更一般地表示二阶偏微分方程的二阶模板，2 表示四阶偏微分方程的二阶离散化，依此类推。

Figure 33.1 DMDACreate2d

```
#include "petscdmda.h"
PetscErrorCode  DMDACreate2d(MPI_Comm comm,DMBoundaryType bx,DMBoundaryType by,DMDA
        StencilType stencil_type,PetscInt M,PetscInt N,PetscInt m,
    PetscInt n,PetscInt dof,PetscInt s,const PetscInt lx[],
    const PetscInt ly[],DM *da)

Input Parameters

comm - MPI communicatorbx,by - type of ghost nodes: DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED,
 DM_BOUNDARY_PERIODIC.stencil_type - stencil type: DMDA_STENCIL_BOX or DMDA_STENCIL_STAR.
M,N - global dimension in each direction ofm,
n - corresponding number of processors in each dimension (or PETSC_DECIDE)
dof - number of degrees of freedom per nodes - stencil widthlx,
 ly - arrays containing the number ofnodes in each cell along the x and y coordinates,
 or NULL.

Output Parameter

da  -the resulting distributed array object
```

图 33.2 DMDAGetLocalInfo#include "petscdmda.h"

```
PetscErrorCode  DMDAGetLocalInfo(DM da,DMDALocalInfo *info)
```

- *partitionx,partitiony* 是给出网格在处理器上显式划分的数组，或者是 *PETSC_NULLPTR* 的默认分布。

Code:

```
// dmrhs.c
DM grid;
PetscCall( DMDACreate2d
             ( comm,
               DM_BOUNDARY_NONE,DM_BOUNDARY_NONE,
               DMDA_STENCIL_STAR,
               100,100,
               PETSC_DECIDE,PETSC_DECIDE,
               1,
               1,
               NULL,NULL,
               &grid
               ) );
PetscCall( DMSetFromOptions(grid) );
PetscCall( DMSetUp(grid) );
PetscCall( DMViewFromOptions(grid,NULL,"-dm_view") );
```

Output:

```
ld: warning: dylib
    ↪(/Users/eijkhout/Installation/petsc/petsc-3.16.4/
    ↪was built for newer macOS
    ↪version (11.5) than being linked
    ↪(11.0)[0] Local = 0-50 x 0-50,
 halo = 0-51 x↪0-51
[1] Local = 50-100 x 0-50, halo =
↪49-100 x 0-51
[2] Local = 0-50 x 50-100, halo = 0-51
↪x 49-100[3] Local = 50-100 x 50-100,
 halo =↪49-100 x 49-100
```

定义一个 DM 对象后，每个进程拥有整个网格中的一个连续子域。你可以使用 DMDAGetCorners 查询其大小和位置，或者使用 DMDAGetLocalInfo (图 33.2) 查询该信息及所有其他信息，它返回一个 DMDALocalInfo (图 33.3) 结构。

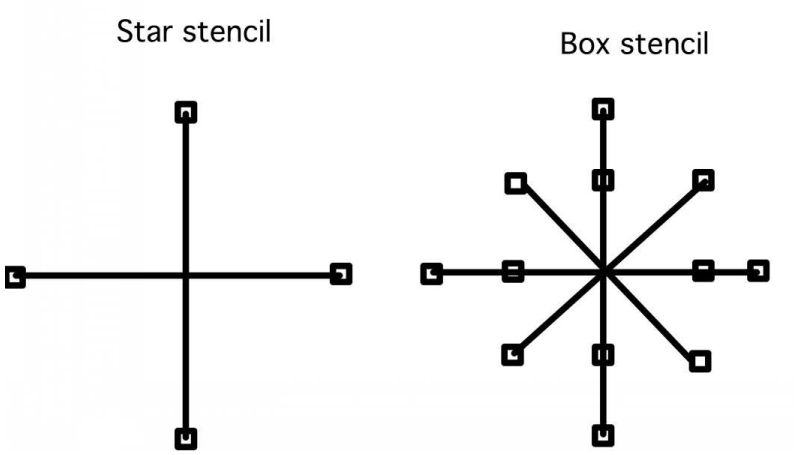


图 33.1：星形和盒形模板

(A DMDALocalInfo struct 对于 1/2/3 维是相同的，因此某些字段可能不适用于您的特定 PDE。)

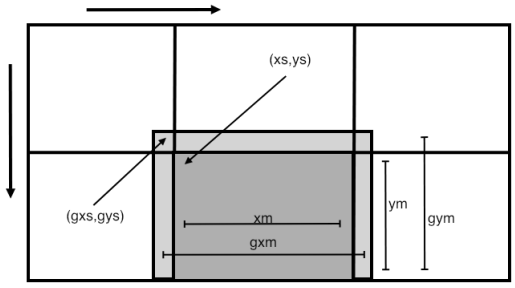


图 33.2：DMDALocalInfo 结构的各字段示意图

33.1.1 关联向量

利用该结构中的字段，每个进程现在可以遍历其自己的子域。例如，所拥有子域的 “左上” 角位于 xs，ys，点的数量为 xm,ym (见图 33.2)，因此我们可以按如下方式遍历子域：

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
for (int i=info.xs; i<info.xs+info.xm; i++) {
// actions on point i,j}}
```

在域的每个点上，我们描述该点的模板。首先，我们现在有信息来计算域点的 x, y 坐标：

```
PetscReal **xyarray;
PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
```

Figure 33.3 DMDALocalInfo

```
typedef struct {PetscInt         dim,dof,sw;PetscInt         mx,my,
mz;    /* global number of grid points in each direction */PetscInt         xs,ys,
zs;    /* starting point of this processor, excluding ghosts */PetscInt         xm,ym,
zm;    /* number of grid points on this processor, excluding ghosts */PetscInt         gxs,gys,
gzs;    /* starting point of this processor including ghosts */PetscInt         gxm,gym,
gzm;    /* number of grid points on this processor including ghosts */DMBoundaryType    bx,by,
bz; /* type of ghost nodes at boundary */DMDAStencilType  st;DM               da;} DMDALocalInfo;


Fortran Notes - This should be declared as

   DMDALocalInfo :: info(DMDA_LOCAL_INFO_SIZE)

and the entries accessed via

   info(DMDA_LOCAL_INFO_DIM)
   info(DMDA_LOCAL_INFO_DOF) etc.

   The entries bx,by,bz, st, and da are not accessible from Fortran.
```

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
for (int i=info.xs; i<info.xs+info.xm; i++) {
PetscReal x = i*hx, y = j*hy;xyarray[j][i] = x*y;}}
PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
```

在某些情况下，我们希望对 DMDA grid 的向量执行模板操作。这需要拥有 *halo region*。上面，你已经看到了 *gxs*,*gxm* 以及与每个进程子域的 halo 相关的其他量。

我们需要的是一种方法来创建包含这些 halo 点的向量。

- 你可以用 DMCreateGlobalVector 制作一个对应于网格的传统向量；如果你只需要这个向量短时间使用，使用 DMGetGlobalVector 和 DMRestoreGlobalVector。
- 你可以用 DMCreateLocalVector 制作一个包含 halo 点的向量；如果你只需要这个向量短时间使用，使用 DMGetLocalVector 和 DMRestoreLocalVector。
- 如果你有一个 “ 全局 ” 向量，你可以用 DMGlobalToLocal 制作相应的 “ 局部 ” 向量，填充其 halo 点；在对局部向量进行操作后，你可以用 DMLocalToGlobal 将其非 halo 部分复制回全局向量。

这里我们设置一个局部向量以进行操作：

```
Vec ghostvector;
PetscCall( DMGetLocalVector(grid,&ghostvector) );
PetscCall( DMGlobalToLocal(grid,xy,INSERT_VALUES,ghostvector) );
PetscReal **xyarray,**gh;
```

```
PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
PetscCall( DMDAVecGetArray(grid,ghostvector,&gh) );
// computation on the arrays
PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
PetscCall( DMDAVecRestoreArray(grid,ghostvector,&gh) );
PetscCall( DMLocalToGlobal(grid,ghostvector,INSERT_VALUES,xy) );
PetscCall( DMRestoreLocalVector(grid,&ghostvector) );
```

实际操作涉及对 halo 实际存在性的一些测试：

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
        if (info.gxs<info.xs && info.gys<info.ys)
          if (i-1>=info.gxs && i+1<=info.gxs+info.gxm &&
              j-1>=info.gys && j+1<=info.gys+info.gym )
            xyarray[j][i] =
              ( gh[j-1][i] + gh[j][i-1] + gh[j][i+1] + gh[j+1][i] )
              /4.;
```

33.1.2 Associated matrix

We construct a m在每个 (i,j) co 上构建模板，从而在 DMDA 上形成矩阵 ordinate on a process:

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    PetscReal x = i*hx, y = j*hy;
    ...
    // set the row, col, v values
    ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);CHKERRQ(ierr);
  }
}
```

每个矩阵元素 `row,col` 是两个 MatStencil 对象的组合。从技术上讲，这是一个 struct，其成员包括 `i,j,k,s`，用于域坐标和场的编号。

```
MatStencil  row;
row.i = i; row.j = j;
```

W我们可以逐个构造该行中的列，但 MatSetValuesStencil 可以一次设置多行或多列
a一次，因此我们同时构造所有列：

```
MatStencil col[5];
PetscScalar v[5];
PetscInt    ncols = 0;
/**** diagonal element ****/
col[ncols].i = i; col[ncols].j = j;
v[ncols++] = 4.;
/**** off diagonal elements ****/....
```

模板的其他 “ 腿 ” 需要有条件地设置：在域的顶行缺少与 $(i-1,j)$ 的连接，在左列缺少与 $(i,j-1)$ 的连接。总的来说：

```
// grid2d.c
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    MatStencil  row,col[5];
    PetscScalar v[5];
    PetscInt    ncols = 0;
    row.j       = j; row.i = i;
    /**** local connection: diagonal element ****/
    col[ncols].j = j; col[ncols].i = i; v[ncols++] = 4.;
    /* boundaries: top and bottom row */
    if (i>0)         {col[ncols].j = j;   col[ncols].i = i-1; v[ncols++] = -1.;}
    if (i<info.mx-1) {col[ncols].j = j;   col[ncols].i = i+1; v[ncols++] = -1.;}
    /* boundary left and right */
    if (j>0)         {col[ncols].j = j-1; col[ncols].i = i;   v[ncols++] = -1.;}
    if (j<info.my-1) {col[ncols].j = j+1; col[ncols].i = i;   v[ncols++] = -1.;}

    PetscCall( MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES) );
  }
}
```

33.2 在网格上构建向量

A DMDA 对象是网格的描述，因此我们现在需要关注如何构建定义在该 t
网格。

We start with vectors: we need a solution vector and a right-hand side. Here we have two options:

1. 我们可以从头构建一个具有正确结构的向量；或者 2. 我们可以利用网格对象具有可提取的向量这一事实。

33.2.1 创建确认向量

如果我们创建一个向量，使用 VecCreate 和 VecSetSizes，很容易得到正确的全局大小，但默认的分区可能不会与网格分布一致。此外，正确设置索引方案也不是一件简单的事。

首先，需要显式设置本地大小，使用来自 DMDALocalInfo 对象的信息：

```
Vec xy;
PetscCall( VecCreate(comm,&xy) );PetscCall( VecSetType(xy,VECMPI) );
PetscInt nlocal = info.xm*info.ym, nglobal = info.mx*info.my;
PetscCall( VecSetSizes(xy,nlocal,nglobal) );
```

之后，不再使用 VecSetValues，而是直接在通过 DMDAVecGetArray 获得的原始数组中设置元素：

```
PetscReal **xyarray;
PetscCall( DMDAVecGetArray(grid,xy,&xyarray) );
for (int j=info.ys; j<info.ys+info.ym; j++) {
for (int i=info.xs; i<info.xs+info.xm; i++) {
PetscReal x = i*hx, y = j*hy;
```

```
        xyarray[j][i] = x*y;
  }
}
PetscCall( DMDAVecRestoreArray(grid,xy,&xyarray) );
```

33.2.2 从 DMDA 中提取向量

33.2.3 细化

例程 *DMDASetRefinementFactor* 可以通过选项 *-da_refine* 激活，或者分别通过 *-da_refine_x*/y/z 方向激活。

33.3 分布式数组的向量

分布式数组类似于分布式向量，因此有提取数组值为向量形式的例程。这可以通过两种方式完成：（这里的例程实际上涉及更通用的 DM ‘数据管理’ 对象，但我们现在将其在 DMDA 的上下文中讨论。）

1. 你可以创建一个 ‘全局’ 向量，该向量定义在与数组相同的通信器上，并以相同的方式进行不相交划分。这是通过 DMCreateGlobalVector 完成的：

```
PetscErrorCode DMCreateGlobalVector(DM dm,Vec *vec)
```

2. 你可以创建一个 ‘本地’ 向量，它是顺序的并定义在 PETSC_COMM_SELF 上，该向量不仅包含进程本地的点，还包含在 DMDACreate 调用定义中指定范围的 ‘halo’ 区域。为此，使用 DMCreateLocalVector：

```
PetscErrorCode DMCreateLocalVector(DM dm,Vec *vec)
```

值可以在局部向量和全局向量之间移动，方法是：

- DMGlobalToLocal：这会建立一个局部向量，包括来自不相交分布的全局向量的幽灵 / 光晕点。（对于通信和计算的重叠，使用 DMGlobalToLocalBegin 和 DMGlobalToLocalEnd。）
- DMLocalToGlobal：这会将局部向量的不相交部分复制回全局向量。（对于通信和计算的重叠，使用 DMLocalToGlobalBegin 和 DMLocalToGlobalEnd。）

33.4 Matrices of adistributed array

一旦你有了网格，就可以创建其关联的矩阵：

```
DMSetUp(grid);
DMCreateMatrix(grid,&A)
```

有了这个子域信息，你就可以开始创建系数矩阵：

```
DM grid;
PetscInt i_first,j_first,i_local,j_local;
DMDAGetCorners(grid,&i_first,&j_first,NULL,&i_local,&j_local,NULL);
for ( PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
  for ( PetscInt j_index=j_first; j_index<j_first+j_local; j_index++) {
  // construct coefficients for domain point (i_index,j_index)
  }
}
```

注意这里的索引是以网格为单位，而不是以矩阵为单位。

举一个简单的例子，考虑一维平滑。从 `DMDAGetCorners` 我们只需要 i 方向上的参数:

```
// grid1d.c
PetscInt i_first,i_local;
PetscCall( DMDAGetCorners(grid,&i_first,NULL,NULL,&i_local,NULL,NULL) );
for (PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
```

然后我们使用一个循环来设置 i 方向上本地范围内的元素:

```
MatStencil  row = {0},col[3] = {{0}};
PetscScalar v[3];
PetscInt    ncols = 0;
row.i = i_index;
col[ncols].i = i_index; v[ncols] = 2.;
ncols++;
if (i_index>0)          { col[ncols].i = i_index-1; v[ncols] = 1.; ncols++; }
if (i_index<i_global-1) { col[ncols].i = i_index+1; v[ncols] = 1.; ncols++; }
PetscCall( MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES) );
```

第 34 章

有限元支持

34.1 通用数据管理

```
ierr = DMCreate(PETSC_COMM_WORLD, &dm);
ierr = DMSetType(dm, DMPLEX);
```

A DMPLEX 默认是二维的。使用

```
plexprogram  -dm_plex_dim k
```

对于其他维度。在二维中，有三个层次的单元:

- 0-cells 是顶点,
- 1-cells 是边，且
- 2-cells 是三角形。

默认的 2×2 grid 依次有:

Code:

```
ierr = DMSetFromOptions(dm);
ierr = PetscObjectSetName((PetscObject) dm, "Sphere");
ierr = DMViewFromOptions(dm, NULL, "-dm_view");
```

输出:

```
mpiexec -n 1 plexsphere -dm_view
DM Object: Sphere 1 MPI processes
  type: plex
Sphere in 2 dimensions:
  Number of 0-cells per rank: 9
  Number of 1-cells per rank: 16
  Number of 2-cells per rank: 8
Labels:
  celltype: 3 strata with value/size
    ↪(0 (9), 3 (8), 1 (16))
  depth: 3 strata with value/size (0
    ↪(9), 1 (16), 2 (8))
  marker: 1 strata with value/size (1
    ↪(16))
  Face Sets: 1 strata with value/size
    ↪(1 (8))
```

以及并行:

代码:

```
ierr = DMSetFromOptions(dm);
ierr = PetscObjectSetName((PetscObject) dm, "Sphere");
ierr = DMViewFromOptions(dm, NULL, "-dm_view");
```

Output:

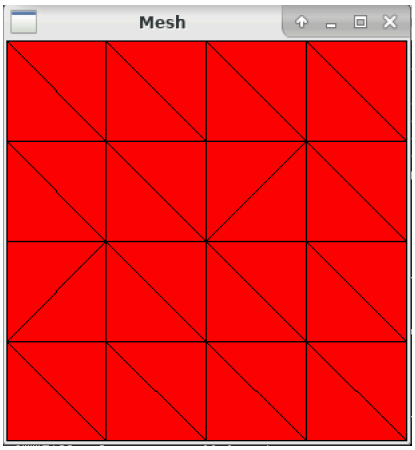
```
mpiexec -n 4 plexsphere -dm_view
DM Object: Sphere 4 MPI processes
  type: plex
Sphere in 2 dimensions:
  Number of 0-cells per rank: 5 5 6 4
  Number of 1-cells per rank: 6 6 6 5
  Number of 2-cells per rank: 2 2 2 2
Labels:
  depth: 3 strata with value/size (0
    ↪(5), 1 (6), 2 (2))
  celltype: 3 strata with value/size
    ↪(0 (5), 1 (6), 3 (2))
  marker: 1 strata with value/size (1
    ↪(7))
  Face Sets: 1 strata with value/size
    ↪(1 (3))
```

对于较大的网格:

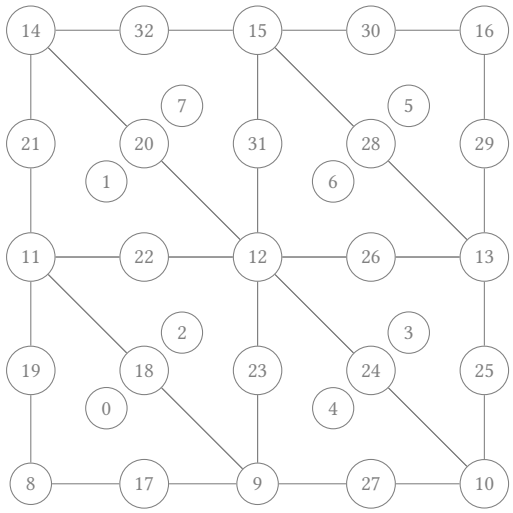
```
plexprogram -dm_plex_box_faces 4,4
```

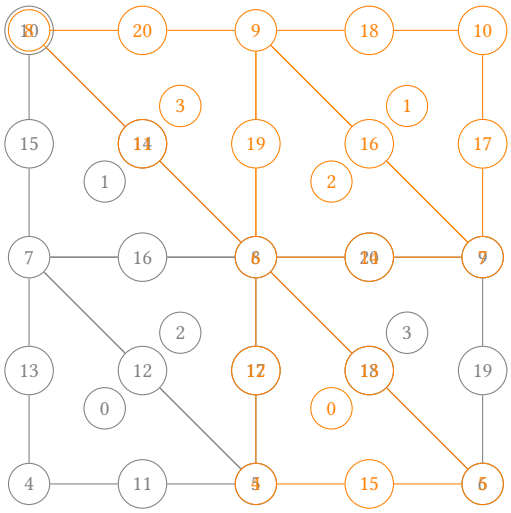
Graphics output from

```
plexprogram -dm_view draw -draw_pause 20
```




```
plexprogram -dm_view :outputfile.tex:ascii_latex \
            -dm_plex_view_scale 4
```





34.1.1 Matrix fromdmplex

遍历元素批次 (e)：遍历元素矩阵 e

ntries(f,fc,g,gc –> i,j: 遍历高斯积分点 (q): 生成 `u_q` 和 `gradU_q`（遍历场，Nb，Ncomp 的循环）

$$\texttt{elemMat[i,j]} \mathrel{+}= \psi_f^{fc}(q) g_{fc}^{(0)}{}_{,} \; {}_{gc}(u, \nabla u) \phi_g^{gc}(q)_{,,} + \psi_f^{fc}(q) \cdot g^{(1)} w_{fc\,gc} \; {}_{dg}(u, \nabla u) \nabla \phi_g^{gc}(q)_{,,} \; {}_{df}(u, \nabla u) \phi_g^{gc}(q)_{,,,} + \nabla \psi_f^{fc}(q) \cdot g_{fc\,gc}^{(2)} + \nabla \psi_f^{fc}(q) \cdot g_{fc\,gc\,df}^{(3)} \; {}_{dg}(u, \nabla u) \nabla \phi_g^{gc}(q)$$

```
// plexsphere.c
ierr = DMPlexGetDepthStratum(dm, 0, &vStart, &vEnd);
ierr = PetscSectionCreate(PetscObjectComm((PetscObject) dm), &s);
ierr = DMSetLocalSection(dm, s);
ierr = PetscSectionDestroy(&s);
```

```
ierr = PetscSectionSetNumFields(s, 1);
ierr = PetscSectionSetFieldComponents(s, 0, 1);ierr = PetscSectionSetChart(s, vStart, vEnd);
//  printf("start-end: %d -- %d\n",vStart,vEnd);
for (v = vStart; v < vEnd; ++v) {
  ierr = PetscSectionSetDof(s, v, 1);
  ierr = PetscSectionSetFieldDof(s, v, 0, 1);}
ierr = PetscSectionSetUp(s);
```

第 35 章

PETSc 求解器

PETSc 中最重要的活动可能是求解线性系统。这是通过一个求解器对象完成的：一个类为 `KSP` 的对象。（这代表 Krylov 空间求解器。）求解例程 `KSPSolve` 接受一个矩阵和一个右端项并给出解；然而，在调用它之前需要进行一定的设置。

求解线性系统有两种非常不同的方法：通过直接方法，本质上是高斯消元的变体；或者通过迭代方法，对解进行连续逼近。在 PETSc 中只有迭代方法。我们稍后会展示如何实现直接方法。PETSc 中的默认线性系统求解器是完全并行的，能够处理许多线性系统，但有许多设置和自定义选项可以使求解器适应您的具体问题。

35.1 KSP：线性系统求解器

35.1.1 数学背景

许多科学应用在某些阶段归结为求解线性方程组：

$$?_x : \ Ax = b$$

解决此问题的基础教科书方法是通过 LU 分解，也称为高斯消元：

$$LU \leftarrow A, \qquad Lz = b, \qquad Ux = z.$$

虽然 PETSc 支持此方法，但其基本设计是针对所谓的迭代求解方法。它们不是直接计算系统的解，而是计算一系列近似值，幸运的话，这些近似值会收敛到真实解：

while not converged $x_{i+1} \leftarrow f(x_i)$

迭代方法的有趣之处在于迭代步骤仅涉及矩阵 - 向量乘积：

while not converged
$r_i = Ax_i - b$
$x_{i+1} \leftarrow f(r_i)$

这个 *residual* 是决定是否停止迭代的关键。由于我们（显然）无法测量到真实解的距离，我们使用残差的大小作为代理测量。

Figure 35.1 KSPCreate

```
C:
PetscErrorCode KSPCreate(MPI_Comm comm,KSP *v);

Python:
ksp = PETSc.KSP()
ksp.create()
# or:
ksp = PETSc.KSP().create()
```

剩下需要了解的一点是迭代方法具有一个预处理器。在数学上，这相当于将线性系统转换为

$$M^{-1}Ax = M^{-1}b$$

因此理论上我们可以对变换后的矩阵和右侧向量进行迭代。然而，实际上我们在每次迭代中应用预处理器：

$$\begin{array}{l} r_i = Ax_i - b \\ \text{当未收敛时} \\ z_i = M^{-1}r_i \\ x_{i+1} \leftarrow f(z_i) \end{array}$$

在这个示意图中，我们没有具体说明 $f()$ 更新函数的性质。这里存在许多可能性；主要的选择是迭代方法类型，例如 “ 共轭梯度 ” 、 “ 广义最小残差 ” 或 “ 稳定双共轭梯度 ” 。（我们将在第 35.2 节中讨论直接求解器。）

量化收敛速度的问题很困难；参见 HPC 书籍，第 5.5.14 节。

35.1.2 求解器对象

首先我们创建一个 KSP 对象，它包含系数矩阵，以及各种参数，如期望的精度和特定方法的参数：KSPCreate (图 35.1)。

之后，基本场景是：

```
Vec rhs,sol;
KSP solver;
KSPCreate(comm,&solver);
KSPSetOperators(solver,A,A);
KSPSetFromOptions(solver);
KSPSolve(solver,rhs,sol);
KSPDestroy(&solver);
```

u使用各种默认设置。向量和矩阵必须进行一致的划分。 KSPSetOperators
c全部使用两个算子：一个是实际的系数矩阵，另一个是预处理器所用的算子
d派生自。在某些情况下，在这里指定不同的矩阵是有意义的。（您可以使用 KSPGetOperators。）调用 KSPSetFromOptions 几乎可以涵盖接下来讨论的所有设置。

KSP 对象有许多选项来控制它们，因此调用 KSPView （或使用命令行选项 -ksp_view ）以获取所有设置的列表是很方便的。

Figure 35.2 KSPSetTolerances

```
#include "petscksp.h"
PetscErrorCode  KSPSetTolerances
   (KSP ksp,PetscReal rtol,PetscReal abstol,PetscReal dtol,PetscInt maxits)

Logically Collective on ksp

Input Parameters:
ksp- the Krylov subspace contextrtol- the relative convergence tolerance,
 relative decrease in the(possibly preconditioned) residual norm
abstol- the absolute convergence tolerance absolute size of the
(possibly preconditioned) residual normdtol- the divergence tolerance,
 amount (possibly preconditioned)
residual norm can increase before KSPConvergedDefault() concludes that
the method is divergingmaxits- maximum number of iterations to use

Options Database Keys
-ksp_atol <abstol>- Sets abstol
-ksp_rtol <rtol>- Sets rtol
-ksp_divtol <dtol>- Sets dtol
-ksp_max_it <maxits>- Sets maxits
```

35.1.3 Tolerances

由于既不保证解的存在也不保证解的速度，迭代求解器需要满足一些容差:

- 残差已足够减少时的相对容差;
- 残差客观上很小时的绝对容差;
- 如果残差增长过多则停止迭代的发散容差; 以及
- 迭代次数的上限，无论过程可能仍在取得任何进展。

T这些容差通过 KSPSetTolerances (图 35.2) 设置，或通过选项 *-ksp_atol*, *-ksp_rtol*, *-ksp_divtol*, *-ksp_max_it*. 指定为 PETSC_DEFAULT 保持值不变。

在下一节中，我们将看到如何确定是哪一个容差导致求解器停止。

35.1.4 为什么我的求解器停止了? 它工作了吗?

在 KSPSolve 例程返回时，并不能保证系统已成功求解。因此，您需要调用 KSPGetConvergedReason (图 35.3) 来获取一个 KSPConvergedReason 参数，该参数指示求解器停止时的状态:

- 迭代可能已成功收敛; 这对应于 *reason*> 0 ;
- 迭代可能发散，或以其他方式失败: *reason*< 0 ;
- 或者迭代可能在达到最大迭代次数时停止，但仍在取得进展; *reason*$= 0$。

更多细节， KSPConvergedReasonView (版本 3.14 之前: KSPReasonView) 可以以可读形式打印出原因; 例如

```
KSPConvergedReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
// before 3.14:
KSPReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
```

图 35.

```
Input Parameter
ksp -the KSP context

Output Parameterreason -negative value indicates diverged,
 positive value converged,see KSPConvergedReason


Python:
r = KSP.getConvergedReason(self)
where r in PETSc.KSP.ConvergedReason
```

图 35.4 **KSPSetType**`#include "petscksp.h"`

```
PetscErrorCode  KSPSetType(KSP ksp, KSPType type)

Logically Collective on ksp

Input Parameters:
ksp : the Krylov space context
type : a known method
```

(这也可以通过 *-ksp_converged_reason* 命令行选项激活。)

在成功收敛的情况下，您可以使用 `KSPGetIterationNumber` 来报告进行了多少次迭代。

以下代码片段分析了一个 `KSP` 对象停止迭代的状态:

```
// shellvector.c
PetscInt its; KSPConvergedReason reason;
Vec Res; PetscReal norm;
ierr = KSPGetConvergedReason(Solve,&reason);
ierr = KSPConvergedReasonView(Solve,PETSC_VIEWER_STDOUT_WORLD);
if (reason<0) {
  PetscPrintf(comm,"Failure to converge: reason=%d\n",reason);
} else {
  ierr = KSPGetIterationNumber(Solve,&its);
  PetscPrintf(comm,"Number of iterations: %d\n",its);
}
```

35.1.5 迭代器的选择

有许多迭代方法，可能需要几次函数调用才能完全指定它们。基本例程是 `KSPSetType` (图 35.4)，或者使用选项 *-ksp_type*。

以下是一些值（完整列表见 *petscksp.h* ）：

图 35.5 KSPMatSolve

```
PetscErrorCode KSPMatSolv          e(KSP ksp, Mat B, Mat X)

Input Parameters
ksp - iterative context
B - block of right-hand sides

Output Parameter
X - block of solutions
```

- KSPCG：仅适用于对称正定系统。其工作量和存储成本在迭代次数上是常数。

  存在一些变体，如 KSPPIPECG，在数学上等价，但可能在大规模下性能更高。 g
  在大规模下。
- KSPGMRES：一种适用于非对称和不定系统的最小化方法。然而， o
  满足此理论属性需要存储完整的残差历史，以正交化每个计算的残差，这意味着存储需求是线性的，工作量是迭代次数的平方。因此，GMRES 总是以截断变体使用，定期重新启动正交化。重启长度可以通过例程 KSPGMRESSetRestart 或选项 *-ksp_gmres_restart* 设置。
- KSPBCGS：一种准最小化方法；比 GMRES 使用更少的内存。

根据迭代方法，可能有多个例程来调整其工作方式。特别是如果您仍在试验选择哪种方法，通过命令行选项指定这些选择可能更方便，而不是显式编码的例程。在这种情况下，调用一次 KSPSetFromOptions 就足以包含这些。

35.1.6 多个右端项

对于多个右端项的情况，使用 KSPMatSolve (图 35.5)。

35.1.7 预处理器

迭代求解器的另一个部分是预处理器。其数学背景见第 35.1.1 节。预处理器的作用是使系数矩阵的条件数更好，这将提高收敛速度；甚至可能没有合适的预处理器，求解器根本无法收敛。

35.1.7.1 背景

预处理器 M 满足 $M \approx A$ 的数学要求可以有两种形式：

1. 我们构造 A^{-1} 的显式近似；这被称为 稀疏近似逆。 2. 我们构造一个算子 M （通常以分解或其他隐式形式给出），使得 $M \approx A$，并且求解系统 $Mx = y$ 得到 x 可以相对快速完成。

In 在决定预处理器时，我们现在必须平衡以下因素 s.

1. 构造预处理器的成本是多少？这不应超过迭代方法解算时间的节省。 2. 应用预处理器每次迭代的成本是多少？显然，使用一个减少迭代次数但每次迭代成本增加更多的预处理器是没有意义的。

3. 许多预处理器具有参数设置，这使得这些考虑更加复杂：较低的参数值可能会产生一个应用成本低但对收敛性提升不大的预处理器，而较高的参数值则使应用成本增加，但减少迭代次数。

35.1.7.2 使用

与大多数其他 PETSc 对象类型不同， `PC` 对象通常不会被显式创建。相反，它作为 `KSP` 对象的一部分被创建，并且可以从中获取。

```
PC prec;
KSPGetPC(solver,&prec);
PCSetType(prec,PCILU);
```

除了设置预处理器的类型外，还有各种特定类型的例程用于设置各种参数。其中一些可能相当繁琐，通过命令行选项设置它们会更方便。

35.1.7.3 Types

方法	PCType	选项数据库名称
Jacobi	PCJACOBI	jacobi
块 Jacobi	PCBJACOBI	bjacobi
SOR （及 SSOR ）	PCSOR	sor
带 Eisenstat 技巧的 SOR	PCEISENSTAT	eisenstat
不完全 Cholesky	PCICC	icc
不完全 LU	PCILU	ilu
加法 Schwarz	PCASM	asm
广义加法 Schwarz	PCGASM	gasm
代数多重网格	PCGAMG	gamg
平衡域分解	PCBDDC	bddc
通过约束 线性求解器		
使用迭代方法	PCKSP	ksp
预处理器的组合	PCCOMPOSITE	composite
LU	PCLU	lu
Cholesky	PCCHOLESKY	cholesky
无预处理	PCNONE	none
用户定义 PC 的 Shell	PCSHELL	shell

以下是一些可用的预处理器类型。

The *Hypre* 包（需要在配置时安装）本身包含多个预处理器。在代码中，可以将预处理器设置为 `PCHYPRE`，并使用 `PCHYPRESetType` 设置为以下之一：euclid 、 pilut 、 parasails 、 boomeramg 、 ams 、 ads。然而，由于这些预处理器本身具有选项，通常更方便使用命令行选项：

```
-pc_type hypre -pc_hypre_type xxxx
```

35.1.7.3.1 稀疏近似逆稀疏矩阵的逆（至少是那些来自 PDE 的矩阵）通常是稠密的。因此，我们的目标是构造一个稀疏近似逆。

PETSc 提供了两种此类预处理器，二者都需要一个外部包。

- PCSPAI 这是一种只能在单处理器运行中使用的预处理器，或者作为块预处理器中的局部求解器；参见章节 35.1.7.3.3。

- 作为 PCHYPRE 包的一部分，提供了并行变体 *parasails*。

```
-pc_type hypre -pc_hypre_type parasails
```

35.1.7.3.2 **不完全分解** 源自 PDE 问题的矩阵的 *LU* 分解存在若干实际问题：

- 它占用的存储空间（显著地）多于系数矩阵，且
- 相应地需要更多时间来应用。

F例如，对于一个三维 PDE 中 N 变量，系数矩阵可能占用存储空间 $7N$，而 t该 *LU* 分解需要 $O(N^{5/3})$。

F因此，通常不完全的 *LU* 分解更受欢迎。

- PETSc 本身有一个 PCILU 类型，但这只能顺序使用。这听起来像是一个限制，但在并行中它仍然可以作为块方法中的子域求解器使用；见章节 35.1.7.3.3。
- 作为 *Hypre* 的一部分，*pilut* 是一个并行 ILU。

ILU 类型有许多选项，例如 PCFactorSetLevels（选项 *-pc_factor_levels*），它设置允许的填充级别数。

35.1.7.3.3 **块方法** 某些预条件器似乎几乎本质上是顺序的。例如，ILU 解在变量之间是顺序的。存在一定程度的并行性，但很难挖掘。

退一步讲，平行预条件器的问题之一在于矩阵中的跨进程连接。如果没有这些连接，我们就可以在每个进程上独立地求解线性系统。既然预条件器本身就是一个近似解，忽略这些连接只会引入额外的近似度。

有两种预条件器基于这个概念运行：

- PCBJACOBI：块 Jacobi。这里每个进程局部求解由耦合本地变量的矩阵系数组成的系统。实际上，每个进程在一个子域上求解一个独立的系统。接下来的问题是子域上使用什么求解器。这里可以使用任何预条件器，特别是那些仅存在于顺序版本中的预条件器。在代码中指定所有这些会很繁琐，通常通过命令行选项指定这样复杂的求解器更为简单：

  ```
  -pc_type jacobi -sub_ksp_type preonly \-sub_pc_type ilu -sub_pc_factor_levels 1
  ```

  （注意这里也提到了一个 sub_ksp：子域求解器实际上是一个 KSP 对象。通过将其类型设置为 *preonly*，我们声明求解器应仅由应用其预条件器组成。）块 Jacobi 预条件器在渐近意义上只能将系统求解加速与子域数量相关的因子，但在实践中它可能非常有价值。

- PCASM：加法 Schwarz 方法。这里每个进程基于局部变量以及与邻近进程的一层（或几层）连接，局部求解一个稍大的系统。实际上，进程在重叠子域上求解系统。该预处理器在渐近意义上可以将迭代次数减少到 $O(1)$，但这需要在子域上精确求解，且在实际中可能无法实现。

图 35.1 以矩阵和子域的形式说明了这些预处理器。

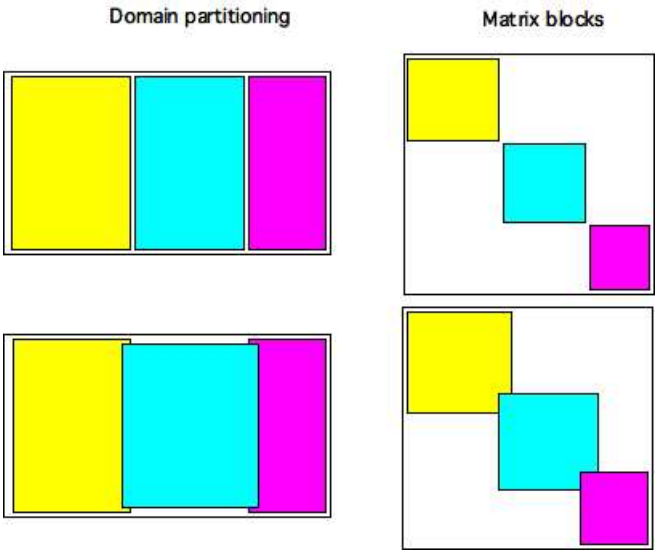


图 35.1：块 Jacobi 和加性 Schwarz 预处理器示意图：左侧为域和子域，右侧为对应的子矩阵

35.1.7.3.4 多重网格预处理器

- PETSc 内置了一种代数多重网格（AMG）类型：`PCGAMG`;
- 外部包 *Hypre* 和 *ML* 具有 AMG 方法。
- 还有一种通用多重网格（MG）类型：`PCMG`。

35.1.7.3.5 场分割预条件器

背景请参见章节 32.4.8。

练习 35.1. 示例代码 `ksp.c` 生成一个五点矩阵，可能是非对称的，定义在单位正方形上。你的任务是探索不同求解器在该系数矩阵线性系统上的收敛行为。

示例代码接受两个命令行参数：

- `-n 123` 设置域大小，这意味着矩阵大小将是该值的平方；
- `-unsymmetry .5` 向矩阵引入一个斜对称组件。

调查以下内容：

- 一些迭代方法，例如共轭梯度法（CG），仅在数学上定义于对称（且正定）矩阵。迭代方法实际上对非对称性的容忍度如何？
- 迭代次数有时可以证明依赖于矩阵的条件数，而条件数本身与矩阵的大小有关。你能找到矩阵大小与迭代次数之间的关系吗？
- 更复杂的迭代方法（例如，增加 GMRES 的重启长度）或更复杂的预处理器（例如，在 ILU 预处理器中使用更多填充级别），可能会导致更少的迭代次数。（实际上是这样吗？）但这不一定会带来更快的求解时间，因为每次迭代的开销更大。

有关背景以及各种具体子章节，参见 35.1.1 章节。

35.1.7.3.6 Shell 预条件器 你已经看到，在迭代方法中，系数矩阵可以操作性地表示为一个 *shellmatrix*；参见章节 32.4.7。类似地，预条件矩阵也可以通过指定类型 PCSHELL 来操作性地指定。

这需要通过 PCShellSetApply 指定应用例程：

```
PCShellSetApply(PC pc,PetscErrorCode (*apply)(PC,Vec,Vec));
```

and probably specification of a context pointer through PCShellSetContext:

```
PCShellSetContext(PC pc,void *ctx);
```

应用函数随后通过 PCShellGetContext 获取此上下文：

```
PCShellGetContext(PC pc,void **ctx);
```

如果 shell 预条件器需要设置，可以通过 PCShellSetSetUp 指定一个设置例程：

```
PCShellSetSetUp(PC pc,PetscErrorCode (*setup)(PC));
```

35.1.7.3.7 组合预处理器 可以将预处理器与 PCCOMPOSITE 结合使用

```
PCSetType(pc,PCCOMPOSITE);
PCCompositeAddPC(pc,type1);
PCCompositeAddPC(pc,type2);
```

默认情况下，预处理器是以加法方式应用的；对于乘法应用

```
PCCompositeSetType(PC pc,PCCompositeType PC_COMPOSITE_MULTIPLICATIVE);
```

35.1.8 自定义：监控和收敛测试

PETSc 求解器可以对用户函数执行各种 *callback*。

35.1.8.1 收敛性测试

例如，您可以使用 KSPSetConvergenceTes 设置您自己的收敛性测试 t.

```
KSPSetConvergenceTest
   (KSP ksp,
    PetscErrorCode (*test)(
        KSP ksp,PetscInt it,PetscReal rnorm,
          KSPConvergedReason *reason,void *ctx),
    void *ctx,PetscErrorCode (*destroy)(void *c     tx));
```

This routines accepts

- 自定义停止测试函数，
- 一个 ‘context’ void 指针，用于向测试器传递信息，和
- 可选的上下文信息自定义析构函数。

默认情况下，PETSc 的行为就像此函数已被调用，参数为 KSPConvergedDefault 。

35.1.8.2 收敛监控

还有一个用于监控每次迭代的回调函数。它可以通过 `KSPMonitorSet` 设置。

```
KSPMonitorSet
   (KSP ksp,
    PetscErrorCode (*mon)(
        KSP ksp,PetscInt it,PetscReal rnorm,void *ctx),
    void *ctx,PetscErrorCode (*mondestroy)(void**));
```

默认情况下没有设置监控器，这意味着迭代过程在没有输出的情况下运行。选项 `-ksp_monitor` 激活残差范数的打印。这相当于将 `KSPMonitorDefault` 设置为监控器。

这实际上输出的是残差的“预条件范数”，它不是 L2 范数，而是 $r^t M^{-1} r$ 的平方根，这是在迭代过程中计算的一个量。指定 `KSPMonitorTrueResidualNorm`（对应选项 `-ksp_monitor_true_residual`）作为监控器会打印实际范数 $\sqrt{r^t r}$。然而，计算这个需要额外的计算，因为这个量通常不会被计算。

35.1.8.3 辅助例程

`KSPGetSolution` `KSPGetRhs` `KSPBuildSolution` `KSPBuildResidual`

```
KSPGetSolution(KSP ksp,Vec *x);
KSPGetRhs(KSP ksp,Vec *rhs);
KSPBuildSolution(KSP ksp,Vec w,Vec *v);KSPBuildResidual(KSP ksp,Vec t,Vec w,Vec *v);
```

35.2 直接求解器

PETSc 对直接求解器（即 LU 分解的变体）有一定支持。在顺序环境中，`PCLU`
p预处理器可以用于此：直接求解器等同于在一次预处理器应用后停止的迭代方法。
d这可以通过指定类型为 `KSPPREONLY` 的 KSP 来强制执行。

分布式直接求解器更为复杂。PETSc 在其基础代码中没有实现此功能，但通过配置 PETSc 使用 *scalapack* 库后即可使用。

您需要指定哪个包提供 LU 分解：

```
PCFactorSetMatSolverType(pc, MatSolverType solver )
```

其中 solver 变量的类型为 `MatSolverType`，并且在源码中指定时可以是 *`MATSOLVERMUMS`* 等。

```
// direct.c
PetscCall( KSPCreate(comm,&Solver) );
PetscCall( KSPSetOperators(Solver,A,A) );
PetscCall( KSPSetType(Solver,KSPPREONLY) );
{
  PC Prec;
  PetscCall( KSPGetPC(Solver,&Prec) );
  PetscCall( PCSetType(Prec,PCLU) );
  PetscCall( PCFactorSetMatSolverType(Prec,MATSOLVERMUMPS) );
}
```

Figure 35.6 **KSPSetFromOptions**

```
Synopsis

#include "petscksp.h"
PetscErrorCode  KSPSetFromOptions(KSP ksp)

Collective on ksp

Input Parameters
ksp - the Krylov space context
```

如命令行中指定

`yourprog -ksp_type preonly -pc_type lu -pc_factor_mat_solver_type mumps` 可选项有 mumps、superlu、 umfpack 或其他多个包。请注意，这些包的可用性取决于 PETSc 在您的系统上的安装方式。

35.3 通过命令行选项进行控制

从上面你可能会觉得设置一个 PETSc 线性系统和求解器需要调用很多函数。如果你想尝试不同的求解器，是不是意味着你必须编辑大量代码？幸运的是，有一种更简单的方法。如果你调用例程 `KSPSetFromOptions`（图 35.6）并以 *`solver`* 作为参数，PETSc 会查看你的命令行选项，并在定义求解器时考虑这些选项。因此，你可以省略在源代码中设置选项，或者用这种方式快速尝试不同的可能性。示例：

```
myprogram -ksp_max_it 200 \
    -ksp_type gmres -ksp_type_gmres_restart 20  \
    -pc_type ilu -pc_type_ilu_levels 3
```

第 36 章

PETSC 非线性求解器

36.1 非线性系统

非线性系统求解意味着找到一般非线性函数的零点，即：

$$\underset{x}{?}: \ f(x) = 0$$

带有 $f: \mathbb{R}^n - \mathbb{R}^n$。在线性函数的特殊情况下，

$$f(x) = Ax - b,$$

我们通过第 35 章中的任一方法来求解。

一般情况可以通过多种方法解决，最主要的是 *Newton'smethod*，它通过迭代实现

$$x_{n+1} = x_n - F(x_n)^{-1} f(x_n)$$

其中 F 是 *Hessian* $F_{ij} = \partial f_i / \partial x_j$。

你会看到需要指定两个依赖于具体问题的函数：目标函数本身，以及它的 Hessian。

36.1.1 基本设置

PETSc 非线性求解器对象的类型是 *SNES*：‘简单非线性方程求解器’。与线性求解器类似，我们在通信器上创建该求解器，设置其类型，整合选项，并调用求解例程 *SNESSolve*(图 36.1)：

```
Vec value_vector,solution_vector;
/* vector creation code missing */
SNES solver;
SNESCreate( comm,&solver );
SNESSetFunction( solver,value_vector,formfunction, NULL );
SNESSetFromOptions( solver );
SNESSolve( solver,NULL,solution_vector );
```

该函数的类型为

```
PetscErrorCode formfunction(SNES,Vec,Vec,void*)
```

Figure 36.1 **SNESSolve**

```
#include "petscsnes.h"
PetscErrorCode  SNESSolve(SNES snes,Vec b,Vec x)

Collective on SNES

Input Parameters
snes - the SNES c
    ontextb     - the constant part of the equation F(x) = b,
 or NULL to use zero.x     - the solution vector.
```

where the parameters are:

- 求解器对象，以便您可以访问其内部参数
- 用于评估函数的 x 值
- 给定输入的结果向量 $f(x)$
- 用于进一步应用特定信息的上下文指针。

Example:

```
PetscErrorCode evaluation_function( SNES solver,Vec x,Vec fx, void *ctx ) {
  const PetscReal *x_array;
  PetscReal *fx_array;
  VecGetArrayRead(fx,&fx_array);
  VecGetArray(x,&x_array);
  for (int i=0; i<localsize; i++)
    fx_array[i] = pointfunction( x_array[i] );
  VecRestoreArrayRead(fx,&fx_array);
  VecRestoreArray(x,&x_array);
};
```

将上述内容与介绍性描述进行比较，你会发现这里没有指定 Hessian。如果你指示 PETSc 通过有限差分来近似它，则可以省略解析 Hessian：

$$H(x)y \approx \frac{f(x+hy) - f(x)}{h}$$

使用 h 某些有限差分。命令行选项 `-snes_fd` 强制使用此有限差分近似。然而，这可能导致大量的函数评估。选项 `-snes_fd_color` 对变量应用着色，从而大幅减少函数评估的次数。

如果你能构造解析的 Jacobian / Hessian，可以用 `SNESSetJacobian` (图 36.2) 指定，其中 Jacobian 是类型为 `SNESJacobianFunction` (图 36.3) 的函数。

指定雅可比矩阵：

```
Mat J;
ierr = MatCreate(comm,&J); CHKERRQ(ierr);
ierr = MatSetType(J,MATSEQDENSE); CHKERRQ(ierr);
ierr = MatSetSizes(J,n,n,N,N); CHKERRQ(ierr);
ierr = MatSetUp(J); CHKERRQ(ierr);
ierr = SNESSetJacobian(solver,J,J,&Jacobian,NULL); CHKERRQ(ierr);
```

Figure 36.2 **SNESSetJacobian**

```
#include "petscsnes.h"
PetscErrorCode  SNESSetJacobian(SNES snes,Mat Amat,Mat Pmat,PetscErrorCode (*J)(SNES,Vec,Mat,Mat,void*),void *ctx)

Logically Collective on SNES

Input Parameters
snes - the SNES c
    ontextAmat - the matrix that defines the (approximate) Jacobian
Pmat - the matrix to be used in constructing the preconditioner, usually the same as Amat.
J    - Jacobian evaluation routine (if NULL then SNES retains any previously set value)
ctx  - [optional] user-defined context for private data for the Jacobian evaluation routine
```

图 36.3 **SNESJacobianFunction**

```
#include "petscsnes.h"
PetscErrorCode SNESJacobianFunction(SNES snes,Vec x,Mat Amat,Mat Pmat,void *ctx);

Collective on snes

Input Parametersx   - input vector,
 the Jacobian is to be computed at this value
ctx - [optional] user-defined Jacobian context

Output Parameters
Amat - the matrix
    that defines the (approximate) JacobianPmat - the matrix to be used in constructing the preconditioner,
 usually the same as Amat.
```

36.2 时间步进

对于情况

$$u_t = G(t,u)$$

调用 *TSSetRHSFunction*.

```
#include "petscts.h"
PetscErrorCode TSSetRHSFunction
   (TS ts,Vec r,
    PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,void*),
    void *ctx);
```

对于隐式情况

$$F(t,u,u_t) = 0$$

调用 *TSSetIFunction*

```
#include "petscts.h"
PetscErrorCode TSSetIFunction
   (TS ts,Vec r,TSIFunction f,void *ctx)
```

第 37 章

PETSc GPU 支持

37.1 使用 GPU 的安装

PETSc 可以通过选项进行配置

`--with-cuda  --with-cudac=nvcc?`

您可以通过以下方式测试 CUDA 是否存在:

```
// cudainstalled.c
#ifndef PETSC_HAVE_CUDA
#error "CUDA is not installed in this version of PETSC"
#endif
```

一些 GPU 可以通过 *GPUDirect* 远程内存访问（ RMA ）直接连接到网络，从而支持 MPI。如果不能，则使用此运行时选项:

`-use_gpu_aware_mpi 0`

更方便的是，将此添加到您的 `.petscrc` 文件；章节 38.3.3。

37.2 GPU 设置

GPU 需要初始化。这可以在创建 GPU 对象时隐式完成，或者通过 *PetscDeviceInitialize* 显式完成。（ PETSc 版本在 PETSc-3.17 之前有一个显式例程 `PetscCUDAInitialize`。）

```
// cudainit.c
PetscDeviceType cuda = PETSC_DEVICE_CUDA;
ierr = PetscDeviceInitialize(cuda);
PetscBool has_cuda;
has_cuda = PetscDeviceInitialized(cuda);
```

37.3 分布式对象

诸如矩阵和向量的对象需要使用 CUDA 类型显式创建。之后，大多数 PETSc 调用与 GPU 的存在无关。

如果需要测试，有一个 C 预处理器 (CPP) 宏 *PETSC_HAVE_CUDA*。

37.3.1 向量

类似于之前的向量创建，有特定的创建调用 VecCreateSeqCUDA,VecCreateMPICUDAWithArray，或者类型可以在 VecSetType 中设置：

```
// kspcu.c
#ifdef PETSC_HAVE_CUDA
  ierr = VecCreateMPICUDA(comm,localsize,PETSC_DECIDE,&Rhs);
#else
  ierr = VecCreateMPI(comm,localsize,PETSC_DECIDE,&Rhs);
#endif
```

类型 VECCUDA 是顺序的或并行的，取决于运行；具体类型有 VECSEQCUDA, VECMPICUDA。

37.3.2 矩阵

```
ierr = MatCreate(comm,&A);
#ifdef PETSC_HAVE_CUDA
ierr = MatSetType(A,MATMPIAIJCUSPARSE);
#else
ierr = MatSetType(A,MATMPIAIJ);
#endif
```

稠密矩阵可以通过特定调用创建 MatCreateDenseCUDA, MatCreateSeqDenseCUDA, 或通过设置类型 MATDENSECUDA, *MATSEQDENSECUDA*, MATMPIDENSECUDA.

稀疏矩阵： MATAIJCUSPARSE 这取决于程序如何启动，可以是顺序的或分布式的。具体类型有： *MATMPIAIJCUSPARSE*, *MATSEQAIJCUSPARSE*.

37.3.3 数组访问

各种 ‘数组’ 操作，例如 MatDenseCUDAGetArray, VecCUDAGetArray,

设置 PetscMalloc 使用 GPU： PetscMallocSetCUDAHost，并通过 PetscMallocResetCUDAHost 切换回去。

37.4 其他

CPU 和 GPU 的内存不是一致的。这意味着诸如 PetscMalloc1 之类的例程不能立即用于 GPU 分配。使用例程 PetscMallocSetCUDAHost 和 PetscMallocResetCUDAHost 来切换分配器到 GPU 内存及返回。

```
// cudamatself.c
Mat cuda_matrix;
PetscScalar *matdata;
ierr = PetscMallocSetCUDAHost();
ierr = PetscMalloc1(global_size*global_size,&matdata);
ierr = PetscMallocResetCUDAHost();
ierr = MatCreateDenseCUDA
  (comm,global_size,global_size,global_size,
  global_size,matdata,&cuda_matrix);
```

第 38 章

PETSc 工具

38.1 错误检查和调试

38.1.1 调试模式

安装过程中（参见章节 31.3），有一个开启调试模式的选项。开启调试的安装：

- 对数值或数组索引进行更多的运行时检查；
- 当你插入 CHKMEMQ 宏（章节 38.1.3）时，会进行内存分析；
- 宏 *PETSC_USE_DEBUG* 已设置为 1。

38.1.2 错误代码

PETSc 执行大量的运行时错误检查。其中一些用于内部一致性，但它也可以检测某些数学错误。为了便于错误报告，使用了以下方案。

每个 PETSc 调用都会返回一个错误代码；通常成功时为零，针对各种情况则为非零。你应该将每个此类函数调用包装在 *PetscCall* 宏中：

```
PetscCall( SomePetscRoutine( arguments ) );
```

(在许多代码中你可能会看到一个宏 CHKERRQ ；这是 PETSc-3.18 之前的机制；参见章节 38.1.2.2。) 这个宏检测任何错误代码，报告它，并退出当前例程。

为了获得良好的回溯，将任何子程序的可执行部分包围在 PetscFunctionBeginUser 和 PetscFunctionReturn 之间，后者以返回值作为参数。（例程 PetscFunctionBegin 也这样做，但应仅用于 PETSc 库例程。）

38.1.2.1 错误抛出

你可以使用可变参数函数 SETERRQ (图 38.1) 来实现你自己的错误返回。（在 PETSc-3.17 之前，有单独的函数 SETERRQ1， SETERRQ2，等等。）

示例。我们编写一个设置错误的例程：

```
// backtrace.c
PetscErrorCode this_function_bombs() {
  PetscFunctionBegin;
  SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
```

图 38.1 SETERRQ

```
#include <petscsys.h>
PetscErrorCode SETERRQ (MPI_Comm comm,PetscErrorCode ierr,char *message)
PetscErrorCode SETERRQ1(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1)
PetscErrorCode SETERRQ2(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1,arg2)
PetscErrorCode SETERRQ3(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1,arg2,arg3)

Input Parameters:
comm - A communicator, so that the error can be collectiveierr - nonzero error code,
 see the list of standard error codes in include/petscerror.h
message - error message in the printf formatarg1,arg2,
arg3 - argument (for example an integer, string or double)
```

```
  PetscFunctionReturn(0);
}
```

运行此操作将在进程零中输出

```
[0]PETSC ERROR: We cannot go on like this
[0]PETSC ERROR: See https://www.mcs.anl.gov/petsc/documentation/faq.html for trouble shooting.
[0]PETSC ERROR: Petsc Release Version 3.12.2, Nov, 2019 年 22
[0]PETSC ERROR: backtrace on a [computer name][0]PETSC ERROR: Configure options [all options]
[0]PETSC ERROR: #1 this_function_bombs() line 20 in backtrace.c
[0]PETSC ERROR: #2 main() line 30 in backtrace.c
```

Fortran note 29: Backtrace on error. 在 Fortran 中，backtrace 并不完全是　s elegant.

```
!! backtrace.F90
  Subroutine this_function_bombs(ierr)
    implicit none
    integer,intent(out) :: ierr

    SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this")
    ierr = -1

  end Subroutine this_function_bombs
```

```
[0]PETSC ERROR: ----- Error Message --------------------------------
[0]PETSC ERROR: We cannot go on like this
[....]
[0]PETSC ERROR: #1 User provided function() line 0 in User file
```

备注 42 在此示例中，使用 *PETSC_COMM_SELF* 表示该错误是在某个进程上单独产生的；仅当相同错误会在所有地方被检测到时，才使用 *PETSC_COMM_WORLD* 。

练习 38.1. 查找 SETERRQ1 的定义。编写一个计算平方根的例程，其用法如下：

```
x = 1.5; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
x = -2.6; ierr = square_root(x,&rootx); CHKERRQ(ierr);
```

```
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
```

这应该输出：`Root of 1.500000 is 1.224745`

```
[0]PETSC ERROR: ----- Error Message --------------------------------------------------------------
[0]PETSC ERROR: Cannot compute the root of -2.600000[...]
[0]PETSC ERROR: #1 square_root() line 23 in root.c
[0]PETSC ERROR: #2 main() line 39 in root.c
```

38.1.2.2 传统错误检查

In PETSc versions pre-PETSc-3.18, errors were handled slightly differently.

1. 每个 PETSc 例程都是返回类型为 `PetscErrorCode` 的函数。2. 调用宏 `CHKERRQ` 对错误代码进行处理时，会打印错误信息并终止当前例程。递归地，这会给出错误发生位置的回溯。

```
PetscErrorCode ierr;
ierr = AnyPetscRoutine( arguments ); CHKERRQ(ierr);
```

3. 其他错误检查宏有 *CHKERRABORT*，它会立即中止，以及 *CHKERRMPI*。

Fortran note 30: Errorcode handling. 在主程序中，使用 `CHKERRA` 和 `SETERRA`。还要注意这些错误“命令”是宏，展开后可能会干扰 *Fortran* 行长度，因此它们应仅在 `.F90` 文件中使用。

C++ note 27: Exception handling. The macro *CHCKERCXX* handles exceptions.

38.1.3 Memory corruption

PETSc 有自己的内存管理（章节 38.5），这有助于发现内存破坏错误。宏 `CHKMEMQ` (`CHKMEMA` （在 void 函数中）检查所有由 PETSc 分配的内存，无论是内部分配还是通过分配例程分配，是否存在破坏。在代码中适当使用此宏可以在内存问题导致 *segfault* 之前检测到它们。

此测试仅在提供命令行参数 `-malloc_debug` （`-malloc_test` 调试模式）时执行，因此对生产运行没有开销。

38.1.3.1 Valgrind

Valgrind 的输出相当冗长。要限制在 valgrind 下运行的进程数量：

```
mpiexec -n 3 valgrind --track-origins=yes ./app -args : -n 5 ./app -args
```

38.2 Program output

PETSc 有多种机制来导出或可视化程序数据。我们将在这里考虑几种可能性。

图 38.2 PetscPrintfC:

```
PetscErrorCode  PetscPrintf(MPI_Comm comm,const char format[],...)

Fortran:
PetscPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

Python:
PETSc.Sys.Print(type cls, *args, **kwargs)
kwargs:
comm : communicator object
```

图 38.3 PetscSynchronizedPrintfC:

```
PetscErrorCode  PetscSynchronizedPrintf(
MPI_Comm comm,const char format[],...)

Fortran:
PetscSynchronizedPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

python:
PETSc.Sys.syncPrint(type cls, *args, **kargs)
kwargs:comm : communicator object
flush : if True, do synchronizedFlush
other keyword args as for python3 print function
```

38.2.1 屏幕 I/O

并行打印屏幕输出是棘手的。如果两个进程几乎同时执行打印语句，屏幕上它们出现的顺序无法保证。（即使尝试让它们一个接一个地打印，也可能无法得到正确的顺序。）此外，两个进程的多行打印操作的行可能会交错显示在屏幕上。

38.2.1.1 printf 替代

PETSc 有两个例程可以解决这个问题。首先，通常打印的信息在所有进程中是相同的，因此只需一个进程，例如进程 0，进行打印即可。这是通过 `PetscPrintf` (图 38.2) 完成的。

如果所有进程都需要打印，可以使用 `PetscSynchronizedPrintf` (图 38.3) 强制输出按进程顺序出现。

为了确保输出正确地从所有系统缓冲区刷新，使用 `PetscSynchronizedFlush` (图 38.4)，普通屏幕输出则使用 `stdout` 用于文件。

Fortran 注释 *31*：打印字符串构造。 Fortran 没有 C 语言中的可变参数机制，因此你只能使用 `PetscPrintf` 在你用 `Write` 语句构造的缓冲区上：

```
Character*80 :: message
write(message,10) xnorm,ynorm10 format("Norm x: ",f6.3,", y

    : ",f6.3,"\n")call PetscPrintf(comm,message,ierr)
```

Figure 38.4 **PetscSynchronizedFlush**

```
C:
PetscErrorCode  PetscSynchronizedFlush(MPI_Comm comm,FILE *fd)
fd : output file pointer, needs to be valid on process zero

Fortran:
PetscSynchronizedFlush(comm,fd,err)
Integer :: comm
fd is usually PETSC_STDOUT
PetscErrorCode :: err

python:
PETSc.Sys.syncFlush(type cls, comm=None)
```

图 38.5 **PetscViewerRead**Synopsis

```
#include "petscviewer.h"
PetscErrorCode  PetscViewerRead(PetscViewer viewer, void *data, PetscInt num, PetscInt *count, PetscDataType dtype)

Collective

Input Parameters
viewer  - The viewer
data  - Location to write the data
num  - Number of items of data to read
datatype  - Type of data to read

Output Parameters
count -number of items of data actually read, or NULL
```

Fortrannote 32: 打印和换行。 Fortran 调用只是围绕 C 例程的包装，因此你可以使用 `\n` 换行符在 Fortran 字符串参数中 `PetscPrintf`。

要刷新（ flush ）的文件通常是 `PETSC_STDOUT`。

Python 注释 45：*Petsc print 和 python print*。 由于打印例程使用了 python `print` 调用，它们会自动包含尾随换行符。你不必像在 C 调用中那样指定它。

38.2.1.2 scanf 替代

在 Petsc 中使用 `scanf` 是棘手的，因为整数和实数的大小可能因安装而异。相反，使用 `PetscViewerRead` (图 38.5)，它以 `PetscDataType` 为操作单位。

38.2.2 查看器

I为了导出 PETSc 矩阵或向量数据结构，有一个 `PetscViewer` 对象类型。这是一个相当通用的
c查看概念：它涵盖了屏幕上的 ascii 输出、文件的二进制转储，或与正在运行的
MMatlab 进程的通信。诸如 `MatView` 或 `KSPView` 的调用接受一个 `PetscViewer` 参数。

在适用的情况下，也有一个逆向的 “ 加载 ” 操作。有关向量，请参见第 32.3.5 节。

一些 viewer 是预定义的，例如 PETSC_VIEWER_STDOUT_WORLD 用于 ascii 渲染到标准输出。（在 C 中，指定零或 NULL 也使用此默认 viewer；Fortran 使用 PETSC_NULL_VIEWER。）

38.2.2.1 Viewer 类型

对于诸如转储到文件等操作，您首先需要使用 PetscViewerCreate 创建 viewer 并用 PetscViewerSetType 设置其类型。

```
PetscViewerCreate(comm,&viewer);
PetscViewerSetType(viewer,PETSCVIEWERBINARY);
```

流行的类型包括 PETSCVIEWERASCII, PETSCVIEWERBINARY, PETSCVIEWERSTRING, PETSCVIEWERDRAW,PETSCVIEWERSOCKET, PETSCVIEWERHDF5, PETSCVIEWERVTK；完整列表可见于 include/petscviewer.h。

38.2.2.2 Viewer formats

Viewers 可以通过使用 PetscViewerPushForma 来接受进一步的格式规范 t:

```
PetscViewerPushFormat
   (PETSC_VIEWER_STDOUT_WORLD,
    PETSC_VIEWER_ASCII_INFO_DETAIL);
```

随后是相应的 PetscViewerPopFormat

Python 注释 *46*：*HDF5* 文件生成。

```
## hdf5.py
file_name = "hdf5.dat"
viewer = PETSc.Viewer().createHDF5(file_name, 'w', comm)
x.view(viewer)
viewer = PETSc.Viewer().createHDF5(file_name, 'r', comm)
x.load(viewer)
```

38.2.2.3 观察器的命令行选项

Petsc 对象观察器可以通过诸如 MatView 的调用来激活，但通常通过命令行选项更为方便，例如 *-mat_view*, *-vec_view*, 或 *-ksp_view*。默认情况下，这些输出到 stdout，格式为 ascii，但这可以通过进一步的选项值来控制：

```
program -mat_view binary:matrix.dat
```

其中 binary 强制进行二进制转储（默认是 ascii），并且显式给出文件名。

二进制转储可能不支持所有数据类型，特别是 DM。对于这种情况，请执行

```
program -dm_view draw \
    -draw_pause 20
```

这会弹出一个 *X11* 窗口，持续显示指定的 pause 时间。

如果需要在特定位置触发查看器，可以使用诸如 VecViewFromOptions 的调用。这些例程都有类似的调用顺序：

```
#include "petscsys.h"
PetscErrorCode  PetscObjectViewFromOptions(PetscObject obj,PetscObject bobj,const char
    ↪optionname[])
PetscErrorCode  VecViewFromOptions(Vec A,PetscObject obj,const char name[])
```

AOViewFromOptions, DMViewFromOptions, ISViewFromOptions, ISLocalToGlobalMappingViewFromOptions, KSPConvergedReasonViewFromOptions, KSPViewFromOptions, MatPartitioningViewFromOptions, MatCoarsenViewFromOptions, MatViewFromOptions, PetscObjectViewFromOptions, PetscPartitionerViewFromOptions, PetscDrawViewFromOptions, PetscRandomViewFromOptions, PetscDualSpaceViewFromOptions, PetscSFViewFromOptions, PetscFEViewFromOptions,PetscFVViewFromOptions, PetscSectionViewFromOptions, PCViewFromOptions, PetscSpaceViewFromOptions,PFViewFromOptions, PetscLimiterViewFromOptions, PetscLogViewFromOptions, PetscDSViewFromOptions, PetscViewerViewFromOptions, SNESConvergedReasonViewFromOptions, SNESViewFromOptions, TSTrajectoryViewFromOptions, TSViewFromOptions, TaoLineSearchViewFromOptions, TaoViewFromOptions, VecViewFromOptions, VecScatterViewFromOptions,

38.2.2.4 命名对象

一个有助于查看的功能是给对象命名：当查看该对象时，将显示该名称。

```
Vec i_local;
ierr = VecCreate(comm,&i_local); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)i_local,"space local"); CHKERRQ(ierr);
```

给予：

```
Vec Object: space local 4 MPI processes
type: mpiProcess [0]
[ ... et cetera ... ]
```

38.3 命令行选项

PETSc 有大量的命令行选项，我们大多数将在后面讨论。现在我们只提到 *-log_summary*，它将打印出各种例程所花费时间的分析。要解析这些选项，必须传递 `argc,argv` 到 `PetscInitialize` 调用中。

38.3.1 添加你自己的选项

您可以向程序添加自定义命令行选项。各种例程如 `PetscOptionsGetInt` 会扫描命令行选项并相应地设置参数。例如，

```
// ksp.c
PetscBool flag;
PetscInt domain_size = 100;
ierr = PetscOptionsGetInt
  (NULL,NULL,"-n",&domain_size,&flag);
PetscPrintf(comm,"Using domain size %d\n",domain_size);
```

声明存在一个选项 `-n`，后面跟一个整数。

现在执行

```
mpiexec yourprogram -n 5
```

will

1. 将 *flag* 设置为 true，2. 将参数 *domain_size* 设置为命令行上的值。

省略 `-n` 选项将保持 *domain_size* 的默认值不变。

对于标志，使用 `PetscOptionsHasName`。

Pythonnote 47:Petsc 选项。在 Python 中，不要指定选项名称的初始连字符。此外，诸如 *getInt* 之类的函数不会返回布尔标志；如果需要测试命令行选项的存在，请使用：

```
hasn = PETSc.Options().hasName("n")
```

有一个相关机制使用 `PetscOptionsBegin` / `PetscOptionsEnd`：

```
// optionsbegin.c
PetscOptionsBegin(comm,NULL,"Parameters",NULL);
PetscCall( PetscOptionsInt("-i","i value",__FILE__,i_value,&i_value,&i_flag) );
PetscCall( PetscOptionsInt("-j","j value",__FILE__,j_value,&j_value,&j_flag) );
PetscOptionsEnd();
if (i_flag)
  PetscPrintf(comm,"Option `-i' was used\n");
if (j_flag)
  PetscPrintf(comm,"Option `-j' was used\n");
```

这种方法的卖点是运行你的代码时

```
mpiexec yourprogram -help
```

将以块的形式显示这些选项。不幸的是，还有大量其他选项。

38.3.2 选项前缀

在许多情况下，yourcode 只有一个 `KSP` 求解器对象，因此指定 *-ksp_view* 或 *-ksp_monitor* 将显示 / 跟踪该对象。然而，你可能有多个求解器，或者嵌套的求解器。此时你可能不想显示所有求解器。

作为嵌套求解器情况的一个例子，考虑 块 *Jacobi* 预处理器 的情况，其中块本身是用迭代方法求解的。你可以用 *--sub_ksp_monitor* 跟踪它。

`sub_` 是一个选项前缀，你可以用 `KSPSetOptionsPrefix` 定义你自己的前缀。（其他 PETSc 对象类型也有类似的例程。）

示例：

```
KSPCreate(comm,&time_solver);
KSPCreate(comm,&space_solver);
KSPSetOptionsPrefix(time_solver "time_");
KSPSetOptionsPrefix(space_solver,"space_");
```

图 38.6 **PetscTime**Synopsis

Returns the CPU time in seconds used by the process

.

```
#include "petscsys.h"
#include "petsctime.h"
PetscErrorCode PetscGetCPUTime(PetscLogDouble *t)
PetscErrorCode PetscTime(PetscLogDouble *v)
```

你可以使用选项 `-time_ksp_monitor` 等。注意前缀没有前导短横线，但有尾部下划线。

类似例程 : `MatSetOptionsPrefix`, `PCSetOptionsPrefix`, `PetscObjectSetOptionsPrefix`,`PetscViewerSetOptionsPrefix`, `SNESSetOptionsPrefix`, `TSSetOptionsPrefix`, `VecSetOptionsPrefix`, 以及一些更晦涩的例程。

38.3.3 选项指定位置

命令行选项显然可以放在命令行上。然而，还有更多地方可以指定它们。

选项可以通过 `PetscOptionsSetValue` 以编程方式指定 :

```
PetscOptionsSetValue( NULL, // for global options
"-some_option","value_as_string");
```

Options can be sp[ecifie] 在用户主目录中的文件 `.petscrc` 中指定，或者在 cu rrent directory.

最后，可以设置环境变量 `PETSC_OPTIONS` 。

The `rc` 文件首先被处理，然后是环境变量，最后是任何命令行参数。这个解析过程是在 `PetscInitialize` 中完成的，因此任何来自 `PetscOptionsSetValue` 的值都会覆盖此设置。

38.4 计时与性能分析

PETSc 有许多计时例程，使得无需使用系统例程如 `getrusage` 或 MPI 例程如 `MPI_Wtime`。主要的（实时时钟）计时器是 `PetscTime` (图 38.6)。注意 `PetscLogDouble` 的返回类型，其精度可能与 `PetscReal` 不同。

例程 `PetscGetCPUTime` 用处较小，因为它只测量计算时间，忽略了通信等因素。

38.4.1 日志记录

Petsc does a lot of logging on its own operations. Additionally, you can introduce your own routines into this log.

显示统计信息的最简单方法是使用选项 *-log_view* 运行。这可以带一个可选的文件名参数:

```
mpiexec -n 10 yourprogram -log_view :statistics.txt
```

The corresponding routine is `PetscLogView`.

图 38.7 **PetscMalloc1**Synopsis

```
Allocates an array of memory aligned to PETSC_MEMALIGN

C:
#include <petscsys.h>
PetscErrorCode PetscMalloc1(size_t m1,type **r1)

Input Parameter:
m1 - number of elements to allocate (may be zero)

Output Parameter:
r1 - memory allocated
```

图 38.8 **PetscFree**Synopsis

```
Frees memory,  not collective

C:
#include <petscsys.h>
PetscErrorCode PetscFree(void *memory)

Input Parameter:
memory - memory to free (the pointer is ALWAYS set to NULL upon sucess)
```

38.5 内存管理

为给定指针分配内存： `PetscNew`，使用 `PetscMalloc` 分配任意内存，使用 `PetscMalloc1` (图 38.7) 分配多个对象（这不会将分配的内存清零，使用 `PetscCalloc1` 来获取已清零的内存）；使用 `PetscFree` (图 38.8) 来释放。

```
PetscInt *idxs;
PetscMalloc1(10,&idxs);
// better than:
// PetscMalloc(10*sizeof(PetscInt),&idxs);
for (PetscInt i=0; i<10; i++)
  idxs[i] = f(i);
PetscFree(idxs);
```

分配的内存是对齐的 `PETSC_MEMALIGN`.

内存分配的状态可以写入文件或标准输出，使用 `PetscMallocDump`。命令行选项 *-malloc_dump* 在 `PetscFinalize` 期间输出所有未释放的内存。

38.5.1 GPU allocation

CPU 和 GPU 的内存是不一致的。这意味着诸如 `PetscMalloc1` 的例程不能立即用于 GPU 分配。详情见第 37.4 节。

第 39 章

PETSc 主题

39.1 通信器

PETSc 有一个 “world” 通信器，默认等于 `MPI_COMM_WORLD`。如果你想在部分进程上运行 PETSc，可以在 `MPI_Init` 和 `PetscInitialize` 调用之间，将一个子通信器赋值给变量 `PETSC_COMM_WORLD`。Petsc 通信器的类型是 *PetscComm*。

第四部分

其他编程模型

还有许多其他的并行编程系统。特别是线程模型多如牛毛。

这里我们讨论：

- Co-array Fortran (CAF)：一种针对 Fortran 数组的分布式并行模式；章节 40。
- Kokkos 和 Sycl，两个面向多核和加速器编程的异构编程模型（ Sycl 还支持现场可编程门阵列 (FPGAs) ）；分别见章节 41 和 42。
- Python multiprocessing ；章节 43。

将这些与上面已经讨论的系统进行比较，我们可以指出 ：

- CAF 是分布式内存编程中为数不多的 MPI 替代方案之一。它对笛卡尔数组的处理更优雅；否则它缺少许多 MPI 功能。
- Kokkos 和 Sycl 是 OpenMP offloading 的竞争者；章节 27. 在这些系统中，CPU 和 GPU 模式之间的切换比 OpenMP 更容易。
- python 的 multiprocessing 工具箱比 Python 中的 *mpi4py* 模块更偏向于基于任务。

在现有的各种线程模型中，我们提到

- *pthreads*，它更适合系统编程而非科学计算；参见 HPC 书，章节 -2.6.1.3。
- C++ 有一个本地线程库；参见 Programming 书，章节 -25.1。C++ 标准库 ‘algorithms’ 中也有执行策略机制。
- *Intel TBB* 通常用作其他线程实现的底层，例如本地 C++parallelexecution 策略。

我们不讨论的并行模型包括

- *Compute-Unified Device Architecture (CUDA)* ，其目标是 GPU。
- *NVIDIA Collective Communication Library (NCCL)*，它针对 GPU 优化了 MPI 集合操作。
- Chapel，一种完全独立的并行计算语言；参见 HPC 书籍，第 2.6.5.5 节。关于并行语言的主题，更一般地参见 HPC 书籍，第 2.6.5 节。

第 40 章

Co-array Fortran

本章解释了 CAF 的基本概念，并帮助您开始运行第一个程序。

40.1 历史与设计

https://en.wikipedia.org/wiki/Coarray_Fortran

40.2 Compilingand running

CAF 建立在与 MPI 相同的 SPMD 设计之上。MPI 讨论的是进程或秩，而 CAF 将你的程序运行实例称为 *image*。

Intel 编译器使用标志 `-coarray=xxx`，其取值为 `single`, `shared`, `distributed` `gpu`。

可以将 “images” 的数量编译进可执行文件，但默认情况下不这样做，数量由变量 `FOR_COARRAY_NUM_IMAGES` 在运行时确定。

CAF can not be mixed with OpenMP.

40.3 基础

共数组通过在 `Dimension` 之外，赋予它们一个 `Codimension` 来定义

```
Complex,codimension(*) :: numberInteger,dimension(:,:,:),
codimension[-1:1,*] :: grid
```

T这意味着我们分别在每个图像上声明一个包含单个数字的数组，或者一个三维网格
s在二维处理器网格上分布。

也可以使用传统的语法:

```
Complex :: number[*]
Integer :: grid(10,20,30)[-1:1,*]
```

与通常只支持线性进程编号的 MPI 不同，CAF 允许多维进程网格。最后一个维度总是指定为 `*`，意味着它在运行时确定。

40.3.1 图像识别

与其他模型一样，在 CAF 中可以分别用 `num_images` 和 `this_image` 来查询图像 / 进程的数量以及当前图像的编号。

```
!! hello.F90
  write(*,*) "Hello from image ", this_image(), &
  "out of ", num_images()," total images"
```

如果调用 `this_image` 并传入一个协同数组作为参数，它将返回图像索引，作为 `cosubscript` 的元组，而不是线性索引。给定这样一组下标，`image_index` 将返回线性索引。

函数 `lcobound` 和 `ucobound` 分别给出图像下标的下界和上界，作为线性索引，或者如果用协同数组变量调用，则作为元组。

40.3.2 Remote operations

CAF 的吸引力在于在 images 之间移动数据看起来（几乎）像一个普通的复制操作：

```
real :: x(2)[*]
integer :: p
p = this_image()
x(1)[ p+1 ] = x(2)[ p ]
```

使用数组语法优雅地交换网格边界：

```
Real,Dimension( 0:N+1,0:N+1 )[*] :: grid
grid( N+1,: )[p] = grid( 0,: )[p+1]
grid(   0,: )[p] = grid( N,: )[p-1]
```

40.3.3 同步

Fortran 标准禁止竞态条件：

> 如果一个变量在一个图像的一个段中被定义，则除非这些段是有序的，否则它不得在另一个图像的段中被引用、定义或变为未定义。

也就是说，你不应该让它们发生。语言和运行时环境当然不会帮你解决这个问题。

嗯，有一点。在远程更新之后，你可以使用 `sync` 调用来同步 images。最简单的方式是全局同步：

```
sync all
```

将此与 MPI 非阻塞调用后的等待调用进行比较。

更细粒度的，可以与特定的 images 进行同步：

```
sync images( (/ p-1,p,p+1 /) )
```

虽然 CAF 中的远程操作很好地实现了一边式，但同步不是：如果 image `p` 发出调用

```
sync(q)
```

然后 q 也需要发出一个镜像调用以与 p 同步。

作为说明，以下代码不是一个正确的 *ping-pong* 实现 :

```
!! pingpong.F90
  sync all
  if (procid==1) then
     number[procid+1] = number[procid]
  else if (procid==2) then
     number[procid-1] = 2*number[procid]
  end if
  sync all
```

我们可以通过全局同步来解决这个问题：

```
sync all
if (procid==1) &
     number[procid+1] = number[procid]
sync all
if (procid==2) &
     number[procid-1] = 2*number[procid]
sync all
```

或者是本地的：

```
if (procid==1) &
     number[procid+1] = number[procid]
if (procid<=2) sync images( (/1,2/) )
if (procid==2) &
     number[procid-1] = 2*number[procid]
if (procid<=2) sync images( (/2,1/) )
```

请注意，本地同步调用在涉及的两个图像上都执行。

集体同步的示例：

```
if ( this_image() .eq. 1 ) sync images( * )
if ( this_image() .ne. 1 ) sync images( 1 )
```

这里 image 1 与所有其他进程同步，但其他进程之间不相互同步。

```
if (procid==1) then
   sync images( (/procid+1/) )
else if (procid==nprocs) then
   sync images( (/procid-1/) )
else
   sync images( (/procid-1,procid+1/) )
end if
```

40.3.4 Collectives

截至 2008 年 Fortran 标准，Collectives 不是 CAF 的一部分。

第 41 章

Kokkos

本材料大部分基于 Jeff Miles 和 Christian Trott 于 2020 年 4 月 21 日至 24 日举办的 Kokkos 教程。

包含文件：

```
// hello.cxx
#include "Kokkos_Core.hpp"
```

41.1 并行代码执行

在并行执行中，我们基本上有两个问题：

1. 算法的并行结构；这是我们在本节中讨论的内容。2. 数据的内存结构布局；这将在第 41.2 节中讨论。

The algorithmic 并行结构由以下 co 指示 nstructs.

```
Kokkos::parallel_for
Kokkos::parallel_reduce
Kokkos::parallel_scan
```

41.1.1 Example:1D loop

Hello world:

```
Kokkos::parallel_for
  ( 10,
    [](int i){ cout << "hello " << i << "\n"; }
    );
```

parallel_for 的两个参数是：

- 迭代次数，
- 迭代次数的函数。这里可以使用函数指针，但我们通常会使用 lambda 表达式。

可选地，parallel 构造可以接受一个字符串参数，用于命名 .

41.1.2 归约

归约为构造添加了一个参数：归约变量。以下是通过积分计算 π 的传统方法：

```
double pi{0.};
int n{100};
Kokkos::parallel_reduce
  ( "PI",
    n,
    KOKKOS_LAMBDA ( int i, double& partial ) {
    double h = 1./n, x = i*h;
    partial += h * sqrt( 1-x*x );
  },
    pi
    );
```

- 并行构造有一个可选名称。这对于性能分析和调试很有用。
- 我们没有使用显式的 lambda 捕获，而是使用 `KOKKOS_LAMBDA`，它执行 `[=]` 捕获，并在需要时添加 GPU 执行的子句。

- lambda 表达式现在接受两个参数：迭代次数和归约变量。这是线程私有变量，而不是最终变量。

- 最后一个参数是全局归约变量。

对于除求和以外的归约，需要一个 `reducer`。

```
// reduxmax.cxxdouble max=0.;
Kokkos::parallel_reduce( npoints,
KOKKOS_LAMBDA (int i,double& m) {
if (x(i)>m)m = x(i);},
Kokkos::Max<double>(max));
cout << "max: " << max << "\n";
```

41.1.3 示例：多维循环

当然，你可以对外层循环进行并行化，在函数对象中执行内层循环。此代码计算 $r \leftarrow y^t Ax$：

```
Kokkos::parallel_reduce( "yAx", N,
  KOKKOS_LAMBDA ( int j, double &update ) {
    double temp2 = 0;

    for ( int i = 0; i < M; ++i ) {
      temp2 += A[ j * M + i ] * x[ i ];
    }

    update += y[ j ] * temp2;
  },
  result
);
```

你也可以将所有循环交给 Kokkos，使用 *RangePolicy* 或 *MDRangePolicy*。这里你需要指明对象的秩（即维度数），以及起始 / 结束值的数组。在上述示例中

```
Kokkos::parallel_reduce( N, ... );
// equivalent:
Kokkos::parallel_reduce( Kokkos:RangePolicy<>(0,N), ... );
```

一个秩大于一的示例：

```
// matyax.cxx
Kokkos::parallel_reduce
  ( "ytAx product",
    Kokkos::MDRangePolicy<Kokkos::Rank<2>>( {0,0}, {m,n} ),
    KOKKOS_LAMBDA (int i,int j,double &partial ) {
        partial += yvec(i) * matrix(i,j) * xvec(j); },
    sum
    );
```

注意此示例中的多维索引：这种括号表示法会根据代码是在 CPU 还是 GPU 上运行，转换为正确的行 / 列主序；参见章节 19.6.2。

41.2 Data

Kokkos 解决的问题之一是主处理器与附加设备（如 GPU ）之间数据的一致性。这是通过 *Kokkos::View* 机制来处理的。

```
// matsum.cxx
int m=10,n=100;
Kokkos::View<double**> matrix("flat",m,n);
assert( matrix.extent(0)==10 );
```

这些行为类似于 C++ *shared_ptr*，因此按值捕获它们实际上是按引用获取数据。存储会在它们超出作用域时自动释放，采用 RAII 风格。

索引最好使用 Fortran 风格的表示法：

```
matrix(i,j)
```

这使得算法中的索引与实际布局无关。

可以容纳编译时维度：

```
View<double*[2]>  tallskinny("tallthin",100);
View<double*[2][3]>  tallthin(100);
```

编译时维度位于末尾。命名是可选的。

方法：

- *extent*(int) 给出某个维度的范围；
- *data* 给出指向数据的原始指针。

41.2.1 数据布局

视图声明有一个可选的模板参数用于数据布局。

```
View<double***, Layout, Space> name(...);
```

值为

- *LayoutLeft* 其中，Fortran 风格中，最左边的索引是步长 1；这是 *CudaSpace* 的默认值。
- *LayoutRight* 其中，C 风格中，最左边的索引是步长 1；这是 *HostSpace* 的默认值。
- *LayoutStride*, *LayoutTiled* 以及其他。
- 用户定义。

实际上，二维数组的遍历现在是一个

- 布局的函数，可能由内存空间决定，且
- the indexing in in the functor:

```
Kokkos:parallel_whatever(
  N,
  KOKKOS_LAMBDA ( size_t i ) {
    matrix(i,j) or matrix(j,i); }
);
```

最好遵循以下经验法则：

> 在由内存空间决定的布局中，让迭代器索引排在第一位，并让函子范围内的循环遍历后续索引。

41.3 执行和内存空间

函子的主体可以在 CPU 或 GPU 上执行。这些就是执行空间。Kokkos 需要安装时支持这些空间。

要表示一个函数或 lambda 表达式可以在多个可能的执行空间上执行：

- 使用 *KOKKOS_LAMBDA* 作为 lambda 表达式的捕获，或者
- 显式地在定义的函数前加上 *KOKKOS_INLINE_FUNCTION* 前缀。

执行空间可以使用 *RangePolicy* 关键字显式指示 ：

```
Kokkos::parallel_for
  ( Kokkos::RangePolicy<>( 0,10 ), # default execution space
    [] (int i) {} );
Kokkos::parallel_for
  ( Kokkos::RangePolicy<SomeExecutionSpace>( 0,10 ),
    [] (int i) {} );
```

默认

```
Kokkos::parallel_for( N, ...
```

等同于

```
Kokkos::parallel_for( RangePolicy<>(N), ...
```

41.3.1 内存空间

数据存储的位置是一个独立的话题。每个执行空间都有一个内存空间。创建 `View` 时，可以选择性地指定一个内存空间参数：

```
View<double***,MemorySpace> data(...);
```

可用的内存空间包括： `HostSpace`, `CudaSpace`, `CudaUVMSpace`。省略内存空间参数等同于

```
View<double**,DefaultExecutionSpace::memory_space> x(1,2);
```

示例：

```
View<double*,HostSpace> hostarray(5);
View<double*,CudaSpace> cudaarray(5);
```

只有在 Kokkos 已使用 CUDA 进行配置时， `CudaSpace` 才可用

41.3.2 空间一致性

Kokkos 从不进行隐式深拷贝，因此你不能立即在 Host 空间的 view 上运行 Cuda 执行空间中的 functor。

你可以在主机上创建 CUDA 数据的镜像：

```
CuMatrix matrix(m,n);
CuMatrix::HostMirror hostmatrix =
    Kokkos::create_mirror_view(matrix);
// populate matrix on the host
for (i) for (j) hostmatrix(i,j) = ....;
// deep copy to GPU
Kokkos::deep_copy(matrix,hostmatrix);
// do something on the GPU
Kokkos:parallel_whatever(
    RangePolicy<CudaSpace>( 0,n ),
    some lambda );
// if needed, deep copy back.
```

41.4 配置

无加速器的 OpenMP 安装：

```
cmake \
    -D Kokkos_ENABLE_SERIAL=ON -D Kokkos_ENABLE_OPENMP=ON
```

线程不兼容 OpenMP：

```
-D Kokkos_ENABLE_THREADS=ON
```

Cuda 安装：

```
cmake \
    -D Kokkos_ENABLE_CUDA=ON -D Kokkos_ARCH_TURING75=ON -D Kokkos_ENABLE_CUDA_LAMBDA=ON
```

41.5 杂项

有 init/finalize 调用，但并非总是需要。

```
// pi.cxx
Kokkos::initialize(argc,argv);
Kokkos::finalize();
```

41.5.1 OpenMP 集成

Cmake flag to enable OpenMP: `-D Kokkos_ENABLE_OPENMP=ON`

之后，所有常用的 OpenMP 环境变量都生效。

或者:

```
int nthreads = Kokkos::OpenMP::concurrency();
Kokkos::initialize(Kokkos::InitializationSettings().set_num_threads(nthreads))
```

并行控制:

```
--kokkos-threads=123 # threads
--kokkos-numa=45     # numa regions
--kokkos-device=6    * GPU id to use
```

第 42 章

Sycl，OneAPI，DPC++

This chapter explains the basic concepts of Sycl/Dpc++, and helps you get started on running your first program.

- *SYCL* 是一种基于 C++ 的可移植并行编程语言。
- *Data Parallel C++ (DPCPP)* 是 Intel 对 Sycl 的扩展。
- *OneAPI* 是 Intel 的编译器套件，其中包含 DPCPP 编译器。

Intel DPC++ extension. The various Intel extensions are listed here: https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html#extensions-table

42.1 物流

Headers:

```
#include <CL/sycl.hpp>
```

你现在可以包含命名空间，但要小心！如果你使用

```
using namespace cl;
```

你必须给所有 SYCL 类加上 `sycl::` 前缀，这有点麻烦。然而，如果你使用

```
using namespace cl::sycl;
```

你会遇到 SYCL 有自己版本的许多标准模板库（ STL ）命令，因此会发生名称冲突。最明显的例子是 `cl::sycl` 命名空间有自己版本的 `cout` 和 `endl`。因此你必须显式使用 `std::cout` 和 `std::end`。使用错误的 I/O 会导致大量难以理解的错误信息。此外，SYCL 还有自己版本的 `free` 和若干数学例程。

Intel DPC++ 扩展。

```
    using namespace sycl;

```

42.2 Platformsand devices

由于 DPCPP 是跨平台的，我们首先需要发现设备。

First we list the platforms:

```
// devices.cxx
std::vector<sycl::platform> platforms = sycl::platform::get_platforms();
for (const auto &plat : platforms) {
// get_info is a template. So we pass the type as an `arguments`.
  std::cout << "Platform: "
            << plat.get_info<sycl::info::platform::name>() << " "
            << plat.get_info<sycl::info::platform::vendor>() << " "
            << plat.get_info<sycl::info::platform::version>()
              << '\n';
```

然后对于每个平台，我们列出设备：

```
std::vector<sycl::device> devices = plat.get_devices();
for (const auto &dev : devices) {
  std::cout << "-- Device: "
            << dev.get_info<sycl::info::device::name>()
            << (dev.is_host() ? ": is the host" : "")
            << (dev.is_cpu() ? ": is a cpu" : "")
            << (dev.is_gpu() ? ": is a gpu" : "")
            << std::endl;
```

您可以通过 `is_cpu`, `is_gpu` 查询您正在处理的设备类型。（函数 `is_host` 在 SYCL-2020 中已被弃用。）

42.3 队列

SYCL 的执行机制是 *queue*：一系列将在选定设备上执行的操作。用户唯一的操作是将操作提交到队列；队列在其声明的作用域结束时执行。

队列执行与主机代码是异步的。

42.3.1 设备选择器

您需要选择一个设备来执行队列。单个队列只能分派到单个设备。

队列与一个特定设备绑定，因此它不能将工作分散到多个设备上。您可以使用以下方法找到队列的默认设备

```
  sycl::queue myqueue;
```

下面的示例使用 `sycl::cpu_selector` 明确地将队列分配给 CPU 设备。

```
// cpuname.cxx
sycl::queue myqueue( sycl::cpu_selector{} );
```

备注 43 *Pre-SYCL-2020:* `sycl::host_selector` 绕过任何设备，使代码在主机上运行。

`cpu_selector` 在 *SYCL-2020* 中已被弃用，取而代之的是 `cpu_selector_v`。

打印你正在运行的设备名称对你的理智有好处：

```
// devname.cxx
std::cout << myqueue.get_device().get_info<sycl::info::device::name>()
          << std::endl;
```

如果你尝试选择一个不可用的设备，将会抛出一个 `sycl::runtime_error` 异常。

Intel DPC++ 扩展。

```
    #include "CL/sycl/intel/fpga_extensions.hpp"
    fpga_selector
```

42.3.2 队列提交和执行

看起来队列内核也会在它们超出作用域时执行，但队列本身不会：

```
// doubler.cxx
sycl::range<1> mySize{SIZE};sycl::buffer<int, 1> bufferA
                                (myArray.data(), mySize);
myqueue.submit
  ( [&](sycl::handler &myHandle) {
    auto deviceAccessorA =
      bufferA.get_access<sy
      cl::access::mode::read_write>(myHandle);/ queue goes out of scope, executes
} /
```

42.3.3 Kernel ordering

内核不一定按照提交的顺序执行。您可以通过指定一个 *in-orderqueue* 来强制执行这一点：

```
sycl::queue myqueue{property::queue::inorder()};
```

42.4 Kernels

每次提交一个 kernel。

```
myqueue.submit( [&] ( handler &commandgroup ) {
    commandgroup.parallel_for<uniquename>
      ( range<1>{N},
        [=] ( id<1> idx ) { ... idx }
      )
    } );
```

注意，kernel 中的 lambda 是按值捕获的。按引用捕获没有意义，因为 kernel 在设备上执行。

```
cgh.single_task(
  [=]() {
    // kernel function is executed EXACTLY once on a SINGLE work-item
});
```

The `submit` 调用产生一个事件对象：

```
auto myevent = myqueue.submit( /* stuff */ );
```

这可以用于两个目的：

1. 可以等待这个特定事件：

   ```
   myevent.wait();
   ```

2. 它可以用来指示内核依赖关系：

   ```
   myqueue.submit( [=] (handler &h) {
       h.depends_on(myevent);
       /* stuff */
       } );
   ```

42.5 Parallel operations

42.5.1 循环

```
cgh.parallel_for(range<3>(1024,1024,1024),// using 3D in this example
[=](id<3> myID) {
// kernel function is executed on an n-dimensional range (NDrange)});
cgh.parallel_for(nd_range<3>( {1024,1024,1024},{16,16,16} ),
// using 3D in this example[=](nd_item<3> myID) {
// kernel function is executed on an n-dimensional range (NDrange)});
cgh.parallel_for_work_group(range<2>(1024,1024),
// using 2D in this example[=](group<2> myGroup) {
// kernel function is executed once per work-group});
grp.parallel_for_work_item(range<1>(1024),
```

```
  // using 1D in this example
  [=](h_item<1> myItem) {
    // kernel function is executed once per work-item
});
```

在 SYCL-2020 中， *nd_range* 上的偏移已被弃用。

42.5.1.1 循环边界：范围

SYCL 采用了现代 C++ 的理念，即不通过显式枚举索引来迭代，而是通过指示它们的范围来实现。这是通过 *range* 类实现的，该类是针对空间维数进行模板化的。

```
sycl::range<2> matrix{10,10};
```

某些编译器对整数参数的类型比较敏感：

```
sycl::range<1> array{ static_cast<size_t>(size)} ;
```

42.5.1.2 循环索引

内核如 *parallel_for* 期望两个参数：

- 一个 *range* 用于索引；以及
- 一个带有一个参数的 lambda：一个索引。

有几种索引方式。 *id<nd>* 多维索引的类。

```
myHandle.parallel_for<class uniqueID>
   ( mySize,
     [=]( id<1> index ) {
       float x = index.get(0) * h;
       deviceAccessorA[index] *= 2.;
     }
   )
```

```
cgh.parallel_for<class foo>(
    range<1>{D*D*D},
    [=](id<1> item) {
        xx[ item[0] ] = 2 * item[0] + 1;
    }
)
```

虽然 C++ vectors 仍然是一维的，但 DPCPP 允许你创建多维缓冲区：

```
std::vector<int> y(D*D*D);
buffer<int,1> y_buf(y.data(), range<1>(D*D*D));
cgh.parallel_for<class foo2D>
   (range<2>{D,D*D},
    [=](id<2> item) {
        yy[ item[0] + D*item[1] ] = 2;
    }
   );
```

Intel DPC++ 扩展 . 存在从一维 `sycl::id<1>` 到 `size_t` 的隐式转换，因此

```
[=](sycl::id<1> i) {
   data[i] = i;
}
```

is legal, which in SYCL requires

```
data[i[0]] = i[0];
```

42.5.1.3 多维索引

```
// stencil2d.cxx
sycl::range<2> stencil_range(N, M);
sycl::range<2> alloc_range(N + 2, M + 2);
std::vector<float>
  input(alloc_range.size()),
  output(alloc_range.size());
  sycl::buffer<float, 2> input_buf(input.data(), alloc_range);
  sycl::buffer<float, 2> output_buf(output.data(), alloc_range);
```

```
constexpr size_t B = 4;sycl::range<2> local_range(B, B);
sycl::range<2> tile_range = local_range + sycl::range<2>(2, 2); // Includes boundary cells
auto tile = local_accessor<float, 2>(tile_range, h); // see templated def'n above
```

我们首先将全局数据复制到工作组本地的数组中:

```
sycl::id<2> offset(1, 1);
h.parallel_for
  ( sycl::nd_range<2>(stencil_range, local_range, offset),
    [=] ( sycl::nd_item<2> it ) {
// Load this tile into work-group local memory
    sycl::id<2>    lid    = it.get_local_id();
    sycl::range<2> lrange = it.get_local_range();
    for   (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
      for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
        int gi = ti + B * it.get_group(0);
        int gj = tj + B * it.get_group(1);
        tile[ti][tj] = input[gi][gj];
      }
    }
```

输入中的全局坐标是根据 `nd_item` 的坐标和组计算得出的:

```
[=] ( sycl::nd_item<2> it ) {
for   (int ti ... ) {
  for (int tj ... ) {
    int gi = ti + B * it.get_group(0);
    int gj = tj + B * it.get_group(1);
    ... = input[gi][gj];
```

块内的局部坐标，包括边界，我还不太明白这一点。

```
[=] ( sycl::nd_item<2> it ) {
sycl::id<2>    lid    = it.get_local_id();
sycl::range<2> lrange = it.get_local_range();
for   (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
  for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
    tile[ti][tj] = ..
```

42.5.2 任务依赖

Each `submit` call can be said to correspond to a 'task'. Since it returns a token, it becomes possible to specify task dependencies by refering to a token as a dependency in a later specified task.

```
queue myQueue;
auto myTokA = myQueue.submit
   ( [&](handler& h) {
       h.parallel_for<class taskA>(...);
     }
   );
auto myTokB = myQueue.submit
   ( [&](handler& h) {
       h.depends_on(myTokA);
       h.parallel_for<class taskB>(...);
     }
   );
```

42.5.3 竞争条件

Sycl has t与其他共享 竞争条件 存在相同的问题　　内存　　system have:

```
// sum1d.cxx
auto array_accessor =
  array_buffer.get_access<sycl::access::mode::read>(h);
auto scalar_accessor =
  scalar_buffer.get_access<sycl::access::mode::read_write>(h);
h.parallel_for<class uniqueID>
  ( array_range,
    [=](sycl::id<1> index)
    {
      scalar_accessor[0] += array_accessor[index];
    }
    ); // end of parallel for
```

要使其正确工作，需要一个归约原语或对累加器的原子操作。2020 年提议的标准改进了原子操作。

```
// reduct1d.cxx
auto input_values = array_buffer.get_access<sycl::access::mode::read>(h);
auto sum_reduction = sycl::reduction( scalar_buffer,h,std::plus<>() );
h.parallel_for
  ( array_range,sum_reduction,
```

```
    [=]( sycl::id<1> index,auto& sum )
    {
      sum += input_values[index];
    }
    ); // end of parallel for
```

42.5.4 归约

归约操作被添加到了 SYCL 2020 临时标准中，这意味着它们尚未最终确定。

这里有一种方法，适用于 *hipsycl*:

```
// reductscalar.cxx
auto reduce_to_sum =
  sycl::reduction( sum_array, static_cast<float>(0.), std::plus<float>() );
myqueue.parallel_for// parallel_for<reduction_kernel<T,BinaryOp,__LINE__>>
  ( array_range,    // sycl::range<1>(input_size),
    reduce_to_sum,  // sycl::reduction(output, identity, op),
    [=] (sycl::id<1> idx, auto& reducer) { // type of reducer is impl-dependent, so use auto
    reducer.combine(shared_array[idx[0]]); //(input[idx[0]]);
//reducer += shared_array[idx[0]]; // see line 216: add_reducer += input0[idx[0]];
  } ).wait();
```

这里从目标数据和归约操作符创建了一个 `sycl::reduction` 对象。然后将其传递给 `parallel_for`，并调用其 `combine` 方法。

42.6 内存访问

SYCL 中的内存处理有点复杂，因为至少存在主机内存和设备内存，这两者不一定是相干的。

还有三种机制:

- 基于普通 C/C++ “ 星号 ” 指针的统一共享内存。
- 使用 `buffer` 类的缓冲区；这需要 `accessor` 类来访问数据。
- 图像。

表 42.1：内存类型及处理

位置	分配	一致性	复制到 / 从设备
Host	`malloc`	explicit transfer	`queue::memcpy`
	`malloc_host`	coherent host/device	
Device	`malloc_device`	explicit transfer	`queue::memcpy`
Shared	`malloc_shared`	coherent host/device	

42.6.1 Unifiedshared memory

使用 *malloc_host* 分配的内存在主机上可见:

```
// outshared.cxx
floattype
  *host_float = (floattype*)malloc_host( sizeof(floattype),ctx ),
  *shar_float = (floattype*)malloc_shared( sizeof(floattype),dev,ctx );
    cgh.single_task
      ( [=] () {
        shar_float[0] = 2 * host_float[0];
        sout << "Device sets " << shar_float[0] << sycl::endl;
      } );
```

设备内存通过 *malloc_device* 分配，传递队列作为参数:

```
// reductimpl.cxx
floattype
  *host_float = (floattype*)malloc( sizeof(floattype) ),
  *devc_float = (floattype*)malloc_device( sizeof(floattype),dev,ctx );
   [&](sycl::handler &cgh) {
     cgh.memcpy(devc_float,host_float,sizeof(floattype));
   }
```

注意对应的 *free* 调用也将队列作为参数。

注意你需要处于并行任务中。以下会导致分段错误 r:

```
  [&](sycl::handler &cgh) {
    shar_float[0] = host_float[0];
  }
```

普通内存，例如来自 *malloc*，必须复制到内核中:

```
[&](sycl::handler &cgh) {
  cgh.memcpy(devc_float,host_float,sizeof(floattype));
}
[&](sycl::handler &cgh) {
  sycl::stream sout(1024, 256, cgh);
  cgh.single_task
    (
     [=] () {
       sout << "Number " << devc_float[0] << sycl::endl;
     }
     );
} // end of submitted lambda
free(host_float);
sycl::free(devc_float,myqueue);
```

42.6.2 Buffers andaccessors

数组需要以一种可以从任何设备访问的方式声明。

```
// forloop.cxx
std::vector<int> myArray(SIZE);
  range<1> mySize{myArray.size()
      };buffer<int, 1> bufferA(myArray.data(), myArray.size());
```

备注 44 *sycl::range* 接受一个 *size_t* 参数；指定一个 *int* 可能会导致编译器关于窄化转换的警告。

在内核内部，数组随后从缓冲区中解包：

```
myqueue.submit( [&] (handler &h) {
    auto deviceAccessorA =
      bufferA.get_access<access::mode::read_write>(h);
```

然而， get_access 函数返回一个 *sycl*::accessor，而不是指向简单类型的指针。确切的类型是模板化且复杂的，因此这里是使用 auto 的好地方。

访问器可以关联一个模式： *sycl*::*access*::*mode*::*read* *sycl*::*access*::*mode*::*write*

Intel DPC++ 扩展。

```
    array<floattype,1> leftsum{0.};
#ifdef __INTEL_CLANG_COMPILER
    sycl::buffer leftbuf(leftsum);
#else
    sycl::range<1> scalar{1};
    sycl::buffer<floattype,1> leftbuf(leftsum.data(),scalar);
```

Intel DPC++ 扩展。 有多种模式

```
// standard
sycl::accessor acc = buffer.get_access<sycl::access::mode:write>(h);
// dpcpp extension
sycl::accessor acc( buffer,h,sycl::read_only );
sycl::accessor acc( buffer,h,sycl::write_only );
```

42.6.2.1 多维缓冲区

要创建多维缓冲区对象，使用 *sycl*::*range* 来指定维度：

```
// jordan.cxx
vector<double> matrix(vecsize*vecsize);
sycl::range<2> mat_range{vecsize,vecsize};
sycl::buffer<double,2> matrix_buffer( matri     x.data(),mat_range );
```

42.6.3 查询

函数 *get_range* 可以查询缓冲区或访问器的大小：

```
// range2.cxx
sycl::buffer<int, 2>
a_buf(a.data(), sycl::range<2>(N, M)),
b_buf(b.data(), sycl::range<2>(N, M)),
c_buf(c.data(), sycl::range<2>(N, M));

sycl::range<2>
  a_range = a_buf.get_range(),
  b_range = b_buf.get_range();

if (a_range==b_range) {
```

```
sycl::accessor c = c_buf.get_access<sycl::access::mode::write>(h);

sycl::range<2> c_range = c.get_range();
if (a_range==c_range) {
  h.parallel_for
    ( a_range,
      [=]( sycl::id<2> idx ) {
      c[idx] = a[idx] + b[idx];
    } );
```

42.7 并行输出

有一个 `sycl::cout` 和 `sycl::endl`。

```
// hello.cxx
[&](sycl::handler &cgh) {
  sycl::stream sout(1024, 256, cgh);
  cgh.parallel_for<class hello_world>
    (
     sycl::range<1>(global_range), [=](sycl::id<1> idx) {
       sout << "Hello, World: World rank " << idx << sycl::endl;
     }); // End of the kernel function
}
```

由于队列末尾不会刷新 stdout，可能需要调用 `sycl::queue::wait`

```
myQueue.wait();
```

42.8 其他

异常:

```
  try {
    sycl::whatever
  } catch ( sycl::errc::runtime &e ) { ....}
```

Deprecated as of SYCL-2020:

```
} catch ( sycl::runtime_error &e ) { ... }
```

42.9 DPCPP 扩展

Intel 对 SYCL 做了一些扩展:

- 统一共享内存 ,
- 有序队列 .

42.10 Intel devcloud 笔记

`qsub -I` for interactive session.

`gdb-oneapi` 用于调试。

https://community.intel.com/t5/Intel-oneAPI-Toolkits/ct-p/oneapi 用于支持。

42.11 示例

42.11.1 循环中的内核

The following idiom works:

```
sycl::event last_event = queue.submit( [&] (sycl::handler &h) {
for (int iteration=0; iteration<N; iteration++) {
  last_event = queue.submit( [&] (sycl::handler &h) {
      h.depends_on(last_event);
```

42.11.2 Stencil computations

模板计算的问题在于只有内部点会被更新。转换为 SYCL：我们需要遍历定义缓冲区的范围的子范围。首先让我们定义这些范围:

```
// jacobi1d.cxx
sycl::range<1> unknowns(N);
sycl::range<1> with_boundary(N + 2);
std::vector<float>
  old_values(with_boundary.size(),0.),
  new_values(with_boundary.size(),0.);
old_values.back() = 1.; new_values.back() = 1.;
```

注意右边界上的边界值 1.

通过 *parallel_for* 的 *offset* 参数来限制迭代到内部点:

```
sycl::id<1> offset(1);
h.parallel_for
    ( unknowns, offset,
      [=] (sycl::id<1> idx) {
          int i = idx[0];
          float self = old_array[i];
          float left = old_array[i - 1];
          float righ = old_array[i + 1];
          new_array[i] = (self + left + righ) / 3.0f;
    } );
```

第 43 章

Python multiprocessing

Python 有一个 *multiprocessing* 工具箱。这是一个并行处理库，依赖于子进程，而不是线程。

43.1 软件和硬件

`multiprocessing` 工具箱会自行进行硬件检测；它使用单个节点，以及系统告诉它的所有核心数。

```
## pool.py
nprocs = mp.cpu_count()
print(f"I detect {nprocs} cores")
```

43.2 进程

进程是将执行一个 python 函数的对象：

```
## quicksort.py
import multiprocessing as mp
import random
import os

def quicksort( numbers ) :
    if len(numbers)==1:
        return numbers
    else:
        median = numbers[0]
        left  = [ i for i in numbers if i<median ]
        right = [ i for i in numbers if i>=median ]
        with mp.Pool(2) as pool:
            [sortleft,sortright] = pool.map( quicksort,[left,right] )
        return sortleft.append( sortright )

if __name__ == '__main__':
    numbers = [ random.randint(1,50) for i in range(32) ]
    process = mp.Process(target=quicksort,args=[numbers])
```

```
    process.start()
    process.join()
```

创建一个进程并不会启动它：要启动进程，请使用 `start` 函数。只有调用了 `join` 函数，进程的执行才有保证：

```
if __name__ == '__main__':
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

通过使 start 和 join 调用不像下面循环中那样规律，可以编写任意复杂的代码。

43.2.1 Arguments

可以通过 `args` 关键字将参数传递给进程的函数。该关键字接受一个参数列表（或元组），这导致单个参数时语法有些奇怪：

```
proc = Process(target=print_func, args=(name,))
```

43.2.2 进程细节

注意对 `__main__` 的测试：启动的进程读取当前文件以执行指定的函数。没有这条语句，import 会先执行更多的进程启动调用，然后才执行函数。

进程有一个名称，可以通过 `current_process().name` 获取。默认是 `Process-5` 等，但你可以指定自定义名称：

```
Process(name="Your name here")
```

进程的目标函数可以通过 `current_process` 函数获取该进程。

当然你也可以查询 `os.getpid()`，但这并不提供任何进一步的可能性。

```
def say_name(iproc):
    print(f"Process {os.getpid()} has name: {mp.current_process().name}")
if __name__ == '__main__':
    processes = [ mp.Process(target=say_name,name=f"proc{iproc}",args=[iproc])
                  for iproc in range(6) ]
```

43.3 池和映射

通常你希望多个进程应用于多个参数，例如在参数扫描中。为此，创建一个 `Pool` 对象，并对其应用 `map` 方法：

```
pool = mp.Pool( nprocs )
results = pool.map( print_value,range(1,2*nprocs) )
```

注意，这也是从进程获取返回值的最简单方法，而使用 `Process` 对象则无法直接实现。其他方法是使用共享对象，或在 `Queue` 或 `Pipe` 对象中使用对象；见下文。

43.4 共享数据

处理共享数据的最佳方法是创建一个 `Value` 或 `Array` 对象，这些对象配备了用于安全更新的锁。

```
pi = mp.Value('d')
pi.value = 0
```

例如，可以通过随机计算 π 来实现

1. 在 $[0,1)^2$中生成随机点，2. 记录落在单位圆内的点数，之后 3. π 是圆内点数与总点数的比率。

```
## pi.py
def calc_pi1(pi,n):
    for i in range(n):
        x = random.random()
        y = random.random()
        with pi.get_lock():
            if x*x+y*y<1:
                pi.value += 1.
```

Exercise 43.1. Do you see a way to improve the speed of this calculation?

43.4.1 管道

一个管道，对象类型 `Pipe`，对应于旧并行编程系统中曾被称为通道的东西：一个先进先出（ FIFO ）对象，一个进程可以向其中放入项目，另一个进程可以从中取出项目。然而，管道并不与任何特定的进程对关联：创建管道即给出了管道的入口和出口

```
q_entrance,q_exit = mp.Pipe()
```

它们可以传递给任何进程

```
producer1 = mp.Process(target=add_to_pipe,args=([1,q_entrance]))
producer2 = mp.Process(target=add_to_pipe,args=([2,q_entrance]))
printer = mp.Process(target=print_from_pipe,args=(q_exit,))
```

然后可以使用 `send` 和 `recv` 命令来放入和取出项目。

```
## pipemulti.py
def add_to_pipe(v,q):
    for i in range(10):
        print(f"put {v}")
        q.send(v)
        time.sleep(1)
    q.send("END")
```

```
def print_from_pipe(q):
    ends = 0
    while True:
```

```
        v = q.recv()
        print(f"Got: {v}")
        if v=="END":
            ends += 1
        if ends==2:
            break
    print("pipe is empty")
```

43.4.2 队列

第五部分

其余部分

第 44 章

探索计算机架构

关于计算机架构有很多内容可以讲述。然而，在并行编程的背景下，我们主要关注以下几点：

- 有多少个网络节点，网络是否具有我们需要关注的结构？
- 在一个计算节点上，有多少个插槽（或其他非统一内存访问（ NUMA ）域）？
- 对于每个插槽，有多少个核心和超线程？缓存是否共享？

44.1 发现工具

发现并了解你的并行机器结构的一个简单方法是使用专门为此目的编写的工具。

44.1.1 Intel cpuinfo

Intel compiler suite 附带了一个工具 *cpuinfo* ，它可以报告你运行节点的结构。它会报告包的数量，即插槽、核心和线程。

44.1.2 hwloc

开源包 `hwloc` 的报告功能类似于 cpuinfo，但它已被移植到许多平台。此外，它还能生成架构的 ascii 和 pdf 图形渲染。

第 45 章

混合计算

到目前为止，您已经学会了使用 MPI 进行分布式内存编程和使用 OpenMP 进行共享内存并行编程。然而，分布式内存架构实际上具有共享内存组件，因为每个集群节点通常是多核设计。因此，您可以使用 MPI 进行节点间编程，使用 OpenMP 进行节点内并行。

您现在必须在进程和线程之间找到合适的平衡，因为每个都可以使一个核心充分忙碌。使情况复杂化的是，一个节点可以有多个 *socket*，以及相应的 *NUMA* 域。图 45.1

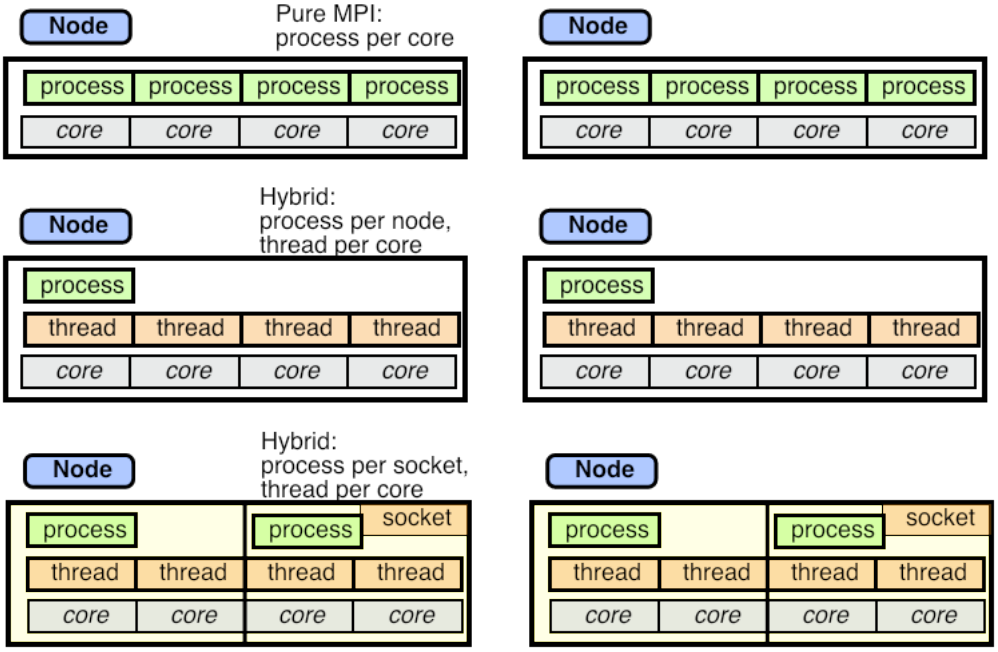


图 45.1：多核集群上 MPI/OpenMP 使用的三种模式

展示了三种模式：纯 MPI 且不使用线程；每个节点一个 MPI 进程并完全多线程；每个节点两个 MPI 进程，每个 socket 一个，并且每个 socket 上有多个线程。

45.1 并发

在混合多进程 / 多线程计算中，一个被抛弃的概念是每个 MPI 进程的顺序语义。例如，单个发送者和单个接收者之间的消息是 *non-overtaking* 非超越的这一事实，如果消息来源于不同线程，则不再成立。

```
// anytag.c
#pragma omp parallel sections
    {
#pragma omp section
    MPI_Isend
      ( &x,1,MPI_DOUBLE,
        receiver,xtag,comm,requests+0);
#pragma omp section
    MPI_Isend
      ( &y,1,MPI_DOUBLE,
        receiver,ytag,comm,requests+1);
    }

    ↪MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
```

```
#pragma omp section
    MPI_Irecv
      ( &xy1,1,MPI_DOUBLE,
        sender, MPI_ANY_TAG, comm,
    ↪requests+0);
#pragma omp section
    MPI_Irecv
      ( &xy2,1,MPI_DOUBLE,
        sender, MPI_ANY_TAG, comm,
    ↪requests+1);
    }
    MPI_Waitall(2,requests,statuses);
```

来自同时线程的消息被称为 *concurrent* 并发：它们之间没有时间或因果关系。

45.2 亲和性

在前面的章节中，我们大多将所有 MPI 节点或 OpenMP 线程视为一个扁平池。然而，为了高性能，你需要关注 *affinity*：即哪个进程或线程放置在哪里，以及它们如何高效地交互的问题。

以下是一些亲和性成为关注点的情况。

- 在纯 MPI 模式下，同一节点上的进程通常比不同节点上的进程通信更快。由于进程通常是顺序放置的，这意味着进程 p 主要与 $p+1$ 交互的方案将更高效，而跨越较大距离的通信效率较低。
- 如果集群网络具有结构（处理器网格 而非 *fat-tree*），进程的放置会影响程序效率。MPI 试图通过图拓扑来解决这个问题；详见第 11.2 节。
- 即使在单个节点上也可能存在不对称。图 45.2 展示了 *Ranger* 超级计算机（已停产）四个插槽的结构。两个核心之间没有直接连接。这种不对称影响该节点上的 MPI 进程和线程。
- 多插槽设计的另一个问题是每个插槽都有其附属内存。虽然每个插槽都可以寻址节点上的所有内存，但访问本地内存更快。这种不对称在 *first-touch* 现象中非常明显；详见第 25.2 节。
- 如果一个节点上的 MPI 进程数少于核心数，你会希望控制它们的放置位置。此外，操作系统可能会迁移进程，这对性能有害，因为它会破坏数据局部性。出于这个原因，可以使用诸如 `numactl` (以及在 TACC `tacc_affinity`) 这样的工具来 *pina thread* 或将进程绑定到特定核心。
- 具有超线程 或硬件线程 的处理器引入了另一个需要关注线程去向的层面。

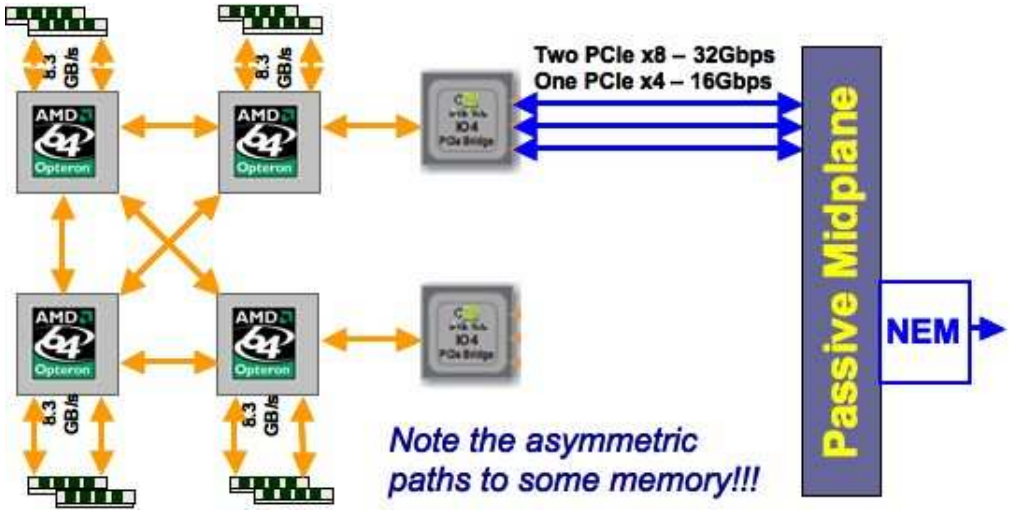


图 45.2：Ranger 节点的 NUMA 结构

45.3 硬件长什么样?

如果你想优化亲和性，首先应该了解硬件长什么样。这里 `hwloc` 工具非常有价值 [11] (https://www.open-mpi.org/projects/hwloc/)。

图 45.3 展示了一个 *Stampede* 计算节点，它是一个双插槽 *Intel Sandybridge* 设计；图 45.4 展示了一个 *Stampede largemem* 节点，它是一个四插槽设计。最后，图 45.5 展示了一个 *Lonestar5* 计算节点，一个双插槽设计，配备了每个有两个硬件线程的 12 核 *Intel Haswell* 处理器。

45.4 亲和性控制

参见第 25 章关于 OpenMP 亲和性控制。

45.5 讨论

纯 MPI 策略与混合策略的性能影响是微妙的 le.

- 首先，我们注意到没有明显的加速：在一个负载均衡良好的 MPI 应用中，所有核心始终处于忙碌状态，因此使用线程并不会带来即时的改进。
- MPI 和 OpenMP 都受阿姆达尔定律的约束，该定律量化了串行代码的影响；在混合计算中，有一个该定律的新版本，涉及 MPI 并行但非 OpenMP 并行的代码量。
- MPI 进程运行时不同步，因此可以容忍负载或处理器行为上的小幅变化。OpenMP 结构中频繁的屏障使得混合代码同步更紧密，因此负载均衡变得更加关键。
- 另一方面，在 OpenMP 代码中，将工作划分为比线程更多的任务更容易，因此从统计学上讲，某种程度的负载均衡会自动发生。
- 每个 MPI 进程都有自己的缓冲区，因此混合模式占用的缓冲开销更少。

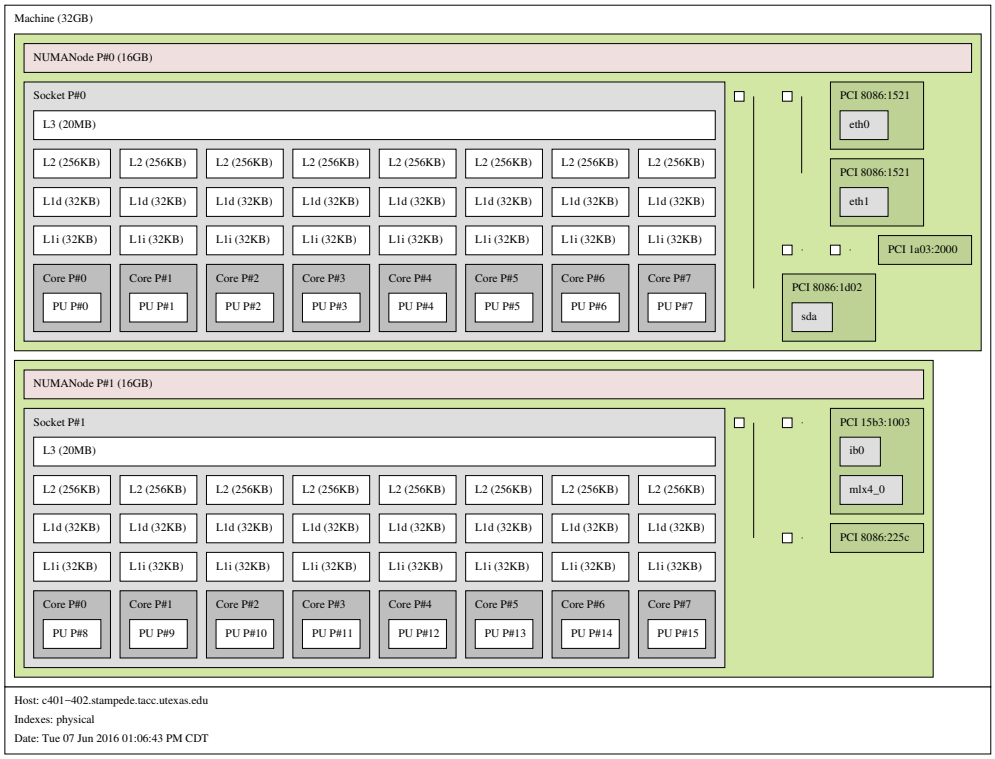


图 45.3：Stampede 计算节点结构

练习 45.1. 回顾 HPC 书中第 7.2 节关于 1D 与 2D 矩阵分解的可扩展性论证。你是否能通过对 MPI 进程进行 1D 分解（例如按行分解），并对 OpenMP 线程进行另一方向（列）的分解，获得可扩展的性能?

我们需要考虑的另一个性能论点涉及消息流量。如果让所有线程都进行 MPI 调用（见第 13.1 节），差别将很小。然而，在一种流行的混合计算策略中，我们会将 MPI 调用排除在 OpenMP 区域之外，实际上由主线程执行。在这种情况下，只有节点之间存在 MPI 消息，而不是核心之间。这导致消息流量减少，尽管这很难量化。消息数量大约减少了每个节点的核心数，因此如果平均消息大小较小，这是一个优势。另一方面，只有当消息内容有重叠时，发送的数据量才会减少。

Limiting MPI traffic to the master 线程也意味着不需要缓冲区空间 the on-node communication.

45.6 进程与核心及亲和性

在 OpenMP 中，线程纯粹是软件构造，你可以创建任意数量的线程。可用核心的硬件限制可以通过 `omp_get_num_procs` (章节 17.5) 查询。在混合环境中这如何工作? ‘proc’ 计数返回的是核心总数，还是 MPI 调度器将其限制为每个 MPI 进程独占的数量?

下面的代码片段对此进行了探讨：

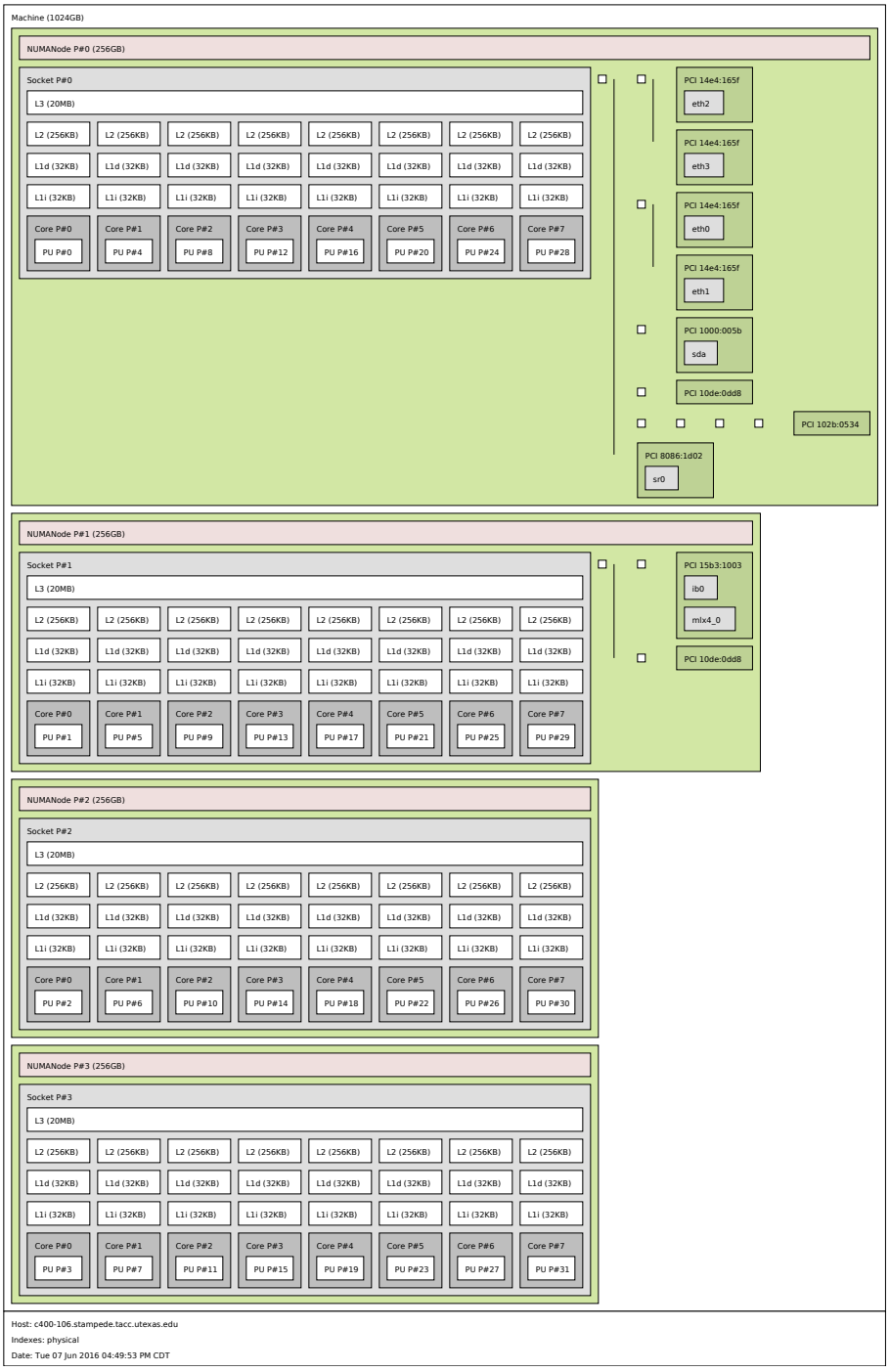


图 45.4：Stampede largemem 四插槽计算节点的结构

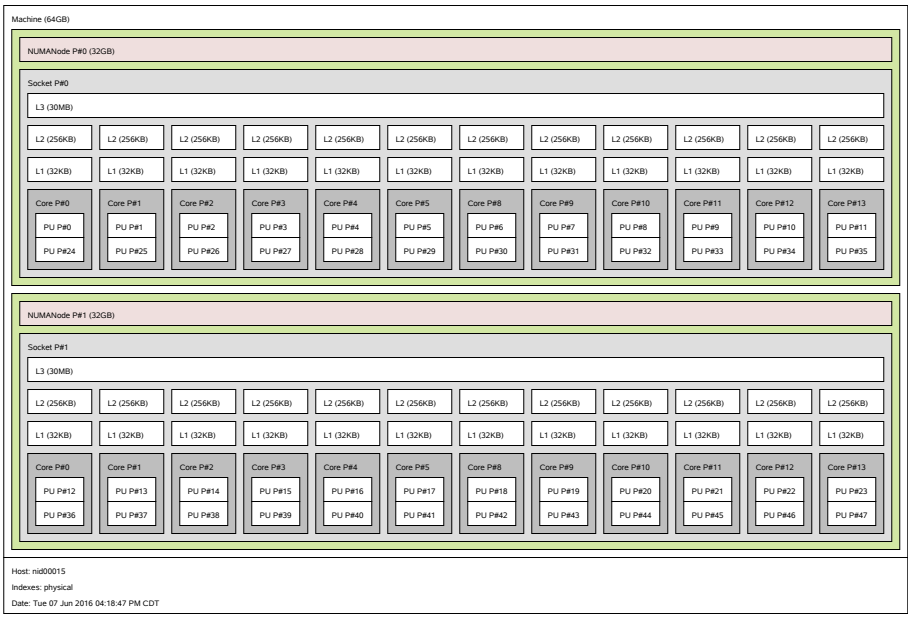


图 45.5：Lonestar5 计算节点结构

```
// procthread.c
int ncores;
#pragma omp parallel
#pragma omp master
ncores = omp_get_num_procs();

int totalcores;
MPI_Reduce(&ncores,&totalcores,1,MPI_INT,MPI_SUM,0,comm);
if (procid==0) {
  printf("Omp procs on this process: %d\n",ncores);
  printf("Omp procs total: %d\n",totalcores);
}
```

使用 *Intel MPI*（版本 19）运行此程序，结果如下：

```
---- nprocs: 14
Omp procs on this process: 4
Omp procs total: 56
---- nprocs: 15
Omp procs on this process: 3
Omp procs total: 45
---- nprocs: 16
Omp procs on this process: 3
Omp procs total: 48 我们看到
```

- 每个进程获得相同数量的核心，并且
- 一些核心将被闲置。

虽然 OpenMP 的 ‘proc’ 计数确保 MPI 进程不会超额订阅核心，但这里并未表达进程和线程的实际放置位置。这个分配称为亲和性，它由 MPI/OpenMP 运行时系统决定。通常可以通过环境变量进行控制，但希望默认分配是合理的。图 45.6 展示了 *Intel Knights Landing* 的情况：

图 45.6：Intel Knights Landing 上的进程和线程放置

- 在 68 个核心上放置四个 MPI 进程，每个进程分配 17 个核心。
- 每个进程接收一组连续的核心。
- 然而，核心以两个为一组的 “ 瓦片 ” 形式分组，因此进程 1 和 3 从瓦片中间开始。
- 因此，该进程的线程零绑定到第二个核心。

45.7 实用规范

假设你使用 100 个集群节点，每个节点有 16 个核心。你可以启动 1600 个 MPI 进程，每个核心一个，但你也可以启动 100 个进程，并让每个进程访问 16 个 OpenMP 线程。

在你的 slurm 脚本中，第一个方案将被指定为 `-N 100 -n 1600`，第二个方案为

```
#$ SBATCH -N 100
#$ SBATCH -n 100

export OMP_NUM_THREADS=16
```

在这两个极端之间还有第三种选择，这种选择是合理的。一个集群节点通常有多个 *socket*，所以你可以在每个 *socket* 上放置一个 MPI 进程，并使用等于每个 *socket* 核心数的线程数。

该脚本为: `#$ SBATCH -N 100`

```
#$ SBATCH -n 200

export OMP_NUM_THREADS=8
ibrun tacc_affinity yourprogram
```

`tacc_affinity` 脚本取消设置以下变量:

```
export MV2_USE_AFFINITY=0
export MV2_ENABLE_AFFINITY=0
export VIADEV_USE_AFFINITY=0
export VIADEV_ENABLE_AFFINITY=0
```

如果你不使用 `tacc_affinity`，你可能想手动完成这一步，否则 *mvapich2* 将使用它自己的亲和性规则。

第 46 章

支持库

有许多与并行编程相关的库，可以让你的工作更轻松，或者至少更有趣。

46.1 SimGrid

SimGrid [17] 是一个分布式系统模拟器。例如，它可以用来探索架构参数的影响。它已被用于模拟大规模操作，如高性能 Linpack (HPL) [4]。

46.2 其他

ParaMesh
Global
Arrays Hdf5
and Silo

第六部分

类项目

第 47 章

项目提交风格指南

H以下是一些关于如何提交作业和项目的指导原则。一般来说，将编程视为
a一门实验科学，而你的写作则是对你所做测试的报告：解释你所
a解决的问题、你的策略、你的结果。

47.1 General approach

作为一般规则，将编程视为一门实验科学，而你的写作则是对你所做测试的报告：解释你所解决的问题、你的策略、你的结果。

提交一份以 pdf 格式的写作（不接受 Word 和文本文件），该文件应由文本处理程序生成（最好是） LaTEX。有关教程，请参见 Tutorialsbook，第 15 节。

47.2 Style

你的报告应遵守正确英语的规则。

- 遵守正确的拼写和语法是不言而喻的。
- 使用完整的句子。
- 尽量避免使用贬义或其他不当的冗词赘句。 *Google developer documentationstyleguide* [12] 是一个很好的资源。

47.3 你的写作结构

47.3.1 写作如同撰写文章

将此项目写作视为练习撰写科学文章的机会 .

从显而易见的内容开始。

- 你的写作应有一个标题。不是 ‘Project’ 或 ‘parallel programming’，而应是类似 ‘Parallelization of Chronosynclastic Enfundibula in MPI’ 这样的标题。
- 作者和联系信息。此信息因出版物而异。这里是：你的姓名，EID，TACC 用户名和电子邮件。
- 引言部分，层次较高：问题是什么，你做了什么，发现了什么。
- 结论：你的发现意味着什么，局限性是什么，未来扩展的机会。
- 参考文献。

47.3.2 考虑你的受众

一篇文章是为特定的受众而写：期刊、会议，或者在本例中：你的指导老师。所以不要深入讲述对你的受众毫无意义的细节，尽量提供他们感兴趣的内容。

换句话说：提供足够的应用背景，但不要过多。你写作的对象不是你的论文导师，而是对你领域感兴趣的外部人士。

另一方面，你的指导老师对并行性了如指掌。所以不要展示一个微分方程然后说 “ 我用 OpenMP 实现了并行 ”。详细说明你是如何将问题转化为计算问题的，然后展示相关的代码片段。

这并不意味着只提交代码就足够了，也不仅仅是代码加示例输出。请写一篇文章。

47.3.3 观察、测量、假设、推断

将你的项目视为对某种现象的科学调查。提出你期望观察到的假设，报告你的观察结果，并得出结论。

你的程序经常会表现出意想不到的行为。注意这一点非常重要，并假设导致你观察到的行为的原因可能是什么。

在大多数计算机应用中，我们关心的是找到解决方案的效率。因此，确保你进行测量。一般来说，进行观察以判断你的程序是否按预期运行。

47.3.4 Reporting

你的报告应包括代码片段和图表。

代码片段的截图不可接受。请至少在您的文本处理器中使用逐字 / 等宽模式，但更好的是，使用 LaTeX `listings` 包或同等工具。

图表可以通过多种方式生成。如果您能搞懂 LaTeX `tikz` 包，那真是太棒了，但 Matlab 、 gnuplot 、 Excel 或 Google Sheets 都是可接受的。同样：不要使用截图。

对于并行运行，您可以使用，但不必使用 TAU 图；详见教程书，第 18 节。

47.3.5 代码仓库组织

如果您通过代码仓库提交您的工作，请将您的 pdf 文件放在顶层；将源代码组织在命名清晰的子目录中。对象文件和二进制文件不应放入代码仓库，因为它们依赖于硬件和编译器等因素。

47.4 并行部分

并行化部分是你报告中最重要的部分。所以不要写 3 页关于你的应用程序，1 页关于并行代码。详细讨论：

- 你的问题的并行结构是什么？将代码结构与应用结构联系起来。
- 你使用了 MPI 还是 OpenMP？为什么？
- 你使用了哪种并行方式？主要是 MPI 集合操作还是点对点操作？OpenMP 循环并行还是任务？为什么？

47.4.1 性能

The most important reason for parallel programming is to achieve faster performance.
为了衡量代码的效率，请在不同的处理器 / 核心数量下运行。你是追求强扩展性还是弱扩展性?

确保你的问题规模足够大，以克服并行化的开销。你能在给定的核心 / 进程数量下运行多个问题规模吗?

考虑到应用程序的结构，你期望获得多少加速？实际获得了多少？如果加速较低，请分析问题可能出在哪里。

47.4.2 运行你的代码

单次运行不能证明任何问题。为了获得良好的报告，您需要针对多个输入数据集（如果有）以及多种处理器配置运行代码。当您选择问题规模时，请牢记以下因素。

- 并行引入了开销，OpenMP 屏障需要数百个周期，MPI 消息则需要几微秒。因此，并行区域内或消息之间的工作量可能需要达到数千次操作。
- 系统中的各种因素会引入随机波动，时间过短的测量可能毫无意义。经验法则是，计时部分至少需要达到秒级别。
- 并行代码中存在各种启动现象，例如 OpenMP 线程创建或动态内存分配。确保您只计时代码中相关的部分；如果不确定，可以在其周围加一个循环进行多次测量，并使用平均时间。

当您并行运行代码时，请注意在集群上，单节点和多节点的并行代码行为总是不同的。在单节点上，MPI 实现可能已针对共享内存进行了优化。这意味着单节点运行得到的结果不具代表性。事实上，在计时和扩展性测试中，您经常会看到从一个节点到两个节点时（相对）性能下降。因此，您需要在多种场景下运行代码，使用多个节点。

47.4.3 图表

在并行编程中，加速比和扩展性是衡量你工作好坏的标准。因此，如何报告这些结果取决于你自己。

数字不是图表的重点：重点是你能从中得出的结论。因此，如果你进行扩展性分析，报告运行时间的图表应使这一点清晰可见。特别是，不要使用线性时间轴，因为曲线图难以阅读。尝试找到一种方法，将你的结果与一条直线进行比较，比如恒定时间或线性增加的加速比。

由你决定报告什么量。这可能取决于你的应用。 .
使用足够的数据点！编写一个简短的脚本多次运行你的程序所花费的时间非常少。

47.5 备注

47.5.1 并行性能或其缺失

在一个理想的世界中，代码的性能应该随着可用资源数量的增加而提升。如果你的程序表现出令人失望的性能，请考虑以下几点。

在并行区域结束时同步 OpenMP 线程可能需要几百个周期。这意味着该区域内的工作量应该大得多。

如果你的 OpenMP 程序在某个核心数时停止扩展，考虑亲和性设置；见第 25.1 节。
MPI 消息传递需要几微秒。同样，这意味着消息之间的工作量需要足够大。

47.5.2 代码格式

包含的代码片段应当可读。至少你可以在将代码包含到 `verbatim` 环境之前，在编辑器中正确缩进代码。（终端窗口的截图显然是一个次优的解决方案。）但更好的方法是使用 `listing` 包，它可以格式化你的代码，并包含语法着色。例如，

```
\lstset{language=C++} % or Fortran or so
\begin{lstlisting}
for (int i=0; i<N; i++)s += 1;
\end{lstlisting}
```

输出：

```
for (int i=0; i<N; i++)
  s += 1;
```

第 48 章

热身练习

我们从一些简单的练习开始。

48.1 Hello world

背景请参见第 2.3 节。

首先，我们需要确保你有一个可用的并行作业环境。示例程序 `helloworld.c` 执行以下操作：

```
// helloworld.c
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ntids);
MPI_Comm_rank(MPI_COMM_WORLD,&mytid);
printf("Hello, this is processor %d out of %d\n",mytid,ntids);
MPI_Finalize();
```

编译这个程序并并行运行。确保处理器不全都说它们是 `processor0 out of 1`！

48.2 集合操作

能够收集统计数据是个好主意，所以在我们做任何有趣的事情之前，我们先来看一下 MPI 集合操作；章节 3.1。

看看 `time_max.cxx`。这个程序会随机睡眠几秒钟：

```
wait = (int) ( 6.*rand() / (double)RAND_MAX );
tstart = MPI_Wtime();sleep(wait);tstop = MPI_Wtime();
jitter = tstop-tstart-wait; 并测量实际睡眠了多久：if (mytid==0)
sendbuf = MPI_IN_PLACE;else sendbuf = (void*)&jitter;
MPI_Reduce(sendbuf,(void*)&jitter,1,MPI_DOUBLE,MPI_MAX,0,comm)
```

;

在代码中，这个量被称为 “jitter”，这是系统中随机偏差的术语。

Exercise 48.1. 通过更改归约操作符，修改此程序以计算平均 jitter。

Exercise 48.2. 现在计算标准差

$$\sigma = \sqrt{\frac{\sum_i (x_i - m)^2}{n}}$$

其中 m 是你在前一个练习中计算的平均值。

- 请完成此练习两次：一次通过先进行 reduce 再进行 broadcast 操作，另一次使用 `Allreduce`。

- 在单个集群节点和多个节点上运行你的代码，并检查 TAU 跟踪。一些 MPI 实现针对共享内存进行了优化，因此单节点上的跟踪可能不会如预期那样。
- 你能从跟踪中看出 allreduce 是如何实现的吗?

Exercise 48.3. 最后，使用 gather 调用将所有值收集到处理器零上，并打印出来。是否有任何进程的行为与其他进程有很大不同?

48.3 Linear arrays of processors

在本节中，你将编写多个关于一个非常简单操作的变体：所有处理器将一个数据项传递给编号更高的下一个处理器。

- 在文件 `linear-serial.c` 中，你会找到一个使用阻塞发送和接收调用的实现。
- 你将修改此代码以使用非阻塞发送和接收；它们需要一个 `MPI_Wait` 调用来完成。

- 接下来，你将使用 `MPI_Sendrecv` 来实现一个同步但无死锁的实现。
- 最后，你将使用两种不同的单边场景。

在参考代码 `linear-serial.c` 中，每个进程定义了两个缓冲区：

`// linear-serial.cint my_number = mytid, other_number=-1.;`，其中 `other_number` 是存储来自左邻居数据的位置。

To check the correctness of the program, there is a gather operation on processor zero:

```
int *gather_buffer=NULL;if (mytid==0) {
gather_buffer = (int*) malloc(ntids*sizeof(int));
if (!gather_buffer) MPI_Abort(comm,1);}
MPI_Gather(&other_number,1,MPI_INT,gather_buffer,1,MPI_INT, 0,
comm);if (mytid==0) {int i,error=0;for (i=0; i<ntids; i++)
if (gather_buffer[i]!=i-1) {
printf("Processor %d was incorrect: %d should be %d\n",i,
gather_buffer[i],i-1);error =1;
```

```
    }
  if (!error) printf("Success!\n");
  free(gather_buffer);
}
```

48.3.1 使用阻塞调用编程

将数据传递给邻近处理器应该是一个非常并行的操作。然而，如果我们天真地用 MPI_Send 和 MPI_Recv 来编写代码，就会出现意想不到的串行行为，正如第 4.1.4 节中所解释的那样。

```
if (mytid<ntids-1)
MPI_Ssend( /* data: */ &my_number,1,MPI_INT,
/* to: */ mytid+1, /* tag: */ 0, comm);if (mytid>0)
  MPI_Recv( /* data: */ &other_number,1,MPI_INT,
            /* from: */ mytid-1, 0, comm, &status);
```

(注意这使用了一个 Ssend ; 详见第 15.8 节的解释。)

练习 48.4. 编译并运行此代码，生成一个 TAU 跟踪文件。确认执行是串行的。将 Ssend 替换为 Send 会改变这一点吗?

让我们稍微整理一下代码。**练习 48.5.** 首先编写这

段代码更优雅地使用 MPI_PROC_NULL .

48.3.2 更好的阻塞解决方案

防止前面练习中串行化问题的最简单方法是使用 MPI_Sendrecv 调用。该例程认识到通常处理器在有发送时会有接收调用。对于发送或接收不匹配的边界情况，可以使用 MPI_PROC_NULL。

练习 48.6. 使用 MPI_Sendrecv 重写代码。通过 TAU 跟踪确认执行不再是串行的。

N注意 Sendrecv 调用本身仍然是阻塞的，但至少其组成的 send 和 recv 的顺序不再
l按时间顺序排列。

48.3.3 非阻塞调用

另一种避免阻塞行为的方法是使用 Isend 和 Irecv 调用，这些调用不会阻塞。当然，现在你需要保证这些发送和接收操作已经完成；在这种情况下，使用 MPI_Waitall。

练习 48.7. 使用 MPI_Isend 和 MPI_Irecv 实现一个完全并行的版本。

48.3.4 单边通信

另一种实现非阻塞行为的方法是使用单边通信。在 Put 或 Get 操作期间，执行只会在数据从源进程传出或传入时阻塞，但不会因目标进程而阻塞。同样，你需要保证传输完成；这里使用 MPI_Win_fence。

练习 48.8. 编写两个版本的代码：一个使用 MPI_Put ，另一个使用 MPI_Get。制作 TAU 跟踪。

通过 TAU 可视化调查阻塞行为。

练习 48.9. 如果你传输大量数据，而目标处理器正忙，你能看到对源进程有任何影响吗？栅栏是同步的吗?

第 49 章

Mandelbrot set

如果你从未听说过 *Mandelbrot set*，你可能认得这张图；图 49.1 它的正式定义

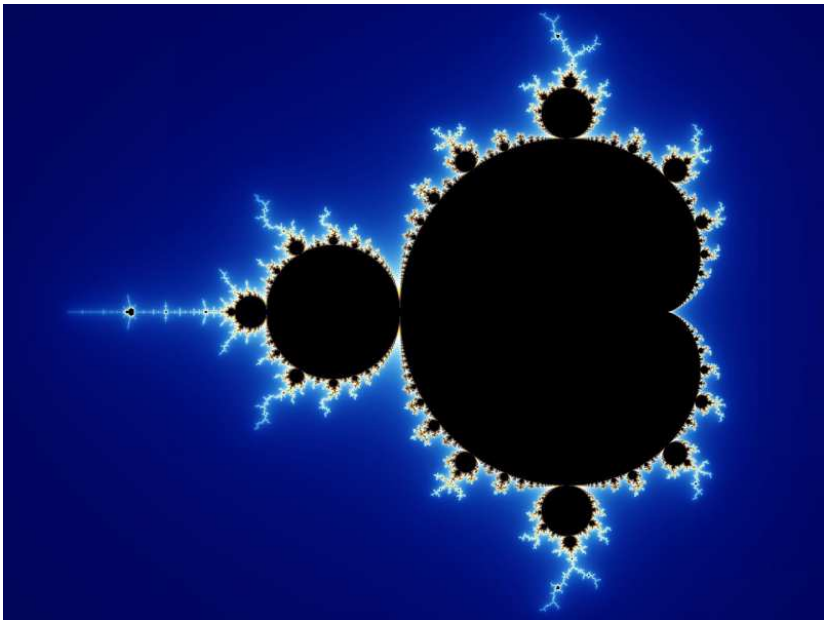


图 49.1：Mandelbrot set

定义如下：

复平面上的一点 c 如果由下式定义的级数 x_n 是

$$\begin{cases} x_0 = 0 \\ x_{n+1} = x_n^2 + c \end{cases}$$

满足

$$\forall_n : |x_n| \leq 2.$$

很容易看出，只有边界圆 $|c| < 2$ 内的点 c 符合条件，但除此之外，如果没有更多的思考或者计算，很难说太多。而这正是我们将要做的。

在这一组练习中，您将以示例程序 `mandel_main.cxx` 为基础，扩展它以使用各种 MPI 编程结构。该程序已设置为 *manager-worker* 模型：有一个管理器处理器（这次是最后一个处理器，而不是零号处理器），它负责分配工作给工作处理器并接收它们的结果。然后，它将结果汇总并构建成图像文件。

49.1 MPI 解决方案

49.1.1 调用

`mandel_main` 程序的调用方式为

```
mpirun -np 123 mandel_main steps 456 iters 789
```

其中 `steps` 参数表示图像中在 x, y 方向上的步数，`iters` 给出了 `belong` 测试中的最大迭代次数。

如果你忘记了参数，可以用以下方式调用程序

```
mandel_serial -h
```

它将打印出使用信息。

49.1.2 工具

Mandelbrot 程序的驱动部分很简单。有一个可以生成坐标的 circle 对象

```
class circle {
 public :
  circle(double stp,int bound);
  void next_coordinate(struct coordinate& xy);
  int is_valid_coordinate(struct coordinate xy);
  void invalid_coordinate(struct coordinate& xy);
```

以及一个全局例程，用于测试一个坐标是否属于该集合，至少在迭代界限内是如此。如果从给定起点开始的序列没有发散，则返回零；如果发散，则返回发散时的迭代次数。

```
int belongs(struct coordinate xy,int itbound) {double x=xy.x,
 y=xy.y; int it;
  for (it=0; it<itbound; it++) {
    double xx,yy;
    xx = x*x - y*y + xy.x;
    yy = 2*x*y + xy.y;
    x = xx; y = yy;
    if (x*x+y*y>4.) {
      return it;
    }
  }
  return 0;
}
```

在前一种情况下，该点可能属于 Mandelbrot 集，我们将其着色为黑色；在后一种情况下，我们根据迭代次数为其着色。

```
if (iteration==0)
  memset(colour,0,3*sizeof(float));
else {
  float rfloat = ((float) iteration) / workcircle->infty;
  colour[0] = rfloat;
  colour[1] = MAX((float)0,(float)(1-2*rfloat));
  colour[2] = MAX((float)0,(float)(2*(rfloat-.5)));
}
```

我们为工作进程使用了一个相当简单的代码：它们执行一个循环，在循环中等待输入，处理输入，返回结果。

```
void queue::wait_for_work(MPI_Comm comm,circle *workcircle) {
  MPI_Status status; int ntids;
  MPI_Comm_size(comm,&ntids);
  int stop = 0;

  while (!stop) {
    struct coordinate xy;
    int res;

    MPI_Recv(&xy,1,coordinate_type,ntids-1,0, comm,&status);
    stop = !workcircle->is_valid_coordinate(xy);
    if (stop) break; //res = 0;
    else {
      res = belongs(xy,workcircle->infty);
    }
    MPI_Send(&res,1,MPI_INT,ntids-1,0, comm);
  }
  return;
}
```

给出了一个使用阻塞发送的管理进程的非常简单的解决方案：

```
// mandel_serial.cxxclass serialqueue : public queue {
private :int free_processor;public :
serialqueue(MPI_Comm queue_comm,circle *workcircle)
: queue(queue_comm,workcircle) {free_processor=0;};/**
The `addtask' routine adds a task to the queue. In this
simple case it immediately sends the task to a worker
and waits for the result,  which is added to the image.
This routine is only called with valid coordinates;
the calling environment will stop the process once
an invalid coordinate is encountered.*/
int addtask(struct coordinate xy) {
```

```
  MPI_Status status; int contribution, err;

  err = MPI_Send(&xy,1,coordinate_type,
           free_processor,0,comm); CHK(err);
  err = MPI_Recv(&contribution,1,MPI_INT,
           free_processor,0,comm, &status); CHK(err);

  coordinate_to_image(xy,contribution);
  total_tasks++;
  free_processor = (free_processor+1)%(ntids-1);

  return 0;
};
```

Exercise 49.1. 解释为什么这个解决方案非常低效。制作一个执行跟踪以证明这一点。

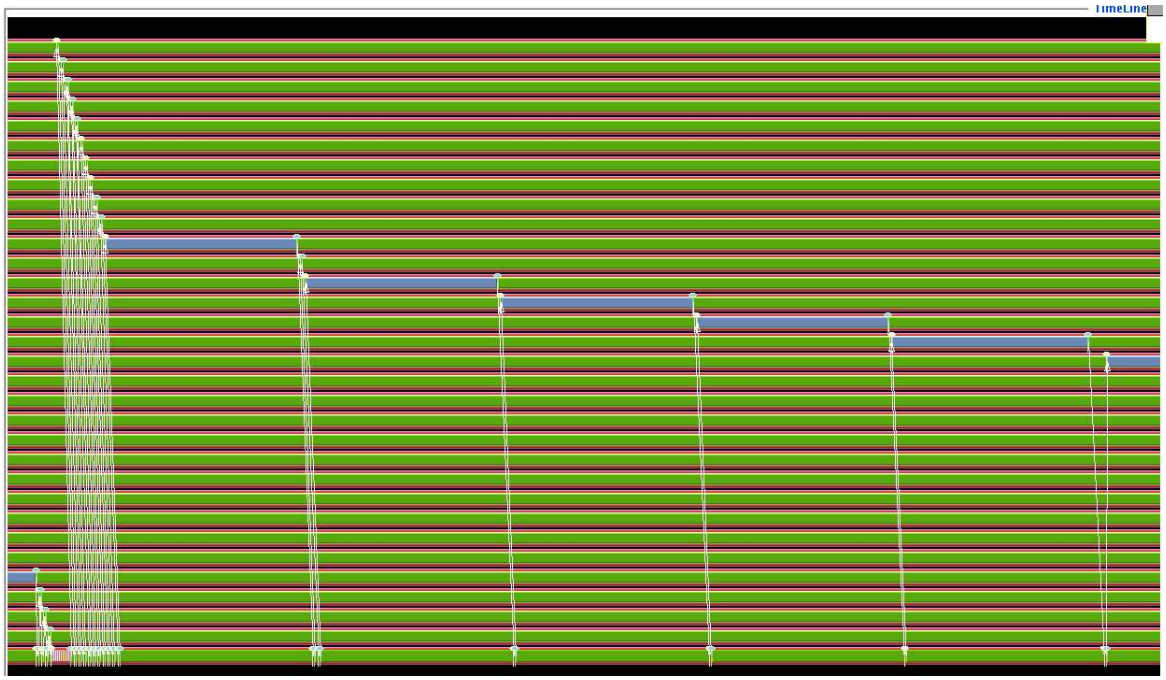


图 49.2：串行 Mandelbrot 计算的跟踪

49.1.3 Bulktask scheduling

T上一节展示了一个非常低效的解决方案，但那主要是为了搭建代码基础。如果所有
t任务所需时间大致相同，你可以给每个进程分配一个任务，然后等待它们全部完成。
A 第一种方法是使用非阻塞发送。

练习 49.2. 编写一个解决方案，使用非阻塞发送和接收将任务分配给所有工作进程，然后等待这些任务使用 `MPI_Waitall` 完成，之后再给所有工作进程新一轮的数据。制作该执行过程的跟踪并报告总时间。

你可以通过编写一个继承自 queue 的新类，并提供它自己的 addtask 方法来实现：

```
// mandel_bulk.cxx
class bulkqueue : public queue {
public :
  bulkqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
```

你还必须重写 complete 方法：当 circle 对象指示所有坐标都已生成时，并非所有工作线程都会忙碌，因此你需要提供适当的 MPI_Waitall 调用。

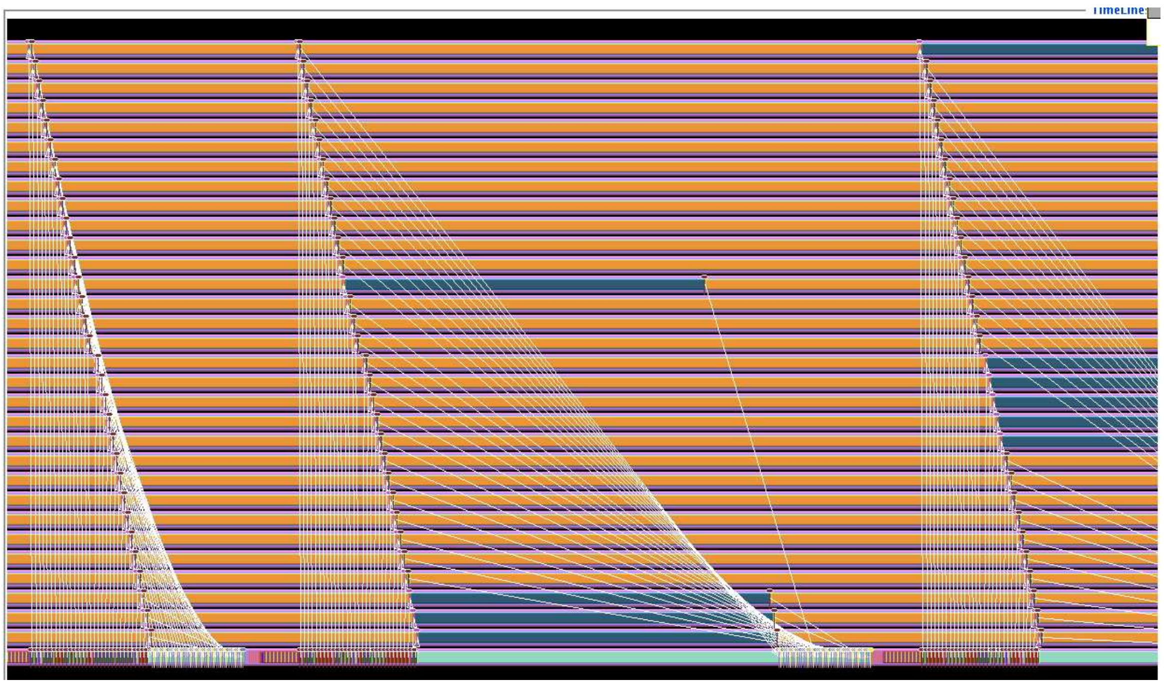


Figure 49.3: Trace of a bulk Mandelbrot calculation

49.1.4 集体任务调度

上一节批量调度的另一种实现方式是通过集体操作

练习 49.3. 编写一种使用 scatter 进行数据分发工作任务，使用 gather 收集结果的解决方案。这个解决方案比之前的更高效还是更低效?

49.1.5 异步任务调度

在第 49.1.3 节开始时我们说过，批量调度主要在所有任务完成时间相近时才有意义。在 Mandelbrot 案例中显然不是这样。

练习 49.4. 编写一个使用 MPI_Probe 或 MPI_Waitany 的完全动态解决方案。制作执行轨迹并报告总运行时间。

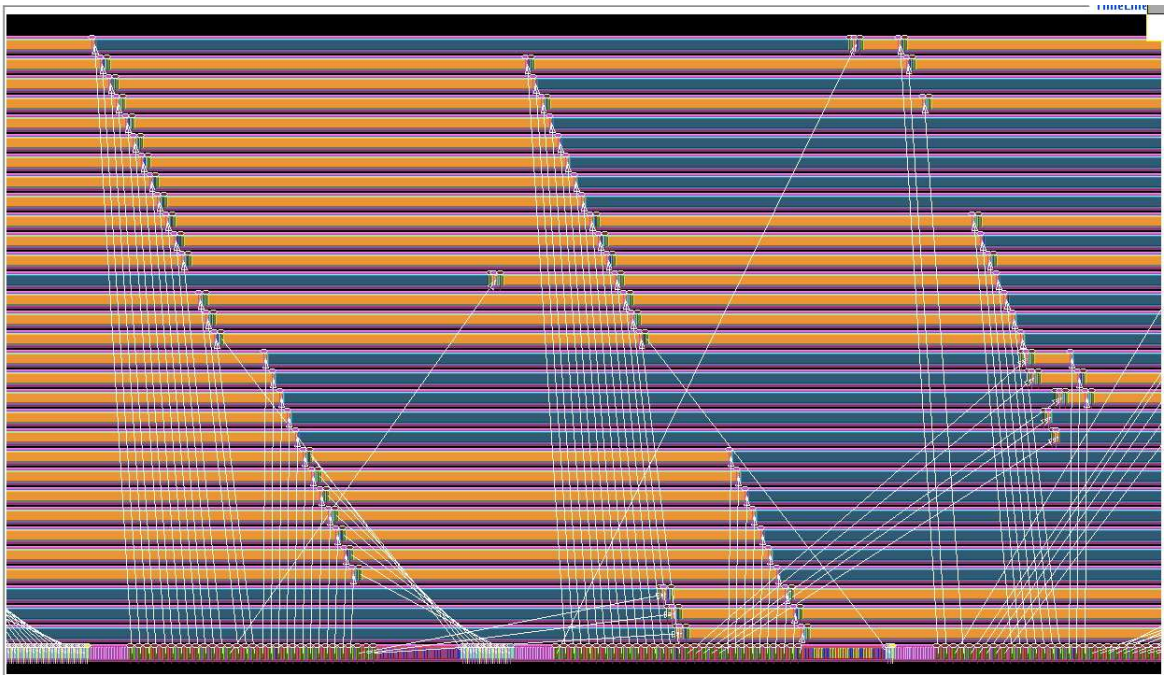


图 49.4：异步曼德尔布罗计算的跟踪

49.1.6 单边解决方案

让我们来推理是否有可能（或是否建议）编写一个用于计算曼德尔布罗集的单边解决方案。使用主动目标同步，你可以在主机上有一个 exposurewindow，工作任务会向其写入。为了防止冲突，你会分配一个数组，并让每个工作任务写入其中的不同位置。这里的问题是，由于计算时间不同，工作任务之间可能没有足够的同步。

那么考虑被动目标同步。现在工作任务可以在有结果需要报告时写入管理者上的窗口；通过锁定窗口，它们防止其他任务干扰。工作任务写入结果后，可以从管理者上的所有坐标数组中获取新数据。

随着结果的不断产生，很难将它们及时放入图像中。为此，管理者必须持续扫描结果数组。因此，构建图像最简单的方式是在所有任务完成后进行。

49.2 OpenMP 解决方案

49.2.1 基于循环

可以对所有坐标进行双重循环，并使用动态或引导调度。

49.2.2 生产者 - 消费者

你也可以使用生产者 - 消费者模型：一个线程生成坐标，其他线程随后处理这些坐标。为了使其尽可能异步，我们保持一个单一的 FIFO 对象（在 C++ 中可以使用 `deque`），并对其加锁：写线程和读线程都会锁定该 FIFO。

This constant locking probably means that threads will keep each other from doing useful work part of the time.
This can be alleviated by writing and reading the FIFO with a block of values at a time.

```
[c205-013 cxx:30] for t in 2 3 4 5 6 ; do OMP_NUM_THREADS=$t ./mandeldeque -s .001 -l 10000 -b 10  ; do
================ #threads = 2 ================
Time: 62.9568
Number of points: 16000000
Interior: 1506920
Compute time: 60.288
================ #threads = 3 ================
Time: 33.2471
Compute time: 60.5359
================ #threads = 4 ================
Time: 23.6858
Compute time: 60.4749
================ #threads = 5 ================
Time: 19.2046
Compute time: 60.4396
================ #threads = 6 ================
Time: 16.7634
Compute time: 60.4717
```

Note that the aggregate compute time is more or less constant, and equals to the case with just one consumer thread.

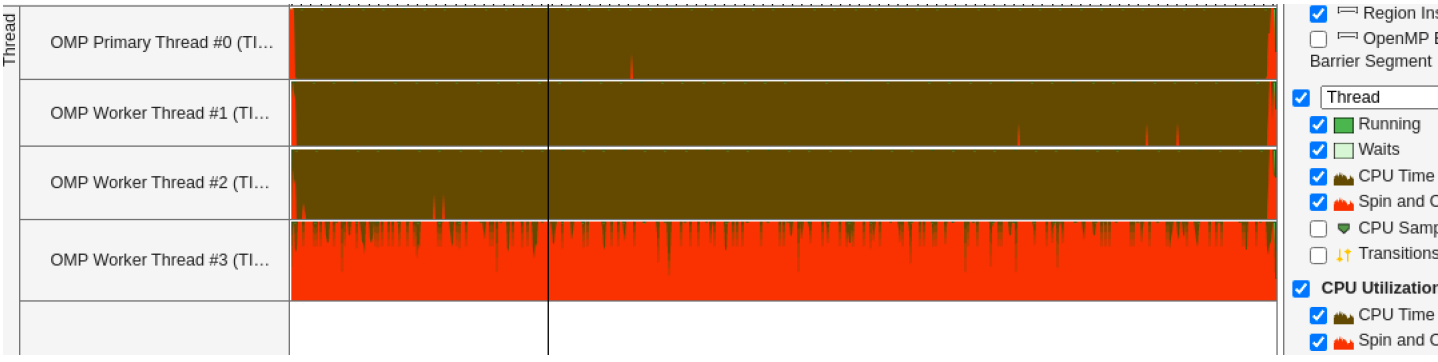
下限破坏可扩展性，总时间再次保持不变，大致等于外部时间 e.

Frontera:

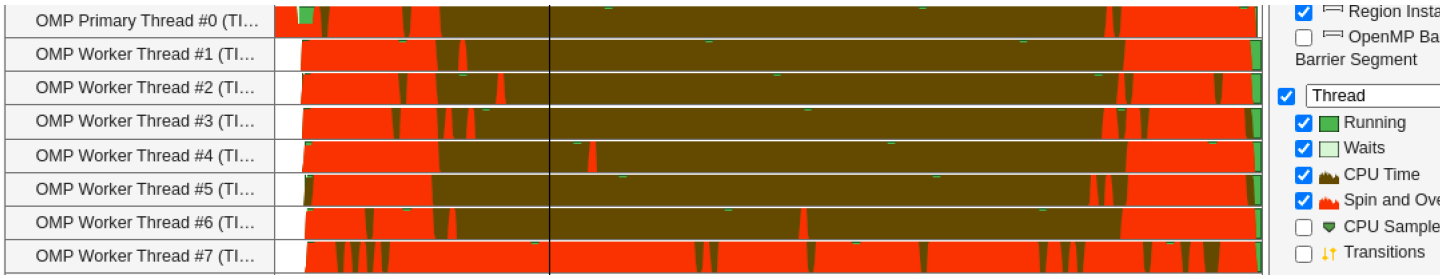
```
[c208-022 cxx:14] for t in 2 3 4 5 6 ; do OMP_NUM_THREADS=$t ./mandeldeque -t $t -s .001 -l 1000 -b 20
=============== #threads = 2 ================8.01974
Time:
Number of points: 16000000
Interior: 1510209
Compute time: 6.47284
================ #threads = 3 ================
Time: 4.85859
Number of points: 16000000
Interior: 1510209
Compute time: 6.37375
================ #threads = 4 ================
Time: 4.31056
Number of points: 16000000
Interior: 1510209
Compute time: 6.40085
================ #threads = 5 ================
Time: 4.26642
Number of points: 16000000
Interior: 1510209
Compute time: 6.37777
================ #threads = 6 ================
Time: 4.18028
Number of points: 16000000
```

```
Interior: 1510209
Compute time: 6.41134

Blocksize 200:
================ #threads = 2 ================
Time: 6.51065
Number of points: 16000000
Interior: 1510209
Compute time: 6.19076
================ #threads = 3 ================
Time: 3.44035
================ #threads = 4 ================
Time: 2.42226
================ #threads = 5 ================
Time: 1.90706
================ #threads = 6 ================
Time: 1.60447

Blocksize 2000:
================ #threads = 2 ================
Time: 6.50357
Number of points: 16000000
Interior: 1510209
Compute time: 6.43529
================ #threads = 3 ================
Time: 3.23416
================ #threads = 4 ================
Time: 2.21621
================ #threads = 5 ================
Time: 1.66643
================ #threads = 6 ================
Time: 1.36181
```

Four threads:



Eight:



目前尚不清楚开头和结尾的空闲时间来源。

第 50 章

数据并行网格

在本节中，我们将逐步构建一个半现实的示例程序。为了帮助你入门，一些部分已经编写好了：作为起点，请查看 `code/mpi/c/grid.cxx`。

50.1 问题描述

通过这个示例，你将研究几种实现简单迭代方法的策略。假设你有一个二维数据点网格 $G = \{g_{ij} : 0 \leq i < n_i, 0 \leq j < n_j\}$，你想计算 G'，其中

$$g'_{ij} = 1/4 \cdot (g_{i+1,j} + g_{i-1,j} + g_{i,j+1} + g_{i,j-1}). \tag{50.1}$$

这在顺序实现时相当简单，但在并行实现时需要一些注意 .

让我们划分网格 G 并将其划分到一个二维的 $p_i \times p_j$ 处理器网格上。（还有其他策略，但这个扩展性最好；参见 HPC book，第 7.5 节。）形式上，我们定义两组点序列

$$0 = i_0 < \cdots < i_{p_i} < i_{p_i+1} = n_i, \quad 0 < j_0 < \cdots < j_{p_j} < i_{p_j+1} = n_j$$

并且我们说处理器 (p, q) 计算 g_{ij} 对于

$$i_p \leq i < i_{p+1}, \quad j_q \leq j < j_{q+1}.$$

从公式（ 50.1 ）可以看出，处理器需要其网格部分两侧各一行的点。图示更为清晰；参见图 50.1。这些环绕处理器自身部分的元素称为 *halo* 或 *ghost region* 。

问题在于 halo 中的元素存储在不同的处理器上，因此需要通信来收集它们。在接下来的练习中，你将需要使用不同的策略来实现这一点。

50.2 Code basics

程序需要从命令行读取网格大小和处理器网格大小的值，以及迭代次数。该例程会进行一些错误检查：如果处理器数量与 `MPI_COMM_WORLD` 的大小不匹配，则返回非零错误代码。

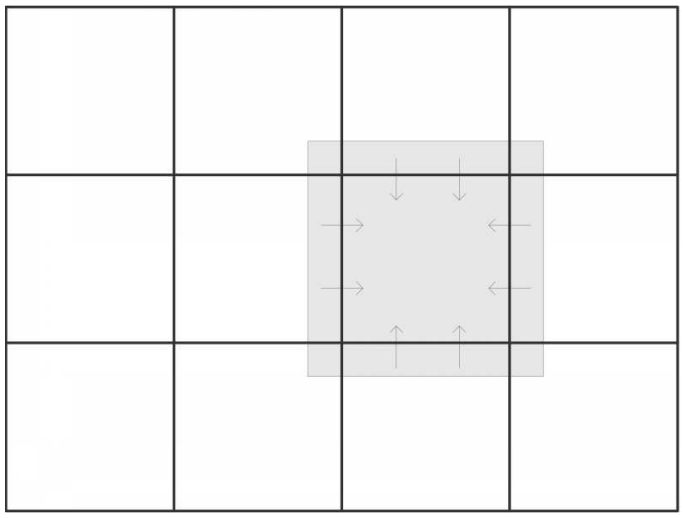


图 50.1：一个划分到多个处理器的网格，标出了 “ 幽灵 ” 区域

```
ierr = parameters_from_commandline
  (argc,argv,comm,&ni,&nj,&pi,&pj,&nit);
if (ierr) return MPI_Abort(comm,1);
```

From the processor parameters we make a processor grid object:

```
processor_grid *pgrid = new processor_grid(comm,pi,pj);
```

and from the numerical parameters we make a number grid:

```
number_grid *grid = new number_grid(pgrid,ni,nj);
```

Number grids 定义了多种方法。要将属于某个处理器的所有元素的值设置为该处理器的编号：

```
grid->set_test_values();
```

To set random values:

```
grid->set_random_values();
```

如果你想可视化整个网格，下面的调用会将所有值收集到处理器零上并打印出来：

```
grid->gather_and_print();
```

接下来我们需要查看一些数据结构的细节。

`number_grid` 对象的定义开始如下：

```
class number_grid {public:

processor_grid *pgrid;

double *values,*shadow;
```

其中 `values` 包含处理器拥有的元素， `shadow` 旨在包含数值加上幽灵区域。那么 `shadow` 是如何接收那些数值的呢？调用看起来是这样的

```
grid->build_shadow();
```

你需要提供该调用的实现。一旦完成，就有一个例程可以打印出每个处理器的影子数组

```
grid->print_shadow();
```

在文件 `code/mpi/c/grid_impl.cxx` 中，你可以看到宏 `INDEX` 的几种用法。它将二维坐标系转换为一维。它的主要用途是让你使用 (i,j) 坐标来索引处理器网格和数值网格：对于处理器，你需要转换为线性秩 (rank)，而对于网格，你需要转换为存储数值的线性数组。

一个使用 `INDEX` 的好例子是在 `number_grid::relax` 例程中：它从 `shadow` 数组中取点并将它们平均到 `values` 数组的一个点中。（要理解这种特定平均的原因，请参见 HPC 书，章节 -4.2.3 和 HPC 书，章节 -5.5.3。）注意宏 `INDEX` 是如何用来索引 `ilength×jlength` 目标数组 `values`，同时从 $(\texttt{ilength}+2)\times(\texttt{jlength}+2)$ 源数组 `shadow` 读取的。

```
for (i=0; i<ilength; i++) {for (j=0; j<jlength; j++) {
int c=0;double new_value=0.;for (c=0; c<5; c++) {
int ioff=i+1+ioffsets[c], joff=j+1+joffsets[c];
new_value += coefficients[c] *
shadow[ INDEX(ioff,joff,ilength+2,jlength+2) ];}
values[ INDEX(i,j,ilength,jlength) ] = new_value/8.;}}
```

第 51 章

N 体问题

N 体问题描述了粒子在诸如引力等力的作用下的运动。对此问题有许多方法，有些是精确的，有些是近似的。这里我们将探讨其中的若干方法。

有关背景阅读，请参见 HPC 书籍，第 11 节。

51.1 解决方法

本课程不涵盖对所有求解 N 体问题的方法（无论是精确还是近似）进行系统性论述，因此我们只考虑具有代表性的一些方法。

1. 全 N^2 方法。这些计算所有相互作用，是最准确的策略，但也是计算量最大的。 2. 基于截断的方法。这些使用 N^2 相互作用的基本思想，但通过对相互作用距离施加截断来降低复杂度。

3. 基于树的方法。这些对远距离相互作用应用粗化方案以降低计算复杂度。

51.2 共享内存方法

51.3 分布式内存方法

51. N 体问题

第七部分

教学法

第 52 章

教学指南

基于每周两次讲座，以下是如何在大学课程中教授 MPI 的提纲。相关链接 t
练习。

主题	练习	Week
	模块 1：SPMD 和集合操作	
介绍：集群结构 功能性并行	`hello`: 2.1, 2.2 `commrank`: 2.4, 2.5, `prime`: 2.6	第 1 周
Allreduce, broadcast	3.1, `randommax`: 3.2 `jordan`: 3.8	week 2
Scan, Gather	3.13, `scangather`: 3.11, 3.15	
	模块 2：双向点对点	第 3 周
发送和接收	`pingpong`: 4.1, `rightsend`: 4.4	
Sendrecv 非阻塞	`bucketblock`: 4.6, `sendrecv`: 4.8, 4.9 `isendirecv`: 4.13, `isendirecvarray`: 4.14 `bucketpipenonblock`: 4.11	第 4 周
	Block 3: Derived datatypes	第 5 周
连续的，向量的，索引的	`stridesend`: 6.4, `cubegather`: 6.6	
Extent and resizing		
	Block 4: Communicators	
Duplication, split 组	`procgrid`: 7.1, 7.2	week 6
	模块 5：I/O	
File open, write, 视图	blockwrite: 10.1, viewwrite 10.4	
	模块 6：邻域集合操作	第 7 周
邻居 allgather	rightgraph: 11.1	

第 53 章

基于心理模型的教学

分布式内存编程，通常通过 MPI 库，是编写大规模并行程序（可达数百万个独立进程）的事实标准。其主导范式是单程序多数据（ SPMD ）编程，这与线程和多核并行有很大不同，以至于学生很难切换模型。与允许通过中央控制和中央数据代码仓库来查看执行的线程编程不同，SPMD 编程采用对称模型，所有进程始终处于活动状态，且无任何特权，数据是分布式的。

该模型对初学的并行程序员来说是反直觉的，因此在灌输正确的 “ 心理模型 ” 时需要格外小心。采用错误的心理模型会导致代码出错或效率低下。

我们指出了当前常见的 MPI 教学方式存在的问题，并提出了一种结构化的 MPI 课程设计，旨在明确强化对称模型。此外，我们主张从现实场景出发，而不是仅仅编写人为代码来练习新学的例程。

53.1 引言

The MPI library [25, 22] 是事实上的大型并行计算工具，因为它被用于工程科学中。本文我们想讨论它通常的教学方式，并提出重新思考。

我们认为这些主题通常按照实现复杂度的顺序进行教学，而不是基于概念上的考虑。我们的论点是，主题的排序和示例的使用应基于 MPI 库的典型应用，并明确针对 MPI 所依赖的并行模型所需的思维模型。

我们编写了一本开源教科书 [9] ，其中的练习集遵循所提议的主题排序和激励应用。

53.1.1 MPI 简要背景

MPI 库起源于集群计算的早期，即 1990 年代上半期。它是一次学术与工业的合作，旨在统一早期常常是厂商特定的消息传递库。MPI 通常用于编写大规模有限元方法（ FEM ）和其他物理仿真应用，这些应用具有相对静态的大量数据分布的特点 —— 因此使用集群来增加目标问题的规模 —— 并且需要非常高效地交换少量数据。

MPI 的主要动机在于它可以扩展到或多或少任意规模，目前可达数百万核心 [1]。与此形成对比的是线程编程，其规模更多地受限于单个节点上的核心数，目前约为 70 个。

C考虑到这一背景，MPI 教学的目标受众包括高年级本科生，
g研究生，甚至首次参与大规模模拟的博士后研究人员。
TMPI 课程的典型参与者可能已经理解了线性代数的基础知识以及一些
a偏微分方程（ PDE ）数值计算的内容。

53.1.2 分布式内存并行

对应其在集群计算中的起源，MPI 以分布式内存并行为目标 1。在这里，网络连接的集群节点运行不共享数据的代码，但通过网络上的显式消息进行同步。其主要的并行模型被描述为单程序多数据（ SPMD ）：单个程序的多个实例在处理单元上运行，每个实例操作自己的数据。MPI 库随后实现了允许进程组合和交换数据的通信调用。

虽然 MPI 程序可以解决许多或所有可以用多核方法解决的问题，但编程方法不同，并且需要调整程序员对并行执行的 “ 心理模型 ” [7, 30] 。本文探讨了如何教授 MPI 以最好地实现这种思维转变的问题。

本文大纲。我们使用第 53.2 节明确讨论支配并行思维和并行编程的心理模型，指出 MPI 为什么不同且起初难以掌握。在第 53.3 节，我们考虑 MPI 通常的教学方式，而在第 53.4 节，我们提供一种不太可能导致错误心理模型的替代方法。

我们提出的教学方式的一些细节在章节 53.5 、 53.6 、 53.7 中进行了探讨。我们在章节 53.8 和 53.9 中以讨论作结。

53.2 隐含的心理模型

Denning [8] 论述了计算思维在于找到一个抽象机器（ ‘ 计算模型 ’ ），以简单的算法方式解决问题。在我们教授并行编程的案例中，复杂之处在于待解决的问题本身已经是一个计算系统。这并不减少构建抽象模型的必要性，因为对初学者来说，MPI 工作原理的完整解释是难以驾驭的，且在实际应用中通常也不需要。

本节我们将更详细地考虑学生可能隐含采用的心理模型及其存在的问题；针对正确心理模型的定位将成为后续章节的主题。分布式内存编程正确心理模型的两个（相互关联的）方面是控制和同步。我们在此讨论学生可能对这两者的误解。

53.2.1 并行性的传统观点

掌握 MPI 库的问题在于初学者需要一段时间来克服对并行性的某种思维模型。在这个模型中，我们可以称之为 “ 顺序语义 ” （或者更俏皮地称为 “ 大食指 ” 模型），只有一条执行线 2，我们可以把它看作是一根沿着源代码向下移动的大食指。

1. 最近对 MPI 标准的补充也针对共享内存。 2. 我们谨慎避免使用 “thread” 一词，因为它在并行编程的语境中带有许多含义。

这个心理模型与并行数学文献中描述算法的方式密切相关，并且在诸如 OpenMP 这样的线程库环境中在某种程度上是正确的，因为最初确实只有一个执行线程，在某些地方会生成一个线程组以并行执行某些代码段。然而，在 MPI 中这个模型实际上是不正确的，因为始终有多个进程处于活动状态，且本质上没有任何进程享有特权，也不存在共享或中央数据存储。

53.2.2 中央控制的误解

如上所述，顺序语义心理模型是并行理论讨论的基础，它促使学习者采用某些编程技术，例如主 - 工作者（ master-worker ）并行编程方法。虽然这在基于线程的编码中通常是正确的方法，因为我们确实有一个主线程和生成的线程，但对于 MPI 来说通常是不正确的。MPI 运行中的执行线索都是长期存在的进程（而非动态生成的线程），并且在它们的能力和执行上是对称的。

缺乏对这一过程对称性的认识也导致学生通过在一个进程上设置 “ 中央数据存储 ” 的方式来解决问题，而不是采用对称的分布式存储模型。例如，我们曾见过学生通过将所有数据收集到进程 0 上，然后再以转置形式重新分发来解决数据转置问题。虽然这在共享内存和 OpenMP 中可能是合理的[3]，但在 MPI 中这是不现实的，因为没有哪个进程可能拥有足够的存储空间来容纳整个问题。此外，这还在执行中引入了顺序瓶颈。

总之，我们认为初学 MPI 的程序员可能存在一种心理模型，使他们无法充分认识到 MPI 进程的对称性，从而导致效率低下且不可扩展的解决方案。

53.2.3 The realityof distributed control

一个 MPI 程序运行由多个独立的控制线程组成。识别这一点的一个问题是只有单一的源代码，因此人们倾向于将程序执行想象成单线程控制：上述的 “ 食指 ” 沿着源代码语句向下执行。促成这种观点的第二个因素是并行代码包含在所有进程中复制的值（ `int x = 1.5;` ）的语句。很容易将这些视为集中执行的。

有趣的是，Ben-David Kolikant 的研究 [2] 表明，没有并发先验知识的学生，在被邀请考虑并行活动时，仍然会以集中式解决方案的思维方式来思考。这表明分布式控制，比如在 MPI 中出现的，是反直觉的，需要在其思维模型中明确强化。特别是，我们明确针对进程对称性和进程差异化。

集中式模型在 MPI 中仍然可以在一定程度上维持，因为本应由单个线程执行的标量操作变成了 MPI 进程中的复制操作。许多学生一开始难以区分顺序执行和复制执行，实际上，由于解释这一点没有任何收益，我们也就不做解释。

53.2.4 The misconception of synchronization

即使有多个控制线程和分布式数据，仍然存在将执行视为 “ 批量同步处理 ” （ BSP [29] ）的倾向。在这里，执行通过超级步骤进行，意味着进程大体上是同步的。（ BSP 模型还有更多组件，通常被忽略，特别是一侧通信和处理器超额订阅。）

3. 一阶效应；亲和性等二阶效应使情况变得复杂。

超级步骤作为一种计算模型允许控制流中存在小的差异，例如在一个大型可并行循环内的条件语句，但除此之外，在主要算法步骤层面上暗示了一种集中控制的形式（如上所述）。然而，使用流水线并行模型的代码，如

```
MPI_Recv( /* from: */ my_process-1)
// do some major work
MPI_Send( /* to  : */ my_process+1)
```

完全超出了顺序语义或 BSP 模型的范畴，需要理解一个进程的控制依赖于另一个进程的控制。获得这种非同步执行的心智模型并非易事。我们在第 53.5.1 节中明确针对这一点进行讨论。

53.3 以常规方式教学 MPI

MPI 库的教学通常如下进行。在介绍并行性后，涵盖加速比以及共享内存与分布式内存并行等概念，学生们学习初始化和终结例程，以及用于查询进程数量和当前进程排名的 `MPI_Comm_size` 和 `MPI_Comm_rank` 调用。

之后，典型的顺序是

1. 双边通信，先是阻塞版本，后是非阻塞版本；2. 集合操作；3. 以及任意数量的高级主题，如派生数据类型、单边通信、子通信器、 MPI I/O 等，顺序不限。

从底层实现的角度来看，这个顺序是有道理的：双边通信调用与硬件行为紧密对应，集合操作在概念上等同于点对点通信调用序列，并且可以作为点对点通信调用序列来实现。然而，这并不足以作为教授这一系列主题的充分理由。

53.3.1 Criticism

我们针对这种传统的 MPI 教学方法提出三点批评。

首先，没有真正的理由在双边例程之后再教授集合操作。它们并不更难，也不需要后者作为前提。事实上，对于初学者来说，它们的接口更简单，一个集合操作只需一行代码，而发送 / 接收对至少需要两行代码，且可能还要加上测试进程秩的条件语句。更重要的是，它们强化了对称进程的视角，尤其是在 `MPI_All...` 例程的情况下。

我们的第二个批评点是关于阻塞和非阻塞的双边通信例程。阻塞例程通常先被教授，并讨论阻塞行为如何导致负载不平衡，从而导致效率低下。然后，非阻塞例程的动机是为了隐藏延迟并解决阻塞中固有的问题。在我们看来，这样的性能考虑应该是次要的。非阻塞例程应被作为一个概念性问题的自然解决方案来教授，如下所述。

第三，起始于点对点例程源于 Communicating Sequential Processes (CSP)[14] 视角：每个进程独立存在，任何全局行为都是运行的涌现属性。这对于了解 “ 底层 ” 概念实现的教师来说可能有意义，但对学生并不会带来额外的洞见。我们认为，更有成效的 MPI 编程方法应从全局行为出发，然后以自顶向下的方式推导 MPI 进程。

53.3.2 TeachingMPI 和 OpenMP

在科学计算中，另一个常用的并行编程系统是 OpenMP [23]。OpenMP 和 MPI 通常一起教授，OpenMP 通常先教，因为它据说更简单，或者因为它的并行性更容易理解。无论我们对第一个估计的看法如何，我们认为 OpenMP 应该在 *MPI* 之后教授，因为它采用 “ 中央控制 ” 的并行模型。如果学生将并行性与具有 “ 主线程 ” 和 “ 并行区域 ” 的模型联系起来，他们会发现更难以习惯性地使用 MPI 的对称模型。

53.4 TeachingMPI，我们的提议

作为引入 MPI 概念的上述顺序的替代方案，我们提出了一个侧重于实际场景的顺序，并积极强化 SPMD 执行的心理模型。

这种强化通常是我们在应用上下文中说明 MPI 构造策略的直接结果：大多数 MPI 应用（正如我们接下来将简要讨论的）操作大型 “ 分布式对象 ”。这立即导致了每个进程的工作方式的心理模型，即该进程对全局计算的 “ 投影 ”。相反的观点是，上述的 CSP 模型，其中整体计算是由各个进程共同产生的。

53.4.1 来自应用的动机

MPI 的典型应用来自计算科学与工程领域，如 N 体问题、空气动力学、浅水方程、格子玻尔兹曼方法、利用快速傅里叶变换的天气建模。在这些应用中，基于 PDE 的应用可以很容易地解释为何需要多种 MPI 机制。

存在非数值应用：

- 图算法如最短路径或 PageRank 在顺序解释时非常直观。然而，分布式内存算法需要从根本上不同于更简单的共享内存变体来处理。因此，它们需要大量的背景知识。此外，它们不具备一维 PDE 应用中常见的规律性通信。可扩展性论证使得这个问题更加复杂。因此，这些算法实际上是并行 PDE 算法讨论之后的一个合乎逻辑的下一个主题 *after*。

  • N 体问题，在其简单实现中，对于任何了解反平方定律（如引力）的学生来说都很容易解释。它很好地展示了一些集合操作，但除此之外没有更多内容。

- 排序。基于排序网络的排序算法（这包括冒泡排序，但不包括快速排序）可以用作示例。事实上，我们使用奇偶换位排序作为 “ 期中 ” 考试作业，这可以用 `MPI_Sendrecv` 解决。诸如 bitonic sort 之类的算法可以用来说明一些高级概念，但快速排序作为一个相对容易解释的串行算法，甚至在共享内存中，也在 MPI 中相当困难。
- 点对点操作也可以通过图形操作来说明，比如 “ 模糊 ”，因为这些对应于应用于像素簇的 “ 模板 ”。不幸的是，这个例子存在这样的问题：既没有集合通信，也没有不规则通信在该应用中有用。此外，使用图形来说明简单的 MPI 点对点通信在两个方面是不现实的：首先，为了简单起见，我们必须假设一个一维像素数组；其次，图形几乎从未达到需要分布式内存的规模，所以这个例子远非 “ 现实世界 ”。（光线追踪自然是分布式完成的，但那有完全不同的计算结构。）

基于对可能应用的讨论，并考虑到课程参与者的可能背景，我们将偏微分方程的有限差分解法视为一个典型应用，它既涵盖了最简单的机制，也涵盖了更复杂的机制。在典型的 MPI 培训中，即使是一天的短期课程，我们也会插入一节关于稀疏矩阵及其计算结构的讲座，以激发对各种 MPI 构造需求的理解。

53.4.2 进程对称性

具有讽刺意味的是，让学生理解 MPI 中进程对称性概念的第一种方法是运行一个非 MPI 程序。因此，要求学生编写一个 “hello world” 程序，并用 `mpiexec` 执行它，就好像它是一个 MPI 程序一样。每个进程都以相同的方式执行打印语句，体现了进程之间的完全对称性。

接下来，要求学生插入初始化和终结语句，并在它们之前、之间和之后各放置三个不同的 “hello world” 语句。这将防止将初始化和终结之间的代码视为 OpenMP 风格的 “ 并行区域 ”。

一个简单的测试用来展示虽然进程是对称的，但它们并不相同，这个测试是使用 `MPI_Get_processor_name` 函数的练习，该函数对于某些或所有进程会有不同的输出，具体取决于 hostfile 的排列方式。

53.4.3 功能并行

`MPI_Comm_rank` 函数被引入作为区分 MPI 进程的一种方式。要求学生编写一个程序，其中只有一个进程打印 `MPI_Comm_size` 的输出。

具有不同执行但不一定不同数据的情况称为 “ 功能并行 ”。此时我们能分配的例子很少。例如，为了通过黎曼和计算积分（ $\pi/4 = \int_0^1 \sqrt{1-x^2}dx$ 是一个流行的例子）需要一个最终求和的 collective 操作，而这在此时尚未教授。

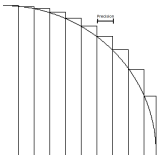


图 53.1：通过黎曼和计算 $\pi/4$

一个可能的例子是素数测试，其中每个进程尝试通过遍历 $[2, \sqrt{N}]$ 的一个子区间来寻找某个大整数 N 的因子，如果找到因子则打印一条消息。布尔可满足性问题是另一个例子，同样是将搜索空间划分而不涉及任何数据空间；找到满足输入的进程可以简单地打印这一事实。然而，这个例子需要学生通常不具备的背景知识。

53.4.4 引入集合通信

此时我们可以引入集合操作，例如用来寻找随机数的最大值
在每个进程上本地计算的值。这需要为随机数生成编写代码
随机数生成，重要的是设置与进程相关的随机数种子。生成随机的二维或三维坐标并找到质心是一个示例，这需要长度大于 1 的发送和接收缓冲区，并且说明归约操作是逐点完成的。

这些示例既展示了进程对称性，也展示了局部数据的第一种形式。然而，关于分布式并行数据的全面讨论将在点对点例程的部分进行。

是否应该放弃 “ 有根 ” 集合操作（如 `MPI_Reduce` ）而从 `MPI_Allreduce`[4] 开始，这是一个有趣的问题。后者本质上更对称，且其缓冲区处理更为简便

4. The `MPI_Reduce` call 对所有进程上的数据执行归约操作，结果保留在 “root” 进程上。使用 `MPI_Allreduce` 时，结果保留在所有 进程上。

解释；这无疑加强了对称思维方式。效率上基本没有差别。

当然，在大多数应用中，“allreduce” 是更常见的机制，例如算法需要计算诸如

$$\bar{y} \leftarrow \bar{x}/\|\bar{x}\|$$

其中 x, y 是分布式向量。量 $\|x\|$ 需要在所有进程上使用，因此 `Allreduce` 是自然的选择。根归约通常只用于最终结果。因此，我们主张引入根和非根集合操作，但让学生最初先做非根变体的练习。

这还有一个额外的好处，就是最初不会让学生为根进程和其他所有进程之间接收缓冲区的非对称处理感到困扰。

53.4.5 Distributed data

作为对下面点对点例程讨论的动机，我们现在引入分布式数据的概念。在最简单的形式中，一个并行程序操作一个线性数组，其维度超过任何单个进程的内存。

```
int n;
double data[n];
```

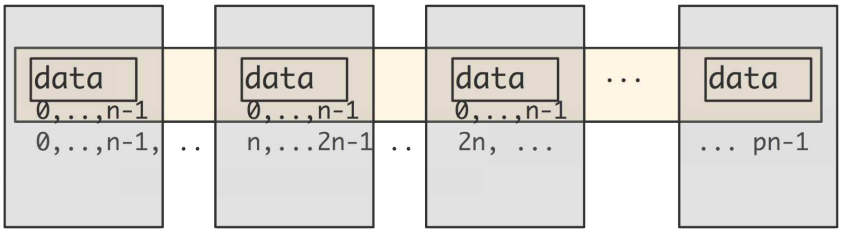


图 53.2：分布式数组与多个本地数组

讲师强调分布式数组的全局结构仅存在于 “ 程序员的脑海中 ”： 每个 MPI 进程看到的数组索引都是从零开始。以下代码片段供学生在后续练习中使用：

```
int myfirst = .....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
   int iglobal = myfirst+ilocal;
   array[ilocal] = f(iglobal);
}
```

此时，学生们可以编写上述素数测试练习的第二个变体，但使用一个数组来存储整数范围。由于现在已了解 collectives，可以实现由一个进程给出单一汇总语句，而不是每个进程给出部分结果语句。

两个分布式向量的内积是处理分布式数据的第二个示例。在这种情况下，用于收集全局结果的归约操作比前面示例中的集合操作更有用。对于这个例子，不需要从局部编号转换到全局编号。

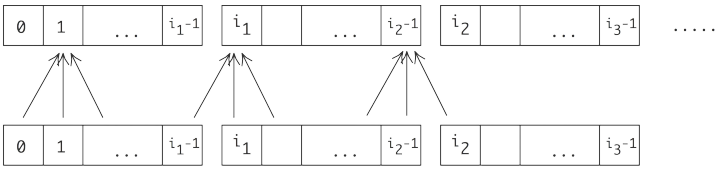
53.4.6 基于分布式数据操作的点对点通信动机

我们现在说明局部组合操作（例如

$$y_i = \big(x_{i-1} + x_i + x_{i+1}\big)/3 : i = 1, \ldots, N-1$$

应用于数组的重要性。了解偏微分方程（ PDEs ）的学生会认识到，带有不同系数时这就是热方程；对于其他人，如果他们接受一维像素数组是实际图形的替代，那么图形的 “ 模糊 ” 操作可以作为说明。

在 “ 所有者计算 ” 机制下，存储位置 y_i 的进程执行该量的完整计算，我们看到为了计算 y 局部部分的第一个和最后一个元素，需要通信：



然后我们说明，这种数据传输在 MPI 中通过双边的发送 / 接收对来实现。

53.4.7 绕行：死锁和串行化

现在引入了 “ 阻塞 ” 的概念，并讨论了这如何导致死锁。一种更微妙的行为是 “ 意外串行化 ”：进程交互导致在概念上应并行的代码表现为串行行为。（课堂协议在第 53.5.1 节中有详细讨论。）为了完整性，也可以讨论 “ 急切限制 ”。

这使学生了解并行正确性概念中的一个有趣现象：程序可能给出正确结果，但并没有达到适当的并行效率。让一个班级提出一个运行时间不与进程数成正比的解决方案，通常会至少有一名学生建议将进程分为奇数和偶数子集。向学生解释这种方法的局限性、代码复杂性以及对规则进程连接的依赖，作为非阻塞发送动机的前奏；详见第 53.4.10 节。

53.4.8 绕行：乒乓

此时我们暂时放弃进程对称性，考虑两个进程 A 和 B 之间的乒乓操作 [5]。我们让学生思考 A 和 B 的乒乓代码是什么样的。由于我们使用的是 SPMD 代码，我们得到的程序中 A 代码和 B 代码是条件语句的两个分支。

我们让学生实现这个，并用 `MPI_Wtime` 进行计时。乒乓操作的实现本身是 SPMD 思维的一个很好的练习；找到正确的发送者 / 接收者值通常会让学生花费不少时间。他们中的许多人最初会写出死锁的代码。

可以引入延迟和带宽的概念，当学生测试不同大小消息的乒乓代码时。通过让一半的进程与另一半的伙伴进程执行乒乓操作，可以引入半带宽的概念。

5. 在这些操作中，进程 A 发送给 B，随后 B 发送给 A。因此，一条消息的时间是 ping-pong 时间的一半。无法直接测量单条消息的时间，因为进程无法做到如此精细的同步。

53.4.9 数据交换回顾

T前面关于双向发送和接收调用行为的绕行是必要的，但它们引入了
进程中的非对称行为。我们回到上述的平均操作，并据此回到一个对所有进程对称处理的代码。特别地，我们认为，除了第一个和最后一个，每个进程都与其左邻居和右邻居交换信息。

这可以用阻塞发送和接收调用来实现，但学生们会意识到这可能介于繁琐和易出错之间。相反，为了防止如上所述的死锁和串行化，我们现在提供了 `MPI_Sendrecv` 例程 [6]。学生们被要求用 sendrecv 例程实现上述课堂练习。理想情况下，他们使用计时或跟踪来收集没有发生串行化的证据。

作为一个非平凡的例子（事实上，这需要足够的编程量，可能会被作为考试题目而非研讨会中的练习题），学生们现在可以使用 `MPI_Sendrecv` 作为主要工具实现奇偶换位排序算法。为简化起见，他们可以为每个进程使用单个数组元素。（如果每个进程有一个子数组，则必须确保他们的解决方案具有正确的并行复杂度。在这里很容易出错，实现一个正确的算法，但其性能却过慢。）

请注意，到目前为止，学生们除了乒乓练习外，还没有进行任何使用阻塞通信调用的严肃练习。实际上也不会进行此类练习。

53.4.10 非阻塞发送

非阻塞发送现在被引入作为解决一个特定问题的方案：上述方案需要成对的进程，或者对发送和接收序列进行精心协调。在不规则通信的情况下，这已不再可能或可行。如果我们能够声明 “ 这部分数据需要发送 ” 或 “ 这些消息是预期的 ”，然后集体等待这些消息，生活将会变得简单。基于这一动机，立即可以看出需要多个发送或接收缓冲区，并且需要收集请求。

Implementing the th使用非阻塞调用的三点平均此时是一个 n excellent exercise.

请注意，我们在这里以解决对称问题为动机引入非阻塞例程。这样做应当教会学生一个关键点：每个非阻塞调用都需要自己的缓冲区并生成自己的请求。将非阻塞例程视为阻塞例程的性能替代方案，可能导致学生重复使用缓冲区或未能保存请求对象。这样做是一个非常难以发现的正确性错误，并且在大规模情况下会导致内存泄漏，因为许多请求对象会丢失。

53.4.11 从这里开始

此时可以讨论各种高级主题。例如，可以引入笛卡尔拓扑，将线性平均操作扩展到更高维度。还可以引入子通信器，以便对矩阵的行和列应用集合操作。递归矩阵转置算法也是子通信器的一个极佳应用。

然而，从教学角度来看，这些主题不需要像基本概念引入那样的细致关注，因此我们在这里不会进一步详细讨论。

6. `MPI_Sendrecv` 调用结合了发送和接收操作，为每个进程指定了发送和接收通信。执行时保证不会发生死锁或串行化。

53.5 ‘并行计算机游戏’

开发准确的并行计算心理模型的部分问题在于没有简单的方法来可视化执行过程。虽然可以用 “ 大拇指 ” 模型来想象顺序执行（参见章节 53.2.1 ），但 MPI 程序可能不同步的执行使得这种想象过于简化。在并行图形环境中运行程序（如 DDT 调试器或 Eclipse PTP IDE ）可以解决这个问题，但它们带来了较大的学习负担。具有讽刺意味的是，低技术含量的解决方案

```
mpiexec -n 4 xterm -e gdb program
```

相当有见地，但必须学习 gdb 又是一个很大的障碍。

我们采用了一个有些不同寻常的解决方案，让学生在课堂上扮演程序的角色。每个学生扮演程序中的一部分，任何交互都能清晰可见，这种效果是其他方式难以达到的。

53.5.1 顺序化

我们的主要示例是为了说明 `MPI_Send` 和 `MPI_Recv`[7] 的阻塞行为。死锁作为阻塞的结果很容易理解 —— 在最简单的死锁情况下，两个进程都被阻塞，等待从对方接收 —— 但还有一些更微妙的效果会让学生感到惊讶。（这在 53.4.7 节中有所提及。）

考虑以下基本程序：

- 将一个数据项传递给编号更高的下一个进程 .

N注意，这在概念上是一个完全并行的程序，因此它应该在 $O(1)$ 的时间内执行，时间单位以进程数计 p。

就发送和接收调用而言，程序变为

- 向下一个更高的进程发送数据；
- 从下一个更低的进程接收数据。

最后一个细节涉及边界条件：第一个进程没有数据可接收，最后一个进程没有数据可发送。这使得程序的最终版本为：

- 如果你不是最后一个进程，向下一个更高的进程发送数据；然后
- 如果你不是第一个进程，从下一个更低的进程接收数据。

为了让学生们表演这个场景，我们告诉他们右手拿笔，左手放进口袋或背后。这样，他们只有一个 “ 通信通道 ”。“ 发送数据 ” 指令变成 “ 向右转并递出你的笔 ”，“ 接收数据 ” 变成 “ 向左转并接收一支笔 ”。

执行该程序时，学生们首先全部向右转，他们发现无法将数据传递给邻居，因为没有人执行接收指令。最后一个进程没有发送，因此继续执行接收指令，之后倒数第二个进程才能接收，依此类推。

这个练习让学生比任何解释或图示都更清楚地看到，一个并行程序如何能够计算出正确的结果，但由于进程之间的相互作用，性能却出乎意料地低。（事实上，我们收到了明确的反馈，称这个游戏是课堂上最大的顿悟时刻。）

7. 阻塞被定义为执行发送或接收调用的过程暂停，直到相应的操作正在执行。

53.5.2 Ping-pong

虽然我们通常强调 MPI 进程的对称性，但在讨论发送和接收调用时，我们通过演示乒乓操作（一个进程向另一个进程发送数据，随后另一个进程将数据发送回去）来展示如何处理非对称操作。为此，两个学生互相抛接一支笔，并在抛接时喊出 “send” 和 “receive”。

T然后老师询问每个学生执行了什么程序，一个是 “send-receive”，另一个是 “receive-send”。将此纳入 SPMD 模型后，会得到带有条件语句的代码，用以确定
t正确进程的正确操作。

53.5.3 集合操作和其他游戏

其他操作可以由类来扮演。例如，老师可以让一个学生将所有学生的成绩相加，作为归约操作的代理。全班很快就会发现这将花费很长时间，并且像在教室里按行求和这样的策略很快就会被提出。

我们曾经尝试让一对学生扮演共享内存编程中的 “ 竞态条件 ”，但这种模拟很快变得过于复杂，难以令人信服。

53.5.4 剩余问题

然而，即使采用我们当前的方法，我们仍然看到学生编写的习惯用法与对称模型相悖。例如，他们会写

```
for (p=0; p<nprocs; p++)
  if (p==myrank)
    // do some function of p
```

这段代码计算出了正确的结果，并且具有正确的性能表现，但它仍然显示出一个概念上的误解。作为 ‘ 并行计算机游戏 ’ 之一（章节 53.5 ），我们让一个学生站在班级前面，举着一个写着 ‘ 我是进程 5’ 的牌子，并大声朗读上述循环（ ‘ 我是进程 0 吗？不是。我是进程 1 吗？不是。’ ），这很快让大家明白了这个结构的无用之处。

53.6 进一步课程总结

我们基于上述思想以两种方式教授 MPI。首先，我们开设一门学术课程，涵盖 MPI 、 OpenMP 以及并行理论的一般知识，课程周期为一个学期。典型的报名人数约为 30 人，学生们完成实验练习和一个自选的编程项目。我们还开设一个为期两天的强化研讨班（根据情况，参加人数为 10 至 40 人），每天 6 至 8 小时。学术课程的学生通常是研究生或高年级本科生；研讨班的参与者则包括博士后、学者和业界人士。典型的背景是应用数学、工程和物理科学。

我们涵盖以下主题，分为两天的研讨会形式 :

- 第 1 天：熟悉 SPMD 、集合操作、阻塞和非阻塞的双边通信。
- 第 2 天：接触子通信器、派生数据类型。以下内容中任选两项：MPI-I/O 、单边通信、进程管理、性能分析和工具接口、邻域集合操作。

53.6.1 练习

第一天学生完成大约 10 个编程练习，大多是完成由讲师提供的骨架代码。对于第二天的内容，学生每个主题做两个练习，同样从给定的骨架开始。（骨架代码作为代码仓库的一部分提供 [9]。）

考虑到我们对心理模型的关注，这些骨架代码的设计是一个有趣的问题。骨架代码旨在减轻学生的繁重工作，既指示基本的代码结构，又避免他们犯与学习 MPI 无关的基础编码错误。另一方面，骨架代码应留有足够的未指定部分，使得多种解决方案成为可能，包括错误的方案：我们希望学生面对他们思维中的概念性错误，而过于完善的骨架代码会阻止他们这样做。

示例：上述提到的寻找素数的练习（教授函数式并行的概念）具有以下骨架：

```
int myfactor;// Specify the loop header:
// for ( ... myfactor ... )for (
/**** your code here ****/) {
if (bignum%myfactor==0)
printf("Process %d found factor %d\n",
procno,myfactor);}
```

这留下了搜索空间既可以按块分配也可以按循环分配的可能性，同时也存在错误的解决方案，即每个进程遍历整个搜索空间。

53.6.2 项目

我们学术课程中的学生以编程项目代替期末考试。学生可以选择一组标准项目中的一个，或者选择自己喜欢的项目。在后一种情况下，一些学生会在其研究生研究的背景下做项目，这意味着他们已有现成的代码库；其他人则从零开始编写代码。正是最后这一类学生，最能清楚地展示他们对 SPMD 程序底层思维模型的正确理解。然而，我们注意到，这只占我们课程学生的一小部分，这一比例因我们还提供了选择使用 OpenMP 而非 MPI 进行项目的选项而进一步减少。由于 OpenMP 至少对初学者来说更易使用，因此在选择自己项目的学生中，OpenMP 实际上更受欢迎。

53.7 在线课程的前景

目前，作者以学术课程或短期研讨会的形式教授 MPI，如第 53.6 节所述。在这两种情况下，讲课时间远少于实验时间，使得教师投入非常密集。这也意味着这种设置无法扩展到更多的学生。实际上，虽然研讨会通常通过网络直播，但我们尚未充分解决支持远程学生的问题。（匹兹堡超级计算中心提供有远程助教的课程，这似乎是一种有前景的方法。）在完全没有面对面支持的在线课程中，这类支持问题将更加严重。

在线教学的一个明显解决方案是自动评分：学生提交一个练习，然后通过一个检查程序运行，测试正确的输出。特别是如果编程作业需要输入，检查脚本可以发现编程错误，尤其是在边界情况中。

然而，本文的全部目标是揭示概念上的误解，例如那些可能导致正确结果但性能不佳的情况。在课堂环境中，这类误解很快被发现并澄清，但要在自动评分的环境中实现这一点，我们需要更进一步。

我们已经开始尝试实际解析学生提交的代码。这项工作始于作者本人教授的一门初级编程课程，但现在正在扩展到 MPI 课程中。

通过检测此类误解的典型表现，可以发现学生理解中的误区。例如，53.5.4 节中的代码可以通过检测一个循环来发现，该循环的上界涉及由 `MPI_Comm_size` 设置的变量。许多 MPI 代码不需要对所有进程进行这样的循环，因此检测到这样的循环会向学生发出警告。

请注意，目前不存在用于此类自动化评估的工具。所需的源代码分析远未达到完整解析的程度。另一方面，它应该检测的那类结构，通常并不是编译器和源代码转换器的编写者感兴趣的内容。这意味着通过编写相当简洁的解析器（例如，不超过 200 行的 python 代码），我们就能对学生的代码进行复杂的分析。我们希望在后续论文中对此进行更详细的报告。

53.8 Evaluation and discussion

目前，还没有对上述想法的有效性进行严格评估。我们打算通过比较两个（或更多）不同机构和不同教学大纲的课程，来对所提教学方法与传统方法的结果进行比较。评估将基于对独立编程项目的评价。

然而，轶事证据表明，学生不太可能开发如第 53.2.2 节所述的 “ 集中式 ” 解决方案。特别是在我们为期一个学期的课程中，学生必须设计并实现一个自己选择的并行编程项目。在教授了 “ 对称 ” 方法后，没有学生编写基于管理者 - 工作者模型或使用集中存储的代码。在之前的学期中，我们曾见过学生这样做，尽管这种模型从未被作为教学内容。

53.9 总结

在本文中，我们介绍了一种非标准的顺序来呈现 MPI 中的基本机制。我们不是从发送和接收开始并逐步构建，而是从强调 SPMD 编程模型中进程之间固有对称性的机制开始。这种对称性要求程序员的思维方式发生重大转变，因此我们明确地针对这一点。

I一般来说，作者认为从灌输一个心理模型的基础开始教学是值得的，
t而不是按照某种（感知的）复杂性或精细程度的顺序来呈现主题。

将我们上述的呈现方式与标准呈现方式进行比较时，我们注意到对阻塞发送和接收调用的淡化。虽然学生会学习这些，实际上会在学习其他发送和接收机制之前学习它们，但他们会认识到使用它们的危险和困难，并且会将组合的 sendrecv 调用以及非阻塞例程作为他们工具库中的标准工具。

53. 从心理模型教学

第八部分

参考文献、索引和缩略语列表

第 54 章

参考文献

[1] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur 和 Jesper Larsson Träff。MPI 在百万核心上的应用。《 Parallel Processing Letters 》，21(01):45–60，2011 年。

[2] Y. Ben-David Kolikant。园丁与电影票：高中生对并发的先入之见。*ComputerScience Education*, 11:221–245, 2001。 [Cited on page 587.]

[3] Ernie Chan, Marcel Heimlich, Avi Purkayastha 和 Robert van de Geijn。集体通信：理论、实践与经验。《 Concurrency and Computation: Practice and Experience 》，19:1749–1783，2007 年。第 84 页。

[4] Tom Cornebize, Franz C Heinrich, Arnaud Legrand, 和 Jérôme Vienne。 在超级计算机规模的商用服务器上模拟高性能 Linpack。 工作论文或预印本，2017 年 12 月。 [Cited on 页 559。]Lisandro Dalcin。 MPI for Python，主页。 https://github.com/mpi4py/mpi4py。 [ ]

[5] Cited on page 19.

[6] Lisandro Dalcin 和 Yao-Lung L. Fang。mpi4py：经过 12 年开发后的状态更新。*Computing inScience & Engineering*, 23(4):47–54, 2021。 [Cited on page 19.]

[7] Saeed Dehnadi 、 Richard Bornat 和 Ray Adams。关于一致性对早期编程学习成功影响的元分析。发表于《 Psychology of Programming Interest Group PPIG 2009 》，第 1–13 页。爱尔兰默里克大学，2009 年。第 586 页。

[8] Peter J. Denning. 科学中的计算思维。 *American Scientist*, 第 13–17 页，2017 年。 [Cited 第 586 页。]Victor Eijkhout. *Parallel*

[9] *Programmingin MPI and OpenMP*。2016 年。可下载地址： https://bitbucket.org/VictorEijkhout/parallel-computing-book/src。 [Cited 第 585 和 596 页。]

[10] Victor Eijkhout。非连续数据的 MPI 发送性能。arXiv 电子预印本，第 arXiv:1809.10778 页，2018 年 9 月。第 312 页。

[11] Brice Goglin. 使用硬件局部性（ hwloc ）管理异构集群节点的拓扑结构。发表于 2014 年 7 月意大利博洛尼亚国际高性能计算与仿真会议（ HPCS 2014 ），IEEE，第 553 页。

[12] Google。Google 开发者文档风格指南。 https://developers.google.com/style/. [Cited on 页 562。]

[13] W. Gropp, E. Lusk, 和 A. Skjellum。《 Using MPI 》。MIT 出版社，1994 年。 [ ]

Cited on page 317.

[14] C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985. ISBN-10: 0131532715, ISBN-13: 978-0131532717. 第 588 页。

[15] Torsten Hoefler 、 Prabhanjan Kambadur 、 Richard L. Graham 、 Galen Shipman 和 Andrew Lumsdaine。关于标准非阻塞集合操作的论证。载于《 Proceedings, Euro PVM/MPI 》，法国巴黎，2007 年 10 月，第 78 页。

[16] Torsten Hoefler 、 Christian Siebert 和 Andrew Lumsdaine。用于动态稀疏数据交换的可扩展通信协议。*SIGPLANNot.*, 45(5):159–168, 2010 年 1 月。 [Cited on page 78.]

[17INRIA。SimGrid 主页。

[18] L. V. Kale 和 S. Krishnan。Charm++：基于消息驱动对象的并行编程。载于 *Parallel Prog ramming using C++*，*G. V. Wilson* 和 *P. Lu*，编辑，第 175–213 页。MIT Press，1996 年。 [Cited 第 16 页。]

[19] M. Li, H. Subramoni, K. Hamidouche, X. Lu 和 D. K. Panda。基于用户态内存注册的高性能 mpi 数据类型支持：挑战、设计与收益。载于 *2015 IEEE International Conference onCluster Computing*，第 226–235 页，2015 年 9 月。 [Cited 第 312 页。]

[20] Zhenying Liu, Barbara Chapman, Tien-Hsiung Weng 和 Oscar Hernandez。通过数组私有化提升 openmp 性能。载于 *Proceedings of the OpenMP Applicationsand Tools 2003 InternationalConference on OpenMPShared Memory Parallel Programming*，WOMPAT'03，第 244–259 页，柏林，海德堡，2003 年。Springer-Verlag。 [Cited 第 405 页。]

[21] Robert McLay. T3pio: TACC 出色的并行 I/O 工具。 https://github.com/TACC/t3pio. [Cited 第 262 页。]MPI 论坛：MPI 文档。

[22] http://www.mpi-forum.org/docs/docs.html. [Cited 第 585 页。]

[23] OpenMP 并行编程的 API 规范。http://openmp.org/wp/openmp-specifications/ [Cited onpage 589.]

[24] Amit Ruhela、 Hari Subramoni、 Sourav Chakraborty、 Mohammadreza Bayatpour、 Pouya Kousha 和 Dhabaleswar K. Panda。高效的无专用资源 MPI 异步通信进展。EuroMPI'18，纽约，纽约州，美国，2018。计算机协会。第 307 页。

[25] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker 和 Jack Dongarra。*MPI: The Complete Reference* ，第一卷，*MPI-1* 核心。MIT Press，第二版，1998 年。 [Cited on page 585。]

[26] Jeff Squyres. Mpi-request-free 是邪恶的。Cisco Blogs，2013 年 1 月。 https://blogs.cisco.com/performance/mpi_request_free-is-evil. [Cited onpage 121.]

[27] R. Thakur, W. Gropp, 和 B. Toonen. 优化 MPI 单边通信中的同步操作。*Int'l Journal of High Performance Computing Applications*, 19:119–128, 2005. [Cited on page 253.]

[28] 瓦伦西亚理工大学。SLEPC – 可扩展特征值问题计算软件。 http://www.grycap.upv.es/slepc/. [Cited onpage 455.]

[29] Leslie G. Valiant. 并行计算的桥接模型。*Commun.ACM*, 33:103–111, 1990 年 8 月。 [Cited 第 587 页。]

[30] P.C. Wason 和 P.N. Johnson-Laird. 思考与推理 .Harmondsworth: Penguin, 1968. [Cited onpage 5 86.]

第 55 章

缩略语列表

API 应用程序接口 **AMG** 代数多重网格 **AVX** 高级向量扩展 **BLAS** 基础线性代数子程序 **BSP** 批量同步并行 **CAF** 共数组 Fortran**CPP** C 预处理器 **CRS** 压缩行存储 **CSP** 通信顺序进程 **CG** 共轭梯度 **CUDA** 统一计算设备架构 **DAG** 有向无环图 **DFS** 深度优先搜索 **DPCPP** 数据并行 C++ **DSP** 数字信号处理 **FEM** 有限元方法 **FIFO** 先进先出 **FPU** 浮点单元 **FFT** 快速傅里叶变换 **FPGA** 现场可编程门阵列 **FSA** 有限状态自动机 **GPU** 图形处理单元 **HPC** 高性能计算 **HPF** 高性能 Fortran**HPL** 高性能 Linpack**ICV** 内部控制变量 **LAPACK** 线性代数包 **MG** 多重网格 **MIC** 多集成核心 **MIMD** 多指令多数据 **MPI** 消息传递接口 **MPL** 消息传递层 **NCCL** NVIDIA 集合通信库 **MPMD** 多程序多数据 **MTA** 多线程架构 **NIC** 网络接口卡 **NUMA** 非统一内存访问 **OO** 面向对象 **OOP** 面向对象编程 **OS** 操作系统 **PGAS** 分区全局地址空间 **PDE** 偏微分方程 **PRAM** 并行随机存取机 **RDMA** 远程直接内存访问 **RMA** 远程内存访问 **SAN** 存储区域网络 **SaaS** 软件即服务 **SFC** 填充空间曲线 **SIMD** 单指令多数据 **SIMT** 单指令多线程 **SLURM** 简单 Linux 资源管理工具 **SM** 流多处理器 **SMP** 对称多处理 **SOR** 连续超松弛法 **SP** 流处理器 **SPMD** 单程序多数据 **SPD** 对称正定 **SSE** SIMD 流扩展 **STL** 标准模板库 **TACC** 德州高级计算中心 **TBB** 线程构建块 **TLB** 转换后备缓冲 **UMA** 统一内存访问 **UPC** 统一并行 C**URI** 统一资源标识符 **WAN** 广域网

第 56 章

通用索引

索引

第 57 章

笔记列表

57.1 MPI-4 笔记

MPI-4 笔记列表

6 窗口内存对齐 .7

57.2 Fortran 笔记

Fortran 笔记列表

57.3 C++notes

C++ notes 列表

57.4 TheMPL C++ 接口

Mpl 笔记列表

57.5 Python 笔记

Python 笔记列表

第 58 章

MPI 命令和关键字索引

58.1 来自标准文档

这是 MPI 标准文档中每个函数、类型和常量的自动生成列表。在本书中出现这些内容时，会给出页码参考。

58.1.1 所有函数列表

- MPI_Abort 27
- MPI_Accumulate 238
- MPI_Address ??
- MPI_Add_error_class 305
- MPI_Add_error_code 305
- MPI_Add_error_string 305
- MPI_Aint_add 154
- MPI_Aint_diff 154
- MPI_Allgather 61
- MPI_Allgatherv 69
- MPI_Allgatherv_init 139
- MPI_Allgather_init 139
- MPI_Alloc_mem 229
- MPI_Alloc_mem_cptr ??
- MPI_Allreduce 40
- MPI_Allreduce_init 138
- MPI_Alltoall 62
- MPI_Alltoallv 63
- MPI_Alltoallv_init 139
- MPI_Alltoallw ??
- MPI_Alltoallw_init 139
- MPI_Alltoall_init 139
- MPI_Attr_delete ??
- MPI_Attr_get 299
- MPI_Attr_put ??
- MPI_Accumulate 238
- MPI_Barrier 68
- MPI_Barrier_init 139
- MPI_Bcast 50
- MPI_Bcast_init 139
- MPI_Bsend 144
- MPI_Bsend_init 146
- MPI_Buffer_attach 145
- MPI_Buffer_detach 145
- MPI_Buffer_flush 145
- MPI_Buffer_iflush ??
- MPI_Cancel 316
- MPI_Cartdim_get 269
- MPI_Cart_coords 269
- MPI_Cart_create 267
- MPI_Cart_get 269
- MPI_Cart_map 272
- MPI_Cart_rank 269
- MPI_Cart_shift ??
- MPI_Cart_sub 271
- MPI_Close_port 217
- MPI_Comm_accept 217
- MPI_Comm_attach_buffer 145
- MPI_Comm_call_errhandler ??
- MPI_Comm_compare 196
- MPI_Comm_connect 218
- MPI_Comm_create 205
- MPI_Comm_create_errhandler 305
- MPI_Comm_create_from_group ??
- MPI_Comm_create_group 205
- MPI_Comm_create_keyval 301
- MPI_Comm_delete_attr 301
- MPI_Comm_detach_buffer 145
- MPI_Comm_disconnect 200
- MPI_Comm_dup 195
- MPI_Comm_dup_with_info 195
- MPI_Comm_flush_buffer 145
- MPI_Comm_free 200
- MPI_Comm_free_keyval 301
- MPI_Comm_get_attr 299
- MPI_Comm_get_errhandler 303
- MPI_Comm_get_info 298
- MPI_Comm_get_name ??
- MPI_Comm_get_parent 210
- MPI_Comm_group 204
- MPI_Comm_idup 195
- MPI_Comm_idup_with_info 195
- MPI_Comm_iflush_buffer ??
- MPI_Comm_join 220
- MPI_Comm_rank 32
- MPI_Comm_remote_size 210
- MPI_Comm_set_attr 299
- MPI_Comm_set_errhandler 303
- MPI_Comm_set_info 298
- MPI_Comm_set_name 195
- MPI_Comm_size 32
- MPI_Comm_spawn 213
- MPI_Comm_spawn_multiple 217
- MPI_Comm_split 201
- MPI_Comm_split_type 203
- MPI_Comm_test_inter 210
- MPI_Compare_and_swap 242
- MPI_Compare_and_swap 242
- MPI_Dims_create 267
- MPI_Dist_graph_create 274
- MPI_Dist_graph_create_adjacent ??
- MPI_Dist_graph_neighbors 279
- MPI_Dist_graph_neighbors_count 279
- MPI_Errhandler_create 304
- MPI_Errhandler_free 304
- MPI_Errhandler_get ??
- MPI_Errhandler_set ??
- MPI_Error_class ??
- MPI_Error_string 305
- MPI_Exscan 55
- MPI_Exscan_init 139
- MPI_Fetch_and_op 240
- MPI_File_call_errhandler 303
- MPI_File_close 256
- MPI_File_create_errhandler ??
- MPI_File_delete 257
- MPI_File_get_amode ??
- MPI_File_get_atomicity ??
- MPI_File_get_byte_offset ??
- MPI_File_get_errhandler ??
- MPI_File_get_group ??
- MPI_File_get_info ??
- MPI_File_get_position ??
- MPI_File_get_position_shared ??
- MPI_File_get_size 263
- MPI_File_get_type_extent ??
- MPI_File_get_view 263
- MPI_File_iread 260
- MPI_File_iread_all 260
- MPI_File_iread_at 260
- MPI_File_iread_at_all 260
- MPI_File_iread_shared 263
- MPI_File_iwrite 260
- MPI_File_iwrite_all 260
- MPI_File_iwrite_at 260
- MPI_File_iwrite_at_all 260
- MPI_File_iwrite_shared 263
- MPI_File_open 256
- MPI_File_preallocate 263
- MPI_File_read 258
- MPI_File_read_all 258
- MPI_File_read_all_begin 260
- MPI_File_read_all_end 260
- MPI_File_read_at 258
- MPI_File_read_at_all 258

- MPI_Win_flush_local_all??
- MPI_Win_free 229
- MPI_Win_get_attr 248
- MPI_Win_get_group??
- MPI_Win_get_info 298
- MPI_Win_get_name??
- MPI_Win_lock 245
- MPI_Win_lock_all 246
- MPI_Win_post??
- MPI_Win_set_attr 301
- MPI_Win_set_info 298
- MPI_Win_shared_query 285
- MPI_Win_shared_query_cptr??
- MPI_Win_start 233
- MPI_Win_sync 248
- MPI_Win_test 232
- MPI_Win_unlock 246
- MPI_Win_unlock_all 246
- MPI_Win_wait 232
- MPI_Wtick 309
- MPI_Wtime 308
- PMPI_??
- PMPI_Aint_add??
- PMPI_Aint_diff??
- PMPI_Isend??
- PMPI_Wtick??
- PMPI_Wtime??

58.1.2 所有 dtypes 列表

58.1.3 所有 ctypes 列表

- MPI_Offset 260
- _Bool??
- bool??
- char??
- double??
- enum??
- float??
- int??
- long??
- short??
- wchar_t??

58.1.4 所有 ftypes 列表

- ALLOCATABLE??
- ASYNCHRONOUS??
- BLOCK??
- CHARACTER??
- COMMON??
- COMPLEX??
- CONTAINS??
- CONTIGUOUS??
- C_F_POINTER??
- C_PTR??
- EXTERNAL??
- FUNCTION??
- IN??
- INCLUDE??
- INOUT??
- INTEGER??
- INTENT??
- INTERFACE ??
- ISO_C_BINDING??
- ISO_FORTRAN_ENV??
- KIND??
- LOGICAL??
- MODULE??
- MPI_User_function??
- OPTIONAL??
- OUT??
- POINTER??
- PROCEDURE??
- REAL??
- SEQUENCE??
- TARGET??
- TYPE??
- USER_FUNCTION??
- VOLATILE??

58.1.5 所有常量列表

- MPI_ADDRESS_KIND 154
- MPI_ANY_SOURCE 96
- MPI_ANY_TAG 97
- MPI_APPNUM 217
- MPI_ARGVS_NULL 217
- MPI_ARGV_NULL 214
- MPI_ASYNC_PROTECTS_NONBLOCKING??
- MPI_BAND 74
- MPI_BOR 74
- MPI_BOTTOM??
- MPI_BSEND_OVERHEAD 145
- MPI_BUFFER_AUTOMATIC 145
- MPI_BXOR 74
- MPI_CART 266
- MPI_COMBINER_CONTIGUOUS??
- MPI_COMBINER_DARRAY??
- MPI_COMBINER_DUP??
- MPI_COMBINER_HINDEXED??
- MPI_COMBINER_HINDEXED_BLOCK??
- MPI_COMBINER_HINDEXED_INTEGER??
- MPI_COMBINER_HVECTOR??
- MPI_COMBINER_HVECTOR_INTEGER??
- MPI_COMBINER_INDEXED??
- MPI_COMBINER_INDEXED_BLOCK??
- MPI_COMBINER_NAMED??
- MPI_COMBINER_RESIZED??
- MPI_COMBINER_STRUCT??
- MPI_COMBINER_STRUCT_INTEGER??
- MPI_COMBINER_SUBARRAY??
- MPI_COMBINER_VALUE_INDEX??
- MPI_COMBINER_VECTOR 189
- MPI_COMM_NULL??
- MPI_COMM_SELF??
- MPI_COMM_TYPE_HW_GUIDED??
- MPI_COMM_TYPE_HW_UNGUIDED??
- MPI_COMM_TYPE_RESOURCE_GUIDED??
- MPI_COMM_TYPE_SHARED??
- MPI_COMM_WORLD??
- MPI_CONGRUENT 197
- MPI_COUNT_KIND??
- MPI_DATATYPE_NULL 159
- MPI_DISPLACEMENT_CURRENT 262

- MPI_WIN_FLAVOR_CREATE??
- MPI_WIN_FLAVOR_DYNAMIC??
- MPI_WIN_FLAVOR_SHARED??
- MPI_WIN_MODEL 248
- MPI_WIN_NULL??
- MPI_WIN_SEPARATE??
- MPI_WIN_SIZE??
- MPI_WIN_UNIFIED??
- MPI_WTIME_IS_GLOBAL??

58.1.6 所有回调函数列表

- COMM_COPY_ATTR_FUNCTION??
- COMM_DELETE_ATTR_FUNCTION??
- COPY_FUNCTION??
- DELETE_FUNCTION??
- MPI_Comm_copy_attr_function??
- MPI_Comm_delete_attr_function??
- MPI_Comm_errhandler_function??
- MPI_Copy_function??
- MPI_Datarep_conversion_function??
- MPI_Datarep_conversion_function_c??
- MPI_Delete_function??
- MPI_File_errhandler_function??
- MPI_Handler_function??
- MPI_Session_errhandler_function??
- MPI_Type_delete_attr_function??
- MPI_User_function??
- MPI_User_function_c??
- MPI_Win_errhandler_function??

58.2 MPI for Python

58.2.1 Buffer specifications

58.2.2 Listing of python routines

ClassComm: 类 Cartcomm: 类 Distgraphcomm: 类 Graphcomm: 类 Intercomm: 类 Intracomm: 类 Topocomm: 类 Group:

Class Request: 类 请求 Grequest: 类 Prequest: 类 ClassStatus:

类 Win:

类 Datatype: 类 File: 类 Info: 类 Op:

类 Errhandler: 类 Message:

第 59 章

OpenMP 关键字索引

第 60 章

PETSc 关键词索引

,464, 464, 466, 464, 464,466,
487, 460,462, 468,514, 515

第 61 章

KOKKOS 关键词索引

第 62 章

Index of SYCL keywords

ISBN 978-1-387-40028-7