计算科学

高性能计算的艺术，第一卷

演进中的副本 —— 开放评论

Victor Eijkhout 与 Edmond Chow 、 Robert van de Geijn 合著

第三版 2022 年，2024 年 4 月 2 日排版 丛书参考编号： https://theartofhpc.com

前言

高性能科学计算领域横跨多个学科与技能体系，因此，若要在科学领域成功运用高性能计算，至少需掌握这些领域的基础知识与技能。计算源于应用场景，故需对物理学与工程科学有所涉猎。随后，这些应用领域的问题通常被转化为线性代数问题，有时则是组合数学问题，因此计算科学家需具备数值分析、线性代数及离散数学等多方面知识。要高效实现应用问题的实际公式化表达，还需理解计算机体系结构，包括 CPU 层面与并行计算层面。最后，除精通上述科学领域外，计算科学家还需掌握特定的软件管理技能。

尽管已有关于数值建模、数值线性代数、计算机体系结构、并行计算及性能优化的优秀教材，但尚无书籍以统一方式整合这些分支。作者在计算中心工作时深刻意识到对此类书籍的需求：用户多为领域专家，未必全面掌握使其成为高效计算科学家所需的所有背景知识。本书旨在为从事大规模计算的科学家讲授这些不可或缺的主题。

本书主要内容为高性能计算（ HPC ）的理论知识。关于编程与实践教程，请参阅本系列其他卷册。书中各章节均配有习题，适合课堂布置，但习题在文中的编排方式使得不愿做题的读者可直接将其视为事实陈述。

公开草案 本书开放征求意见。哪些内容缺失、不完整或表述不清？材料呈现顺序是否不当？欢迎通过邮件向我反馈您的意见。

您可能从多个渠道获取本书；我所有教材的权威发布地址是 https://theartofhpc.com。该页面也提供 lulu.com 的链接，您可在此获取精美印刷版。

Victor Eijkhout eijkhout@tacc.utexas.edu
研究科学家 德克萨斯高级计算中心 德克萨斯大学奥斯汀分校

致谢 衷心感谢与 Kazushige Goto 和 John McCalpin 的富有成效的讨论。感谢 Dan Stanzione 提供的云计算笔记、 Ernie Chan 关于块算法调度的笔记，以及 John McCalpin 对 top500 的分析。感谢 Elie de Brauwer 、 Susan Lindsey 、 Tim Haines 和 Lorenzo Pesce 的校对及诸多建议。Edmond Chow 撰写了分子动力学章节。Robert van de Geijn 贡献了稠密线性代数相关的多个章节。

引言

科学计算是建模科学过程与利用计算机从这些模型中生成定量结果的交叉学科领域。它将领域科学转化为计算活动。作为一种定义，我们可以提出

> 应用数学中构造性方法的高效计算。

这清晰地表明了科学计算所涉及的三个科学分支：

- 应用数学：对现实世界现象的数学建模。此类建模常导致隐式描述，例如以偏微分方程的形式呈现。为了获得实际可感知的结果，我们需要一种构造性方法。
- 数值分析为科学模型提供了算法思维。它通过成本与稳定性分析，提出了一种解决隐式模型的构造性方法。
- 计算科学采用数值算法，并分析在真实存在（而非假设性）的计算引擎上实施这些算法的效能。

可以说，当现实世界现象的数学被要求具有构造性时，即从证明解的存在性转向实际获取解时，‘计算’ 便自成一体成为一个科学领域。此时，算法本身成为研究对象，而不仅仅是工具。

算法的研究在计算机发明后变得尤为重要。由于数学运算现在被赋予了可定义的时间成本，算法的复杂性成为一个研究领域；由于计算不再使用 ‘实数’ 而是有限位串的表示形式，算法的精确性需要被研究。其中一些考量实际上早于计算机的出现，是受机械计算器计算的启发。

科学计算的核心关注点是效率。虽然对一些科学家而言，解存在的抽象事实已足够，但在计算中我们实际需要那个解，且最好是昨天就得到。因此，在本书中，我们将非常具体地讨论算法和硬件的效率。重要的是，不要将效率概念局限于硬件的有效使用。虽然这很重要，但两种算法方法之间的差异可能使针对特定硬件的优化成为次要考虑。

本书旨在涵盖一名成功的计算科学家所需掌握的知识体系基础。它被设计为研究生或高年级本科生的教材；其他人可将其作为参考书使用，通过阅读习题获取信息内容。

目录

目录

第一部分

理论

第一章

单处理器计算

为了编写高效的科学计算代码，理解计算机体系结构至关重要。两个计算结果相同的代码之间，速度差异可能从几个百分点到数个数量级不等，这仅取决于算法针对处理器架构的编码优化程度。显然，仅拥有算法并 ‘将其运行在计算机上’ 是不够的：对计算机体系结构的了解是必要的，有时甚至是关键。

有些问题可以在单个中央处理器（CPU）上解决，而另一些则需要由多个处理器组成的并行计算机。我们将在下一章详细讨论并行计算机，但即使是并行处理，理解单个 CPU 的工作原理也是必要的。

本章将重点探讨 CPU 内部及其内存系统的工作原理。我们首先简要讨论指令处理的一般流程，接着深入处理器核心的算术运算；最后但同样重要的是，我们将重点关注数据在内存与处理器之间以及处理器内部的移动。后者可能出乎意料地重要，因为内存访问速度通常远低于执行处理器指令的速度，这使其成为程序性能的决定性因素；以 ‘浮点运算次数（FLOPS）计数’ 预测代码性能的时代早已过去。这种差距实际上呈扩大趋势，因此处理内存流量的问题随时间推移愈发重要，而非消失。

本章将帮助您基本了解 CPU 设计涉及的问题、其对性能的影响，以及如何编写代码以实现最佳性能。更多细节请参阅计算机体系结构的标准著作 ——Hennessy 和 Patterson 的《计算机体系结构》 [100]。

1.1 冯 · 诺伊曼体系结构

尽管计算机（尤其是本章重点讨论的处理器）在诸多细节上可能各不相同，但它们也有许多共同点。在高度抽象的层面上，许多体系结构可归类为冯 · 诺伊曼体系结构。这种设计采用未分区的内存同时存储程序和数据（“存储程序”概念），并由处理单元以“取指、执行、存储”的循环方式执行指令、操作数据。

备注 1 这种按预设指令序列执行的模型也被称为控制流，与之相对的是数据流，我们将在 *7.12* 节中详述。

这种架构将现代处理器与最早的以及某些特殊用途的当代设计区分开来，后者采用硬接线编程方式。由于指令和数据存储在同一空间，它还允许程序自我修改或生成其他程序。这使我们能够拥有编辑器和编译器：计算机将程序代码视为可操作的数据。

R备注 2 曾几何时，存储程序概念被视为实现运行能力的关键要素，
*p*程序修改自身源代码的做法。然而，人们很快意识到这会导致代码难以维护，
*a*在实践中很少这样做 *[49]*。

I本书中，我们不会明确讨论编译过程，即把高级语言翻译成
m机器指令的过程。（关于使用方面的内容，请参阅教程教程书籍中的第 *2* 节。）不过，偶尔
w我们会探讨如何编写高级程序以确保其在底层的高效性。
然而在科学计算领域，我们通常不太关注程序代码本身，而几乎完全聚焦于数据及其在程序执行过程中的流动方式。从大多数实际应用角度看，程序与数据仿佛是被分开存储的。关于指令处理的核心要点可简述如下。

处理器执行的机器指令（与用户编写的高级语言不同）通常会指定操作名称、操作数位置及结果位置。这些位置并非以内存地址表示，而是通过寄存器：CPU 内部少量具名的存储位置。

备注 3 直连内存架构较为罕见，尽管历史上确实存在过。*1980* 年代的 *Cyber 205* 超级计算机能同时处理三条数据流：两条从内存到处理器，一条从处理器返回内存。这种架构仅在内存速度能跟上处理器时可行，而当今已无法满足这一条件。

例如，这里有一个简单的 C 语言例程

```
void store(double *a, double *b, double *c) {
 *c = *a + *b;
}
```

及其通过 `gcc -O2 -S -o - store.c` 生成的 X86 汇编输出：

```
        .text
        .p2align 4, 15
.globl store
        .type   store, @function
store:
        movsd   (%rdi), %xmm0           # Load *a to %xmm0
        addsd   (%rsi), %xmm0           # Load *b and add to %xmm0
        movsd   %xmm0, (%rdx)           # Store to *c
        ret
```

(此为 64 位系统输出；32 位系统需添加选项 `-m64` 。)

此处指令为:

- 从内存加载到寄存器;
- 另一加载操作，与加法结合;
- 将结果写回内存。

每条指令按以下步骤处理:

- 指令提取：根据程序计数器将下一条指令载入处理器。我们将忽略该过程具体如何实现及数据来源的问题。
- 指令解码：处理器解析指令以确定操作类型及操作数。
- 内存读取：必要时将数据从内存载入寄存器。
- 执行：运行操作，从寄存器读取数据并写入 将其写回寄存器。
- 写回：对于存储操作，寄存器内容会被写回内存。

数组数据的情况稍复杂些：加载（或存储）的元素由数组基址加上偏移量决定。

从某种角度看，现代 CPU 对程序员而言仍像冯 · 诺依曼机器。但这并非全然正确。首先，虽然内存看似可随机寻址 [1]，但实际上存在**局部性概念**：一旦某数据项被加载，邻近项的加载效率更高，重新加载初始项也会更快。

数据简单加载过程的另一复杂之处在于，现代 CPU 会同时处理多条指令（称为 ‘ 飞行中 ’ 指令），这些指令处于不同完成阶段。自然，伴随这些并行指令，其输入输出也在内存与处理器间以重叠方式移动。这就是**超标量 CPU 架构**的基本理念，亦称为**指令级并行**（*ILP*）。因此，尽管单条指令可能需多个时钟周期完成，处理器在理想情况下每周期可完成一条指令，某些情况下甚至能完成多条指令。

关于 CPU 的主要统计数据是其千兆赫兹（ Gigahertz ）评级，这暗示着处理器的速度是计算机性能的主要决定因素。虽然速度显然与性能相关，但实际情况更为复杂。某些算法是 *cpu-bound*（受 CPU 限制），此时处理器的速度确实是最重要的因素；而其他算法则是 *memory-bound*（受内存限制），这时总线速度和缓存大小等方面（后续将讨论）变得尤为重要。

在科学计算中，第二类情况实际上相当突出，因此在本章中，我们将重点关注数据从内存到处理器的传输过程，而对实际处理器的关注则相对较少。

1. 事实上，存在一种称为 ‘ 随机存取机 ’ （ Random Access Machine ）的计算理论模型；我们将在 2.2.2 节简要看到其并行推广版本。

1.2 现代处理器

现代处理器相当复杂，本节我们将简要介绍其组成部分。图 1.1 展示了 *Intel Sandy Bridge* 处理器的晶圆照片。该芯片尺寸约一英寸，包含近十亿个晶体管。

1.2.1 处理核心

在冯 · 诺依曼模型中，执行指令的是单一实体。但自 2000 年代初以来，这种情况已逐渐改变。图 1.1 所示的 Sandy Bridge 拥有八个核心，每个核心都是独立执行指令流的单元。本章主要讨论单个核心的特性；1.4 节将探讨多核心的集成问题。

1.2.1.1 指令处理

冯 · 诺依曼架构模型在假设所有指令严格按顺序执行方面同样不切实际。过去二十年间，处理器越来越多地采用乱序指令处理技术，允许指令以不同于用户程序指定的顺序执行。当然，处理器仅在被允许重新排序指令且能保持执行结果不变的情况下才会这样做！

在框图（图 1.2 ）中可见多个涉及指令处理的单元：这种精巧设计实际上消耗了大量能源及晶体管数量。正因如此，第一代英特尔至强融核处理器 *KnightsCorner* 采用顺序指令处理。然而在下一代 *KnightsLanding* 中，出于性能考虑该决策被推翻。

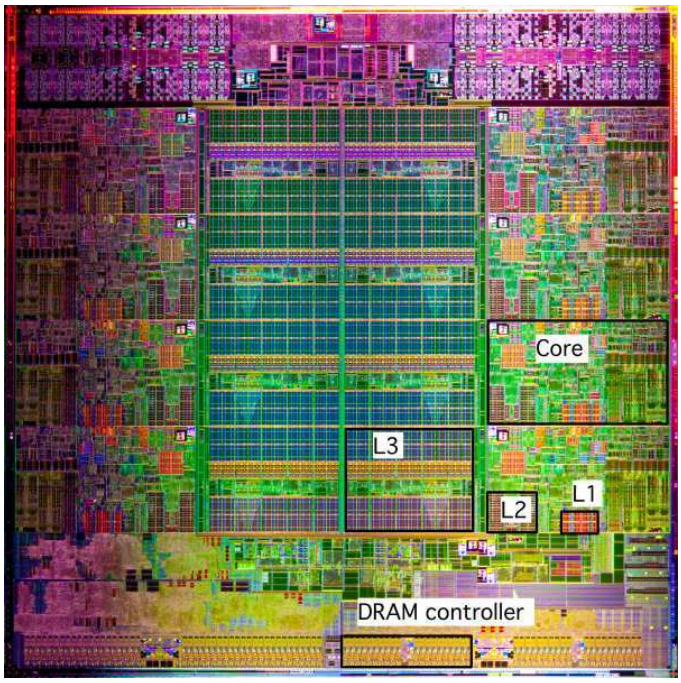


图 1.1：英特尔 Sandy Bridge 处理器晶圆。

1.2.1.2 浮点运算单元

在科学计算领域，我们主要关注处理器如何处理浮点数据。相比之下，整数或布尔运算通常较少受到重视。因此，处理器核心在数值数据处理方面具备高度复杂的架构设计。

例如，早期处理器仅配备单个浮点运算单元（ FPU ），而现代处理器则拥有多个可并行执行的浮点运算单元。

例如，通常会有独立的加法与乘法单元；若编译器能发现互不依赖的加法与乘法运算，便可调度它们同步执行，从而使处理器性能翻倍。某些情况下，处理器会配备多个加法或乘法单元。

另一种提升性能的方式是采用融合乘加 *(FMA)* 单元，该单元执行指令 $x \leftarrow ax + b$ 所需时间与独立加法或乘法操作相同。结合流水线技术（见下文），这意味着处理器每个时钟周期可渐近完成多次浮点运算。

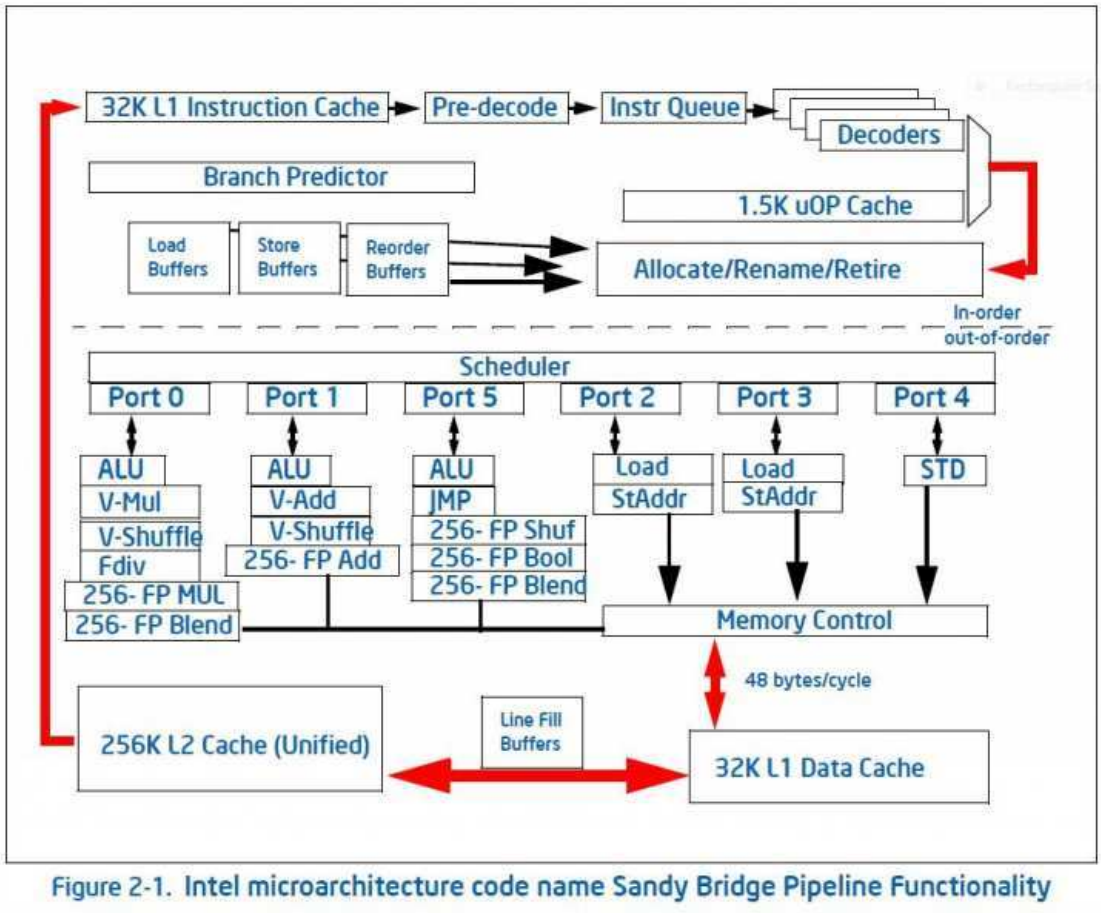


图 1.2: 英特尔 Sandy Bridge 核心结构框图

值得注意的是，除法运算成为性能瓶颈的算法较为罕见。因此现代 CPU 对除法运算的优化程度远不及加法与乘法。除法操作可能消耗 10 至 20 个时钟周期，而 CPU 可配备多个加法 / 乘法单元（渐近情况下）实现每周期输出一个运算结果。

处理器	year	加法器 / 乘法器 / 融合乘加单元 (数量 × 位宽)	daxpy cycles (算术运算 vs 加载 / 存储)
MIPS R10000	1996	$1 \times 1 + 1 \times 1 + 0$	8/24
Alpha EV5	1996	$1 \times 1 + 1 \times 1 + 0$	8/12
IBM Power5	2004	$0 + 0 + 2 \times 1$	4/12
AMD Bulldozer	2011	$2 \times 2 + 2 \times 2 + 0$	2/4
Intel Sandy Bridge	2012	$1 \times 4 + 1 \times 4 + 0$	2/4
Intel Haswell	2014	$0 + 0 + 2 \times 4$	1/2

Table 1.1: Floating point capabilities (per core) of several processor architectures, and DAXPY cycle number for 8 operands.

1.2.1.3 流水线技术

处理器的浮点加法与乘法单元采用流水线设计，这使得一系列独立操作能够以每个时钟周期输出一个结果的渐近速度执行。

流水线背后的核心理念如下：假设一个操作由多个较简单的子操作构成，那么通过为每个子操作配备专用硬件，我们有可能加速整体操作。当需要执行多个操作时，通过让所有子操作同时运作 —— 每个子操作将其结果传递给下一个，并从上一个接收输入数据 —— 即可实现加速。

例如，一条加法指令可能包含以下组件：

- 解码指令，包括定位操作数的存储位置。
- 将操作数复制到寄存器中（即 “ 数据获取 ” 阶段）。
- 对齐指数；此时加法运算 $.35 \times 10^{-1} + .6 \times 10^{-2}$ 变为 $.35 \times 10^{-1} + .06 \times 10^{-1}$。
- 执行尾数相加操作，本例中得到结果为 .41。
- 对结果进行规范化处理，本例中调整为 $.41 \times 10^{-1}$。（此例中的规范化未实际改变数值。请自行验证 $.3 \times 10^{0} + .8 \times 10^{0}$ 与 $.35 \times 10^{-3} + (-.34) \times 10^{-3}$ 中存在非平凡调整的情况。）
- 存储运算结果。

这些部分通常被称为流水线的 ‘ 级 ’ 或 ‘ 段 ’ 。

若每个组件均设计为 1 个时钟周期内完成，则整条指令需 6 个周期。但如果各阶段拥有独立硬件，我们可在少于 12 个周期内执行两条操作：

- 执行第一条操作的解码阶段；
- 为第一个操作执行数据获取，同时为第二个操作进行解码。
- 同时执行第一个操作的第三阶段和第二个操作的第二阶段。
- 以此类推。

可以看到，第一个操作仍需要 6 个时钟周期完成，但第二个操作仅比它晚 1 个周期就完成了。

让我们对流水线带来的加速进行正式分析。在传统浮点运算单元上，生成 n 结果需要 $t(n) = n\ell\tau$，其中 ℓ 代表流水线级数，τ 代表时钟周期时间。结果产出速率是 $t(n)/n$ 的倒数：r 串行 $\equiv (\ell\tau)^{-1}$。

另一方面，对于流水线浮点运算单元 (FPU)，所需时间为 $t(n) = [s + \ell + n - 1]\tau$，其中 s 为初始设置成本：首项运算仍需经历与之前相同的阶段，但此后每个周期可额外产出一个结果。该公式亦可表示为

$$t(n) = [n + n_{1/2}]\tau,$$

表示线性时间加上一个偏移量。

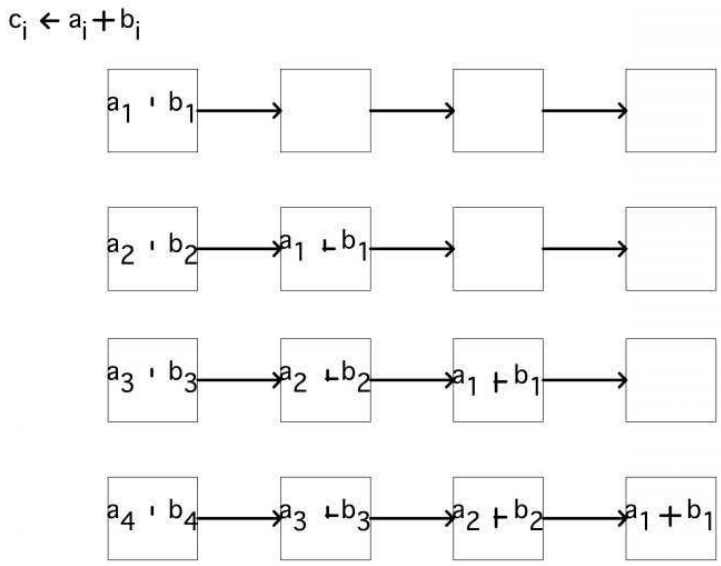


图 1.3：流水线操作示意图。

练习 1.1. 现比较传统 FPU 与流水线 FPU 的运算速度。证明当前结果速率取决于 n：给出 $r(n)$ 的公式及 $r_\infty = \lim_{n\to\infty} r(n)$ 的表达式。相较于非流水线情况，r 的渐进改进程度如何？接下来可探讨接近渐进行为所需时长。证明当 $n = n_{1/2}$ 时可得 $r(n) = r_\infty/2$，此常被用作 $n_{1/2}$ 的定义。

由于向量处理器同时处理多条指令，这些指令必须相互独立。运算 $\forall_i : a_i \leftarrow b_i + c_i$ 包含独立的加法操作；而运算 $\forall_i : a_{i+1} \leftarrow a_i b_i + c_i$ 将一次迭代的结果（a_i）作为下一次迭代的输入（$a_{i+1} = \ldots$），因此这些操作并不独立。

流水线处理器可将运算速度提升至早期 CPU 的 4 、 5 、 6 倍。这类数字在 1980 年代首批成功的向量计算机上市时颇具代表性。如今，CPU 可具备 20 级流水线。这是否意味着它们速度惊人？这个问题稍显复杂。芯片设计者持续提高时钟频率，导致流水线段落无法在一个周期内完成工作，因此被进一步细分。有时甚至存在空转的流水线段落：这些时间用于确保数据能及时传输至芯片的不同区域。

流水线 CPU 所能带来的性能提升是有限的，因此为了追求更高的性能，人们尝试了多种流水线设计的变体。例如，Cyber 205 拥有独立的加法与乘法流水线，且数据无需先返回内存即可将一条流水线的输出直接输入到下一条流水线中。类似 $\forall_i : a_i \leftarrow b_i + c \cdot d_i$ 这样的操作被称为 ‘链接三元组’（由于内存路径数量的限制，其中一个输入操作数必须为标量）。

练习 1.2. 分析链接三元组的加速比与 $n_{1/2}$。

A另一种提升性能的方式是采用多条相同的流水线。这种设计被
tNEC SX 系列发挥到极致。例如，在 4 条流水线的配置下，运算 $\forall_i : a_i \leftarrow b_i + c_i$ 会按模块 4 进行分割，因此
t第一条流水线处理索引 $i = 4 \cdot j$ 的数据，第二条处理 $i = 4 \cdot j + 1$ 的数据，以此类推。

练习 1.3. 分析具有多条并行流水线处理器的加速比及 $n_{1/2}$。假设存在 p 条独立流水线，它们执行相同指令，每条均可处理操作数流。

(您可能好奇为何在此提及一些相当古老的计算机：真正的流水线超级计算机已几乎绝迹。在美国，Cray X1 是该系列的末代产品，日本仅 NEC 仍在制造。然而，当今 CPU 的功能单元仍采用流水线设计，因此这一概念依然重要。)

练习 1.4. 该运算

```
for (i) {
  x[i+1] = a[i]*x[i] + b[i];
}
```

无法通过流水线处理，因为存在操作单次迭代输入与前次输出间的依赖关系。但可通过数学等价转换获得潜在更高计算效率的循环。推导不涉及 `x[i+1]` 即可从 `x[i]` 计算 `x[i+2]` 的表达式，此方法称为递归倍增。假设拥有充足临时存储空间，现可通过以下方式完成计算：

- 进行一些初步计算；
- 计算 `x[i]`、`x[i+2]`、`x[i+4]` 等，并基于这些结果，
- 计算缺失项 `x[i+1]`、`x[i+3]` 等。通过给出 T 的公式来分析此方案的效率

$_0(n)$ 和 $T_s(n)$。你能否想到一个论点，说明在某种情况下初步计算可能不那么重要?

1.2.1.3.1 **脉动计算** 如前所述的流水线技术是脉动算法的一种特例。在 20 世纪 80 至 90 年代，研究者们探索利用流水线算法并构建专用硬件 —— 脉动阵列来实现这些算法 [126]。这也与现场可编程门阵列（FPGA）计算相关，其中脉动阵列是通过软件定义的。

第 6.3.2 节对流水线与循环展开进行了性能研究。

1.2.1.4 峰值性能

得益于流水线技术，现代 CPU 的时钟频率 与峰值性能之间存在简单关系。由于每个浮点运算单元（FPU）在渐进情况下每个周期可产生一个结果，其峰值性能为

时钟频率乘以独立浮点运算单元 (FPU) 的数量。衡量浮点性能的指标是 ‘每秒浮点运算次数’，缩写为 *flops*。考虑到当今计算机的速度，您最常听到的浮点性能表达单位是 ‘千兆浮点运算’：即 10^9 次浮点运算的倍数。

1.2.2 8 位、 16 位、 32 位、 64 位

处理器通常以其能作为一个单元处理的数据块大小来表征。这可能涉及：

- 处理器与内存之间的路径宽度：一个 64 位浮点数能否在一个周期内加载完成，还是分批次到达处理器。
- 内存寻址方式：若地址限制为 16 位，则仅能识别 64,000 字节。早期 PC 采用分段复杂方案突破此限制：地址由段号和段内偏移量共同指定。
- 寄存器中的位数，特别是用于操作数据地址的整数寄存器的大小；参见前一点。（浮点寄存器通常更大，例如 x86 架构中的 80 位。）这也对应于处理器可同时操作的数据块大小。
- 浮点数的大小。如果 CPU 的算术单元设计为高效乘 8 字节数字（‘双精度’；参见章节 3.3.2），则一半大小的数字（‘单精度’）有时可以更高效率处理，而对于更大的数字（‘四倍精度’），则需要一些复杂的方案。例如，四倍精度数字可以通过两个具有固定指数差的双精度数字来模拟。

这些测量值不一定相同。例如，最初的 Pentium 处理器具有 64 位数据总线，但却是 32 位处理器。另一方面，Motorola68000 处理器（原始 Apple Macintosh 所用）拥有 32 位 CPU，但数据总线为 16 位。

英特尔首款微处理器 4004 是一款 4 位处理器，即它每次处理 4 位数据块。如今，64 位处理器正逐渐成为主流。

1.2.3 缓存：片上存储器

T计算机内存的主体位于与处理器分离的芯片中。然而，通常
会有少量（通常为几兆字节）的片上存储器，称为缓存。这将在 1.3.5 节中详细说明。

1.2.4 图形、控制器及专用硬件

“消费级”与“服务器级”处理器的一个区别在于，消费级芯片会将处理器芯片上大量面积用于图形处理。手机和平板电脑的处理器甚至可能配备用于安全或 MP3 播放的专用电路。处理器的其他部分则专用于与内存或 *I/O* 子系统通信。本书将不讨论这些方面。

1.2.5 超标量处理与指令级并行

在冯·诺伊曼架构模型中，处理器通过控制流运行：指令线性或通过分支依次执行，而不考虑涉及的数据。随着处理器变得更强大且能同时执行多条指令，转向数据流模型成为必要。这类超标量处理器会分析多条指令以发现数据依赖性，并并行执行互不依赖的指令。

这一概念也被称为指令级并行（*ILP*），它通过多种机制实现：

- 多发射：互不依赖的指令可同时启动；
- 流水线：算术单元可处理处于不同完成阶段的多项操作（见章节 1.2.1.3）；
- 分支预测与推测执行：编译器可以‘猜测’条件指令是否评估为真，并据此执行相应指令；
- 乱序执行：若指令间无依赖关系且重排后执行效率更高，则可重新排列指令顺序；
- 预取：数据可在实际需要它的指令出现前被推测性请求（详见章节 1.3.6）。

前文在浮点运算场景中介绍了流水线技术。如今整个 CPU 都已流水线化 —— 不仅是浮点运算，任何类型的指令都会尽快进入指令流水线。需注意，此流水线不再局限于相同指令：流水线概念现已泛化为任何同时‘执行中’的部分指令流。

随着时钟频率提升，处理器流水线长度增加以使各段能在更短时间内执行。前文已提及，更长流水线会带来更大的 $n_{1/2}$，因此需要更多独立指令才能使流水线保持全效运行。当指令级并行性达到极限时，进一步增加流水线长度（有时称为‘加深’）将不再具有收益。这普遍被视为芯片设计者转向多核架构的原因 —— 以此更高效利用芯片晶体管（参见章节 1.4）。

这些较长流水线还存在第二个问题：当代码遇到分支点（如条件语句或循环中的测试）时，无法确定接下来要执行哪条指令。此时流水线可能会停滞。CPU 采用推测执行技术应对，例如总是假设测试结果为真。若代码最终走向另一分支（称为分支预测错误），则必须清空并重启流水线。由此导致的执行流延迟称为分支惩罚。

1.3 存储层次结构

现在我们将细化冯·诺依曼架构的描绘，基于第 1 节的内容 1.

To the programmer, a computer appears to execute a sequence of steps:

1. 解码指令以确定操作数 s,

1. 单处理器计算

2. 从内存中检索操作数，3. 执行操作并将结果写回内存。

这幅图被称为冯·诺伊曼架构，由于所谓的内存墙 [194]，也称为冯·诺伊曼瓶颈：内存速度过慢，无法以处理器吸收数据的速率将数据加载到处理流程中。具体而言，单次加载可能产生 1000 个周期的 ‘延迟’，而处理器每个周期可执行多次操作。在漫长的加载等待后，下一次加载可能更快到来，但你仍可能受限于内存的 ‘带宽’，这对处理器而言依然太慢。

为解决这一瓶颈，现代处理器在浮点运算单元（FPU）与主内存之间设有多个内存层级：寄存器与缓存，统称为内存层次结构。它们试图通过使最近使用的数据比从主内存获取更快来缓解内存墙问题。当然，这要求算法及其实现允许数据被多次使用。此类关于数据复用的问题将在 1.6.1 节详细讨论；首先我们将探讨技术细节。

寄存器与缓存均比主内存快，只是程度各异；遗憾的是，某一层级的内存速度越快，其容量往往越小。这种容量与访问速度的差异会引发有趣的编程问题，我们将在后文讨论。详见第 6 章。

接下来我们将探讨内存层次结构的各个组件，以及分析其行为所需的理论概念。

1.3.1 总线

计算机中传输数据的导线 —— 无论是从内存到 CPU、磁盘控制器还是屏幕 —— 被称为总线。对我们而言最重要的是前端总线（*FSB*），它连接处理器与内存。在一种主流架构中，这被称为 ‘北桥’，以区别于连接外部设备（图形控制器除外）的 ‘南桥’。

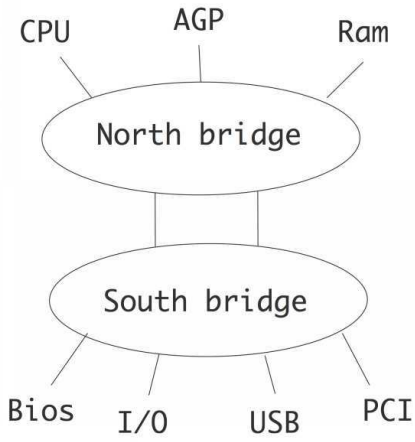


图 1.4：处理器的总线结构。

总线速度通常低于处理器，其时钟频率略高于 1GHz，仅为 CPU 时钟频率的一小部分。这正是需要缓存的原因之一；处理器每个时钟周期可消耗大量数据项的特性加剧了这一需求。除频率外，总线带宽还取决于每个时钟周期可传输的比特数。当前架构中这一数值通常为 64 或 128 位。接下来我们将对此进行更详细的探讨。

1.3.2 延迟与带宽

前文我们笼统提到，访问寄存器数据几乎是瞬时的，而将数据从内存加载到寄存器（任何操作前的必要步骤）则会产生显著延迟。下面我们将对此现象进行更精确的说明。

描述数据移动有两个重要概念：延迟和带宽。其核心假设是：请求数据项会产生初始延迟；若该数据项是数据流（通常是连续内存地址范围）的首个元素，则后续数据流会以固定周期速率持续到达，不再产生额外延迟。

延迟是指处理器发出对内存项的请求到该项实际到达之间的时间间隔。我们可以区分不同类型的延迟，例如从内存到缓存的传输、缓存到寄存器，或者将它们全部汇总为内存与处理器之间的延迟。延迟以（纳）秒或时钟周期为单位进行测量。如果处理器按照汇编代码中的顺序执行指令，那么当从内存中获取数据时，执行往往会停滞；这也被称为内存停滞。因此，低延迟非常重要。实际上，许多处理器具有“乱序执行”指令的能力，使它们能够在等待请求数据的同时执行其他操作。程序员可以考虑到这一点，并通过编码实现延迟隐藏；另见章节 1.6.1。图形处理单元（GPU）（见章节 2.9.3）可以非常快速地在线程之间切换，以实现延迟隐藏。

带宽是指初始延迟被克服后，数据到达目的地的速率。带宽以每秒或每个时钟周期的字节（千字节、兆字节、千兆字节）为单位进行衡量。两个内存层级之间的带宽通常是以下因素的乘积：

- 通道的周期速度（总线速度，以 GT/s 为单位衡量），
- 总线宽度：在总线时钟的每个周期中可以同时发送的比特数。

- 每个插槽的通道数，
- 插槽的数量。

Fo例如，对于 *TACCFrontera* 集群，计算如下

$$\text{带宽} = 2.933\,GT/s \times 8 \text{ 字节 / 传输 / 通道} \times 6 \text{ 通道 / 套接字} \times 2 sockets.$$

延迟与带宽的概念常被结合在一个公式中，用于计算消息从开始到完成的传输时间：

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

其中 α 表示延迟，β 表示带宽的倒数：即每字节所需时间。

在内存层次结构中，离处理器越远，延迟时间越长，带宽越低。因此，编程时应尽可能让处理器使用缓存或寄存器中的数据，而非主内存，这一点至关重要。

1.3.3 流式操作

从性能角度看，最糟糕的操作情形是所谓的 ‘流式’ 内核。此时数据在内存中流进流出，几乎无法利用缓存提升性能。以向量加法为例：

```
for (i)
  a[i] = b[i]+c[i]
```

每次迭代执行一次浮点运算，现代 CPU 通过流水线技术可在一个时钟周期内完成。但每次迭代需要加载两个数字并写入一个，总计产生 24 字节内存流量。（实际上， `a[i]` 在被写入前需先加载，因此每次迭代有 4 次内存访问，总计 32 字节。）典型内存带宽数值（例如图 1.5 所示）远达不到每周期 24 （或 32 ）字节。这意味着若无缓存，算法性能可能受限于内存性能。当然，缓存并非能加速所有操作，事实上对上述例子毫无影响。关于如何通过编程策略显著提高缓存利用率的方法将在第 6 章讨论。

延迟和带宽的概念同样会出现在并行计算机中，当我们讨论将数据从一个处理器发送到另一个处理器时。

1.3.4 寄存器

每个处理器内部都有一小块内存：寄存器，或统称为寄存器文件。寄存器是处理器实际操作的对象：例如像

```
a := b + c
```

实际上是通过以下方式实现的

- 将 `b` 的值从内存加载到寄存器中，
- 将 `c` 的值从内存加载到另一个寄存器中，
- 计算总和并将结果写入另一个寄存器，然后
- 将求和后的值写回 `a` 的内存位置。

查看汇编代码（例如编译器的输出）时，你会看到显式的加载、计算和存储指令。

例如，你会看到如下指令

```
addl    %eax, %edx
```

该指令将一个寄存器的内容加到另一个寄存器上。如本例所示，寄存器不像内存地址那样以数字编号，而是拥有独特的名称，在汇编指令中直接引用这些名称。

指令。通常，处理器拥有 16 或 32 个浮点寄存器；*Intel Itanium* 是个例外，配备了 128 个浮点寄存器。

寄存器因其属于处理器的一部分而具有高带宽和低延迟特性。可以认为数据进出寄存器的传输基本上是瞬时完成的。

本章将重点强调从内存移动数据的相对高成本。因此，尽可能将数据保留在寄存器中是一种简单的优化策略。

例如，对这些指令的严格解释如下：

```
a := b + c
d := a + e
```

w会在第一次操作后将 `a` 写入主内存，随后在第二次操作时重新加载。作为一个明显的优 —— t优化过程中，计算得到的 `a` 值可保留在寄存器中。这一操作通常由编译器执行 *o*优化：我们称 `a` 驻留在寄存器中

将数值保留在寄存器中通常是为了避免重复计算某个量。例如，在

```
t1 = sin(alpha) * x + cos(alpha) * y;
t2 = -cos(alpha) * x + sin(alpha) * y;
```

正弦和余弦量很可能会被保留在寄存器中。您可以通过显式引入临时变量来辅助编译器：

```
s = sin(alpha); c = cos(alpha);
t1 = s * x + c * y;
t2 = -c * x + s * y
```

这一转换无需由程序员手动完成：编译器通常能自行处理。

当然，寄存器能保存的变量数量存在上限；试图保存过多变量会导致寄存器溢出，从而降低代码性能。

K将变量保存在寄存器中尤为重要，尤其是当该变量出现在内层循环时。在 c计算

```
for i=1,length
a[i] = b[i] * c
```

变量 `c` 很可能被编译器保留在寄存器中，但在

```
for k=1,nvectorsfor i=1,
lengtha[i,k] = b[i,k] * c[k]
```

引入一个显式的临时变量来保存 `c[k]` 是个好主意。在 C 语言中，你可以通过声明一个 *register variable* 来向编译器提示将变量保留在寄存器中：

```
register double t;
```

然而，如今的编译器在寄存器分配方面足够智能，这类提示很可能会被忽略。

1.3.5 缓存

在寄存器（存放指令的即时输入输出数据）与主存储器（可长期存储大量数据）之间，存在着多级缓存内存。它们比主存具有更低的延迟和更高的带宽，数据会在其中保留中等时长。

数据从内存经由缓存最终抵达寄存器。缓存内存的优势在于：若某个数据项在首次使用后不久被再次使用，它仍会驻留在缓存中，因此其访问速度远快于从主存重新调入。

本节探讨缓存的基本原理；关于缓存感知编程，请参阅章节 6.4.1。

从历史角度看，内存层次结构的概念早在 1946 [25]，中就有论述，其动机源于当时内存技术的速度瓶颈。

1.3.5.1 反激励示例

举例来说，假设一个变量 `x` 被使用了两次，且它并未驻留在寄存器中：

```
... = ... x ..... // instruction using x
.........         // several instructions not involving x
... = ... x ..... // instruction using x
```

对应的汇编代码将是

- 从内存加载 `x` 到寄存器；对其进行操作；
- 执行中间的指令；
- 将 `x` 从内存加载到寄存器；对其进行操作；

使用缓存后，汇编代码保持不变，但内存系统的实际行为变为：

- 将 `x` 从内存加载到缓存，再从缓存加载到寄存器；对其进行操作；
- 执行中间的指令；
- 向内存发起请求，但由于数据仍在缓存中，直接从缓存加载到寄存器；对其进行操作。

由于从缓存加载比从主内存加载更快，计算速度将得到提升。缓存容量通常较小，因此数据无法永久保存。我们将在后续讨论中看到这一机制的影响。

i

缓存内存与寄存器之间存在一个重要区别：数据通过显式的汇编指令移入寄存器，而从主内存到缓存的移动完全由硬件完成。

因此缓存的使用和重用不在程序员的直接控制范围内。稍后，特别是在章节 1.6.2 和 6 中，你将看到如何间接影响缓存的使用。

1.3.5.2 缓存标签

I在上述示例中，用于确定某个项目是否存在的机制未作具体说明。
i在缓存中。为此，每个缓存位置都有一个标签：包含足够的信息以重建内存
l缓存项目来源的位置。

1.3.5.3 缓存层级、速度与容量

这些缓存被称为 ‘一级’ 和 ‘二级’ 缓存（简称 L1 和 L2）。现代处理器可能还具备 L3 缓存，有时甚至包含 L4 缓存。L1 和 L2 缓存位于处理器芯片的晶粒上，尽管 L2 缓存集成在芯片上是相对较新的技术；更高层级的缓存可能位于芯片外部。缓存可以是某个核心独享的（如 L1 缓存通常如此），也可以由部分或全部核心共享。

L1 缓存容量较小，通常约 16KB。二级（以及存在的三级）缓存容量更大，可达数 MB，但速度也更慢。与可扩展的主内存不同，缓存容量是固定的。若某款处理器芯片存在缓存扩容版本，其价格通常显著更高。

某些操作所需的数据在传输至处理器的过程中会被复制到各级缓存中。若若干指令后再次需要该数据项，系统会先在 L1 缓存中查找；若未找到，则继续在 L2 缓存中搜索；若仍未找到，则从主内存加载。在缓存中找到数据称为缓存命中，未找到则称为缓存未命中。

图 1.5 展示了缓存层级结构的基本事实，本例中以 *IntelSandyBridge* 芯片为例：缓存离浮点运算单元越近，速度越快，但容量也越小。关于此图的几点说明。

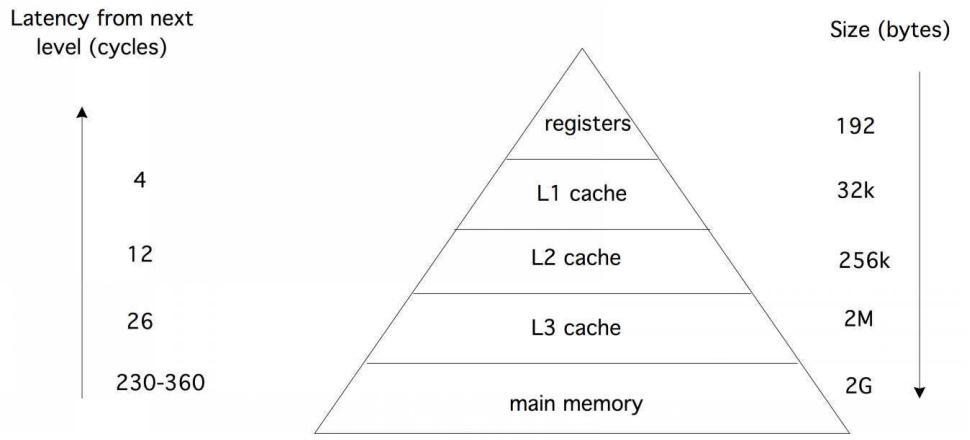


图 1.5：Intel Sandy Bridge 的内存层级结构，以速度和容量为特征。

- 从寄存器加载数据的速度极快，不会构成算法执行速度的限制。但寄存器数量有限，每个核心仅有 16 个通用寄存器和 16 个 SIMD 寄存器。
- L1 缓存虽小，但能维持 32 字节的带宽，即 4 个双精度数，
  每个周期。这足以加载两个操作各自的两个操作数，但需注意核心实际上每个周期可执行 4 次操作。因此，要达到峰值速度，某些操作数需保留在寄存器中：通常，L1 带宽仅能满足约一半的峰值性能需求。
- L2 和 L3 缓存的带宽名义上与 L1 相同。然而，这部分带宽会因一致性问题而部分浪费。
- 主存访问延迟超过 100 个周期，带宽为每周期 4.5 字节，约为 L1 带宽的 1/7 分之一。但该带宽由多个

处理器芯片的核心，因此有效带宽仅为该数值的一部分。大多数集群还会在每个节点配备多个插槽（处理器芯片），通常是 2 个或 4 个，因此部分带宽需用于维护缓存一致性（参见章节 1.4），这进一步降低了每个芯片可用的带宽。

第一级缓存中，指令与数据有独立的缓存区；而 L2 和 L3 缓存则同时包含数据和指令。

可见，容量更大的缓存越来越难以足够快速地向处理器提供数据。因此，编程时必须尽可能让数据停留在最高层级的缓存中。本章后续内容将详细讨论这一议题。

习题 1.5. L1 缓存小于 L2 缓存，若存在 L3 缓存，则 L2 又小于 L3。请从实践和理论角度各给出一个原因说明为何如此。关于缓存大小及其与性能关系的实验探索，详见章节 6.4.1a

nd 25.2.

1.3.5.4 缓存未命中的类型

缓存未命中主要有三种类型。

正如前例所示，首次引用数据时必然会发生缓存未命中。这被称为强制性缓存未命中，因其无法避免。这是否意味着首次需要数据时总会等待？未必：1.3.6 节将解释硬件如何通过预判下一项所需数据来优化此过程。

第二类缓存未命中源于工作集规模：容量性缓存未命中是因缓存容量不足导致数据被覆盖所致。参见图 1.6

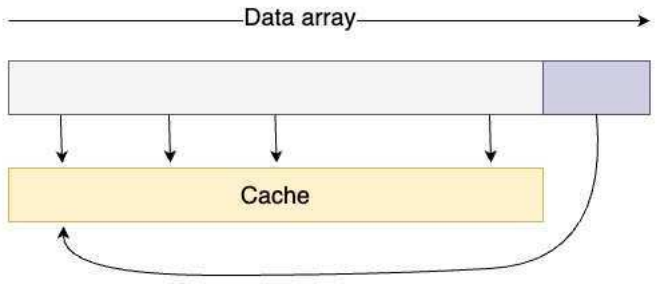


图 1.6：容量性未命中示意图

对此进行了说明。（1.3.5.6 节将讨论处理器实际如何决定覆盖哪些数据。）

若要避免此类缓存未命中，需将问题拆分为足够小的数据块，使其能在缓存中驻留较长时间。当然，这前提是数据项会被多次操作，因此才有必要将其保留在缓存中；具体讨论见章节 1.6.1。

最后，还存在冲突缺失的情况，这是由于一个数据项被映射到了与另一个数据项相同的缓存位置，而两者在计算过程中仍被需要，此时本应有更适合的候选位置。

驱逐。这将在章节 1.3.5.10 中讨论。

在多核 环境下，还存在另一种缓存未命中类型：失效未命中。当缓存中的某个数据项因其他核心修改了对应内存地址的值而失效时，就会发生这种情况。该核心随后必须重新加载该地址的数据。

1.3.5.5 重用是关键

存在一个或多个缓存并不能立即保证高性能：这在很大程度上取决于代码的内存访问模式，以及其利用缓存的效率。首次引用某个数据项时，会将其从内存复制到缓存，再传递至处理器寄存器。此过程的延迟和带宽不会因缓存的存在而得到任何缓解。当第二次引用同一数据项时，可能会在缓存中找到它，此时延迟和带宽成本将显著降低：缓存具有比主内存更短的延迟和更高的带宽。

我们得出的结论是，首先，算法必须具备数据重用的机会。如果每个数据项仅使用一次（如两个向量相加），则无法实现重用，缓存的存在基本无关紧要。只有当缓存中的项目被多次引用时，代码才能从缓存增加的带宽和降低的延迟中受益；详见章节 1.6.1 的详细讨论。一个例子是矩阵 - 向量乘法 $y = Ax$，其中 x 的每个元素在 n 次操作中被使用，其中 n 是矩阵的维度；参见章节 6.10。

其次，算法在理论上可能具备重用的机会，但需要以某种方式编码，以实际暴露这种重用。我们将在章节 1.6.2 中讨论这些要点。尤其是第二点并非易事。

有些问题规模足够小，可以完全放入缓存中，至少能放入 L3 缓存。这是在基准测试时需要注意的一点，因为它会过于乐观地反映处理器的性能。

1.3.5.6 替换策略

缓存与寄存器中的数据由系统自动放置，程序员无法直接控制。同样地，当数据长时间未被引用且其他数据需要存入时，系统会决定何时覆盖缓存或寄存器中的旧数据。下文将详细阐述缓存实现这一机制的原理，但总体遵循最近最少使用（ LRU ）缓存替换策略：若缓存已满且需存入新数据，则从缓存中清除最久未使用的数据，即用新数据覆盖该条目使其不可访问。LRU 是目前最主流的替换策略，其他可选策略包括先进先出（ FIFO ）或随机替换。

练习 1.6. LRU 替换策略如何与直接映射缓存和关联缓存相关联?

练习 1.7. 构建一个简单场景并编写（伪）代码，论证作为替换策略时 LRU 为何优于 FIFO。

1.3.5.7 缓存行

数据在内存与缓存之间或不同缓存之间的传输并非以单个字节甚至字为单位进行。相反，数据移动的最小单位称为缓存行，有时也称为缓存块。典型的

缓存行为 64 或 128 字节长，在科学计算背景下意味着 8 或 16 个双精度浮点数。移入 L2 缓存的数据其缓存行大小可能比移入 L1 缓存的数据更大。

缓存行的首要动机是出于简化的实际考量：若缓存行为 64 字节长，则向负责将数据从内存移至缓存的电路指定地址时可减少 6 位二进制数。其次，缓存行的存在合理，因为许多代码展现出空间局部性：若需要访问内存中的某个字，其相邻字有很大概率也会在不久后被访问。具体讨论参见 1.6.2 节。

反过来，现在有强烈动机通过编码方式利用这种局部性，因为每次内存访问都会引发多个字的传输（示例见 6.2.2 节）。高效程序会尝试使用缓存行上的其他项，因为访问它们的代价实际上为零。这种现象在按跨度访问数组的代码中尤为明显：元素以固定间隔被读写。

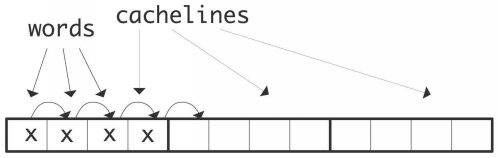


图 1.7：以跨度为 1 访问 4 个元素。

步幅 1 对应于数组的顺序访问：

```
for (i=0; i<N; i++)
  ... = ... x[i] ...
```

让我们以每缓存行 4 字为例进行说明。请求第一个元素时，会将其所在的整个缓存行加载至缓存中。随后对第 2 、 3 、 4 元素的请求便可直接从缓存中满足，这意味着高带宽与低延迟的访问。

更大的跨越

```
for (i=0; i<N; i+=stride)
  ... = ... x[i] ...
```

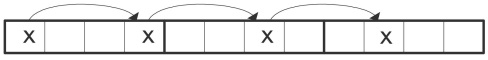


图 1.8：以步长 3 访问 4 个元素

这意味着在每一缓存行中仅使用了特定元素。我们以步幅 3 为例进行说明：请求第一个元素会加载一个缓存行，而这个缓存行

也包含第二个元素。然而，第三个元素位于下一个缓存行上，因此加载它会带来主内存的延迟和带宽消耗。第四个元素同样如此。现在加载四个元素需要加载三个缓存行而非一个，这意味着可用带宽的三分之二被浪费了。（如果不是硬件机制检测到这种规律访问模式并预加载更多缓存行，第二种情况还会产生三倍于第一种情况的延迟；详见章节 1.3.6。）

某些应用天然会导致步长大于 1，例如仅访问复数数组的实部（关于复数实际实现的备注可参见章节 3.8.6 ）。此外，采用递归倍增的方法通常具有呈现非单位步长的代码结构

```
for (i=0; i<N/2; i++)
  x[i] = y[2*i];
```

在讨论缓存行时，我们默认假设缓存行的起始位置也是某个字（无论是整数还是浮点数）的起始位置。但这并非必然：一个 8 字节的浮点数可能会跨两个缓存行的边界存储。可想而知这对性能不利。第 25.1.3 节将探讨实际应用中解决缓存行边界对齐问题的方法。

1.3.5.8 缓存映射

缓存离浮点运算单元越近，速度越快但容量越小，即便如此，最大的缓存也远小于主存容量。在 1.3.5.6 节中我们已讨论过如何决定保留或替换哪些元素。

现在我们将探讨缓存映射问题，即 ‘若某数据项存入缓存，应置于何处’。该问题通常通过将数据项的（主存）地址映射到缓存地址来解决，由此引出 ‘若两个数据项被映射至同一地址该如何处理’ 的命题。

1.3.5.9 直接映射缓存

最简单的缓存映射策略是直接映射。假设内存地址为 32 位长，因此可寻址 4G 字节 [2]；进一步假设缓存有 8K 字，即 64K 字节，需要 16 位寻址。直接映射会从每个内存地址中提取最后（ ‘最低有效’ ） 16 位，并将其用作缓存中数据项的地址；参见图 1.9。

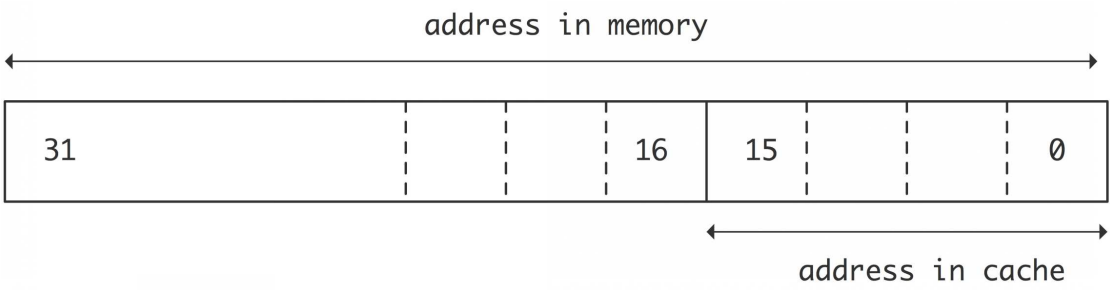


图 1.9：将 32 位地址直接映射到 64K 缓存中。

直接映射非常高效，因为其地址计算可以极快完成，从而实现低延迟，但在实际应用中存在一个问题。如果两个被寻址的项相隔 8K 字，它们将被映射到同一个缓存位置，这会导致某些计算效率低下。示例：

```
double A[3][8192];
for (i=0; i<512; i++)
  a[2][i] = ( a[0][i]+a[1][i] )/2.;
```

或用 Fortran 表
示：

2. We implicitly use the convention that K,M,G suffixes refer to powers of 2 rather than 10: 1K=1024, 1M=1,048,576, 1G=1,073,741,824.

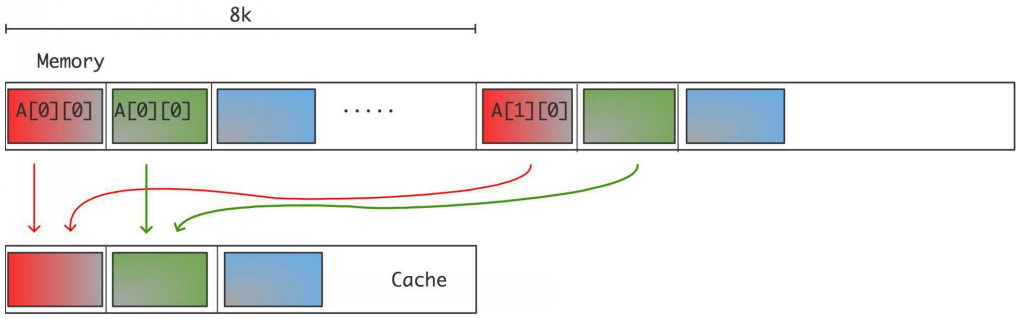


图 1.10：直接映射缓存中的映射冲突。

```
real*8 A(8192,3);do i=1,512
a(i,3) = ( a(i,1)+a(i,2) )/2
end do
```

H此处， a[0][i]、 a[1][i] 和 a[2][i] （或 a(i,1)、 a(i,2)、 a(i,3)）的位置彼此相距 8K，o对于每个 i 而言，其地址的最后 16 位将相同，因此它们会被映射到t缓存中的同一位置；参见图 1.10。

循环的执行过程现在将如下进行：

- 位于 a[0][0] 的数据被载入缓存和寄存器，这会产生一定延迟。与该元素一起，整个缓存行被传输。
- 位于 a[1][0] 的数据连同其整个缓存行被载入缓存（及寄存器，此后不再赘述），并伴随一定延迟。由于此缓存行与第一条映射至相同位置，第一条缓存行被覆盖。
- 为了写入输出，包含 a[2][0] 的缓存行被载入内存。此行再次映射到相同位置，导致刚为 a[1][0] 加载的缓存行被刷新。
- 在下一轮迭代中，需要 a[0][1] （它与 a[0][0] 位于同一缓存行）。但该缓存行已被刷新，因此需重新从主存或更深层缓存调入。此过程中，它会覆盖持有 a[2][0] 的缓存行。
- 类似的情况也发生在 a[1][1] 身上：它位于一个 [1][0], 的缓存线上，不幸的是在前一步骤中被覆盖了。

如果一条缓存线可容纳四个字，我们会发现循环的每四次迭代都涉及 a 元素的八次传输 —— 若没有缓存冲突，仅需两次传输即可完成。

练习 1.8. 在直接映射缓存的示例中，从内存到缓存的映射是通过使用 32 位内存地址的最后 16 位作为缓存地址来实现的。请证明：若改用前（'最高有效'） 16 位作为缓存地址，该示例中的问题会消失。为什么这一方案在通常情况下并非良策?

注释 4 迄今为止，我们一直假设缓存是基于虚拟内存地址的。实际上，缓存依据的是内存中数据的物理地址，而这些地址取决于将虚拟地址映射到内存页的算法。

1.3.5.10 关联缓存

前文所述的缓存冲突问题，若任何数据项可存入任意缓存位置即可解决。此时除缓存填满外不会产生冲突，而缓存替换策略（见章节 1.3.5.6 ）会清空数据为新项腾出空间。此类缓存称为全关联缓存，虽看似最优，但构建成本极高，且使用速度远慢于直接映射缓存。

因此最常见的解决方案是采用 k 路关联缓存，其中 k 至少为二。此时数据项可存入 k 个缓存位置中的任意一个。

本节探讨关联性概念；缓存关联性的实际应用详见章节 6.5。

代码需出现 $k+1$ 路冲突后，数据才会如上例般被提前清空。该例中 $k=2$ 值已足够，但实践中常遇到更高值。图 1.11 展示了直接映射缓存中内存地址到缓存位置的映射关系

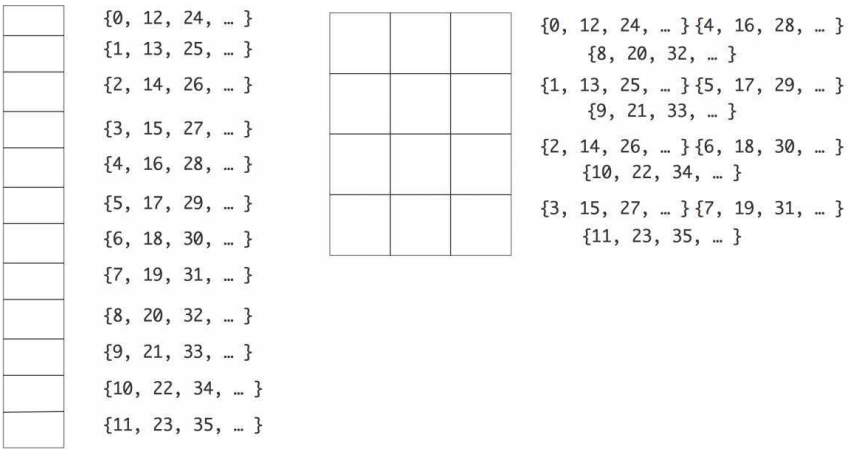


图 1.11：两种 12 元素的缓存结构：直接映射缓存（左）与 3 路组相联缓存（右）。

以及一个 3 路组相联缓存。两种缓存均有 12 个元素，但使用方式不同。直接映射缓存（左）中内存地址 0 和 12 会发生冲突，而在 3 路组相联缓存中这两个地址可映射到任意三个元素上。

以实际产品为例，*Intel Ice Lake* 处理器的 L1 缓存为 48K 字节，采用 12 路组相联结构，缓存行大小为 64 字节；L2 缓存为 1.25M 字节，采用 20 路组相联结构，缓存行大小同样为 64 字节。

习题 1.9. 用你熟悉的编程语言编写一个简易缓存模拟器。假设采用 k 路组相联结构的 32 条目缓存，架构支持 16 位地址。针对 $k=1$ 、 2 、 4 、 ... 分别运行以下实验：

1. 设 k 为模拟缓存的关联度。
2. 编写从 16 位内存地址到 $32/k$ 缓存地址的转换。3. 生成 32 个随机机器地址，并模拟将其存储到缓存中。

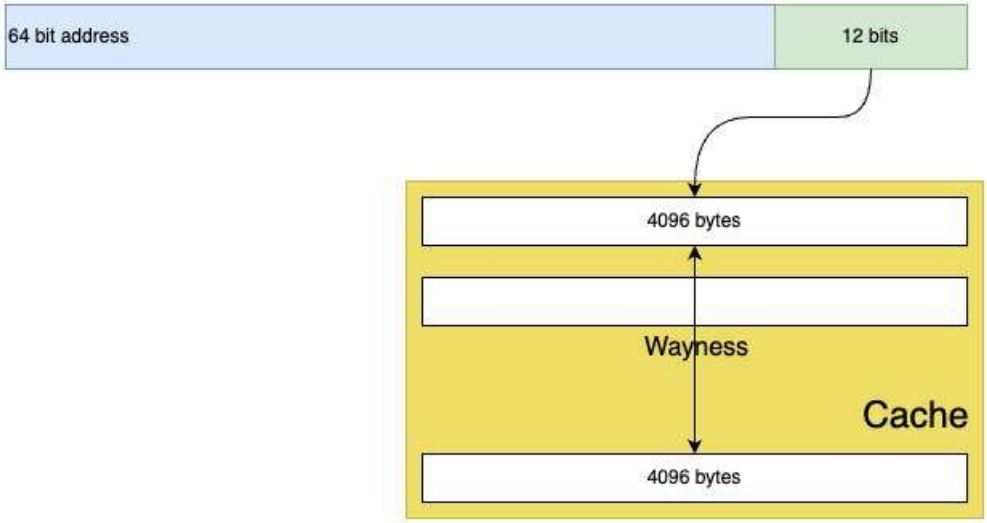


图 1.12：关联缓存结构

由于缓存有 32 个条目，理想情况下所有 32 个地址都能存储在缓存中。但实际发生这种情况的概率很小，通常一个地址的数据会在被驱逐出缓存（即被覆盖）当另一个地址与其冲突时。记录在模拟结束时，32 个地址中实际有多少存储在缓存中。将步骤 3 重复 100 次，并绘制结果；给出中位数、平均值和标准差。观察到增加关联性会提高存储的地址数量。极限行为是什么？（加分项：进行正式的统计分析。）

1.3.5.11 缓存内存与常规内存

那么缓存内存有什么特别之处？为什么我们不将其技术用于所有内存？

缓存通常由静态随机存取存储器（*SRAM*）组成，其速度比动态随机存取存储器（*DRAM*）更快
ccessMemory (DRAM) 用于主内存，但也更昂贵，每个需要 5-6 个晶体管
b而非一个，并且消耗更多电力。

1.3.5.12 加载与存储

在上述描述中，程序中访问的所有数据都需要先移入缓存，才能执行使用它的指令。这既适用于读取的数据，也适用于写入的数据。然而，对于写入且不会再次（在合理时间内）需要的数据，没有理由留在缓存中，这可能会产生冲突或驱逐仍可重用的数据。因此，编译器通常支持流式存储：纯输出的连续数据流将直接写入内存，而不会被缓存。

1.3.6 预取流

在传统的冯 · 诺依曼模型（章节 1.1 ）中，每条指令都包含其操作数的位置信息，因此实现该模型的 CPU 会为每个新操作数发起单独的请求。实际上，后续数据项往往在内存中相邻或呈规律间隔分布。内存系统可通过观察缓存未命中点来检测此类数据模式，并请求一个预取数据流；如图 1.13 所示。

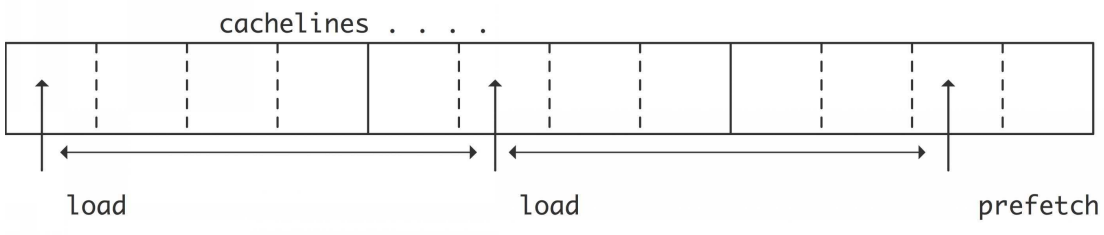


图 1.13：由等距请求生成的预取流。

最简单的实现方式是，CPU 会检测到连续加载的数据来自两个相邻的缓存行，并自动发起对下一个缓存行的请求。如果代码实际请求了第三个缓存行，该过程可重复或扩展。由于这些缓存行在需要之前就已从内存中预取，预取技术有望消除除前几个数据项之外的所有访问延迟。

预取流通常无法跨越小页边界，因此将数据对齐到小页边界是有益的。通常这意味着起始地址应为 4K 的倍数；详见章节 1.3.9。

现在需要稍微重新审视缓存未命中的概念。从性能角度来看，我们只关注缓存未命中导致的停顿，即计算必须等待数据载入的情况。不在缓存中但能在处理其他指令时被载入的数据并不构成问题。若将 ‘L1 未命中’ 仅理解为 ‘未命中导致的停顿’，则术语 ‘L1 缓存重填’ 用于描述所有缓存行的加载，无论处理器是否因此停顿。

1.3.6.1 预取指令

由于预取由硬件控制，它也被称为硬件预取。预取流有时可通过软件控制，例如通过内联函数实现。

由程序员引入预取需要在多种因素间谨慎权衡 [92]。其中最关键的是预取距离：即预取操作开始到数据被实际使用之间的时钟周期数。实践中，这通常表现为循环迭代次数：预取指令会为未来某次迭代请求数据。

1.3.7 并发与内存传输

在讨论内存层次结构时我们曾指出，内存速度低于处理器。更棘手的是，要充分利用内存提供的全部带宽也非易事。

换言之，若编程时不够谨慎，实际获得的性能甚至会低于基于可用带宽预期的水平。让我们对此进行分析。

内存系统通常每个周期能传输超过一个浮点数的带宽，因此需要每个周期发起相应数量的请求才能充分利用可用带宽。即使延迟为零时亦是如此；而实际存在延迟时，数据从内存传输到处理器需要时间。因此，基于第一批数据计算结果所请求的任何后续数据，其请求必须至少延迟一个内存延迟周期才能发出。

要完全利用带宽，必须始终保持传输中的数据量等于带宽乘以延迟。由于这些数据必须相互独立，我们由此得出利特尔法则 [140]：

$$\text{并发度} = \text{带宽} \times \text{延迟}$$

如图 1.14 所示。维持这种并发度的难点并不在于程序无法

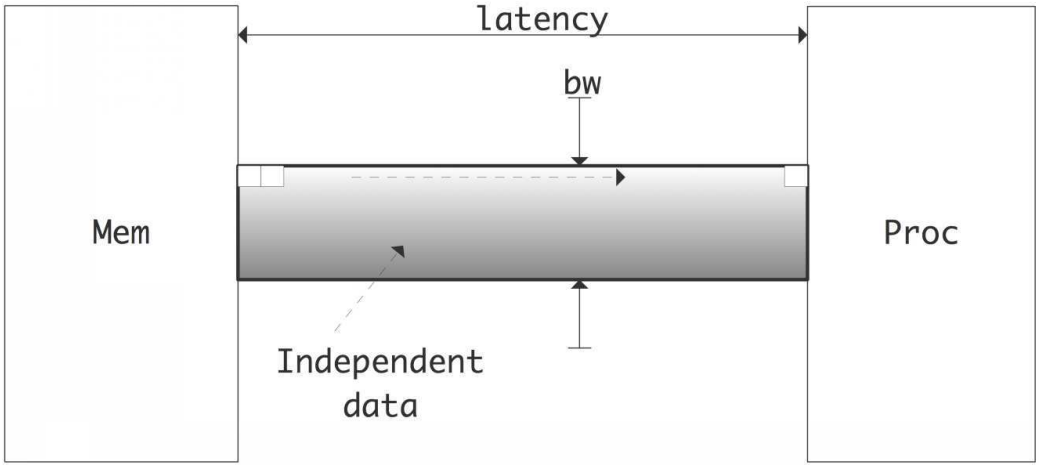


图 1.14：利特尔定律示意图，该定律阐述了独立数据在传输过程中需要保持的并发量。

问题不在于缺乏这种能力，而在于如何让编译器和运行时系统识别它。例如，如果循环遍历一个长数组，编译器不会发出大量内存请求。预取机制（章节 1.3.6 ）会提前发出一些内存请求，但通常数量不足。因此，为了利用可用带宽，需要同时维持多个数据流。基于此，我们也可以将利特尔定律表述为

$$\text{有效吞吐量} = \text{表达并发量} / \text{延迟}。$$

1.3.8 内存库

前文我们讨论了与带宽相关的问题。可以看到，内存（以及较小程度上的缓存）的带宽低于处理器最大吸收能力。实际情况比上述讨论更为严峻。因此，内存通常被划分为内存

banks 交错排列的存储体：在四存储体结构中，字 0、4、8、... 位于存储体 0，字 1、5、9、... 位于存储体 1，以此类推。

假设现在按顺序访问内存，那么这种 4 路交错存储体可维持单存储体四倍的带宽。但若以步长 2 访问，带宽将减半，更大步长则更不理想。若连续两次操作访问同一存储体，则称为存储体冲突 [7]。实际应用中存储体数量会更多，因此小步长跨距访问仍能保持标称带宽。例如 *Cray-1* 有 16 个存储体，*Cray-2* 则达 1024 个。

习题 1.10. 证明当存储体数量为质数时，任何小于该数的步长访问都不会引发冲突。你认为实际存储架构为何不采用此方案?

在现代处理器中，*DRAM* 仍采用存储体（bank）结构，但由于缓存的存在，其影响已有所减弱。然而，*GPU* 具有存储体却无缓存机制，因此仍会遭遇与早期超级计算机类似的某些问题。

习题 1.11：数组元素求和的递归倍增算法如下：

```
for (s=2; s<2*n; s*=2)
  for (i=0; i<n-s/2; i+=s)
    x[i] += x[i+s/2]
```

分析该算法的存储体冲突。假设 $n = 2^p$ 且每个存储体包含 2^k 个元素其中 $k < p$。同时将其视为并行算法时，内层循环的所有迭代相互独立，因此可同步执行。或者，我们也可采用递归折半法：

```
for (s=(n+1)/2; s>1; s/=2)for (i=0; i<n; i+=1)x[i] += x[i+s]
```

再次分析存储体冲突。该算法更优吗？在并行情况下表现如何?

缓存内存同样可采用存储体结构。例如，*AMDBarcelona* 芯片中 L1 缓存的缓存行长度为 16 字，被划分为两个交错排列的 8 字存储体。这意味着对缓存行元素的顺序访问是高效的，但跨步访问的性能则会显著下降。

1.3.9 TLB、页与虚拟内存

程序的所有数据可能不会同时存在于内存中。这种情况可能由多种原因导致：

- 计算机为多用户服务，因此内存并非专供单一用户使用；
- 计算机同时运行多个程序，这些程序所需内存总量超过物理可用内存；
- 单个程序使用的数据量可能超过可用内存容量。

因此，计算机采用虚拟内存机制：当所需内存超过实际可用容量时，特定内存块会被写入磁盘。实际上，磁盘充当了真实内存的扩展。这意味着数据块可能位于内存中的任意位置，甚至若被置换进出，它在不同时间可能处于不同位置。置换操作并非作用于单个内存地址，而是针对内存块进行。

页：连续的内存块，大小从几千字节到几兆字节不等。（在早期的操作系统中，将内存移至磁盘是程序员的责任。相互替换的页被称为覆盖。）

因此，我们需要一种从程序使用的内存地址到实际内存地址的转换机制，且这种转换必须是动态的。程序有一个 ‘逻辑数据空间’ （通常从地址零开始），用于存放编译代码中使用的地址，这些地址需要在程序执行时转换为实际内存地址。为此，存在一个页表，用于指定哪些内存页包含哪些逻辑页。

1.3.9.1 大页

在非常不规则的应用程序中，例如数据库，随着或多或少随机数据被载入内存，页表可能会变得非常大。然而，有时这些页会表现出一定程度的聚集性，这意味着如果页尺寸更大，所需页的数量将大幅减少。因此，操作系统可以支持大页，通常大小约为 2Mb。（有时会使用 ‘巨页’ ；例如 *Intel Knights Landing* 具有千兆字节大小的页。）

大页面的优势取决于具体应用场景：若小页面缺乏足够的聚集性，使用大页面可能导致内存过早被大页面中未使用的部分填满。

1.3.9.2 TLB

然而，通过查表进行地址转换速度较慢，因此 CPU 配备了转译后备缓冲器 （ ***TLB*** ）。TLB 是高频使用页表项的缓存：它为若干页面提供快速地址转换。当程序需要访问某内存位置时，会查询 TLB 以确认该位置是否位于 TLB 记录的页面中。若命中，逻辑地址将极速转换为物理地址；若 TLB 未记录该页面则称为 ***TLB*** 缺失，此时需查询页查找表，必要时将所需页面调入内存。TLB 采用（有时完全）相联结构（章节 1.3.5.10 ），并遵循 LRU 替换策略（章节 1.3.5.6 ）。

典型的 TLB （转译后备缓冲器）通常包含 64 至 512 个条目。若程序按顺序访问数据，通常仅会在少数几个页面间交替访问，因此不会出现 TLB 未命中情况。反之，若程序随机访问大量内存地址，则可能因 TLB 未命中而导致性能下降。当前正在使用的页面集合被称为 ‘工作集’ 。

Section 6.4.4 和附录 25.5 讨论了展示 TLB 行为的简单代码示例。

[实际情况更为复杂。例如，通常存在多个 TLB：第一个与 L2 缓存关联，第二个与 L1 缓存关联。在 *AMD Opteron* 处理器中，L1 TLB 包含 48 个条目且为全相联（ 48 路），而 L2 TLB 虽有 512 个条目但仅为 4 路相联。这意味着可能出现 TLB 冲突。前文讨论仅涉及 L2 TLB。其能与 L2 缓存（而非主存）关联的原因在于：从内存到 L2 缓存的地址转换是确定性的。]

使用大页还能减少潜在的 TLB 未命中次数，因为页面的工作集可以被缩减。

1.4 多核架构

近年来，传统处理器芯片设计的性能极限已被触及。

- 时钟频率无法继续提升，因为这会增加能耗，导致芯片过热；详见章节 1.7.1。
- 无论是由于编译器的限制、内在可用并行度的有限性，还是分支预测的不可行性（参见章节 1.2.5），从代码中提取更多指令级并行（ILP）已无可能。

提高单个处理器芯片利用率的方法之一，是从进一步优化单一处理器的策略转向将芯片划分为多个处理 ‘核心’。这些独立核心可以处理不相关的任务，或通过引入实质上的数据并行（章节 2.3.1）协作完成共同任务，从而实现更高的整体效率 [153]。

备注 5 另一种解决方案是英特尔的超线程技术，它允许处理器混合多个指令流的指令。其优势高度依赖于具体应用场景。然而，这一机制在 *GPU* 中取得了巨大成功；详见章节 *2.9.3*。相关讨论参见章节 *2.6.1.9*。

这解决了上述两个问题：

- 两个低频核心的吞吐量可等同于单个高频处理器，因此多核架构能效更高。
- 原先通过硬件发现的指令级并行 (ILP) 现被程序员管理的显式任务并行所取代。

早期的多核 CPU 仅是简单地将两个处理器集成在同一晶片上，后续世代则加入了两个处理器核心共享的 L3 或 L2 缓存；见图 1.15。这种设计使得

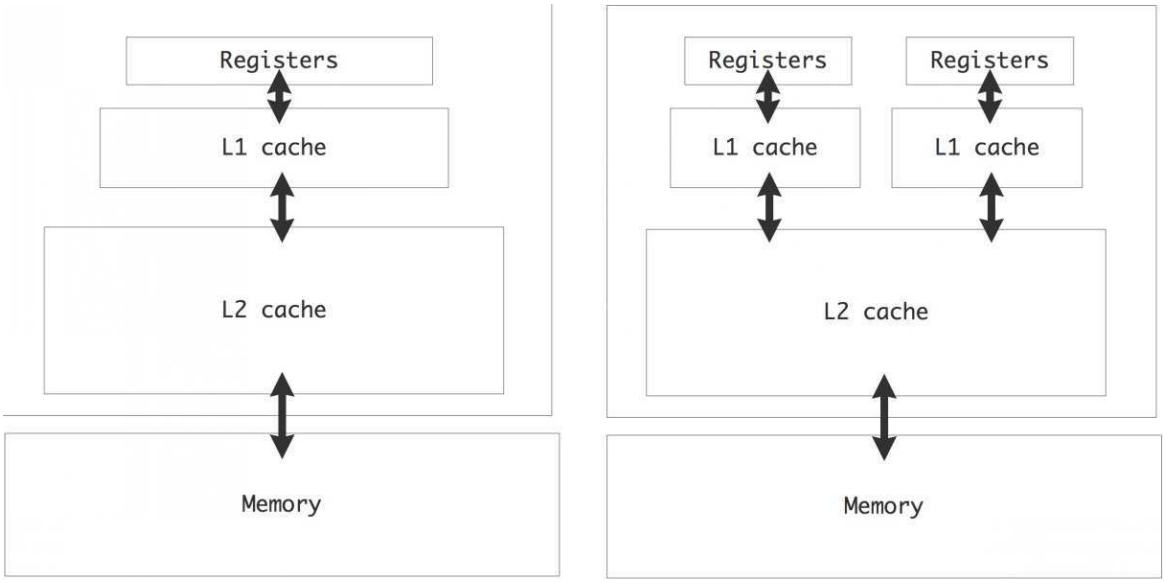


图 1.15：单核与双核芯片中的缓存层次结构。

高效地让核心协同处理同一问题。各核心仍会拥有自己的 L1 缓存，而这些独立的缓存会引发缓存一致性问题；详见 1.4.1 小节。

需注意，“处理器”一词如今存在歧义：既可指代整个芯片，也可指芯片上的处理器核心。因此，我们通常用插槽代指整个芯片，而用核心指代包含独立算术逻辑单元及寄存器的部分。当前，即使是笔记本电脑也普遍配备 4 核或 6 核 CPU，英特尔和 AMD 更已推出 12 核芯片的商用产品。未来核心数量很可能继续增加：英特尔曾展示过 80 核原型机，并在此基础上开发出 48 核的‘单芯片云计算计算机’（见图 1.16）。该芯片采用 24 个双核‘区块’通过 2D 网状网络互联的结构，仅特定区块连接内存控制器，其余区块需通过片内网络访问内存。

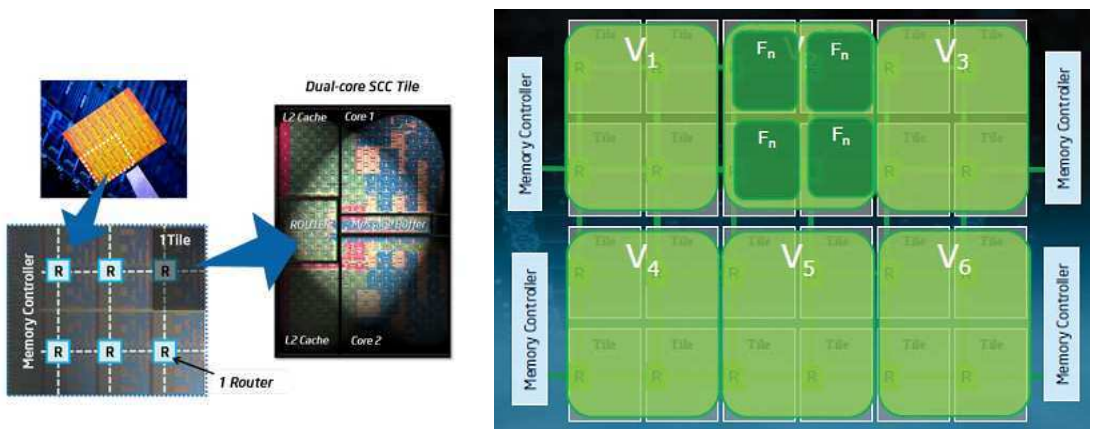


图 1.16：英特尔单芯片云计算芯片的结构。

这种共享与私有缓存的混合配置，使得多核处理器的编程模型正演变为共享内存与分布式内存的混合体：

核心 每个核心拥有私有的 L1 缓存，这属于分布式内存的一种形式。前文提及的英特尔 80 核原型芯片中，各核心间采用分布式内存模式通信。插槽 单个插槽上通常设有共享的 L2 缓存，作为各核心间的共享内存。节点 单个 ' 节点 ' 或主板上可配置多个插槽，共同访问同一共享内存空间。

网络 需采用分布式内存编程（参见下一章）来实现节点间的通信。

历史上，多核架构可追溯至多处理器共享内存设计（见章节 2.4.1），例如 *Sequent Symmetry* 与 *AlliantFX/8*。其编程模型在概念上相同，但现有技术已将多处理器板卡浓缩至多核芯片。

1.4.1 缓存一致性

在并行处理中，若多个处理器持有同一数据项的副本，则存在潜在的冲突风险。确保所有缓存数据均为主内存的准确副本这一问题，被称为缓存一致性：即当一个处理器修改其副本时，其他副本需同步更新。

在分布式内存架构中，数据集通常被不相交地划分到各个处理器上，因此数据的冲突副本只能在用户知晓的情况下产生，并由用户自行处理该问题。共享内存的情况则更为微妙：由于进程访问的是相同的主内存，表面上似乎不可能出现冲突。然而，处理器通常拥有一些私有缓存，其中存储着来自内存的数据副本，因此仍可能出现冲突副本。这种情况尤其会出现在多核设计中。

假设两个核心在其（私有） L1 缓存中拥有同一数据项的副本，且其中一个核心修改了其副本。此时另一个核心缓存的数据将不再是对应项的准确副本：处理器会使该数据项的副本失效，实际上会使整个缓存行失效。当进程再次需要访问该数据项时，必须重新加载该缓存行。另一种方案是让任何修改数据的核心将该缓存行发送给其他核心。这种策略可能带来更高的开销，因为其他核心不太可能拥有该缓存行的副本。

这一更新或使缓存行失效的过程被称为维护缓存一致性，它由处理器在极底层完成，无需程序员介入（这使得内存位置的更新成为原子操作；更多细节见章节 2.6.1.5 ）。然而，这会降低计算速度，并浪费本可用于加载或存储操作数的核心带宽。

缓存行相对于主内存中数据项的状态通常被描述为以下之一：

空白（ Scratch ）：该缓存行不包含该数据项的副本；
有效（ Valid ）：该缓存行是主内存中数据的正确副本；独占（
Reserved ）：该缓存行是该数据块的唯一副本；
脏（ Dirty ）：该缓存行已被修改但尚未写回主内存；无效（ Invalid ）：该缓存行上的数据也存在于其他处理器上（未被独占），且其他进程已修改其数据副本。

该协议的一个简化变体是 Modified-Shared-Invalid （ MSI ）一致性协议，其中缓存行在给定核心上可处于以下状态：

Modified （已修改）：缓存行已被修改，需写回后备存储。此写入操作可在行被驱逐时执行，或根据写回策略立即执行。Shared （共享）：该行存在于至少一个缓存中且未被修改。I
nvalid （无效）：该行不存在于当前缓存中，或存在但另一个缓存中的副本已被修改。

这些状态控制着缓存行在内存与缓存之间的移动。例如，假设某核心读取一个在该核心上处于无效状态的缓存行。随后它可从内存加载该行，或从其他缓存获取（可能更快）。（检查某行是否以 M 或 S 状态存在于其他缓存中的过程称为侦听；另一种方法是维护缓存目录，详见下文。）若该行处于 Shared 状态，可直接复制；若在其他缓存中处于 M 状态，该核心需先将其写回内存。

练习 1.12. 考虑两个处理器、内存中的数据项 x，以及映射到 x 的两个处理器私有缓存中的缓存行 x_1 和 x_2。描述在两个处理器对 x 进行读写操作时， x_1 和 x_2 状态之间的转换。此外，

标明哪些操作会占用内存带宽。（此状态转换列表属于有限状态自动机（*FSA*）；详见章节 21。）

MSI 协议的变种通过增加 ‘独占’ 或 ‘拥有’ 状态来提升效率。

1.4.1.1 缓存一致性解决方案

实现缓存一致性有两种基本机制：监听机制和基于目录的方案。

在监听机制中，所有数据请求会被广播至所有缓存，若数据存在于任一缓存则直接返回；否则从内存中获取。该机制的变体中，核心会 ‘监听’ 所有总线流量，当其他核心修改其缓存行副本时，该核心能使自己的缓存行副本失效或更新。失效操作比更新更高效，因其仅需位操作，而更新涉及整行缓存复制。

练习 1.13. 何时更新会带来收益？编写一个简单的缓存模拟器来评估这个问题。

由于侦听通常涉及向所有核心广播信息，其扩展性在核心数量较少时就会受限。一种更具扩展性的解决方案是使用标签目录：一个中央目录，包含关于哪些数据存在于某个缓存中以及具体位于哪个缓存中的信息。对于具有大量核心的处理器（如 *Intel Xeon Phi*），该目录可以分布在各个核心上。

1.4.1.2 标签目录

在具有分布式但一致性的缓存的多核处理器中（如 *Intel Xeon Phi*），标签目录本身也可以被分布式存储。这会增加缓存查找的延迟。

1.4.2 伪共享

即使核心访问的是不同的数据项，缓存一致性问题也可能出现。例如，一个声明

```
double x,y;
```

很可能会将 `x` 和 `y` 分配在内存中相邻的位置，因此它们有很大概率落在同一缓存行上。此时如果一个核心更新 `x` 而另一个核心更新 `y`，该缓存行就会不断在核心间来回迁移。这种现象被称为伪共享。

T伪共享最常见的情况发生在多个线程更新数组中连续的元素时。
F例如，在以下 OpenMP 代码片段中，所有线程都在更新数组中各自负责的
p部分结果：

```
  local_results = new double[num_threads];
#pragma omp parallel
{
  int thread_num = omp_get_thread_num();
  for (int i=my_lo; i<my_hi; i++)
    local_results[thread_num] = ... f(i) ...
}
global_result = g(local_results)
```

虽然实际上并不存在竞态条件（如果所有线程都更新 `global_result` 变量则会出现），但这段代码可能性能较低，因为包含 `local_result` 数组的缓存行会不断失效。

话虽如此，在现代 CPU 中，伪共享问题已不如过去严重。让我们来看一段实现上述方案的代码：

```
for (int spacing : {16,12,8,4,3,2,1,0} ) {
    int iproc = omp_get_thread_num();
    floattype *write_addr = results.data() + iproc*spacing;
    for (int r=0; r<how_many_repeats; r++) {
      for (int w=0; w<stream_length; w++) {
        *write_addr += *( read_stream+w );
      }
    }
```

在 *Intel Corei5* 处理器（搭载于 *Apple Macbook Air*）上运行时，仅表现出轻微的性能下降：

```
executing: OMP_NUM_THREADS=4 OMP_PROC_BIND=true ./falsesharing-omp
Running with 4 threads; each doing stream length=9000
Spacing 16.. ran for  1393069 usec                          ,

                                  equivalent serial time: 1.39307e+06 usec (efficiency: 100% ); ,
    operation count: 3.6e+09
Spacing 12.. ran for  1337529 usec
                                  equivalent serial time: 1.33753e+06 usec (efficiency: 104% ); ,
    operation count: 3.6e+09
Spacing  8.. ran for  1307244 usec
                                  equivalent serial time: 1.30724e+06 usec (efficiency: 106% ); ,
    operation count: 3.6e+09
Spacing  4.. ran for  1368394 usec
                                  equivalent serial time: 1.36839e+06 usec (efficiency: 101% ); ,
    operation count: 3.6e+09
Spacing  3.. ran for  1603778 usec
                                  equivalent serial time: 1.60378e+06 usec (efficiency:  86% ); ,
    operation count: 3.6e+09
Spacing  2.. ran for  2044134 usec
                                  equivalent serial time: 2.04413e+06 usec (efficiency:  68% ); ,
    operation count: 3.6e+09
Spacing  1.. ran for  1819370 usec
                                  equivalent serial time: 1.81937e+06 usec (efficiency:  76% ); ,
    operation count: 3.6e+09
Spacing  0.. ran for  1811778 usec
                             equivalent serial time: 1.81178e+06 usec (efficiency:  76% );
operation count: 3.6e+09
```

另一方面，在 *TACC Frontera* 集群的 *Intel Cascade Lake* 上，我们并未观察到类似现象：

```
executing: OMP_NUM_THREADS=24 OMP_PROC_BIND=close OMP_PLACES=cores  ./falsesharing-omp
Running with 24 threads; each doing stream length=9000
Spacing 16.. ran for 0.001127 usec  equivalent serial time:       1127 usec (efficiency:   100% );
   Spacing 12.. ran for 0.001103 usec,                                 equivalent serial time:       1103 usec (efficiency: 102.1% ); ,
                                          operation count: 1.08e+07
                                     Spacing  8.. ran for 0.001102 usec
                                   equivalent serial time:       1102 usec (efficiency: 102.2% ); ,
    operation count: 1.08e+07
Spacing  4.. ran for 0.001102 usec
                                   equivalent serial time:       1102 usec (efficiency: 102.2% ); ,
    operation count: 1.08e+07
Spacing  3.. ran for 0.001105 usec
                                   equivalent serial time:       1105 usec (efficiency: 101.9% ); ,
    operation count: 1.08e+07
Spacing  2.. ran for 0.001104 usec
                                   equivalent serial time:       1104 usec (efficiency:   102% ); ,
    operation count: 1.08e+07
Spacing  1.. ran for 0.001103 usec
                                   equivalent serial time:       1103 usec (efficiency: 102.1% ); ,
    operation count: 1.08e+07
Spacing  0.. ran for 0.001104 usec
                              equivalent serial time:       1104 usec (efficiency:   102% );
operation count: 1.08e+07
```

原因在于硬件会将累加器变量缓存起来，直到循环结束时才写入内存。这完全避免了伪共享带来的所有问题。

我们可以通过强制写回内存来引发伪共享问题，例如使用 *OpenMPatomic* 指令：

```
executing: OMP_NUM_THREADS=24 OMP_PROC_BIND=close OMP_PLACES=cores  ./falsesharing-omp
Running with 24 threads; each doing stream length=9000Spacing 16.. ran for  0.01019 usec,
 equivalent serial time:  1.019e+04 usec (efficiency:   100% );operation count: 1.08e+07
Spacing 12.. ran for  0.01016 usec,  equivalent serial time:  1.016e+04 usec (efficiency: 100.2% );
operation count: 1.08e+07Spacing  8.. ran for  0.01016 usec,
 equivalent serial time:  1.016e+04 usec (efficiency: 100.2% );operation count: 1.08e+07
Spacing  4.. ran for   0.1414 usec,  equivalent serial time:  1.414e+05 usec (efficiency:   7.2% );
operation count: 1.08e+07Spacing  3.. ran for   0.1391 usec,
 equivalent serial time:  1.391e+05 usec (efficiency:   7.3% );operation count: 1.08e+07
Spacing  2.. ran for    0.184 usec,  equivalent serial time:   1.84e+05 usec (efficiency:   5.5% );
operation count: 1.08e+07Spacing  1.. ran for   0.2855 usec,
 equivalent serial time:  2.855e+05 usec (efficiency:   3.5% );operation count: 1.08e+07
Spacing  0.. ran for   0.3542 usec,  equivalent serial time:  3.542e+05 usec (efficiency:   2.8% );
operation count: 1.08e+07
```

(Of course, this has a considerable performance penalty by itself.)

1.4.3 多核芯片上的计算

多核处理器可通过多种方式提升性能。首先，在桌面环境下，多核实际上可以同时运行多个程序。更重要的是，我们可以利用并行性来加速单个代码的执行。这可以通过两种不同的方式实现。

MPI 库（章节 2.6.3.3 ）通常用于通过网络连接的处理器间通信。然而，它也可用于单个多核处理器：此时 MPI 调用通过共享内存复制实现。

另一种方法是利用共享内存和共享缓存，并使用 OpenMP 等线程系统进行编程（章节 2.6.2 ）。这种模式的优势在于并行性可以更加动态，因为运行时系统能在程序运行期间设置并更改线程与核心的对应关系。

我们将详细讨论多核芯片上线性代数运算的调度；参见章节 7.12。

1.4.4 TLB 击落

章节 1.3.9.2 解释了 TLB 如何用于缓存从逻辑地址（即逻辑页）到物理页的转换。TLB 是 *socket* 内存单元的一部分，因此在多 socket 设计中，一个 socket 上的进程可能更改页映射，导致其他 socket 上的映射失效。

该问题的一种解决方案称为 *TLB* 击落：更改映射的进程生成一个处理器间中断，触发其他处理器重新构建其 TLB。

1.5 节点架构与套接字

在前面的章节中，我们沿着内存层级结构向下探索，依次了解了寄存器、各级缓存（Cache）及其私有或共享的特性。内存层级的底层是所有核心共享的主存，其容量从普通笔记本电脑的几 GB 到超级计算机中心的几 TB 不等。

尽管所有核心共享此内存，但其内部存在一定的结构。这源于一个集群节点可以拥有多个 *socket*（即处理器芯片）的事实。共享内存位于

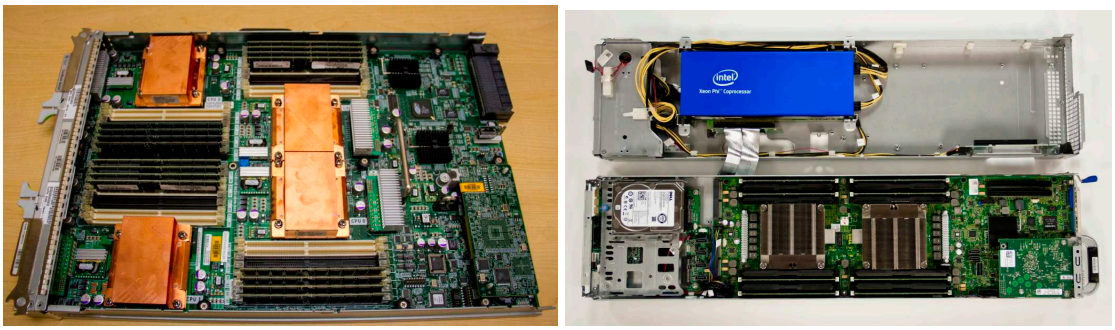


图 1.17：左图展示了一个四插槽设计。右图则是一个带有协处理器的双插槽设计。

节点通常分布在直接连接到特定插槽的内存条上。这一点在图 1.17 中得到了展示，该图显示了 *TACC Ranger* 集群超级计算机（已停产）的四插槽节点，以及 *TACC Stampede* 集群超级计算机的双插槽节点，后者包含一个 *IntelXeon Phi* 协处理器。在这两种设计中，可以清晰地看到直接与插槽相连的内存芯片。

1.5.1 设计考量

设想一个由 N 节点组成的超级计算机集群，每个节点包含 S 个插槽，每个插槽有 C 个核心。此时我们不禁思考：‘如果 $S > 1$，为何不降低 S，并增加 N 或 C？’ 这类问题的答案往往既受性能驱动，也受价格影响。

- 增加每个插槽的核心数量可能会受到芯片制造工艺的限制。
- 增加每个插槽的核心数也可能降低每个核心的可用带宽。这对高性能计算（HPC）应用尤为重要 —— 因其核心常执行相同任务；而对于异构工作负载，此影响可能较小。
- 在 C 固定的情况下，降低 S 并增加 N 通常会导致价格上涨，因为节点价格与 S 的关联性较弱。
- 另一方面，提高 S 会使节点架构更复杂，并可能因维护一致性的复杂度增加（参见章节 1.4.1）而降低性能。同样地，这对异构工作负载影响较小，因此 S 的高值在 Web 服务器中比 HPC 设施更常见。

1.5.2 NUMA 现象

上述节点是非统一内存访问 *(NUMA)* 设计的实例：对于运行在某个核心上的进程而言，访问其所属插槽连接的内存比访问其他插槽连接的内存略快。

这种现象的一个结果是首次接触现象。动态分配的内存在实际首次写入之前并不会真正分配。现在考虑以下 OpenMP （章节 2.6.2 ）代码：

```
double *array = (double*)malloc(N*sizeof(double));
for (int i=0; i<N; i++)
   array[i] = 1;
#pragma omp parallel for
for (int i=0; i<N; i++)
   .... lots of work on array[i] ...
```

B由于首次接触机制，数组会被完全分配在主线程所属插槽的内存中。
I在随后的并行循环中，另一个插槽的核心将较慢地访问内存。
t它们所操作的对象。

此处的解决方案是同样将初始化循环并行化，即使其中工作量可能微不足道。

1.6 局部性与数据复用

至此应当明确，算法执行不仅仅是计算操作数量：涉及的数据传输同样重要，甚至可能主导成本。由于我们拥有缓存和寄存器，通过编程使数据尽可能靠近处理器，可以最小化数据传输量。这部分取决于巧妙的编程，但我们也可以从理论角度审视：算法本身是否允许这种优化。

事实证明，在科学计算中，数据通常主要与某种意义上邻近的数据交互，这将导致数据局部性；章节 1.6.2。这种局部性往往源于应用本身特性，如第 4 章将介绍的偏微分方程案例。而在分子动力学（第 8 章）等情况下，由于所有粒子都相互交互，不存在这种固有局部性，需要极高的编程技巧才能实现高性能。

1.6.1 Data reuse and arithmetic intensity

在前几节中，您了解到处理器设计存在某种不平衡：加载数据比执行实际操作更慢。这种不平衡对于主内存尤为显著，而对于各级缓存则相对较小。因此，我们倾向于将数据保留在缓存中，并尽可能提高数据复用的量。

当然，我们首先需要确定计算是否允许数据被复用。为此，我们将算法的算术强度定义如下：

若 n 表示算法操作的数据项数量， $f(n)$ 表示
所需操作次数，则算术强度为 $f(n)/n$。
（数据项可用浮点数或字节衡量。后者更便于将算术强度与处理器的硬件规格关联。）

算术强度也与延迟隐藏相关：即通过持续进行计算活动来抵消数据加载带来的性能负面影响。要实现此效果，计算量需远超数据加载量，这正是计算强度的核心定义 —— 每加载字节 / 字 / 数字所对应的高操作比率。

1.6.1.1 示例：向量运算

以向量加法为例

$$\forall_i : x_i \leftarrow x_i + y_i.$$

每次迭代涉及三次内存访问（两次加载和一次存储）以及一次运算，算术强度为 $1/3$。

基础线性代数子程序（ BLAS ）中的 *axpy* （即 ‘a 乘以 x 加上 y’ 运算

$$\forall_i : x_i \leftarrow a\, x_i + y_i$$

包含两次运算，但由于 a 的首次加载被分摊，内存访问次数与简单加法相同。因此其效率高于简单加法，算术强度达到 $2/3$。

内积计算

$$\forall_i : s \leftarrow s + x_i \cdot y_i$$

结构与 axpy 运算类似，每次迭代对两个向量和一个标量执行一次乘法和加法。但由于 s 可保留在寄存器中，仅在循环结束时写回内存，故仅需两次加载操作。此处的重用率为 1。

1.6.1.2 示例：矩阵运算

接下来考虑矩阵 - 矩阵乘积：

$$\forall_{i,j} : c_{ij} = \sum_k a_{ik} b_{kj}.$$

这涉及 $3n^2$ 个数据项和 $2n^3$ 次运算，属于更高阶的计算。算术强度为 $O(n)$，意味着每个数据项将被重复使用 $O(n)$ 次。这表明，通过适当的编程，该运算有可能通过将数据保留在快速的缓存内存中来克服带宽 / 时钟速度的差距。

练习 1.14. 矩阵 - 矩阵乘积作为一种运算，根据上述定义显然具有数据复用性。请论证这种复用性并非通过简单实现就能轻易达成。是什么决定了朴素实现是否能复用缓存中的数据?

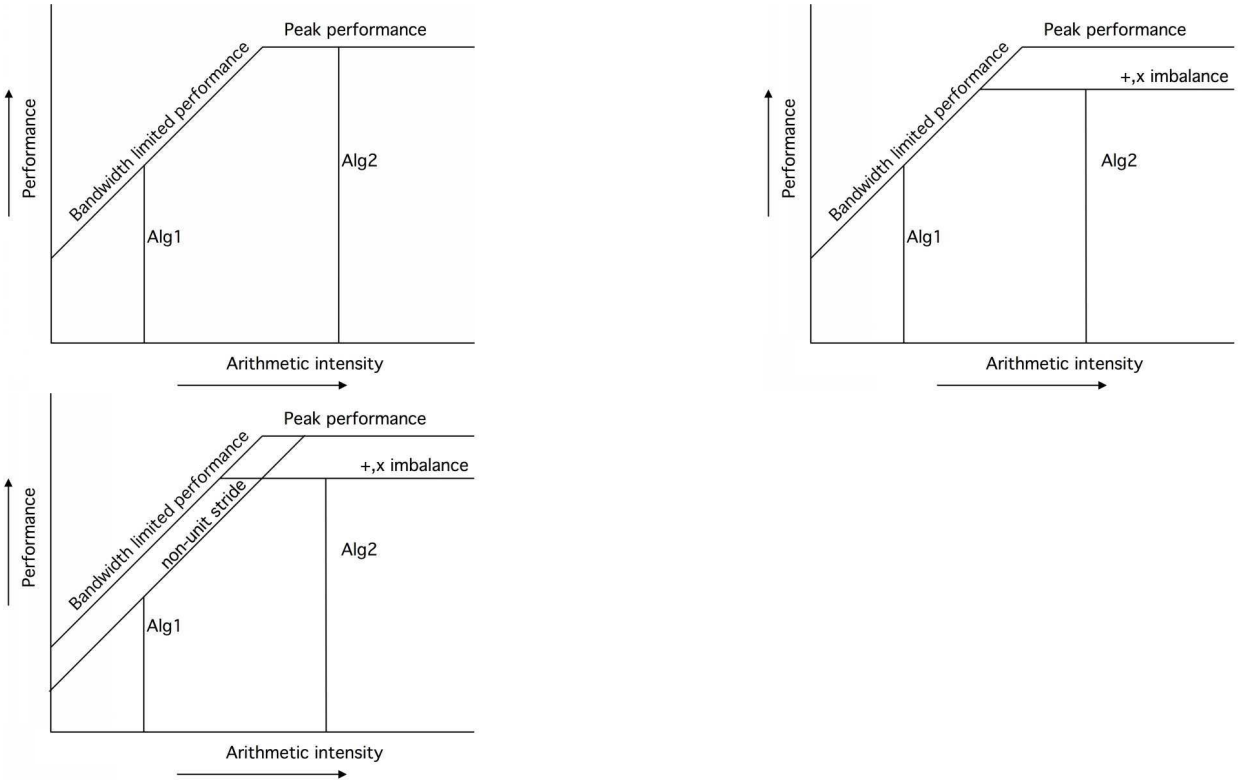


图 1.18：屋顶线模型中决定性能因素的示意图。

I在此讨论中，我们仅关注给定实现的操作次数，而非
数学运算本身。例如，存在方法可以用少于 $O(n^3)$ 次运算完成矩阵乘法与高斯消元算法 [179, 157]。但这需要不同的实现方式，其内存访问与复用特性需另行分析。

矩阵乘积的高性能实现将在 7.4.1 节讨论。

矩阵乘积是 *LINPACK* 基准测试 [51] 的核心（参见 2.11.4 节）。将其作为计算机性能评估的唯一标准可能过于乐观：矩阵乘积具有高度数据复用性，因此对内存带宽及并行计算机的网络特性相对不敏感。通常计算机在 Linpack 测试中能达到峰值性能的 60–90%。涉及稀疏矩阵运算等其他基准测试的结果可能显著更低。

1.6.1.3 屋顶线模型

有一种优雅的方式来描述算术强度（这是关于理想算法的陈述，而非其实现）如何与硬件参数及实际实现相互作用以决定性能。这被称为屋顶线模型 [193], ，它表达了性能受两个因素限制的基本事实，如图 1.18 的第一张图所示。

1. 该峰值性能，由图表顶部的水平线表示，是性能的绝对上限 [3]，仅当 CPU 的每个方面（流水线、多个浮点运算单元）都被完美利用时才能达到。该数值的计算完全基于 CPU 特性和时钟周期；假设内存带宽不是限制因素。
2. 每秒运算次数还受带宽乘积的限制，这是一个绝对数值，与算术强度的乘积：

$$\text{运算次数 秒} = \text{运算次数 数据项} \cdot \frac{\text{数据项}}{\text{秒}}$$

这在图表中表现为线性上升的线段。

屋顶线模型是一种优雅的方式，用于表达各种因素会降低性能上限。例如，若算法未能充分利用 *SIMD* 宽度，这种不平衡会降低可达到的峰值性能。图 1.18 中的第二张图展示了降低性能上限的多种因素。此外，诸如延迟隐藏不完美等因素也会降低可用带宽，这体现在第三张图中倾斜屋顶线的下移。

对于给定的算术强度，性能取决于其垂直线与屋顶线的交点位置。若交点在水平部分，则称该计算为计算受限型：性能由处理器特性决定，带宽不构成瓶颈。反之，若垂直线与屋顶线的倾斜部分相交，则称该计算为带宽受限型：性能由内存子系统决定，处理器的全部算力未被充分利用。

习题 1.15. 如何判断给定程序内核属于带宽受限型还是计算受限型?

1.6.2 局部性

由于使用缓存中的数据比从主内存获取数据成本更低，程序员显然希望通过编码实现缓存数据的重复利用。尽管在大多数 CPU 中（ Cell 处理器和某些 GPU 中程序员可通过低级内存访问控制），缓存数据的放置并非由程序员直接控制，但了解缓存的行为后，仍可推测哪些数据位于缓存中，并在一定程度上加以控制。

这里有两个关键概念：时间局部性与空间局部性。时间局部性最易解释：它描述数据元素在最近被使用后短时间内再次被使用的情况。由于大多数缓存采用 LRU 替换策略（章节 1.3.5.6 ），若两次引用之间所涉及的数据量小于缓存容量，该元素仍会驻留缓存中，从而快速可访问。但对于随机替换等其他策略，则无法保证这一点。

1.6.2.1 时间局部性

以时间局部性为例，考虑对长向量的重复使用 :

```
for (loop=0; loop<10; loop++) {
  for (i=0; i<N; i++) {
```

3. An old joke states that the peak performance is that number that the manufacturer guarantees you will never exceed.

```
    ... = ... x[i] ...
  }
}
```

`x` 的每个元素将被使用 10 次，但如果该向量（加上其他访问的数据）超出缓存容量，每个元素在下一次使用前会被清除。因此，`x[i]` 的使用并未体现时间局部性：后续使用时间间隔过长，无法使其保留在缓存中。

若计算结构允许我们交换循环顺序：`for (i=0; i<N; i++) {`

```
  for (loop=0; loop<10; loop++) {
    ... = ... x[i] ...
  }
}
```

`x` 的元素现在被重复利用，因此更可能保留在缓存中。这种重构代码在使用 `x[i]` 时表现出更好的时间局部性。

1.6.2.2 空间局部性

空间局部性 的概念稍显复杂。若程序引用的内存位置与其先前引用过的内存位置 ‘邻近’，则称该程序展现出空间局部性。在仅含处理器与内存的经典冯·诺依曼架构中，空间局部性本应无关紧要，因为访问内存中任一地址的速度均相同。然而，在现代配备缓存的 CPU 中，情况则截然不同。上文已展示两个体现空间局部性的例子：

- 由于数据以缓存行为单位而非单个字或字节进行传输，若编程时能确保充分利用缓存行的所有元素，将带来显著性能提升。在循环 `for (i=0; i<N*s; i+=s) {... x[i] ...}` 中，空间局部性随步长 `s` 递减。设 `S` 为缓存行大小，当 `s` 取值从 $1 \ldots S$ 变化时，每个缓存行中被利用的元素数量将从 `S` 降至 1。相对而言，这会增加循环中内存流量的成本：若 $s = 1$，则每个元素需加载 $1/S$ 个缓存行；若 $s = S$，则每个元素需加载一整个缓存行。该效应将在 6.2.2 节予以演示。

- 另一个值得注意的空间局部性实例涉及 TLB （见章节 1.3.9.2 ）。若程序访问彼此邻近的元素，这些元素很可能位于同一内存页中，通过 TLB 的地址转换将非常迅速。反之，若程序频繁访问分散的元素，则意味着需要访问多个不同的页。由此产生的 TLB 缺失代价极高；另见章节 6.4.4。所幸现代 TLB 容量已大幅提升，实际应用中很少遇到此问题。

习题 1.16. 考虑以下用于对 n 个数 $x[i]$ 求和的算法伪代码（其中 n 为 2 的幂）：

```
for s=2,4,8,...,n/2,n:
for i=0 to n-1 with steps s:
```

```
    x[i] = x[i] + x[i+s/2]
sum = x[0]
```

分析该算法的空间局部性与时间局部性，并与标准算法进行对比

```
sum = 0for i=0,
1,2,...,n-1
sum = sum + x[i]
```

练习 1.17. 考虑以下代码，并假设 `nvectors` 远小于缓存容量，且 `length` 较大。

```
for (k=0; k<nvectors; k++)
  for (i=0; i<length; i++)
    a[k,i] = b[i] * c[k]
```

以下概念如何影响该代码的性能：

- 复用
- Cache size
- Associativity

如果交换循环顺序，以下代码的性能会更好还是更差？原因是什么？

```
for (i=0; i<length; i++)
  for (k=0; k<nvectors; k++)
    a[k,i] = b[i] * c[k]
```

1.6.2.3 局部性示例

让我们通过一个实际案例来考察局部性问题。矩阵乘法 $C \leftarrow A \cdot B$ 可通过多种方式实现。我们比较两种实现方案，假设所有矩阵均按行存储，且缓存容量不足以存储整行或整列数据。

```
for i=1..n
  for j=1..n
    for k=1..n
      c[i,j] += a[i,k]*b[k,j]
```

```
for i=1..n
  for k=1..n
    for j=1..n
      c[i,j] += a[i,k]*b[k,j]
```

这些实现方式如图 1.19 所示。第一种实现构建了 (i, j) 元素

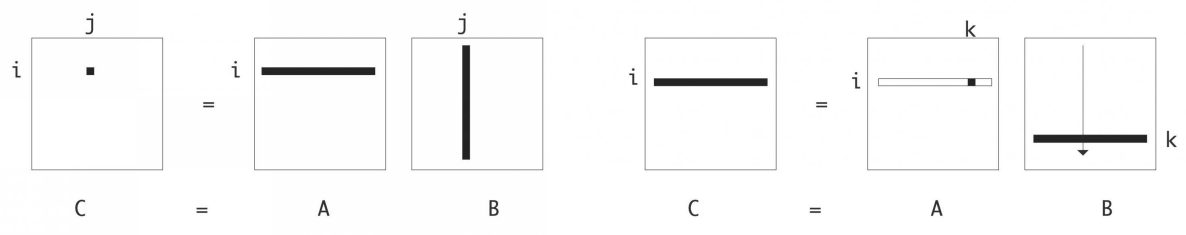


图 1.19: $C \leftarrow A \cdot B$ 矩阵 - 矩阵乘积的两种循环排序方式。

通过 C 与 A 的行和 B 的列的内积计算，在第二种情况下，C 的行通过 B 的行与 A 的元素缩放进行更新。

我们的第一个观察结果是，两种实现确实都计算了 $C \leftarrow C + A \cdot B$，且它们都大约需要 $2n^3$ 次操作。然而，它们的内存行为，包括空间和时间局部性，却大不相同。

`c[i,j]` In the first implementation, `c[i,j]` is invariant in the inner iteration, which constitutes temporal locality, so it can be kept in register. As a result, each element of C will be loaded and stored only once.
在第二种实现中，`c[i,j]` 将在每次内部迭代中被加载和存储。特别是，这意味着现在有 n^3 次存储操作，比第一种实现多出 n 倍。

`a[i,k]` 在两种实现中，`a[i,k]` 的元素都是按行访问的，因此具有良好的空间局部性，因为每个加载的缓存行将被完全使用。在第二种实现中，`a[i,k]` 是不变的 t
在内部循环中，这构成了时间局部性；它可以保持在寄存器中。因此，i n
第二种情况 A 只会被加载一次，而第一种情况则需要加载 n 次。

`b[k,j]` 这两种实现在访问矩阵 B 的方式上有很大差异。首先，`b[k,j]` 不是不变的，因此不会保留在寄存器中，而 B 在两种情况下都会引发 n^3 次内存加载。然而，访问模式不同。在 · 第二种情况下，`b[k,j]` 是按行访问的，因此具有良好的空间局部性：缓存行将会

fully utilized after they are loaded.
In the first implementation, `b[k,j]` is accessed by columns. Because of the row storage of the matrices, a cacheline contains a part of a row, so for each cacheline loaded, only one element is used in the columnwise traversal. This means that the first implementation has more loads for B by a factor of the cacheline length. There may also be TLB effects.

N注意，我们并未对这些实现的代码性能做出任何绝对预测，或
e甚至对它们的运行时间进行相对比较。这类预测非常难以做出。然而，上述
d讨论指出了对广泛经典 CPU 相关的问题。

练习 1.18. 计算乘积 $C \leftarrow A \cdot B$ 的算法不止一种。考虑以下方案：

```
for k=1..n:
  for i=1..n:
    for j=1..n:
      c[i,j] += a[i,k]*b[k,j]
```

分析矩阵 C 的内存流量，并证明其性能逊于前述两种算法。

1.6.2.4 核心局部性

上述空间局部性与时间局部性概念主要体现程序的特性，但硬件特性（如缓存线长度和缓存大小）也会影响局部性分析。第三种与硬件密切相关的局部性类型是**核心局部性**，这在多核多线程程序中尤为重要。

若代码执行期间空间或时间上邻近的写入操作均在同一核心或处理单元上完成，则称其展现出核心局部性。核心局部性不仅取决于程序本身，还在很大程度上与程序的并行执行方式密切相关。

以下因素在此起着关键作用。

- 当两个核心的本地存储中均存在某条缓存线副本时（参见章节 1.4.1），会对缓存一致性产生性能影响。若双方仅执行读取操作则无虞，但若任一方执行写入操作，一致性协议需将该缓存线拷贝至另一核心的本地存储，这将消耗宝贵的内存带宽，应尽量避免。
- 若操作系统（OS）允许线程迁移，会导致缓存内容失效，从而影响核心局部性。关于固定线程亲和性的探讨详见《并行编程》第 *25* 章及《并行程序设计》第 *45* 章。
- 在多插槽系统中，还存在首次接触现象：数据会在其首次初始化的插槽上分配。这意味着从其他插槽访问时可能会显著变慢。详见《并行编程》一书第 *25.2* 节。

1.7 扩展主题

1.7.1 功耗

高性能计算机中另一个重要主题是其功耗问题。在此我们需要区分单个处理器芯片的功耗与整个集群的功耗。

随着芯片上组件数量的增加，其功耗也会相应增长。幸运的是，在反向作用的趋势中，芯片特征的微型化同时降低了所需功耗。假设特征尺寸 λ （可理解为导线厚度）按比例缩小至 $s\lambda$，其中 $s < 1$。为了保持晶体管中的电场恒定，沟道的长度和宽度、氧化物厚度、衬底浓度密度以及工作电压均按相同比例因子进行缩放。

1.7.1.1 缩放特性的推导

恒定场缩放 或 Dennard 缩放 的特性是对电路在微型化过程中特性的理想化描述。一个重要结论是：随着芯片特征尺寸缩小且频率同步提升，功率密度保持恒定。

根据电路理论推导的基本特性是，若我们将特征尺寸按 s 比例缩小：

$$\begin{aligned} &\text{特征尺寸} && \sim s \\ &\text{电压} && \sim s \\ &\text{电流} && \sim s \\ &\text{频率} && \sim s \end{aligned}$$

Then we can derive that

$$\text{Power} = V \cdot I \sim s^2,$$

并且由于电路的总尺寸也随 s^2减小，功率密度保持不变。因此，还可以在电路上放置更多晶体管，而基本上不改变散热问题。

这一结果可视为摩尔定律背后的驱动力，该定律指出处理器中的晶体管数量每 18 个月翻一番。

处理器所需功率中与频率相关的部分来自对电路电容的充放电，因此

$$\begin{array}{ll} \text{电荷} & q = CV \\ \text{Work} & W = qV = CV^2 \\ \text{功率} & W/\text{时间} = WF = CV^2F \end{array} \tag{1.1}$$

该分析可用于论证引入多核处理器的合理性。

Year	# 晶体管数量	处理器
1975	3, 000	6502
1979	30, 000	8088
1985	300, 000	386
1989	1, 000, 000	486
1995	6, 000, 000	Pentium Pro
2000	40, 000, 000	Pentium 4
2005	100, 000, 000	双核 Pentium D
2008	700, 000, 000	8 核 Nehalem
2014	6, 000, 000, 000	18 核 Haswell
2017	20, 000, 000, 000	32 核 AMD Epyc
2019	40,000,000,000	64 核 AMD Rome

Chronology of Moore's Law (courtesy Jukka Suomela, 'Programming Parallel Computers')

1.7.1.2 多核

在撰写本文时（约 2010 年），组件的微型化几乎停滞不前，因为进一步降低电压会导致无法接受的漏电。反之，频率也无法按比例提升。因为这会使得芯片的发热量过高。图 1.20 生动展示了这一现象

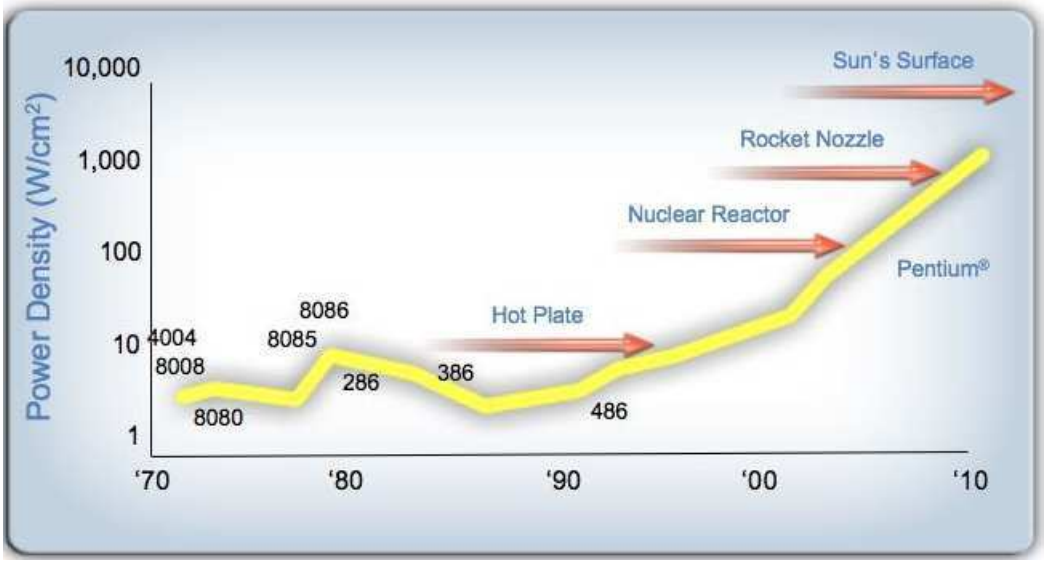


图 1.20：若趋势持续，CPU 的预计散热情况 —— 此图表由 Pat Helsinger 提供。

若单处理器的发展趋势持续，芯片将释放的热量。

一个结论是，计算机设计正面临功耗墙，单核的复杂度已无法进一步提升（例如我们无法再增加 *ILP* 和流水线深度）。

而提升性能的唯一途径就是增加显式可见的并行处理量。这一发展趋势催生了当前这一代多核处理器（参见章节 1.4 ）。这也是为何采用简化处理器设计、因而能耗更低的 GPU 具有吸引力；FPGA 亦是如此。解决功耗墙问题的一种方案是引入多核处理器。回顾公式 1.1，将单个处理器与两个半频处理器进行比较。理论上两者应具有相同的计算能力，对吗？由于降低了频率，若保持相同制程技术，我们还可以降低电压。

理想情况下，两个处理器（核心）的总电功率为

$$\left.\begin{array}{l} C_{\text{multi}} = 2C \\ F_{\text{multi}} = F/2 \\ V_{\text{multi}} = V/2 \end{array}\right\} \Rightarrow P_{\text{multi}} = P/4.$$

实际上电容会略微超过 2 倍增长，而电压无法完全降至 $1/\sqrt{2}$，因此更可能出现 $P_{多} \approx 0.4 \times P$ [30]。当然实际集成过程会稍复杂 [19] ；关键结论是：如今为了降低功耗（或反过来说，在保持功耗不变的前提下进一步提升性能），我们必须开始并行编程。

1.7.1.3 计算机总功耗

并行计算机的总功耗由每颗处理器的功耗和整机处理器数量共同决定。目前这个数值通常达到数兆瓦特。根据上述分析，增加处理器数量带来的功耗增长已无法通过更高能效的处理器来抵消，因此当并行计算机从已实现的千万亿次级别（ 2008 年由 *IBM Roadrunner* 首次达成）向预期的百亿亿次级别迈进时，功耗问题正成为首要考量因素。

在最新几代处理器中，功耗正成为压倒性的考量因素，其影响甚至体现在意想不到的领域。例如，处理器采用单指令多数据（ SIMD ）架构设计（详见章节 2.3.1，特别是 2.3.1.2 小节）就是由指令解码的功耗成本所决定的。

1.7.2 操作系统影响

高性能计算从业者通常不太关心操作系统 |textbf。但有时操作系统的存在会显现出来并影响性能，这源于周期性中断机制 —— 操作系统每秒上百次中断当前进程，以便让其他进程或系统守护进程获得时间片。

I如果你基本上只运行一个程序，就不会希望有这种开销和抖动，以及随之而来的不可预测性
p进程运行时间。因此，历史上存在过一些计算机，它们基本上摒弃了
a操作系统以提升性能。

周期性中断还会带来更多负面影响。例如，它会污染缓存和 *TLB*。作为抖动的细粒度效应，它会降低依赖线程间屏障的代码性能，这在 *OpenMP* 中经常发生（章节 2.6.2 ）。

特别是在金融应用中，对同步性要求极高的场景下，已采用一种 Linux 内核模式，其周期性计时器每秒仅触发一次，而非数百次。这种模式被称为无滴答内核。

1.8 复习题

对于判断题，若选择 ‘错误’ 答案，请给出简要解释。

练习 1.19. 判断正误。代码

```
for (i=0; i<N; i++)
  a[i] = b[i]+1;
```

会访问 `a` 和 `b` 的每个元素一次，因此每个元素都会发生缓存缺失。

练习 1.20. 给出一个代码片段的例子，其中 3 路组相联缓存会发生冲突，但 4 路缓存不会。

练习 1.21. 考虑一个 $N \times N$ 矩阵的矩阵 - 向量乘积。若要使该操作仅发生强制性缓存缺失，所需的缓存大小是多少？你的答案取决于操作的实现方式：分别针对矩阵的行优先遍历和列优先遍历回答，假设矩阵始终按行存储。

第 2 章

并行计算

最大且最强大的计算机有时被称为 ‘超级计算机’。过去二十年间，这一称谓毫无例外地指向并行计算机：即拥有多个 CPU 、可协同处理同一问题的机器。

并行性难以精确定义，因其可体现在多个层面。前一章已说明 CPU 内部如何让多条指令 ‘同时执行’。这被称为指令级并行，它不受用户显式控制：其实现依赖于编译器和 CPU 决定从单一指令流中选取哪些指令进行同步处理。另一种极端情况则是多处理器（通常各自位于独立电路板上）处理多条指令流的并行性。此类并行性通常由用户显式调度。

本章将分析这种更显式的并行类型、支持它的硬件、实现它的编程方法，以及分析它的相关概念。

2.1 引言

在科学计算代码中，通常存在大量需要完成的工作，且这些工作往往具有一定规律性，即对众多数据执行相同的操作。问题在于，是否可以通过并行计算机来加速这一过程。假设有 n 项操作需完成，在单处理器上耗时 t，那么能否在 p 个处理器上将时间缩短至 t/p？

让我们从一个非常简单的例子开始。对长度为 n 的两个向量进行加法运算

```
for (i=0; i<n; i++)
  a[i] = b[i] + c[i];
```

最多可使用 n 个处理器完成。在理想情况下，若拥有 n 个处理器，每个处理器将持有本地标量 `a`、`b`、`c` 并执行单条指令 `a=b+c`。如图 2.1 所示。

在一般情况下，当每个处理器执行类似以下操作时

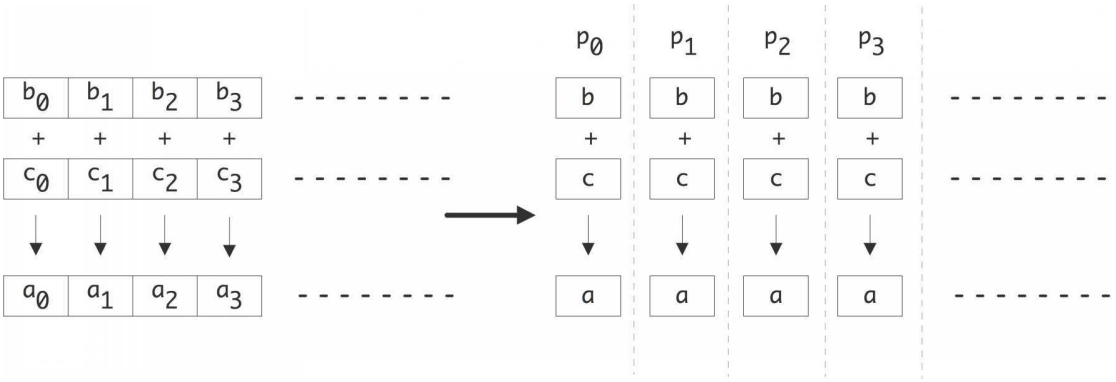


图 2.1：向量加法的并行化示例。

```
for (i=my_low; i<my_high; i++)
  a[i] = b[i] + c[i];
```

e执行时间随处理器数量线性减少。若每个操作耗时一个单位时间，
原始算法耗时为 n，而在 p 个处理器上并行执行耗时 n/p。并行算法速度提升了 p[1] 倍。

接下来，我们考虑对向量元素求和。（这种输入为向量但输出仅为标量的操作通常称为归约。）
再次假设每个处理器仅存储单个数组元素。串行代码如下：

```
s = 0;
for (i=0; i<n; i++)
  s += x[i]
```

不再明显具有并行性，但如果我们将循环重写为

```
for (s=2; s<2*n; s*=2)
  for (i=0; i<n-s/2; i+=s)
    x[i] += x[i+s/2]
```

存在一种并行化的方法：外层循环的每次迭代现在都可以由 n/s 个处理器并行执行。由于外层循环将经历 $\log_2 n$ 次迭代，可见新算法的运行时间缩短为 $n/p \cdot \log_2 n$。并行算法现在提速了 $p/\log_2 n$ 倍。如图 2.2 所示。

即使从这两个简单例子中，我们也能看出并行计算的一些特征：

- 有时需要略微重写算法才能实现并行化。
- 并行算法可能无法展现完美的加速效果。

1. 此处我们忽略当 p 无法被 n 整除时产生的低阶误差。通常我们也会忽略循环开销的问题。

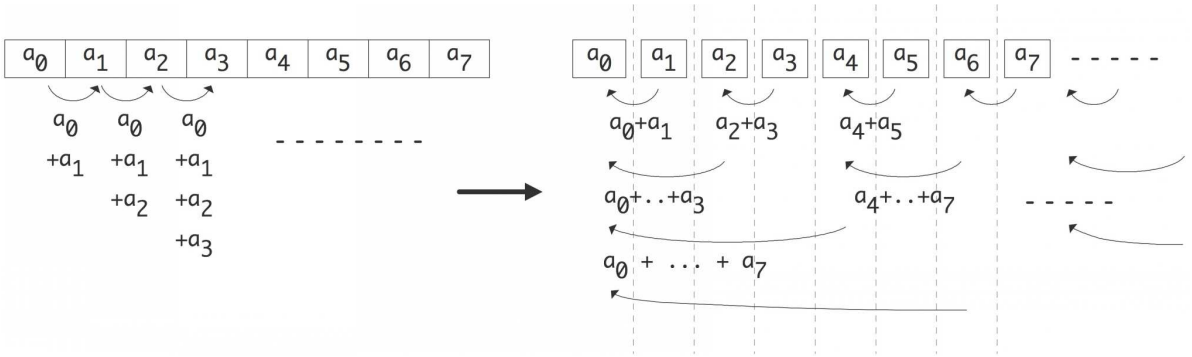


Figure 2.2: Parallelization of a vector reduction.

还有其他需要注意的事项。第一种情况下，如果每个处理器都将其 x_i、y_i 存储在本地存储器中，算法可以无额外复杂性地执行。第二种情况下，处理器需要相互通信数据，而我们尚未为此分配成本。

首先让我们系统地审视通信问题。我们可以将图 2.2 右半部分的并行算法转化为树状图（参见附录 20），具体方法是将输入定义为叶节点，所有部分和定义为内部节点，根节点作为总和。若一个节点是另一个节点（部分）求和的输入，则两者之间存在边连接。这一过程如图 2.3 所示。图中，水平对齐的节点代表可同步执行的计算步骤；每一层级有时被称为计算中的超步。垂直对齐的节点表示它们在同一处理器上计算，而箭头则表示处理器间的通信。图 2.3 中的垂直排列方式并非唯一可能，若在超步内重新调整节点位置

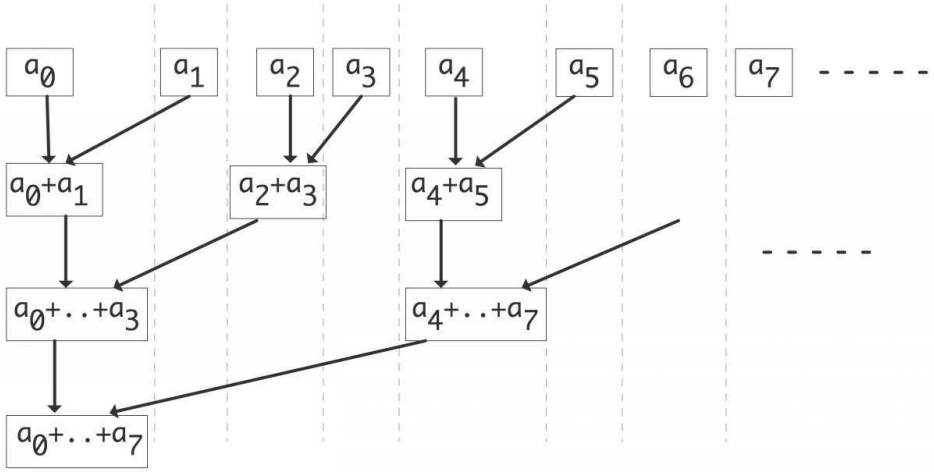


图 2.3：并行向量归约的通信结构。

o在水平层级 r 上，采用不同的通信模式 rises.

练习 2.1. 考虑将超步中的节点随机分配到处理器上。证明，

若无两个节点最终落在同一处理器上，其通信次数最多为图 2.3 所示情形的两倍。

习题 2.2. 能否绘制一个计算图，使得求和结果保留在每个处理器上？存在两种解决方案：一种需要两倍数量的超步，另一种则保持相同步数。这两种情况下的计算图不再是树结构，而是更通用的有向无环图（DAG）。

处理器通常通过网络连接，数据在网络上传输需要时间。这引入了处理器间距离的概念。在图 2.3 中，处理器呈线性排列，距离与其排序中的秩相关。若网络仅连接相邻处理器，外层循环的每次迭代都会增加通信发生的距离。

习题 2.3. 假设加法运算消耗单位时间，且数据在处理器间传输也消耗相同单位时间。证明通信时间等于计算时间。现假设从处理器 p 向 $p \pm k$ 发送数据耗时 k。证明此时并行算法的执行时间与串行算法同阶。

求和示例做出了不切实际的假设，即每个处理器最初仅存储一个向量元素：实际上我们将拥有 $p < n$，且每个处理器存储多个向量元素。显而易见的策略是让每个处理器处理连续的一段元素，但有时显而易见的策略并非最佳选择。

练习 2.4. 考虑用 4 个处理器对 8 个元素求和的情况。证明图 2.3 中某些边不再对应实际通信。现在再次考虑用 4 个处理器对 16 个元素求和。这次通信边的数量是多少？

这些关于算法适配性、效率及通信的问题，对并行计算至关重要。我们将在本章以不同形式反复讨论这些问题。

2.1.1 功能并行与数据并行

通过上述介绍，我们可以将并行性描述为在程序执行过程中寻找独立操作。所有示例中这些独立操作实际上是相同的操作，但应用于不同的数据项。我们可称之为数据并行：相同的操作并行应用于多个数据元素。这实际上是科学计算中的常见场景：并行性通常源于数据集（向量、矩阵、图等）分布在多个处理器上，每个处理器处理其部分数据。

术语数据并行性传统上多用于描述单一指令操作；在子程序情况下则常被称为任务并行性。

也可能发现不基于数据元素、而基于指令本身的独立性。传统上，编译器会从 ILP（指令级并行）角度分析代码：例如可将独立指令分配给不同浮点单元或重新排序，以优化寄存器使用（另见章节 2.5.2）。

ILP 是功能并行性的一种情况；在更高层次上，功能并行性可以通过考虑独立的子程序来实现，这通常被称为任务并行性；参见章节 2.5.3。

功能并行性的一些例子包括蒙特卡洛模拟，以及遍历参数化搜索空间的其他算法，例如布尔可满足性问题。

2.1.2 算法中的并行性与代码中的并行性

我们常常会遇到这样的情况：想要将一个以顺序形式表达的常见算法并行化。在某些情况下，这种顺序形式很容易并行化，比如上文讨论的向量加法。而在其他情况下，算法并没有简单的并行化方法；我们将在 7.10.2 节讨论线性递归问题。还有一种情况是，顺序代码看似无法并行，但算法本身其实具有并行性。

练习 2.5.f

```
or 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]
```

回答以下关于双重 `i,j` 循环的问题：
1. 内层循环的迭代是否独立，即是否可以并行执行？ 2. 外层循环的迭代是否独立？ 3. 如果已知 `x[1,1]` ，证明 `x[2,1]` 和 `x[1,2]` 可被独立计算。4. 这是否为你提供了并行化策略的思路？

W我们将在第 7.10.1 节讨论这一难题的解决方案。一般而言，整个第 7 章
b关于科学计算算法中固有并行量的探讨。

2.2 理论概念

使用并行计算机有两个重要原因：获取更大内存或更高性能。内存增益容易量化，因为总内存即各独立内存之和。而并行计算机的速度则较难表征。本节将深入探讨用于表达和评判从串行架构转向并行架构所获执行速度增益的理论度量方法。

2.2.1 定义

2.2.1.1 加速比与效率

定义加速比的一种简单方法是让同一程序在单处理器和拥有 p 个处理器的并行机器上运行，并比较运行时间。设 T_1 为单处理器上的执行时间，T_p 为 p 个处理器上的时间，我们将加速比 定义为 $S_p = T_1/T_p$。（有时 T_1 被定义为 ‘在单处理器上解决问题的最佳时间’，这允许在单处理器上使用与并行不同的算法。）在理想情况下，$T_p = T_1/p$，但实际上我们并不期望达到这一点，因此 $S_p \leq p$。为了衡量我们与理想加速比的差距，我们引入效率 $E_p = S_p/p$。显然，$0 < E_p \leq 1$。

练习 2.6. 证明 $E = 1$ 意味着所有处理器始终处于活跃状态。（提示：假设所有处理器在时间 T 内完成工作，除了一个处理器在 $T' < T$ 内完成。此时 T_p 是多少？探究上述关系。）

上述定义存在一个实际问题：能在并行机器上求解的问题可能规模过大，无法适配任何单处理器。反之，将单处理器问题分布到多个处理器上可能导致失真，因为每个处理器上分配到的数据量极少。下文我们将讨论更现实的加速比衡量标准。

实际速度低于 p 的原因有多种。其一，使用多个处理器必然涉及通信与同步，这些额外开销是原始计算中不存在的。其二，若各处理器负载不均（称为负载不平衡），部分处理器可能出现空闲，从而再次降低实际达到的加速比。最后，代码可能包含固有串行部分。

处理器间通信是效率损失的重要来源。显然，无需通信即可求解的问题效率极高。这类问题实质上由若干完全独立的计算组成，被称为令人尴尬的并行（或便利并行；参见 2.5.4 节），其加速比和效率近乎完美。

练习 2.7. 当加速比超过处理器数量时，这种情况称为超线性加速。请从理论上论证为何这种情况不可能发生。

实际上，超线性加速是有可能发生的。例如，假设一个问题规模过大无法全部载入内存，单处理器只能通过将数据交换到磁盘来解决。若该问题恰好能装入两个处理器的内存中，由于不再发生磁盘交换，加速比很可能超过 2。数据量减少或数据局部性增强，也可能改善代码的缓存性能。

超线性加速也可能出现在搜索算法中。设想每个处理器从搜索空间的不同位置开始工作：此时可能出现处理器 3 立即找到解的情况。而在顺序执行时，必须遍历处理器 1 和 2 的所有可能性。这种情况下加速比将远超过 3。

2.2.1.2 成本最优性

在加速效果不理想的情况下，我们可以将开销定义为两者之间的差值

$$T_o = pT_p - T_1.$$

我们也可以将其解释为：在单处理器上模拟并行算法与实际最佳串行算法之间的差异。

后续我们将看到两种不同类型的开销：

1. 并行算法可能与串行算法存在本质差异。例如，排序算法的复杂度通常为 $O(n \log n)$，但并行双调排序（章节 9.6 ）的复杂度则是 $O(n \log^2 n)$。2. 并行算法可能产生源自并行化过程的开销，例如消息传递成本。举例来说，章节 7.2.3 分析了矩阵 - 向量乘积中的通信开销。

若某并行算法的开销量级不超过对应串行算法的运行时间，则称该算法为成本最优。

习题 2.8. 上述开销定义隐含假设开销不可并行化。请结合前文两个示例讨论该假设。

2.2.2 渐近分析

若忽略处理器数量必须有限或处理器间互连物理限制等约束条件，我们可以推导出并行计算极限的理论结果。本节将简要介绍这类成果，并探讨其与现实高性能计算的联系。

以矩阵乘法 $C = AB$ 为例，该运算需要 $2N^3$ 次操作（ N 为矩阵维度）。由于 C 元素间的运算不存在依赖关系，所有操作均可并行执行。假设拥有 N^2 个处理器，可为每个处理器分配 C 中的 (i, j) 坐标，使其在 $2N$ 时间内计算 c_{ij}。因此该并行运算效率为 1，达到最优值。

练习 2.9. 证明该算法忽略了内存使用方面的若干关键问题：

- 若矩阵存储于共享内存中，每个内存位置会同时发生多少次读取操作?
- 如果处理器将本地计算的输入和输出保留在本地存储中，矩阵元素会有多少重复?

在 $_2N$ 对数时间内，使用 $N/2$ 个处理器可以完成 N 个数 $\{x_i\}_{i=1\ldots N}$ 的加法运算。若有 $N/2$ 个处理器，则可计算：

1. 定义 $s_i^{(0)} = x_i$。2. 使用 $j = 1$ 、 ... 进行迭代，

对数 $_2N$：3. 计算 $N/2^j$ 个部分和

$s_i^{(j)} = s_{2i}^{(j-1)} + s_{2i+1}^{(j-1)}$

W可见 $N/2$ 个处理器总共执行 N 次操作（理应如此），耗时 $_2N$ 对数时间。
e该并行方案的效率为 $O(1/\log_2 N)$，是 N 的缓慢递减函数。
习题 2.10. 证明采用上述并行加法方案，可在 $_2N$ 对数时间内用 $N^3/2$ 个处理器完成两个矩阵的乘法。最终效率是多少?
现在提出这个问题已是一个合理的理论探讨

- 若我们拥有无限数量的处理器，矩阵乘法的理论最低时间复杂度是多少？或者说，
- 是否存在仍保持 $O(1)$ 效率的更快算法?

此类问题已有研究（例如参见 [98] ），但它们对高性能计算的影响甚微 -
高性能计算。

针对这类理论界限的首要质疑在于，它们隐含假设了某种形式的共享内存。实际上，这些算法的形式化模型被称为并行随机存取机（ *PRAM* ），其假设前提是每个内存位置均可被任意处理器访问。

通常还会附加一个假设：对同一存储位置的多重并发访问实际上是可行的。由于写入和读取操作在实践中的行为模式不同，因此衍生出了 CREW-PRAM （并发读独占写 PRAM ）的概念。

PRAM 模型的基本假设在实践中并不现实，尤其是在问题规模和处理器数量扩展的背景下。对 PRAM 模型的另一个质疑是，即使在单处理器上它也忽略了内存层次结构；详见章节 1.3。

但即便考虑分布式内存，理论结果仍可能脱离实际。上述求和算法确实可以在分布式内存中不加修改地运行，只是我们必须关注活跃处理器间的距离会随着迭代深入而增加。若处理器通过线性阵列连接，活跃处理器间的 ' 跳数 ' 将翻倍，随之而来的是迭代计算时间的渐近增长。总执行时间因此变为 $n/2$—— 考虑到我们投入了如此多的处理器来解决该问题，这个结果令人失望。

如果处理器采用超立方体拓扑连接（章节 2.7.5 ）会怎样？不难看出，求和算法此时确实可以在 $\log_2 n$ 时间内完成。然而，随着 $n \to \infty$，我们能否物理构造一个由 n 个节点组成的超立方体序列，并保持两个连接节点间的通信时间恒定？由于通信时间取决于延迟，而延迟部分取决于导线长度，我们必须关注相邻节点间的物理距离。

这里的关键问题在于，超立方体（一个 n 维对象）能否嵌入三维空间，同时保持相连邻居间的距离（以米为单位）恒定。显然，三维网格可以任意扩展而保持单位导线长度，但对于超立方体这一问题尚不明确。在超立方体中，导线长度可能需随 n 增长而增加，这与电子有限速度的特性相冲突。

我们概述一个证明（详见 [63] 获取更多细节），在我们这个三维世界且光速有限的情况下，加速比被限制在 $\sqrt[3]{n}$ 对于运行于 n 个处理器上的问题，无论互连方式如何。论证如下：考虑一个需要在一个处理器上收集最终结果的操作。假设每个处理器占据单位空间体积，每单位时间产生一个结果，并能每单位时间发送一个数据项。那么，在时间 t 内，最多只有半径为 t 的球体内的处理器，即 $O(t^3)$ 个处理器总数，能对最终结果作出贡献；其余处理器因距离过远无法参与。在时间 T 内，能对最终结果产生贡献的操作数量为 $\int_0^T t^3 dt = O(T^4)$。这意味着最大可实现的加速比是顺序时间的四次方根。

最后，‘ 如果我们拥有无限多处理器会怎样 ’ 这一问题本身并不现实，但我们将以允许的方式提出弱扩展问题（第 2.2.5 节） ‘ 如果我们让问题规模与

处理器数量彼此成比例增长’。这个问题是合理的，因为它对应着一个非常实际的考量：购买更多处理器是否能让人们运行更大的问题，如果是的话，能带来怎样的 ‘性价比’。

2.2.3 阿姆达尔定律

加速效果不完美的原因之一是代码的某些部分本质上是顺序执行的。这从以下方面限制了并行效率。假设 5% 的代码是顺序执行的，那么无论有多少处理器可用，这部分的时间都无法减少。因此，该代码的加速比被限制在 20 倍以内。这种现象被称为阿姆达尔定律 [4],，我们接下来将对此进行阐述。

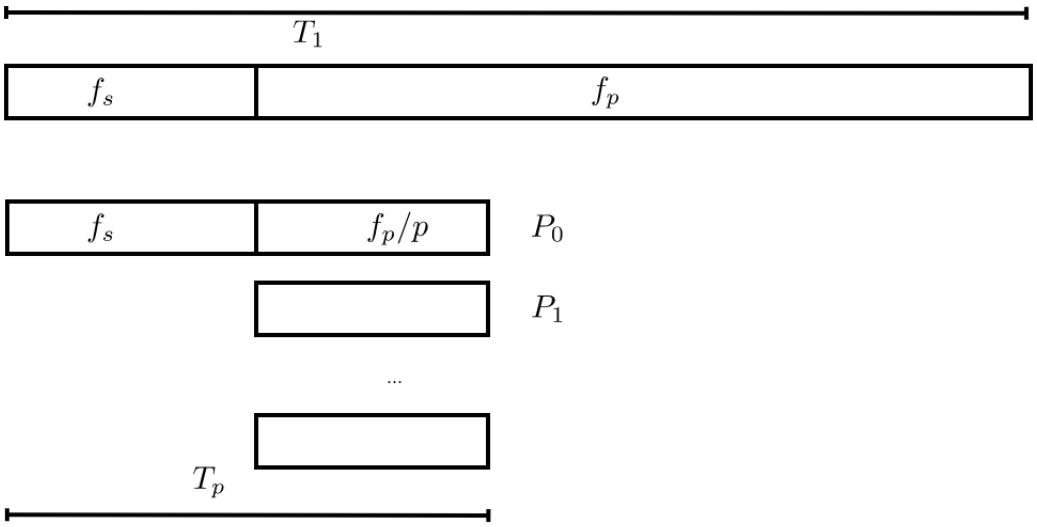


图 2.4：阿姆达尔分析中的顺序执行时间与并行执行时间。

设 F_s 为顺序执行部分的比例，F_p 为并行执行部分的比例（或更严格地说：‘可并行化’ 部分的比例）。那么 $F_p + F_s = 1$。在 p 个处理器上的并行执行时间 T_p 是顺序部分 T_1F_s 与可并行化部分 T_1F_p/P 之和：

$$T_P = T_1(F_s + F_p/P). \tag{2.1}$$

(参见图 2.4) 随着处理器数量增加 $P \to \infty$，并行执行时间逐渐趋近于代码中顺序执行部分的时间：$T_P \downarrow T_1F_s$。我们得出结论：加速比受限于 $S_P \leq 1/F_s$，而效率则是关于 $E \sim 1/P$ 的递减函数。

代码的顺序执行部分可能包含诸如 I/O 操作等内容。然而，也存在某些代码部分实际上起到顺序执行的作用。考虑一个执行单循环的程序，其中所有迭代均可独立计算。显然，这类代码不存在并行化的障碍。但若将循环拆分为若干部分（每个处理器处理一部分），则每个处理器现在都需要处理循环开销：边界计算和完成条件判断。该开销会随着处理器数量成倍复制。实际上，循环开销相当于代码中的顺序执行部分。

习题 2.11. 我们来看一个具体例子。假设某段代码的初始化需要 1 秒，可并行部分在单处理器上运行需要 1000 秒。若使用 100 个处理器执行，其加速比和效率是多少？若使用 500 个处理器呢？答案请保留最多两位有效数字。

习题 2.12. 探究阿姆达尔定律的隐含意义：当处理器数量 P 增加时，代码的并行部分需要如何增长才能维持固定效率？

2.2.3.1 考虑通信开销的阿姆达尔定律

从某种意义上说，尽管阿姆达尔定律发人深省，但它甚至还是乐观的。并行化代码会带来一定的加速，但同时也引入了通信开销，这会降低实际获得的加速比。让我们来完善公式 (2.1) 的模型（参见 [130, 第 367] 页）：

$$T_p = T_1(F_s + F_p/P) + T_c,$$

其中 T_c 为固定通信时间。

为评估此通信开销的影响，我们假设代码完全可并行化，即 $F_p = 1$。随后可得出

$$S_p = \frac{T_1}{T_1/p + T_c}. \tag{2.2}$$

要使该值接近 p，需满足 $T_c \ll T_1/p$ 或 $p \ll T_1/T_c$。换言之，处理器数量不应超过标量执行时间与通信开销之比。

2.2.3.2 古斯塔夫森定律

阿姆达尔定律曾被认为证明了大规模处理器集群永远无法带来效益。然而，该定律隐含的假设是：存在一个固定计算任务被分配到越来越多的处理器上执行。实际情况并非如此：通常存在某种扩展问题规模的方法（第 4 章将介绍 ‘离散化’ 概念），人们会根据可用处理器数量调整问题规模。

更现实的假设是：存在与问题规模无关的串行部分，以及可无限复制的并行部分。为形式化表述，我们不再从串行程序的执行时间出发，而是从并行程序的执行时间出发，设

$$T_p = T(F_s + F_p) \qquad \text{with } F_s + F_p = 1.$$

现在我们有两种可能的定义 T_1。首先，存在从 T_1 通过设置 $p = 1$ 于 T_p 中得到的定义。（请自行验证这与 T_p 实际上是相同的。）然而，我们需要的是 T_1，用于描述并行程序所有操作所需的时间。（参见图 2.5。）因此，我们从一个观测到的 T_p 出发，并重构 T_1 为：

$$T_1 = F_s T_p + p \cdot F_p T_p.$$

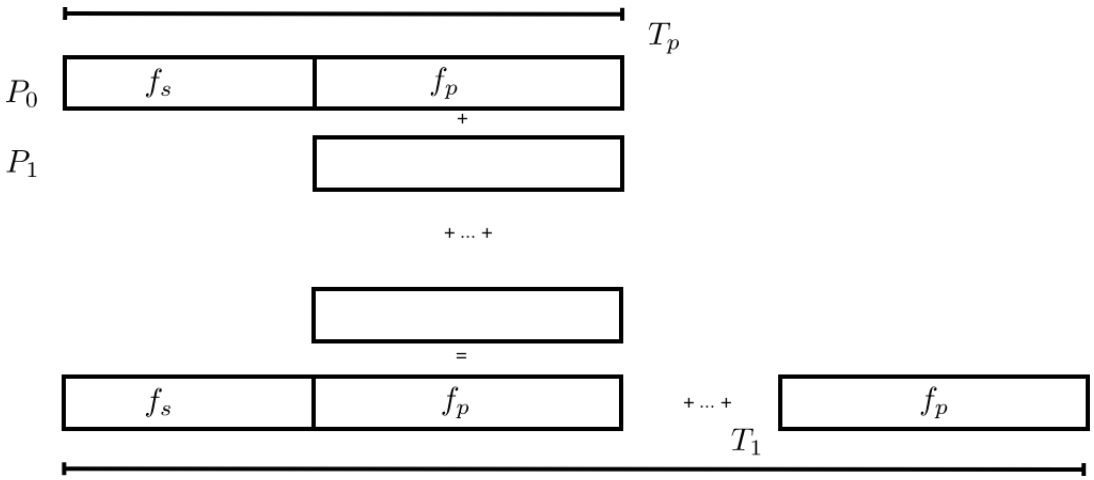


图 2.5：Gustafson 分析中的串行与并行时间。

由此我们得到加速比为

$$S_p = \frac{T_1}{T_p} = \frac{F_s + p \cdot F_p}{F_s + F_p} = F_s + p \cdot F_p = p - (p-1) \cdot F_s. \tag{2.3}$$

从该公式可得出以下结论：

- 加速比仍受 p 限制；
- … 但它是一个正数；
- 对于给定的 p，• 再次成为串行部分比例的递减函数。

习题 2.13. 重写方程 (2.3) 以用和表示加速比。效率的渐近行为是什么？

与阿姆达尔定律类似，若考虑通信开销，我们可以研究古斯塔夫森定律的行为。回到完美可并行问题的方程 (2.2)，可近似表示为

$$S_p = p(1 - \frac{T_c}{T_1} p).$$

现在，在问题规模逐渐扩大的假设下，和成为的函数。可见若，我们将获得与 1 保持恒定比例差距的线性加速比。作为一般性讨论，我们无法进一步推进此分析；在 7.2.3 节您将看到一个详细的案例分析。

2.2.3.3 阿姆达尔定律与混合编程

前文您已了解混合编程 —— 分布式与共享内存编程的混合模式。这催生了阿姆达尔定律的新形式。

假设我们有 p 个节点，每个节点包含 c 个核心，且 F_p 描述了代码中采用 c 路线程并行化的部分比例。我们假定整个代码在 p 个节点间完全并行。理想加速比

理想情况下并行运行时间应为 pc，理想并行运行时间为 $T_1/(pc)$，但实际运行时间是

$$T_{p,c} = T_1\left(\frac{F_s}{p} + \frac{F_p}{pc}\right) = \frac{T_1}{pc}\left(F_s c + F_p\right) = \frac{T_1}{pc}\left(1 + F_s(c-1)\right).$$

习题 2.14. 证明加速比 $T_1/T_{p,c}$ 可近似表示为 p/F_s。
在原始阿姆达尔定律中，加速比受限于串行部分固定值 $1/F_s$；而在混合编程中，加速比受限于任务并行部分 p/F_s。

2.2.4 关键路径与布伦特定理

上述加速比和效率的定义，以及阿姆达尔定律与古斯塔夫森定律的讨论，都隐含了一个假设：并行工作可以无限细分。正如你在 2.1 节求和示例中所见，实际情况并非总是如此 —— 操作间可能存在依赖关系，即某个操作需要前驱操作的结果作为输入。存在依赖关系的操作无法并行执行，因此会限制可实现的并行程度。

我们将关键路径定义为一条（可能不唯一）具有最大长度的依赖链。（该长度有时被称为跨度。）由于关键路径上的任务需要依次执行，其长度是并行执行时间的下限。

为了使这些概念更加精确，我们定义以下术语：

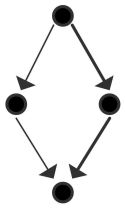
定义 1

T_1 ： 计算在单处理器上消耗的时间 T_p ： 计算在 p 个处理器上消耗的时间 T_∞： 计算在无限处理器可用时消耗的时间 P_∞： 当 $T_p = T_\infty$ 时 p 的值

基于这些概念，我们可以将算法的平均并行度定义为 T_1/T_∞，而关键路径的长度为 T_∞。

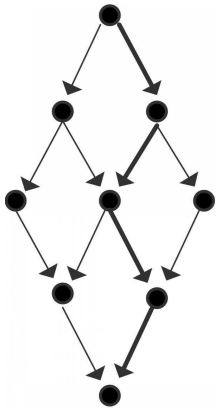
习题 2.15. 假设你有一个算法可以在 $E_p \equiv 1$ 条件下执行。这对关键路径分析意味着什么？

我们将通过展示任务及其依赖关系的图示来提供几个示例。假设
s默认情况下每个节点都是单位时间任务。



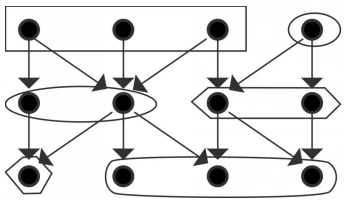
可使用的最大处理器数量为 2，平均并行度为 4/3：

$$T_1 = 4, \quad T_\infty = 3 \quad \Rightarrow T_1/T_\infty = 4/3$$
$$T_2 = 3, \quad S_2 = 4/3, \quad E_2 = 2/3$$
$$P_\infty = 2$$



可使用的处理器最大数量为 3，平均并行度为 $9/5$；效率在 $p = 2$ 时达到最大值：

$$
\begin{aligned}
&T_1 = 9, \quad T_\infty = 5 \quad \Rightarrow T_1/T_\infty = 9/5\\
&T_2 = 6, \quad S_2 = 3/2, \quad E_2 = 3/4\\
&T_3 = 5, \quad S_3 = 9/5, \quad E_3 = 3/5\\
&P_\infty = 3
\end{aligned}
$$



可使用的处理器最大数量为 4，且该数值也是平均并行度；图示展示了采用 $P = 3$ 且效率为 $\equiv 1$ 的并行化方案：

$$
\begin{aligned}
&T_1 = 12, \quad T_\infty = 4 \quad \Rightarrow T_1/T_\infty = 3\\
&T_2 = 6, \quad S_2 = 2, \quad E_2 = 1\\
&T_3 = 4, \quad S_3 = 3, \quad E_3 = 1\\
&T_4 = 3, \quad S_4 = 4, \quad E_4 = 1\\
&P_\infty = 4
\end{aligned}
$$

基于这些示例，您可能已经发现存在两种极端情况：

- 若每个任务严格依赖于另一个任务，则会形成依赖链，且 $T_p = T_1$ 对于任何 p。
- 另一方面，如果所有任务都是独立的（且 p 能整除其数量），你会得到 $T_p = T_1/p$ 对于任何 p。
- 在一个比前例稍复杂的场景中，假设关键路径长度为 m，且在这 m 个步骤的每一步中都有 $p - 1$ 个独立任务（或至少：仅依赖于前一步骤的任务）。那么在这 m 个步骤中每一步都将实现完美并行，我们可以表达 $T_p = T_1/p$ 或 $T_p = m + (T_1 - m)/p$。实际上该陈述普That l遍成立。这被称为布伦特定理：

定理 1 设 m 为任务总数，p 为处理器数量，t 为关键路径的长度。则该计算可在

$$T_p \leq t + \frac{m - t}{p}.$$

证明。将计算划分为若干步骤，使得步骤 $i + 1$ 中的任务彼此独立，且仅依赖于步骤 i。设 s_i 为步骤 i 中的任务数量，则该步骤所需时间为 $\lceil \frac{s_i}{p} \rceil$。对 i 求和可得

$$T_p = \sum_i^t \lceil \frac{s_i}{p} \rceil \leq \sum_i^t \frac{s_i + p - 1}{p} = t + \sum_i^t \frac{s_i - 1}{p} = t + \frac{m - t}{p}.$$

练习 2.16. 考虑一棵深度为 d 的树，即包含 $2^d - 1$ 个节点，并进行搜索

$$\text{最大 } n\in \text{ 个节点 } f(n).$$

假设所有节点都需要被访问：我们对其值没有任何先验知识或排序信息。

分析在 p 个处理器上的并行运行时间，其中可假设 $p = 2^q$，
当 $q < d$ 时。这与 Brent 定理和 Amdahl 定律得出的数值有何关联?

2.2.5 可扩展性

前文提到，将给定问题拆分到越来越多的处理器上并无意义：当超过某个临界点时，每个处理器将因工作量不足而无法高效运作。实践中，并行代码的使用者通常会选择与问题规模相匹配的处理器数量，或通过逐步扩大问题规模并相应增加处理器数量来求解。这两种情况都难以用加速比来衡量，取而代之的是采用可扩展性这一概念。

我们区分两种可扩展性类型。所谓的强可扩展性本质上等同于前文讨论的加速比。当一个问题在越来越多的处理器上分区时，若展现出完美或接近完美的加速比（即执行时间随处理器数量线性下降），我们称该问题具有强可扩展性。用效率术语可表述为：

$$\left.\begin{array}{l} N \equiv \text{constant} \\ P \to \infty \end{array}\right\} \Rightarrow E_P \approx \text{constant}$$

通常我们会看到诸如 ‘该问题可扩展至 500 个处理器’ 的表述，意指在不超过 500 个处理器时，加速比不会显著偏离最优值。此类问题无需能在单处理器上运行：常以较小基数（如 64 个处理器）作为评估可扩展性的基准线。

Exercise2.17. 我们可以将强扩展性表述为运行时间与处理器数量成反比：

$$t = c/p.$$

证明在对数 - 对数坐标图中（即绘制运行时间的对数相对于处理器数量的对数），
你将得到一条斜率为 -1 的直线。对于不可并行化的部分（即运行时间为
$t = c_1 + c_2/p$ 的情况），你能提出一种处理方法吗?

更有趣的是，*weakscalability* 描述了当问题规模和处理器数量同时增长时，但保持每个处理器的工作量恒定的执行行为。这里的 ‘工作量’ 一词含义模糊：有时弱扩展被解释为保持数据量不变，而在其他情况下则是指操作数量保持不变。

诸如加速比之类的度量指标较难准确报告，因为操作数量与数据量之间的关系可能很复杂。如果这种关系是线性的，则可以说明

每个处理器的数据量保持恒定，并报告称随着处理器数量增加，并行执行时间保持恒定。（你能想到工作与数据之间呈线性关系的应用场景吗？哪些情况下不是线性关系？）

从效率角度而言：

$$\left.\begin{array}{l} N \to \infty \\ P \to \infty \\ M = N/P \equiv \text{constant} \end{array}\right\} \Rightarrow E_P \approx \text{constant}$$

练习 2.18. 假设你正在研究某代码的弱可扩展性。在运行了几个不同规模及对应进程数的测试后，你发现每种情况下的浮点运算速率大致相同。请论证该代码确实具有弱可扩展性。

练习 2.19. 上述讨论中我们始终隐式对比同一算法的串行版本与并行版本。但在 2.2.1 节中曾指出，有时加速比被定义为并行算法与解决同一问题的**最优**串行算法之间的比较。基于此，请将运行时间为 $(\log n)^2$ 的并行排序算法（例如双调排序，见 9.6 节）与最优串行算法（其运行时间为 $n \log n$）进行对比。

> 证明在弱扩展情况下 $n = p$ 加速比为 $p/\log p$。证明在强扩展情况下加速比是 n 的递减函数。

Remark6 一则历史轶事。

```
Message:  1023110, 88 linesPosted:  5:34pm EST,
 Mon Nov 25/85,Subject:  Challenge from Alan Karp
To:  Numerical-Analysis, ...  From GOLUB@SU-SCORE.ARPA

I have just returned from the Second SIAM Conference on
Parallel Processing for Scientific Computing in Norfolk、
Virginia.  There I heard about 1、 000 processor systems、 4、
000 processor systems、  and even a proposed 1,000,000
processor system.  Since I wonder if such systems are the
best way to do general purpose、  scientific computing、  I am
making the following offer.

I will pay $100 to the first person to demonstrate a speedup
of at least 200 on a general purpose, MIMD computer used for
scientific computing.  This offer will be withdrawn at 11:59
PM on 31 December 1995.
```

通过扩大问题规模满足了这一要求。

2.2.5.1 等效率

在弱可扩展性的定义中，我们曾指出，在问题规模 N 与处理器数量 p 之间的某种关系下，效率将保持恒定。我们可以对此进行精确表述，并将等效率曲线定义为 N 与 p 之间能保持恒定效率 [83] 的关系。

2.2.5.2 Precisely what do you mean by scalable?

在科学计算中，可扩展性是算法及其在特定架构上并行化方式的属性，尤其关注数据分布的方式。

在计算机行业术语中，“ 可扩展性 ” 一词有时用于描述架构或整个计算机系统：

> 可扩展计算机是指由少量基础组件设计而成、不存在单一瓶颈组件的计算机，使其能在设计扩展范围内逐步扩充，为一组明确定义的可扩展应用提供线性递增的性能。通用可扩展计算机提供广泛的处理能力、内存容量和 I/O 资源。可扩展性是指可扩展计算机性能提升的线性程度 ” [12]。

具体而言，

- 水平扩展 是指增加更多硬件组件，例如向集群添加节点；
- 垂直扩展 则对应使用更强大的硬件，例如通过硬件升级实现。

2.2.6 模拟扩展

在大多数关于弱扩展的讨论中，我们假设工作量与存储量呈线性关系。但情况并非总是如此；例如矩阵 - 矩阵乘法的运算复杂度为 N^3，而数据量为 N^2。若线性增加处理器数量并保持每个进程的数据量不变，工作量可能以更高幂次增长。

类似效应也出现在时间相关偏微分方程（ PDE ）的模拟中（此概念引自第 4 章）。此时总工作量是单时间步工作量与时间步数的乘积。这两个数值相互关联；在 4.1.2 节中可见，时间步长存在与空间离散化相关的最小尺寸限制。因此，随着单时间步工作量的增加，时间步数也会相应增加。

现在考虑诸如天气预报这类应用场景，假设当前需要 4 小时计算时间来预测未来 24 小时的天气。问题在于：若你对当前计算效率满意，并购买更强大的计算机以获得更精确的预测，你能如何描述这台新计算机的性能?

换言之，本节我们将不再从算法执行的角度探讨可扩展性，而是研究模拟时间 S 与运行时间 T 保持恒定的情况。由于新计算机速度更快，我们能在相同运行时间内执行更多运算。但尚不明确涉及的数据量及所需内存将如何变化。为此需结合应用数学知识进行分析。

设 m 为每个处理器的内存， P 为处理器数量，可得：

$$M = Pm \qquad \text{total memory.}$$

若 d 表示问题的空间维度数（通常为 2 或 3 ），则有

$$\Delta x = 1/M^{1/d} \qquad \text{grid spacing.}$$

为保证稳定性，时间步长 Δt 被限制为

$$\Delta t = \begin{cases} \Delta x = 1 \,/\, M^{1/d} & \text{hyperbolic case} \\ \Delta x^2 = 1 \,/\, M^{2/d} & \text{parabolic case} \end{cases}$$

（注意双曲情形未在第 4 章讨论。）给定模拟时间 S，可得

$$k = S/\Delta t \qquad \text{time steps.}$$

若假设各时间步长完全可并行化（即采用显式方法，或使用最优求解器的隐式方法），则运行时间为

$$T = kM/P = \frac{S}{\Delta t} m.$$

设置 $T/S = C$，我们发现

$$m = C\Delta t,$$

即随着处理器数量的增加，每个处理器的内存容量会下降。（最后一句话中缺失的步骤是什么？）

进一步分析这一结果，我们发现

$$m = C\Delta t = c \begin{cases} 1 \,/\, M^{1/d} & \text{hyperbolic case} \\ 1 \,/\, M^{2/d} & \text{parabolic case} \end{cases}$$

代入 $M = Pm$，我们最终得出

$$m = C \begin{cases} 1 \,/\, P^{1/(d+1)} & \text{hyperbolic} \\ 1 \,/\, P^{2/(d+2)} & \text{parabolic} \end{cases}$$

即随着处理器数量的增加，每个处理器可用的内存容量会以更高的幂次下降。

2.2.7 其他扩展性度量

上述阿姆达尔定律是基于单处理器上的执行时间表述的。但在许多实际场景中，这一假设并不现实，因为并行处理的问题规模往往过大，无法在单一处理器上容纳。通过公式变形，我们可以得到某种程度上等效的量化指标，且无需依赖单处理器参数 [150]。

首先，根据 $S_p(n) = \frac{T_1(n)}{T_p(n)}$ 对强扩展性的定义，我们观察到 $T_1(n)/n$ 代表每个操作的串行时间，其倒数 $n/T_1(n)$ 可称为串行计算速率，记作 $R_1(n)$。类似地定义 ' 并行计算速率 '

$$R_p(n) = n/T_p(n)$$

我们可以得出

$$S_p(n) = R_p(n)/R_1(n)$$

在强扩展场景中 $R_1(n)$ 将保持恒定，因此我们仅通过测量 $T_p(n)$ 绘制出对数形式的加速比曲线。

2.2.8 并发；异步与分布式计算

即便在非并行计算机上，也存在多个进程同时执行的问题。操作系统通常采用时间片轮转机制，让所有活跃进程轮流获得 CPU 的短暂控制权。通过这种方式，顺序执行的机器可以模拟并行计算的效果 —— 当然，效率无法与之相提并论。

然而时间片轮转即使在不运行并行应用时也很有用：操作系统需要管理多个独立进程（如文本编辑器、邮件接收监视程序等），这些进程都需要保持活跃状态并按需运行。这类独立进程的难点在于它们有时需要访问相同资源。当两个进程各自持有一个资源，又同时请求对方持有的资源时，就会形成死锁。关于资源竞争最著名的形式化模型当属哲学家就餐问题。

研究此类独立进程的领域有多个称谓：并发、异步计算或分布式计算。" 并发 " 一词描述的是多个任务同时活跃执行且动作间无时序关系的场景；" 分布式计算 " 则源自数据库系统等应用场景 —— 多个独立客户端需要访问共享的数据库。

本书不会过多讨论这一主题。章节 2.6.1 探讨了支持时间切片机制的线程机制；在现代多核处理器上，线程可用于实现共享内存并行计算。

《通信顺序进程》一书对并发进程间的交互进行了分析 [106]。其他作者则运用拓扑学来分析异步计算 [101]。

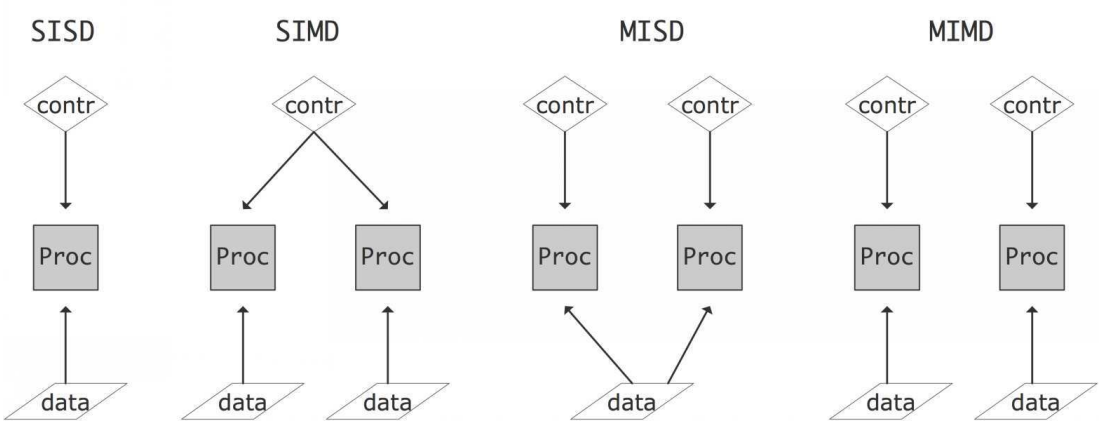


图 2.6：Flynn 分类法中的四种类别。

2.3 并行计算机架构

长期以来，顶级计算机一直是某种形式的并行计算机，即一种允许多条指令或指令序列同时执行的架构。描述这种架构多样性的方法之一源自 Flynn [64]。*Flynn* 分类法通过数据流和控制流是共享还是独立来区分架构。由此得出以下四种类型（另见图 2.6 ）：

SISD 单指令单数据：这是传统的 CPU 架构：任何时候仅执行一条指令，操作单个数据项。
SIMD 单指令多数据：此类计算机可包含多个处理器，每个处理器操作各自的数据项，但均对数据项执行相同指令。向量计算机（第 2.3.1.1 节）通常也被归类为 SIMD。**MISD** 多指令单数据。不存在符合此描述的架构；或可认为安全关键应用中的冗余计算是 MISD 的实例。

MIMD 多指令多数据流（ Multiple Instruction Multiple Data ）：此处多个 CPU 各自对不同的数据项进行操作，每个执行独立的指令。当前大多数并行计算机属于此类。

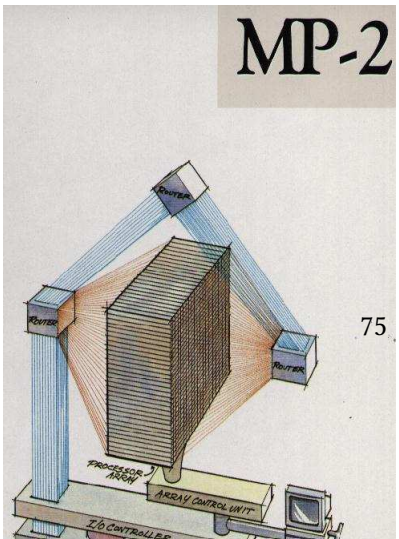
ex-

接下来我们将更详细地讨论 SIMD 和 MIMD 架构。

2.3.1 SIMD

SIMD 类型的并行计算机同时对多个数据项执行相同的操作。这类计算机的 CPU 设计可以相当简单，因为算术单元不需要独立的逻辑和指令解码单元：所有 CPU 同步执行相同的操作。这使得 SIMD 计算机特别擅长处理数组运算，例如

```
for (i=0; i<N; i++) a[i] = b[i]+c[i];
```



因此，它们也常被称为阵列处理器。科学计算代码通常可以编写成大部分时间花费在数组操作上的形式。

另一方面，有些操作无法在阵列处理器上高效执行。例如，计算递推关系 $x_{i+1} = ax_i + b_i$ 的若干项涉及多次加法和乘法运算，但这些运算交替进行，因此每次只能处理一种类型的运算。这里没有同时作为加法或乘法输入的数值数组。

为了允许对数据的不同部分执行不同的指令流，处理器会设置一个 ‘掩码位’，可通过设置该位来阻止指令执行。在代码中，这通常表现为

```
where (x>0) {
  x[i] = sqrt(x[i])
```

这种对多个数据项同时施加相同操作的编程模型，被称为数据并行。

此类数组运算可能出现在物理模拟场景中，但另一个重要来源是图形应用程序。对于这类应用，阵列处理器中的处理单元性能可以远低于个人电脑中的中央处理器：它们通常实际上是位处理器，每次仅能操作单个比特。沿着这一技术路线，*ICL* 在 1980 年代推出了 4096 个处理器的 DAP [112]，而 *Goodyear* 在 1970 年代建造了 16K 处理器的 MPP [10]。

后来，*Connection Machine* (CM-1、CM-2、CM-5) 系列大受欢迎。虽然第一代连接机采用位处理器（每芯片集成 16 个），但后续型号配备了支持浮点运算的传统处理器，并非真正的 SIMD 架构。这些机型均基于超立方体互连网络，详见章节 2.7.5。另一家成功商业化阵列处理器的制造商是 *MasPar*；图 2.7 展示了其架构。可以清晰看到为方形处理器阵列配备的单一控制单元，以及执行全局操作的网络。

基于阵列处理的超级计算机已不复存在，但 SIMD （单指令多数据）的概念仍以各种形式延续。例如，GPU 通过其 *CUDA* 编程语言强制实现 SIMD 架构。此外，*Intel Xeon Phi* 也具备显著的 SIMD 组件。早期 SIMD 架构的设计动机是尽量减少所需晶体管数量，而现代协处理器的驱动力则源于能效考量。处理指令（称为指令发射）在时间、能耗及芯片面积成本上实际远高于浮点运算。采用 SIMD 正是优化后两项指标的有效途径。

2.3.1.1 流水线与流水线处理器

多款计算机采用向量处理器或流水线处理器设计。首批商业成功的超级计算机 Cray-1 和 Cyber-205 即属此类。近年来，

Cray-X1 和 NEC SX 系列采用了向量流水线技术。曾连续三年占据 TOP500 榜单（见章节 [168], 2.11.4 ）的 " 地球模拟器 " 超级计算机，便是基于 NEC SX 处理器构建。流水线技术的基本原理已在章节 1.2.1.3 中阐述。

虽然基于流水线处理器的超级计算机已属少数，但流水线技术如今已成为构成集群的现代超标量 CPU 的标准配置。典型 CPU 配备流水线式浮点运算单元，通常加法与乘法单元分离（详见章节 1.2.1.3 ）。

然而，现代超标量 CPU 与老式向量处理器中的流水线存在重要差异。这些向量计算机的流水线单元并非集成在 CPU 内的浮点运算单元，而应视为外接于 CPU （ CPU 本身已含浮点单元）的向量协处理器。向量单元配备向量寄存器 [2]，通常可存储 64 个浮点数，且一般不设 " 向量缓存 "。其逻辑结构更为简单，常通过显式向量指令寻址。而超标量 CPU 则完全集成于芯片内，专为挖掘非结构化代码中的数据流而优化。

2.3.1.2 CPU 与 GPU 中的 TrueSIMD 技术

真正的 SIMD 阵列处理技术可见于现代 CPU 和 GPU 中，这两种情况均受到图形应用所需并行性的启发。

英特尔、 AMD 的现代 CPU 以及 PowerPC 芯片都具备向量指令集，可同时执行多个相同操作实例。在英特尔处理器上，这被称为 *SIMD* 流式扩展（ *SSE* ）或高级向量扩展（ *AVX* ）。这些扩展最初是为图形处理设计的，因为通常需要对大量像素执行相同操作。数据通常需要总计 128 位，可被划分为两个 64 位实数、四个 32 位实数，或更多更小的数据块（如 4 位）。

AVX 指令基于最高 512 位宽的 SIMD 技术，即能同时处理八个双精度浮点数。正如单精度浮点操作作用于寄存器中的数据（章节 1.3.4 ），向量操作使用向量寄存器。向量寄存器中的位置有时被称为 *SIMD* 通道。

采用 SIMD 技术的主要动机源于功耗考量。解码指令的能耗可能高于执行指令本身，因此 SIMD 并行成为节能的有效手段。

现代编译器可自动生成 SSE 或 AVX 指令；用户有时也能通过插入编译指示（例如使用 Intel 编译器）实现该功能：

```
void func(float *restrict c, float *restrict a,
float *restrict b,  int n){
#pragma vector alwaysfor (int i=0; i<n; i++)
c[i] = a[i] * b[i];}
```

2. The Cyber205 was an exception, with direct-to-memory architecture.

使用这些扩展通常要求数据与缓存行边界对齐（章节 1.3.5.7 ），因此有专门的分配和释放调用用于返回对齐的内存。

OpenMP 的第 4 版也包含了指示 SIMD 并行性的指令。

更大规模的数组处理可以在 *GPU* 中找到。一个 GPU 通常包含大量简单的处理器，按 32 个一组排列。每个处理器组限于执行相同的指令。因此，这是真正的 SIMD 处理实例。更多讨论见章节 2.9.3。

2.3.2 MIMD/SPMD 计算机

目前最常见的并行计算机架构称为多指令多数据（ MIMD ）：处理器执行多个可能不同的指令，每个指令作用于自己的数据。说指令不同并不意味着处理器实际运行不同的程序：大多数这类机器以单程序多数据（ *SPMD* ）模式运行，程序员在并行处理器上启动相同的可执行文件。由于可执行文件的不同实例可以通过条件语句采取不同路径，或执行循环的不同迭代次数，它们通常不会像在 SIMD 机器上那样完全同步。如果这种不同步是由于处理器处理不同数量的数据造成的，则称为负载不平衡，这是导致加速比不理想的主要原因；参见章节 2.10。

MIMD 计算机种类繁多，主要体现在内存组织方式、连接处理器的网络等硬件层面，以及编程方法上的差异。下文将详述这些方面。如今许多机器被称为集群，可由定制或商用处理器构建（若由运行 Linux 的 PC 通过以太网连接，则称为 *Beowulf* 集群 [90] ）；由于处理器相互独立，它们属于 MIMD 或 SPMD 模型的实例。

2.3.3 超级计算机的商用化

1980 至 1990 年代，超级计算机与个人电脑及 DEC PDP 、 VAX 系列等迷你 / 超级迷你计算机截然不同。SIMD 向量计算机通常配备一颗（ *CDC Cyber205* 或 *Cray-1* ），至多几颗（ *ETA-10* 、 *Cray-2* 、 *CrayX/MP* 、 *CrayY/MP* ）性能极强的处理器，多为向量处理器。1990 年代中期，由数千个简易（微）处理器构成的集群开始取代少量向量管道的机器（参见 http://www.top500.org/lists/1994/11 ）。早期这些微处理器（ *IBM Power* 系列、 *Intel i860* 、 *MIPS* 、 *DECAlpha* ）仍远强于家用电脑处理器，但后来这种差异也逐渐淡化。如今，许多最强集群采用消费级市场相同的 Intel Xeon 和 AMD Opteron 芯片驱动，其他则使用 IBM Power 系列等服务器芯片。这段历史自 1993 年以来的演变示例详见章节 2.11.4 。

2.4 不同类型的内存访问

在引言中，我们将并行计算机定义为多处理器协同解决同一问题的架构。除最简单的情况外，这意味着这些处理器需要访问共享的

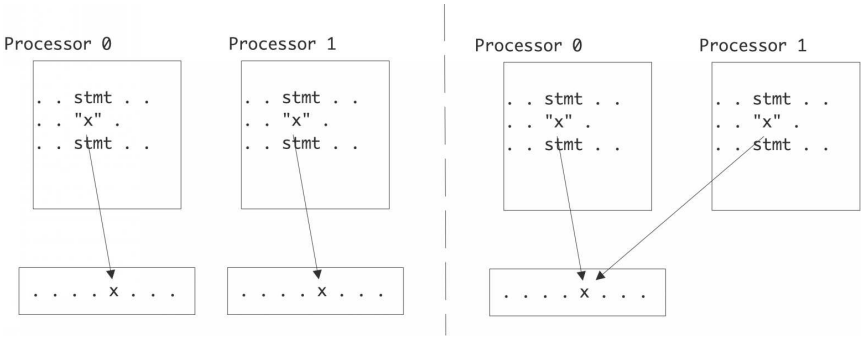


图 2.8：分布式内存与共享内存场景中对同名变量的引用。

数据池。在前一章中您已了解到，即便在单处理器上，内存也往往难以跟上处理器的需求。对于并行计算机而言，当多个处理器可能同时访问同一内存位置时，这一问题会变得更加严重。我们可以通过并行计算机如何处理多进程对共享数据池的并发访问问题，来刻画其特性。

这里的主要区别在于分布式内存与共享内存之间。在分布式内存中，每个处理器拥有独立的物理内存，更重要的是拥有独立的地址空间。因此，若两个处理器引用变量 `x`，它们访问的是各自本地内存中的变量。这是 SPMD 模型的一个实例。

另一方面，在共享内存中，所有处理器访问同一内存；我们亦称之为共享地址空间。参见图 2.8。

2.4.1 对称多处理器：统一内存访问

如果任何处理器都能访问任意内存位置，那么并行编程将相当简单。正因如此，制造商有强烈动机设计出能让处理器对所有内存位置一视同仁的架构：每个处理器均可访问任意内存位置，且访问时间无差异。这种架构被称为统一内存访问 *(UMA)*，基于此原则的架构编程模型通常称为对称多处理 *(SMP)*。

实现 SMP 架构有几种方法。当前的台式计算机可以通过单一内存总线让少数几个处理器访问共享内存；例如，苹果公司就推出了一款配备 2 个六核处理器的机型。处理器共享内存总线的方式仅适用于少量处理器；对于更多数量的处理器，可以采用交叉开关将多个处理器与多个内存库相连；具体可参阅 2.7.6 节。

在多核处理器上，存在另一种类型的统一内存访问方式：各核心通常共享一个缓存，常见的是 L3 或 L2 缓存。

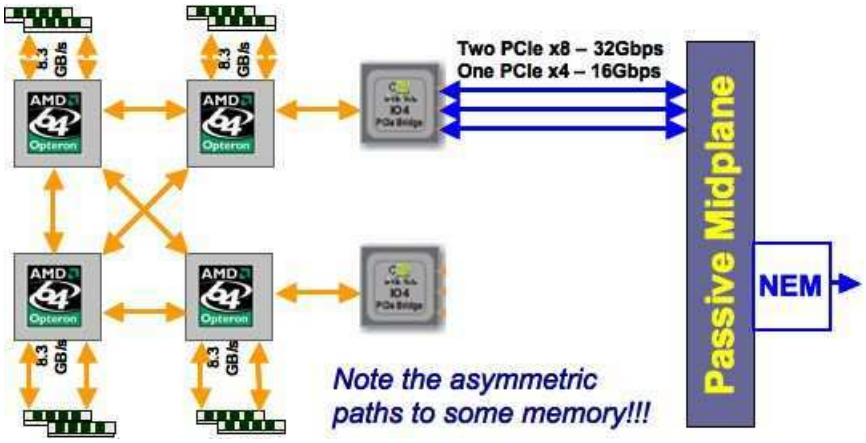


图 2.9：四路主板中的非统一内存访问架构。

2.4.2 非统一内存访问

基于共享内存的 UMA 方法显然仅适用于少量处理器。交叉开关网络具有可扩展性，因此似乎是最佳选择。然而实践中，人们会将带有本地内存的处理器置于交换网络配置中。这导致处理器能快速访问自身内存，而访问其他处理器内存则较慢。这就是所谓的 *NUMA*（非统一内存访问）的一种情况：该策略使用物理分布的内存，放弃统一的访问时间，但保持逻辑上共享的地址空间 —— 每个处理器仍可访问任意内存位置。

2.4.2.1 亲和性

在 NUMA 架构下，数据应放置于何处（相对于将要访问它的进程或线程）这一问题变得至关重要。这被称为亲和性；若从进程或线程的放置视角来看，则称为进程亲和性。

图 2.9 展示了四路主板在 *TACC Ranger* 集群中的 NUMA 架构。每个芯片拥有独立内存（ 8GB ），但主板使处理器仿佛能访问 32GB 的共享内存池。显然，访问其他处理器的内存比访问本地内存更慢。此外需注意，每个处理器虽有三个连接可用于访问其他内存，但最右侧两枚芯片的一个连接用于网络通信。这意味着处理器间内存访问必须通过中间处理器中转，这会降低传输速度并占用该处理器的连接资源。

2.4.2.2 一致性

NUMA 架构虽为程序员提供了便利，却给系统带来挑战。假设两个不同处理器的本地（缓存）内存中各存有同一内存位置的副本，若一个处理器修改该位置内容，变更必须传播到其他处理器。若两个处理器同时试图修改同一内存位置，程序行为可能变得不可预测。

保持内存位置副本同步的过程被称为缓存一致性（详见章节 1.4.1 ）；采用该技术的多处理器系统有时被称为 ‘缓存一致性 NUMA’ 或 *ccNUMA* 架构。

将 NUMA 理念推向极致，可通过软件层使网络连接的处理器呈现为操作共享内存的状态。这被称为分布式共享内存或虚拟共享内存。在此方案中，虚拟机监控程序通过将系统调用转换为分布式内存管理，提供共享内存 API。该共享内存 API 可被 *Linux* 内核利用，支持多达 4096 个线程。

当前供应商中仅 SGI （ *UV* 系列）和 Cray （ *XE6* ）销售大规模 NUMA 产品。两者均对分区全局地址空间（ *PGAS* ）语言提供强力支持（参见章节 2.6.5 ）。另有如 *ScaleMP* 等供应商提供常规集群上实现分布式共享内存的软件解决方案。

2.4.3 逻辑与物理分布式内存

针对内存访问问题最极端的解决方案是提供不仅在物理上、而且在逻辑上也分布的内存：处理器拥有各自的地址空间，无法直接访问其他处理器的内存。这种方法常被称为 ‘分布式内存’，但这一术语并不准确，因为我们实际上需要分别考虑内存是否分布以及是否呈现分布状态。值得注意的是，NUMA 架构同样具备物理分布式内存，但其分布特性对程序员而言并不显见。

在逻辑与物理分布式内存架构中，处理器间交换信息的唯一方式是通过网络显式传递数据。您将在 2.6.3.3 章节中了解更多相关内容。

此类架构具有显著优势 —— 可扩展至海量处理器规模：*IBM BlueGene* 超级计算机就采用了超过 20 万个处理器。但另一方面，这也是最难编程的并行系统类型。

上述类型之间存在多种混合形态。事实上，大多数现代集群会采用 NUMA 节点架构，但节点间仍通过分布式内存网络连接。

2.5 并行粒度

本节我们从工作划分深度的角度探讨并行性。我们将重点解析粒度这一概念：即每个处理单元独立工作量与同步频率之间的平衡关系。若同步点之间存在大量工作，我们称之为「粗粒度并行」；若工作量较小，则称为「细粒度并行」。显然，要使细粒度并行具有效益，同步过程必须极快；而粗粒度并行则可以容忍较高成本的同步操作。

本节讨论主要停留在概念层面；具体并行编程的实践细节将在 2.6 章节深入探讨。

2.5.1 数据并行

程序中常见的情况是存在循环结构，其循环体较为简单，但需对大型数据集中的所有元素执行：

```
for (i=0; i<1000000; i++)
  a[i] = 2*b[i];
```

此类代码被视为数据并行或细粒度并行的实例。若处理器数量与数组元素数量相当，这段代码会显得极为简单：每个处理器仅需对本地数据执行该语句

```
a = 2*b
```

对其本地数据进行操作。

若代码主要由此类数组循环构成，则可通过所有处理器同步锁步高效执行。基于此理念的架构（处理器实际上只能 同步锁步工作）早已存在，详见章节 2.3.1。这类对数组的完全并行操作常见于计算机图形学领域，其中图像的每个像素都被独立处理。正因如此，GPU （章节 2.9.3 ）高度依赖数据并行性。

由 *NVidia* 发明的统一计算设备架构（ CUDA ）语言，能优雅地表达数据并行性。后续开发的 *Sycl* 等语言或 *Kokkos* 等类库虽目标相似，但更侧重于异构并行计算。

Cont延续上述示例稍作扩展，考虑 operation

for $0 \leq i <$ 最大值 **do**^{i}左 = 模 $(i-1,$ 最大值 $)^{i}$右 = 模 $(i+1,$ 最大值 $)$
$a_i = (b_{i_{\text{left}}} + b_{i_{\text{right}}})/2$

在数据并行机器上，这可以实现为

`bleft` $\leftarrow$ **右移** (`b`)
`bright` $\leftarrow$ **左移** (`b`)
`a` $\leftarrow$ (`bleft` + `bright`)/2

其中 `shiftleft/right` 指令会导致数据项被发送到编号减 1 或加 1 的处理器。为了使第二个示例高效运行，必须确保每个处理器能与其直接相邻的处理器快速通信，且首尾处理器之间也能相互通信。

在图形处理中的 ‘ 模糊 ’ 操作等多种场景下，对二维数据进行操作是有意义的：

for $0 < i < m$ **do**
  for $0 < j < n$ **do**
    $a_{ij} \leftarrow (b_{ij-1} + b_{ij+1} + b_{i-1j} + b_{i+1j})$

因此处理器需要具备在二维网格中向相邻节点传输数据的能力。

2.5.2 指令级并行

在 *ILP* 中，并行性仍体现在单个指令层面，但这些指令不必相似。例如，r
在

$$a \leftarrow b + c$$
$$d \leftarrow e * f$$

两个赋值操作相互独立，因此可以同时执行。这类并行性对人类而言过于繁琐难以识别，但编译器却非常擅长。实际上，识别 ILP 对充分发挥现代超标量 CPU 的性能至关重要。

2.5.3 任务级并行

与数据和指令级并行性截然相反的是，任务并行性关注的是识别可以并行执行的整个子程序。举例来说，在树形数据结构中进行搜索可以如下实现：

> **若最优 (** 根节点 **) 则退出，否则并行**：**SearchInTree** (左子节点)，**SearchInTree** (右子节点) **过程** SearchInTree(根节点)

此例中的搜索任务未进行同步，且任务数量不固定：可无限增长。实践中，任务过多并非良策，因为处理器在仅处理单一任务时效率最高。此时任务可按如下方式调度：

> **当**存在剩余任务时**执行**等待至处理器空闲；在其上生成新任务

（前述两段伪代码存在微妙差异。第一段中任务自我调度：每个任务派生出两个新任务。第二段代码展示了管理者 - 工作者范式的实例：存在一个贯穿代码生命周期的中央任务，负责派生并分配工作者任务。）

与上述数据并行示例不同，在此类方案中，数据向处理器的分配并非预先确定。因此，这种并行模式最适合线程编程，例如通过 OpenMP 库实现；详见章节 2.6.2。

让我们考虑一个更严肃的任务级并行示例。

有限元网格在最简单的情况下是覆盖二维对象的一组三角形集合。由于应避免过于尖锐的角，*Delauney* 网格细化过程可以选取某些三角形，并用形状更优的三角形替换它们。如图 2.10 所示：黑色三角形违反了某些角度条件，因此它们自身会被细分，或与相邻三角形（以灰色渲染）合并后重新划分。

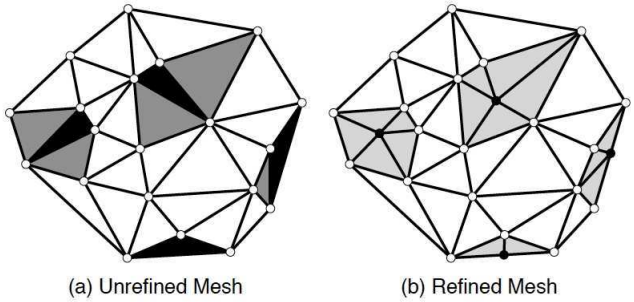


图 2.10：细化前后的网格对比。

网格 m = /* 读入初始网格 */ 工作列表
wl;
wl.add(mesh.badTriangles());**while**
`(wl.size() != 0)`**do** 元素 =
wl.get(); // 获取劣质三角形 if (e 不在网
格中) continue; 腔体 c = new
Cavity(e); c.expand();
c.retriangulate(); mesh.update(c);
wl.add(c.badTriangles());

图 2.11：Delaunay 细化的任务队列实现。

用伪代码表示，其实现如图 2.11 所示。（该图及代码可在 [124], 中找到，其中还包含更详细的讨论。）

显然，该算法由需在所有进程间共享的工作列表（或任务队列）数据结构驱动。结合数据到进程的动态分配，这意味着此类不规则并行适合共享内存编程，而在分布式内存中实现则困难得多。

2.5.4 便利并行计算

在某些场景下，一个简单的、通常由单处理器执行的计算任务需要对大量不同输入进行处理。由于这些计算不存在数据依赖关系，且无需按特定顺序执行，因此常被称为易并行或便利并行计算。此类并行性可能出现在多个层级。例如在计算曼德博集合或评估国际象棋走法时，会针对多个参数值调用子程序级计算。在更宏观的层面，可能需要为多个输入运行简单程序，此时整体计算被称为参数扫描。

2.5.5 中粒度数据并行

上述严格实现的数据并行假设处理器数量与数据元素数量相等。实际上，处理器的内存容量远大于此，且数据元素数量很可能远超最大规模计算机的处理器数量。因此，数组会被分组为子数组分配到各处理器上。代码实现如下所示：

```
my_lower_bound = // some processor-dependent number
my_upper_bound = // some processor-dependent number
for (i=my_lower_bound; i<my_upper_bound; i++)
  // the loop body goes here
```

该模型兼具数据并行特性（因对大量数据项执行相同操作），亦可视为任务并行（因每个处理器执行较大代码段，且处理的数据块大小未必均等）。

2.5.6 任务粒度

在前几小节中，我们探讨了发现并行工作的不同层次，或者说划分工作以寻找并行性的不同方式。还有一种视角：我们将并行方案的粒度定义为处理单元在执行完一定量工作（或任务规模）后必须与其他处理单元通信或同步的程度。

在指令级并行（ILP）中，我们处理的是非常细粒度的并行性，其规模约为单条指令或仅少数几条指令。而在真正的任务并行中，粒度则要粗得多。

此处值得关注的是数据并行性，我们可以自由选择任务规模。在 SIMD 机器上，可以选择单条指令的粒度，但正如你在 2.5.5 节所见，操作可被组合成中等规模的任务。因此，只要处理器数量与总问题规模之间达到适当平衡，数据并行操作就可在分布式内存集群上执行。

练习 2.20. 讨论如何为二维网格上的数据并行操作（如求平均值）选择合适的任务粒度。证明存在表面积 - 体积比效应：通信量比计算量低一个数量级。这意味着即使通信速度远慢于计算速度，增大任务规模仍能实现均衡执行。

遗憾的是，选择大任务规模以克服通信延迟可能会加剧另一个问题：聚合这些操作可能导致任务运行时间不均，引发负载失衡。一种解决方案是采用过分解策略：创建比处理单元数量更多的任务，并将多个任务分配给单个处理器（或动态分配任务）以均衡不规则运行时间。这被称为动态调度，如 2.5.3 节示例所示；另见 2.6.2.1 节。过分解在线性代数中的应用案例可参阅 7.3.2 节。

2.6 并行编程

并行编程比顺序编程更为复杂。虽然在顺序编程中，大多数编程语言遵循相似的原则（除了函数式或逻辑语言等少数例外），但处理并行性问题的方式却多种多样。让我们探讨其中的一些概念和实践方面。

并行编程有多种实现途径。首先，目前似乎不可能出现一种并行化编译器，能够自动将顺序程序转换成并行程序。除了需要判断哪些操作是独立的问题外，主要难点在于并行环境下数据的定位极其困难。编译器需要分析整个代码而非逐个处理子程序。即便如此，现有成果仍不尽如人意。

更高效的方案是让用户主要编写顺序程序，但提供关于哪些计算可并行化及数据应如何分布的指示。在 OpenMP 中通过显式标注操作并行性实现（章节 2.6.2 ）；而 PGAS 语言则通过指定数据分布，将并行性交由编译器和运行时处理（章节 2.6.5 ）。这类方法在共享内存环境中表现最佳。

迄今为止最困难的并行编程方式，但在实践中效果最佳，是将并行性暴露给程序员，让程序员显式地管理一切。这种方法在分布式内存编程的情况下是必要的。我们将在 2.6.3.1 节中讨论分布式编程的一般性问题；2.6.3.3 节将讨论 MPI 库。

2.6.1 线程并行性

作为 OpenMP （ 2.6.2 节）的铺垫，我们将简要介绍 ‘线程’ 。

要解释什么是线程，我们首先需要从技术角度理解什么是进程。Unix 进程对应单个程序的执行。因此，它在内存中拥有：

- 程序代码，以机器语言指令的形式存在；
- 一个堆，包含例如通过 `malloc` 创建的数组；
- 一个存储快速变化信息的栈，例如程序计数器（用于指示当前正在执行的指令），以及具有局部作用域的数据项和计算过程中的中间结果。

一个进程可以拥有多个线程；这些线程的相似之处在于它们共享相同的程序代码和堆，但各自拥有独立的栈。因此，线程是进程中独立执行的 ‘线索’ 。

进程可以属于不同用户，或是同一用户并发运行的不同程序，因此它们拥有独立的数据空间。而线程作为进程的一部分，共享进程的堆。线程可以拥有部分私有数据（例如通过独立的数据栈），但其主要特征在于能够协作处理同一数据。

2.6.1.1 fork-join 机制

线程是动态的，这意味着它们可以在程序执行期间被创建。（这与 MPI 模型不同，在 MPI 中每个处理器运行一个进程，且所有进程都在

同一时间。）程序启动时，有一个活跃线程：主线程。其他线程通过线程派生创建，主线程可等待其完成。这被称为 *fork-join*

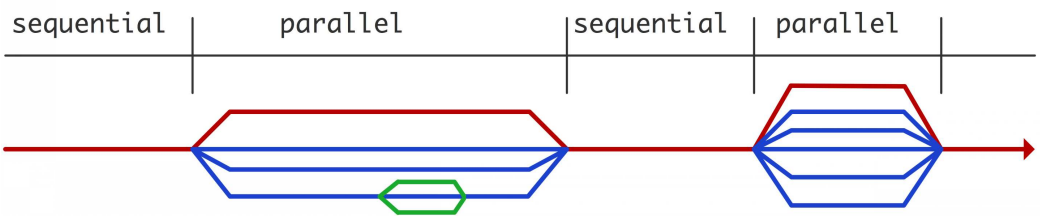


图 2.12：并行执行期间的线程创建与删除。

模型；如图 2.12 所示。从同一线程派生且同时活跃的一组线程称为线程组。

2.6.1.2 线程的硬件支持

如前所述的线程是一种软件构造。在并行计算机出现之前，线程技术就已存在；例如用于操作系统中处理独立活动。若无并行硬件，操作系统会通过多任务处理或时间片轮转管理线程：每个线程定期获得 CPU 的短暂使用权。（技术上，Linux 内核通过任务概念处理进程和线程；任务保存在列表中，并定期激活或停用。）

这可以提高处理器利用率，因为一个线程的指令可以在另一个线程等待数据时被处理。（在传统 CPU 上，线程间切换有一定开销（超线程机制是个例外），但在 GPU 上则不然，事实上它们需要大量线程来实现高性能。）

现代多核处理器支持线程的方式显而易见：每个核心分配一个线程可实现并行执行，从而高效利用硬件资源。共享内存使所有线程都能访问相同数据。但这也可能引发问题，详见章节 2.6.1.5。

2.6.1.3 线程示例

以下示例（严格基于 Unix 系统，无法在 Windows 运行）清晰演示了分叉 - 合并模型。该示例使用 *pthreads* 库生成多个任务，这些任务共同更新全局计数器。由于线程共享相同内存空间，它们确实能访问并更新同一内存地址。

```
#include <stdlib.h>
#include <stdio.h>
#include "pthread.h"

int sum=0;

void adder() {
```

```
  sum = sum+1;
  return;
}

#define NTHREADS 50
int main() {
  int i;
  pthread_t threads[NTHREADS];
  printf("forking\n");
  for (i=0; i<NTHREADS; i++)
    if (pthread_create(threads+i,NULL,&adder,NULL)!=0) return i+1;
  printf("joining\n");
  for (i=0; i<NTHREADS; i++)
    if (pthread_join(threads[i],NULL)!=0) return NTHREADS+i+1;
  printf("Sum computed: %d\n",sum);

  return 0;
}
```

这段代码能给出正确结果纯属巧合：仅仅是因为更新变量的速度远快于创建线程。（在多核处理器上，出错的概率会大幅增加。）如果我们人为延长更新的时间，就再也得不到正确结果了：

```
void adder() {int t = sum; sleep(1); sum = t+1;return;}
```

现在所有线程都读取 `sum` 的值，等待一段时间（可能在进行某些计算）然后更新。

可以通过在应保持 ' 互斥 ' 的代码区域设置锁来解决此问题：

```
pthread_mutex_t lock;void adder() {int t;pthread_mutex_lock(&lock);
t = sum; sleep(1); sum = t+1;pthread_mutex_unlock(&lock);return;}
int main() {....
```

```
pthread_mutex_init(&lock,NULL);
```

加锁与解锁命令能确保两个线程不会互相干扰对方的更新操作。

2.6.1.4 上下文

在上述示例及其使用 `sleep` command 的版本中，我们忽略了一个事实：其中涉及两类数据。首先，变量 `s` 是在线程生成部分之外创建的，因此该变量是共享的。

另一方面，变量 `t` 是在每个生成的线程中单独创建的。我们称其为私有数据。

T一个线程能够访问的所有数据总和称为其上下文。它包含私有和共享数据，
以及该线程正在处理的临时计算结果。（它还包含程序计数器和栈指针。若您不了解这些概念也无需担心。）

完全有可能创建比处理器核心数量更多的线程，因此处理器可能需要在不同线程的执行之间切换。这被称为上下文切换。

在常规 CPU 上，上下文切换并非零成本操作，因此只有当线程化工作的粒度足够大时才具有效益。此规则的例外情况包括：

- 具备硬件多线程支持的 CPU，例如通过超线程技术（参见章节 2.6.1.9 ），或如 *IntelXeon Phi* （章节 2.9 ）；
- GPU，其实际上依赖于快速上下文切换（章节 2.9.3 ）；
- 某些其他 ‘ 异质 ’ 架构，例如 *Cray XMT* （章节 2.8 ）。

2.6.1.5 竞态条件、线程安全与原子操作

共享内存简化了程序员的工作，因为每个处理器都能访问所有数据：处理器之间无需显式数据传输。然而，多个进程 / 处理器也可能同时写入同一个变量，这会引发潜在问题。

假设两个进程都试图递增整型变量 `I` ：

初始值： `I=0`

进程 1： `I=I+2`
进程 2： `I=I+3`

若该变量是独立进程计算值的累加器，则此行为是合法的。这两次更新的结果取决于处理器读写该变量的顺序。

图 2.13 展示了三种场景。这种最终结果取决于线程执行顺序的场景，被称为竞态条件 或 数据竞争。其正式定义为：

> 当存在两个语句 S_1 、 S_2 时，我们称之为数据竞争，
> - 且两者无因果关系；
> - 且均访问同一位置 L ；以及
> - 且至少有一次访问为写入操作。

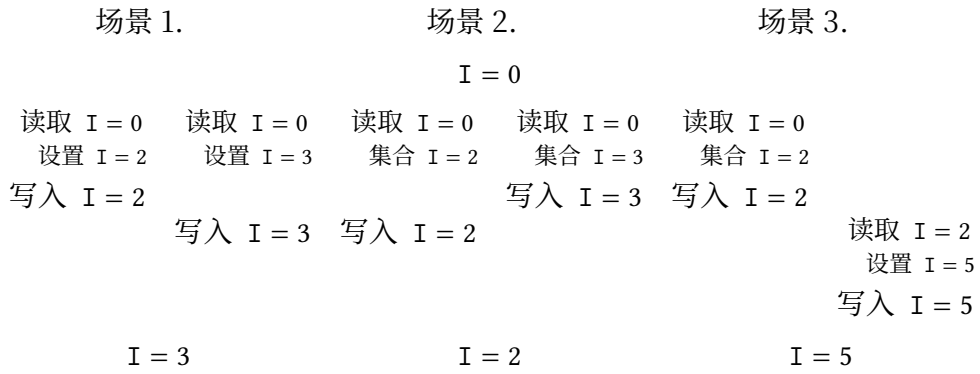


图 2.13：数据竞争场景的三种执行情况。

这类冲突更新的一个非常实际的例子是内积计算：

```
for (i=0; i<1000; i++)
   sum = sum+a[i]*b[i];
```

这里的乘积计算确实是独立的，因此可以选择让循环迭代并行执行这些计算，例如通过各自的线程。然而，所有线程都需要更新同一个变量 sum。这种特定的数据冲突情况被称为归约，它非常常见，以至于许多线程系统都为此设计了专用机制。

无论以顺序方式还是多线程方式执行，行为都相同的代码被称为线程安全的。从上述示例可以看出，线程不安全通常是由于对共享数据的处理方式导致的。这意味着程序中使用的局部数据越多，线程安全的可能性就越高。遗憾的是，有时线程确实需要写入共享 / 全局数据。

解决这个问题本质上存在两种方法。其一是将共享变量的此类更新声明为临界区代码段。这意味着临界区内的指令（在内积示例中即 ‘从内存读取 sum 、更新数据、写回内存’）同一时间只能由一个线程执行。具体而言，这些指令必须由一个线程完整执行完毕后，其他线程才能开始执行，从而避免前文所述的歧义问题。当然，上述代码片段如此常见，因此像 OpenMP （第 2.6.2 节）这样的系统专门为此提供了机制 —— 通过将其声明为归约操作。

临界区可以通过信号量机制 [48] 来实现。每个临界区周围会有两个原子操作控制一个信号量（即标志位）。第一个遇到信号量的进程会降低它，并开始执行临界区。其他进程看到降低的信号量后进入等待状态。当第一个进程完成临界区执行时，它会执行第二条指令来升高信号量，从而允许一个等待中的进程进入临界区。

解决共享数据共同访问的另一种方式是在特定内存区域设置临时锁。如果临界区很可能被共同执行（例如

它实现了对数据库或哈希表的写入操作。在这种情况下，一个进程进入临界区将阻止任何其他进程写入数据，即使它们可能写入的是不同位置；此时锁定正在访问的特定数据项是更优的解决方案。

锁机制的问题在于它们通常存在于操作系统层面。这意味着它们的速度相对较慢。鉴于我们希望上述内积循环的迭代能以浮点运算单元的速度执行，或至少以内存总线的速度执行，这种延迟是不可接受的。

其中一种实现方式是 事务性内存，硬件本身支持原子操作；该术语源自数据库事务，后者存在类似的完整性问题。在事务性内存中，进程会执行正常的内存更新，除非处理器检测到与另一进程的更新存在冲突。此时，更新（ ‘事务’ ）会被取消并重试，一个处理器锁定内存，另一个则等待锁释放。这是一种优雅的解决方案；然而，取消事务可能会带来流水线刷新（章节 1.2.5 ）和缓存行失效（章节 1.4.1 ）的额外开销。

2.6.1.6 内存模型与顺序一致性

上述信号现象表明，竞态条件 会导致某些程序的运行结果具有不确定性，具体取决于指令执行的顺序。此外还存在另一个关键因素，即处理器和 / 或编程语言采用的内存模型 [3]。内存模型决定了线程或核心之间的活动可见性。

举例说明，初始状态：
`A=B=0;`，随后进程 1：
`A=1; x = B;`，进程 2：
`B=1; y = A;`
如前所述，我们通过全局语句序列来描述三种可能场景：

场景 1.	场景 2.	场景 3。
A ← 1	A ← 1	B ← 1
x ← B	B ← 1	y ← A
B ← 1	x ← B	A ← 1
y ← A	y ← A	x ← B
$x = 0, y = 1$	$x = 1, y = 1$	$x = 1, y = 0$

(在第二种场景中，语句 1 、 2 可以调换顺序，语句 3 、 4 同样可以调换，而不会改变最终结果。)

这三种不同的结果可被描述为：通过一个全局排序计算得出，该排序遵循各语句的局部顺序。这被称为顺序一致性：并行执行的结果与按局部语句顺序交错执行并行计算的串行结果保持一致。

维护顺序一致性的代价高昂：这意味着对变量的任何修改必须立即对所有其他线程可见，或者线程对变量的任何访问都需要咨询所有其他线程。我们已在 1.4.1 节讨论过这一点。

I在宽松内存模型中，有可能得到非顺序一致性的结果。假设在
a述示例中，编译器决定对两个进程的语句进行重排序，因为读取
a写入操作是独立的。实际上我们得到了第四种情景：

情景 4。

x ← B
y ← A
A ← 1
B ← 1

$x = 0, y = 0$

leading to the result $x = 0, y = 0$, 这在上述顺序一致性模型中是不可能实现的。（存在寻找此类依赖关系的算法 [121]。）

顺序一致性意味着

```
integer n
n = 0
!$omp parallel shared(n)
n = n + 1
!$omp end parallel
```

应具有与 ... 相同的效果

```
n = 0
n = n+1 ! for processor 0
n = n+1 ! for processor 1
        ! et cetera
```

在顺序一致性模型下，不再需要声明原子操作或临界区；然而，这对模型的实现提出了严格要求，可能导致代码效率低下。

2.6.1.7 亲和性

线程编程非常灵活，能根据需要有效创建并行性。但本书大部分内容强调科学计算中数据移动的重要性，这一方面在线程编程中同样不可忽视。

在多核处理器的背景下，任何线程都可以被调度到任何核心上运行，这本身并无直接问题。然而，若追求高性能，这种灵活性可能会带来意想不到的代价。出于多种原因，您可能希望特定线程仅运行在特定核心上。由于操作系统允许迁移线程，或许您只是希望线程保持原位。

- 若线程迁移至不同核心，而该核心拥有独立缓存，您将丢失原缓存内容，并引发不必要的内存传输。
- 若线程发生迁移，操作系统可能将两个线程置于同一核心，而另一核心完全闲置。即便线程数与核心数相等，这显然会导致无法实现理想加速。

我们将线程（线程亲和性）或进程（进程亲和性）与核心之间的映射称为亲和性。亲和性通常表示为掩码：即描述允许线程运行的位置集合。

例如，考虑一个双插槽节点，每个插槽包含四个核心。 .

Wit在两线程与套接字亲和性的情况下，我们有以下内容 affinity mask:

线程	插槽 0	插槽 1
0	0-1-2-3	
1		4-5-6-7

核心亲和性的掩码取决于亲和类型。典型策略包括 ‘紧密’ 和 ‘分散’。采用紧密亲和性时，掩码可能为：

thread	socket 0	socket 1
0	0	
1	1	

两个线程位于同一插槽意味着它们可能共享 L2 缓存，因此若线程间需共享数据，此策略是合适的。

另一方面，采用分散亲和性策略时，线程会被放置得更远：

线程	插槽 0	插槽 1
0	0	
1		4

此策略更适用于带宽密集型应用，因为现在每个线程可独占一个插槽的带宽，而非在 ‘紧密’ 场景下被迫共享带宽。

若分配所有核心，紧密（ close ）与分散（ spread ）策略会导致不同的排列方式：

	插槽 0	插槽 1		插槽 0	插槽 1
关闭	0-1-2-3		扩展	0-2-4-6	
		4-5-6-7			1-3-5-7

Affinity and data access patterns 亲和性亦可视为一种将执行与数据绑定的策略。

C考虑以下代码：

```
for (i=0; i<ndata; i++) // this loop will be done by threads
  x[i] = ....
for (i=0; i<ndata; i++) // as will this one
  ... = .... x[i] ...
```

第一个循环通过访问 x 的元素，将内存载入缓存或页表。第二个循环按相同顺序访问元素，因此固定亲和性是对性能有利的正确决策。

在其他情况下，固定映射并非正确解决方案：

```
for (i=0; i<ndata; i++) // produces loop
  x[i] = ....
for (i=0; i<ndata; i+=2) // use even indices
  ... = ... x[i] ...
for (i=1; i<ndata; i+=2) // use odd indices
  ... = ... x[i] ...
```

在第二个示例中，要么需要转换程序，要么程序员必须实际维护一个任务队列。

首次接触 人们很自然地会从 ‘将执行放在数据所在之处’ 的角度思考亲和性。然而实践中，相反的观点有时更合理。例如，图 2.9 展示了集群节点的共享内存实际可能如何分布。因此，线程可绑定至某个插槽，而数据可能被操作系统分配到任意插槽上。操作系统常用的机制称为首次接触策略：

- 当程序分配数据时，操作系统实际上并未创建该数据；
- 相反，数据的内存区域是在线程首次访问时创建的；
- 因此，首个触及该区域的线程实际上会导致数据被分配到其所在插槽的内存上。

Exercise 2.21. 解释以下代码的问题。

```
#pragma omp parallel initialization

for (i=0; i<N; i++)
  a[i] = 0.;
#pragma omp parallel for
for (i=0; i<N; i++)
  a[i] = b[i] + c[i];
```

关于内存策略的深入讨论，请参阅 [128]。

2.6.1.8 Cilk Plus

还存在其他基于线程的编程模型。例如，英特尔 *Cilk Plus*（http://www.cilkplus.org/）是一组 C/C++ 的扩展，程序员可用其创建线程。

图 2.14 展示了斐波那契数列的计算，包括顺序执行和使用 Cilk 的线程化版本。在此例中，变量 `rst` 由两个可能独立的线程更新。此更新的语义，即如何解决诸如同时写入等冲突的精确定义，由顺序一致性规定；参见章节 2.6.1.6。

Sequential code:

```
int fib(int n){
    if (n<2) return 1;
    else {
         int rst=0;
         rst += fib(n-1);
         rst += fib(n-2);
         return rst;
         }
}
```

Cilk code:

```
cilk int fib(int n){
         if (n<2) return 1;
         else {
              int rst=0;
              rst += cilk_spawn fib(n-1);
              rst += cilk_spawn fib(n-2);
              cilk_sync;
              return rst;
              }
}
```

图 2.14：斐波那契数列的顺序与 Cilk 代码实现。

2.6.1.9 超线程与多线程对比

在上述示例中，您可以看到程序运行期间生成的线程本质上执行相同的代码，并访问相同的数据。因此，在硬件层面，线程由少量局部变量唯一确定，例如其在代码中的位置（程序计数器）以及当前计算涉及的中间结果。

超线程是英特尔的一项技术，允许多个线程真正同时使用处理器，从而优化处理器的部分资源利用率。

若处理器在执行一个线程与另一个线程之间切换，它会保存当前线程的局部信息，并加载另一线程的信息。这种操作的成本相较于运行整个程序而言较为有限，但可能与单条指令的成本相比显得较高。因此，超线程技术未必总能带来性能提升。

某些架构支持多线程。这意味着硬件实际上为多个线程的本地信息提供了显式存储，线程间切换可以非常快速。GPU （章节 2.9.3 ）和 *Intel Xeon Phi* 架构就是这种情况，其中每个核心可支持多达四个线程。

然而，超线程共享核心的功能单元，即算术处理部分。因此，多个活跃线程不会为计算密集型代码带来成比例的性能提升。当线程在特性上异构时，预期能获得大部分增益。

2.6.2 OpenMP

OpenMP 是 C 和 Fortran 编程语言的扩展。其实现并行化的主要方式是通过循环的并行执行：基于编译器指令，预处理器可以调度循环迭代的并行执行。

由于 OpenMP 基于线程，它具有动态并行性：并行执行流的数量可以在代码的不同部分变化。通过创建并行区域来声明并行性，例如表明循环嵌套的所有迭代都是独立的，运行时系统随后会利用所有可用资源。

OpenMP 并非一门独立语言，而是对现有 C 和 Fortran 语言的扩展。它主要通过向源代码中插入指令来运作，这些指令由编译器解释。它也有少量库调用，但与 MPI （章节 2.6.3.3 ）不同，这些并非主要部分。最后，还有一个运行时系统来管理并行执行。

OpenMP 在可编程性上相比 MPI 有一个重要优势：可以从一个串行代码开始，通过增量式并行化逐步改造。相比之下，将串行代码转换为分布式内存的 MPI 程序则是一种全有或全无的改造。

许多编译器，例如 *gcc* 或 Intel 编译器，都支持 OpenMP 扩展。在 Fortran 中，OpenMP 指令被置于注释语句内；在 C 语言中，它们则被放在 `#pragma` CPP 指令中，这些指令标明了编译器特定的扩展。因此，对于不支持 OpenMP 的编译器而言，OpenMP 代码看起来仍像是合法的 C 或 Fortran 代码。程序需要链接到 OpenMP 运行时库，且其行为可通过环境变量进行控制。

For more information about OpenMP, see [32] and http://openmp.org/wp/.

2.6.2.1 OpenMP 示例

使用 OpenMP 最简单的例子是并行循环。

```
#pragma omp parallel for
for (i=0; i<ProblemSize; i++) {
  a[i] = b[i];
}
```

显然，所有迭代都可以独立且以任意顺序执行。pragma CPP 指令随后将这一事实传达给编译器。

有些循环在概念上是完全并行的，但在实现上并非如此 :

```
for (i=0; i<ProblemSize; i++) {
  t = b[i]*b[i];
  a[i] = sin(t) + cos(t);
}
```

此处看似每次迭代都会对共享变量 `t` 进行读写操作。但实际上， `t` 是一个临时变量，仅属于每次迭代的局部作用域。这种本应可并行化但因类似结构而无法实现的代码，被称为非线程安全。

OpenMP 通过以下方式声明该临时变量是每次迭代私有的：

```
#pragma omp parallel for shared(a,b)、
 private(t)for (i=0; i<ProblemSize; i++) {
t = b[i]*b[i];a[i] = sin(t) + cos(t);}
```

若标量确实被共享，OpenMP 提供了多种机制来处理这种情况。例如，共享变量常出现在归约操作中：

```
  sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (i=0; i<ProblemSize; i++) {
    sum = sum + a[i]*b[i];
  }
```

如你所见，只需极少的努力即可将串行代码并行化。

T迭代任务分配给线程的工作由运行时系统完成，但用户可以指导这一分配过程。
我们主要关注线程数少于迭代次数的情况：若有 P 个线程和 N 次迭代且 $N > P$，迭代 i 将如何分配给某个线程?

最简单的分配方式采用轮询任务调度，一种静态调度策略，其中线程 p 获取迭代次数 $p \times (N/P)$, ...， $(p+1) \times (N/P) - 1$。这样做的好处是，如果某些数据在迭代之间被重复使用，这些数据将保留在执行该线程处理器的数据缓存中。另一方面，如果迭代涉及的工作量不同，该过程可能会因负载不平衡而受到静态调度的影响。在这种情况下，动态调度策略会更有效，即每个线程在完成当前迭代后立即开始处理下一个未处理的迭代。参见 2.10.2 节中的示例。

您可以通过 `schedule` 关键字控制 OpenMP 对循环迭代的调度；其值包括 `static` 和 `dynamic`。还可以指定一个 `chunksize`，用于控制分配给线程的迭代块大小。如果省略块大小，OpenMP 会将迭代分成与线程数量相同的块。

练习 2.22. 假设有 t 个线程，你的代码看起来像这样

```
for (i=0; i<N; i++) {
  a[i] = // some calculation
}
```

若指定分块大小为 1，迭代 $0, t, 2t, \ldots$ 会分配给第一个线程，$1, 1+t, 1+2t, \ldots$ 给第二个线程，以此类推。请从性能角度讨论为何这是种糟糕策略。提示：查阅伪共享的定义。理想的分块大小应是多少?

2.6.3 基于消息传递的分布式内存编程

虽然 OpenMP 程序及其他采用共享内存范式的程序仍与串行程序极为相似，但消息传递代码却非如此。在详细讨论消息传递接口（ MPI ）库之前，我们先观察这种并行代码编写方式的转变。

2.6.3.1 分布式编程中的全局视角与局部视角

并行算法在观察者视角与实际编程实现之间可能存在显著差异。以我们拥有一个由处理器 $\{P_i\}_{i=0..p-1}$ 组成的数组为例，每个处理器包含

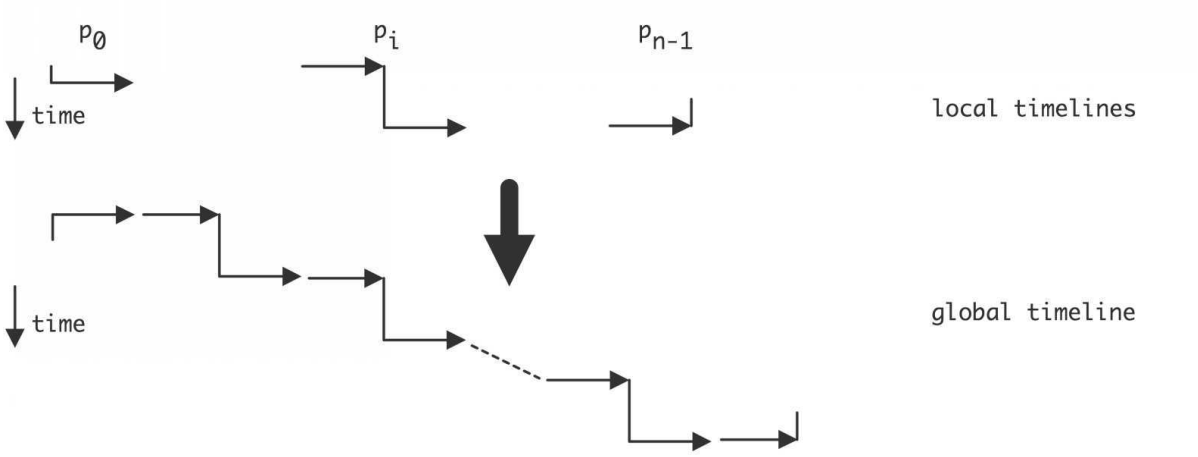


图 2.15：向右发送数据的算法局部视图与全局结果视图。

数组 x 和 y 中的一个元素，且 P_i 计算

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i > 0 \\ y_i \text{ unchanged} & i = 0 \end{cases} \tag{2.4}$$

其全局描述可以是

- 除最后一个处理器外，每个处理器 P_i 都将其 x 元素发送至 P_{i+1}；
- 除第一个处理器外，每个处理器 P_i 都会从其邻居 P_{i-1} 接收一个 x 元素，且
- 他们将其添加到他们的 y 元素中。

然而，通常我们无法用这些全局术语编写代码。在 SPMD 模型（章节 2.3.2）中，每个处理器执行相同的代码，整体算法是这些个体行为的结果。本地程序只能访问本地数据 —— 其他所有内容都需要通过发送和接收操作进行通信 —— 且处理器知道自己的编号。

一种可能的编写方式如下

- 如果我是处理器 0，则什么也不做。否则从左侧接收一个 y 元素，将其添加到我的 x 元素中。
- 如果我是最后一个处理器，则什么也不做。否则将我的 y 元素发送到右侧。

首先我们来看发送和接收是所谓的阻塞通信指令的情况：发送指令在发送的项目实际被接收之前不会完成，而接收指令会等待对应的发送。这意味着处理器之间的发送和接收必须仔细配对。我们现在将看到，这可能导致在编写高效代码的过程中出现各种问题。

上述解决方案如图 2.15 所示，其中展示了描述本地处理器代码的局部时间线及由此产生的全局行为。可见处理器并非同时工作：我们得到了序列化执行。

如果调换发送和接收操作的顺序会怎样 ?

- 若我非末位处理器，则将我的 x 元素向右发送 ;

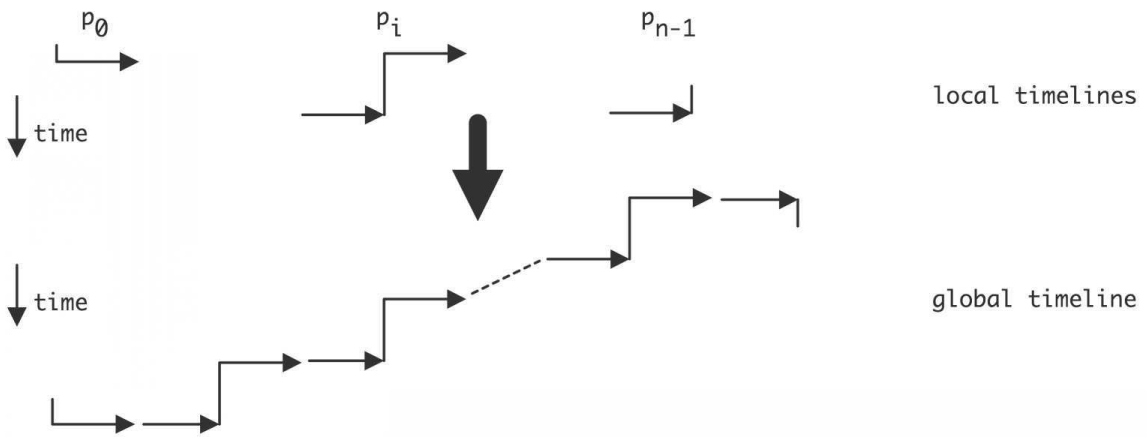


图 2.16：向右侧发送数据的算法局部视图及最终全局视图。

- If I am not the first processor, receive an x element from the left and add it to my y element.

This is illustrated in figure 2.16 and you see that again we get a serialized execution, except that now the processors are activated right to left.

若方程 2.4 中的算法是循环的：

$$\begin{cases} y_i \leftarrow y_i + x_{i-1} & i = 1 \ldots n-1 \\ y_0 \leftarrow y_0 + x_{n-1} & i = 0 \end{cases} \tag{2.5}$$

问题将更加严重。此时最后一个处理器因需向处理器 0 发送 x_{n-1} 而被阻塞，无法启动接收操作。这种因所有处理器相互等待而导致程序无法推进的情况，称为死锁。

实现高效代码的关键在于尽可能使通信并行化。毕竟该算法中不存在串行依赖关系。因此我们按以下方式编写算法：

- 若我是奇数编号的处理器，则先发送后接收；
- 若我是偶数编号的处理器，则先接收数据，再发送。

如图 2.17 所示，可见此时执行过程已实现并行化。

习题 2.23 重新观察并行归约示意图 2.3，其基本操作包括：

- 从相邻节点接收数据
- 将其与自身数据相加
- 将结果继续传递

如图所示，至少存在一个处理器不会继续发送数据，而其他处理器可能在执行可变次数的接收操作后才会转发其结果。编写节点代码以实现 SPMD 程序完成分布式归约。提示：将每个处理器编号用二进制表示。该算法使用的步骤数等于 the length of this bitstring.

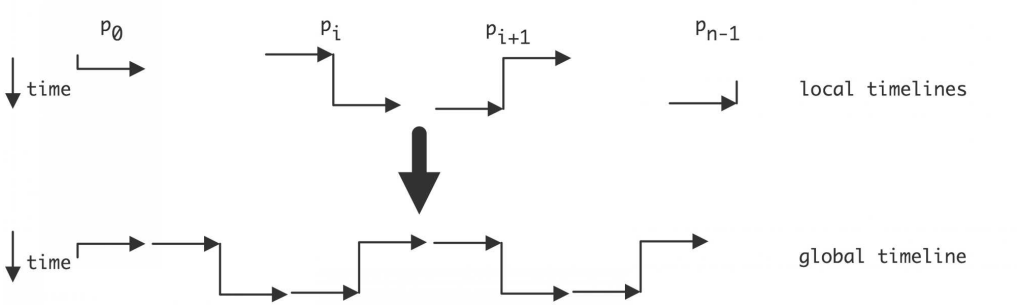


图 2.17：向右发送数据的算法局部视图及最终全局视图。

- 假设处理器接收到一条消息，用步数表示该消息源头的距离。
- 每个处理器最多发送一条消息。用二进制处理器编号表示这一事件发生的步骤。

2.6.3.2 阻塞式与非阻塞式通信

T阻塞指令的作用是防止数据在网络中堆积。如果发送指令
w若要在对应的接收操作开始前完成此处操作，网络必须临时存储数据
s在此期间存储于某处。考虑一个简单例子：

```
buffer = ... ;  // generate some data
send(buffer,0); // send to processor 0
buffer = ... ;  // generate more data
send(buffer,1); // send to processor 1
```

首次发送后，我们开始覆盖缓冲区。若其中数据未被接收，第一组值将不得不暂存于网络某处，这并不现实。通过使发送操作阻塞，数据会保留在发送方缓冲区，直到确保已复制到接收方缓冲区。

解决阻塞指令导致的顺序化或死锁问题的一种方案是使用非阻塞通信指令，这类指令包含显式数据缓冲区。使用非阻塞发送指令时，用户需为每次发送分配缓冲区，并检查何时可安全覆盖缓冲区。

```
buffer0 = ... ;   // data for processor 0
send(buffer0,0);  // send to processor 0
buffer1 = ... ;   // data for processor 1
send(buffer1,1);  // send to processor 1...
// wait for completion of all send operations.
```

2.6.3.3 MPI 库

如果说 OpenMP 是共享内存编程的方式，那么消息传递接口（ MPI ） [174] 则是分布式内存编程的标准解决方案。MPI （ ‘Message Passing Interface’ ）是一套库接口规范，用于在不共享数据的进程间传递数据。MPI 例程大致可分为以下几类：

- 进程管理。包括查询并行环境及构建处理器子集的功能。
- 点对点通信。这是一组两个进程交互的调用接口，主要是发送和接收调用的各种变体。
- 集合通信。这些例程涉及所有处理器（或指定子集的全部成员）。例如广播调用，即一个处理器将其数据共享给所有其他处理器；或聚集调用，即一个处理器从所有参与处理器收集数据。

让我们探讨如何用 MPI 实现 OpenMP 示例。注意：本章并非 MPI 编程教程，因此示例将省略 MPI 调用的诸多细节。若需学习 MPI 编程，请参考 [86, 89, 87] 或本系列第二卷。

首先，我们不再分配

```
double a[ProblemSize];
```

but

```
double a[LocalProblemSize];
```

其中局部大小约为全局大小的 $1/P$ 比例。（实际应用中需考虑是否要求分布尽可能均匀，或存在特定倾向性。）

该并行循环是简单并行的，唯一区别在于它现在仅操作数组的一部分：

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i];
}
```

然而，若循环涉及基于迭代次数的计算，则需将其映射至全局值：

```
for (i=0; i<LocalProblemSize; i++) {
  a[i] = b[i]+f(i+MyFirstVariable);
}
```

(我们假设每个进程已通过某种方式计算出 `LocalProblemSize` 和 `MyFirstVariable` 的值。) 此时局部变量自动成为局部，因为每个进程拥有其自身的实例：

```
for (i=0; i<LocalProblemSize; i++) {
  t = b[i]*b[i];
  a[i] = sin(t) + cos(t);
}
```

然而，共享变量更难实现。由于每个进程都有自己的数据，本地累加必须显式组装：
模拟过程需要显式地进行组装：

```
for (i=0; i<LocalProblemSize; i++) {
  s = s + a[i]*b[i];
}
MPI_Allreduce(s,globals,1,MPI_DOUBLE,MPI_SUM);
```

‘reduce’ 操作将所有本地值 `s` 求和到一个变量 `globals` 中，该变量在每个处理器上接收相同的值。这被称为集合操作。

让我们把这个例子稍微复杂化一些 :

```
for (i=0; i<ProblemSize; i++) {
  if (i==0)
    a[i] = (b[i]+b[i+1])/2
  else if (i==ProblemSize-1)
    a[i] = (b[i]+b[i-1])/2
  else
    a[i] = (b[i]+b[i-1]+b[i+1])/3
}
```

If we had shared memory, we could write the following parallel code:

```
for (i=0; i<LocalProblemSize; i++) {
  bleft = b[i-1]; bright = b[i+1];
  a[i] = (b[i]+bleft+bright)/3
}
```

要将此转化为有效的分布式内存代码，首先需考虑 `bleft` 和 `bright` 需要从不同处理器获取以计算 `i==0` (`bleft`) 和 `i==LocalProblemSize-1` (`bright`) 这一事实。我们通过与左右相邻处理器进行数据交换操作实现：

```
// get bfromleft and bfromright from neighbor processors、
 thenfor (i=0; i<LocalProblemSize; i++) {
if (i==0) bleft=bfromleft;else bleft = b[i-1]
if (i==LocalProblemSize-1) bright=bfromright;
else bright = b[i+1];a[i] = (b[i]+bleft+bright)/3}
```

O获取相邻处理器数值的操作步骤如下。首先需要查询当前处理器编号，以便我们
c能与编号增减一的处理器建立通信连接。

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID);
MPI_Sendrecv
(/* to be sent:  */ &b[LocalProblemSize-1],
/* destination  */ myTaskID+1,
/* to be recvd: */ &bfromleft,
/* source:      */ myTaskID-1,
/* some parameters omitted */);
MPI_Sendrecv(&b[0],myTaskID-1,&bfromright,
 /* ... */ );
```

这段代码仍存在两个问题。首先，sendrecv 操作需为首尾处理器设置异常处理。可通过如下方式优雅实现：

```
MPI_Comm_rank(MPI_COMM_WORLD,&myTaskID);
MPI_Comm_size(MPI_COMM_WORLD,&nTasks);
if (myTaskID==0) leftproc = MPI_PROC_NULL;
  else leftproc = myTaskID-1;
if (myTaskID==nTasks-1) rightproc = MPI_PROC_NULL;
  else rightproc = myTaskID+1;
MPI_Sendrecv( &b[LocalProblemSize-1], &bfromleft,  rightproc );
MPI_Sendrecv( &b[0],                  &bfromright, leftproc);
```

Exercise2.24. 该代码仍存在一个问题：未考虑原始全局版本中的边界条件。请给出解决该问题的代码。

当不同进程需执行不同操作时（例如一个进程需向另一个发送数据），MPI 会变得复杂。问题在于每个进程执行相同的可执行文件，因此需同时包含发送和接收指令，具体执行取决于进程的 rank 值。

```
if (myTaskID==0) {
MPI_Send(myInfo,1,MPI_INT,/* to: */ 1,/* labeled: */,0,
MPI_COMM_WORLD);} else {
MPI_Recv(myInfo,1,MPI_INT,/* from: */ 0,/* labeled: */,0,
/* not explained here: */&status, MPI_COMM_WORLD);}
```

2.6.3.4 阻塞通信

尽管 MPI 因其感知难度和显式程度有时被称为 ‘并行编程的汇编语言’，但实际学习起来并不那么困难，大量科学代码采用 MPI 就是明证。使 MPI 使用稍显复杂的主要问题在于缓冲区管理和阻塞语义。

这些问题相互关联，源于一个理想情况：数据不应同时存在于两个位置。让我们简要考虑处理器 1 向处理器 2 发送数据时的情况。最安全的策略是让处理器 1 执行发送指令，然后等待处理器 2 确认数据已成功接收。这意味着处理器 1 会暂时被阻塞，直到处理器 2 实际执行其接收指令且数据通过网络传输完成。这是 `MPI_Send` 和 `MPI_Recv` 调用的标准行为，这些调用被称为使用阻塞通信。

或者，处理器 1 可以将其数据放入缓冲区，告知系统确保数据在某个时间点发送，稍后再检查缓冲区是否可以安全重用。第二种策略被称为非阻塞通信，它需要使用临时缓冲区。

2.6.3.5 集体操作

在上述示例中，您看到了 `MPI_Allreduce` 调用，它计算了全局总和并将结果留在每个处理器上。还有一个本地版本 `MPI_Reduce`，仅在一个处理器上计算结果。这些调用是集体操作或集合体的示例。集合体包括：

r规约操作：每个处理器持有一个数据项，需要通过加法、乘法、取最大值或最小值等算术运算将这些数据项合并。结果可以保留在单个处理器上，或分发给所有处理器（此时称为**全规约**操作）。

广播：单个处理器持有的数据项需要分发给所有处理器。**聚集**：每个处理器持有一个数据项，这些数据项需被收集到一个数组中（不进行加法等运算合并）。结果可以保留在单个处理器上，或分发给所有处理器（此时称为**全聚集**）。

散播：单个处理器持有一个数据项数组，每个处理器接收该数组中的一个元素。**全交换**：每个处理器持有一个数据项数组，需将该数组分发给所有其他处理器。

集合操作是阻塞式的（参见章节 2.6.3.4），不过 MPI 3.0 （当前仅为草案版本）将引入非阻塞式集合操作。我们将在章节 7.1 详细分析集合操作的开销。

2.6.3.6 非阻塞通信

在简单的计算机程序中，每条指令的执行都需要一定时间，具体取决于处理器内部的状态。而在并行程序中，情况则更为复杂。发送操作的最基本形式是声明需要发送某个数据缓冲区，随后程序执行将暂停，直到该缓冲区被安全发送并被另一处理器接收。这类操作被称为非本地操作，因为它依赖于其他进程的行为；同时也被称为阻塞通信操作，因为执行会暂停直到特定事件发生。

阻塞操作的缺点在于可能导致死锁。在消息传递的上下文中，死锁描述的是进程等待一个永远不会发生的事件的情况；例如，某个进程可能正在等待接收消息，而该消息的发送者却在等待其他事件。当两个进程互相等待时就会发生死锁，更一般地说，如果存在一个进程循环，其中每个进程都在等待循环中的下一个进程，就会产生死锁。示例：

```
if ( /* this is process 0 */ )
   // wait for message from 1
else if ( /* this is process 1 */ )
   // wait for message from 0
```

在此处进行阻塞式接收会导致死锁。即使未发生死锁，也可能造成处理器大量空闲时间，因为它们等待期间未执行任何有效工作。但这种方式具有明确缓冲区可重用时机的优势：操作完成后，能确保数据已安全送达接收端。

通过使用非阻塞通信操作，可以避免阻塞行为，但代价是使缓冲区语义复杂化。非阻塞发送 (`MPI_Isend`) 声明需要发送数据缓冲区，但不会等待对应接收操作完成。另有操作 `MPI_Wait` 会实际阻塞直至接收完成。这种发送与阻塞解耦的优势在于，现在可以编写如下代码：

```
MPI_ISend(somebuffer,&handle); // start sending, and
// get a handle to this particular communication
{ ... }  // do useful work on local data
MPI_Wait(handle); // block until the communication is completed;
{ ... }  // do useful work on incoming data
```

若运气稍佳，本地操作耗时将超过通信时间，这样你便完全消除了通信时间。

除了非阻塞发送外，还存在非阻塞接收。典型的代码片段如下所示：

```
MPI_ISend(sendbuffer,&sendhandle);
MPI_IReceive(recvbuffer,&recvhandle);
{ ... }  // do useful work on local data
MPI_Wait(sendhandle); Wait(recvhandle);
{ ... }  // do useful work on incoming data
```

练习 2.25. 重新审视方程 (2.5)，给出使用非阻塞发送与接收解决问题的伪代码。相较于阻塞式解决方案，此代码有何劣势？

2.6.3.7 MPI 版本 *1* 、 *2* 及 *3*

首个 MPI 标准 [154] 存在若干显著缺失，这些功能在 MPI 2 标准中得以补充 [88]。其中之一涉及并行输入 / 输出：当时没有为多进程访问同一文件提供支持，即使底层硬件允许这样做。独立的 MPI-I/O 项目现已被纳入 MPI-2 标准。本书将探讨并行 I/O 相关内容。

MPI 缺失的第二个功能是进程管理（虽然早于 MPI 的 *PVM* [50, 71] 已具备该功能）：无法创建新进程使其加入并行运行。

最后，MPI-2 支持单边通信：一个进程将数据直接放入另一进程的内存，而无需接收进程执行实际接收指令。我们将在 2.6.3.8 小节进行简要讨论。

到了 MPI-3 版本，该标准新增了多项特性，如非阻塞集合通信、邻域集合通信以及性能分析接口，单边通信机制也得到更新。

2.6.3.8 单边通信

MPI 中编写匹配的发送和接收指令的方式因多种原因并非理想选择。首先，它要求程序员对同一数据描述两次，分别在发送和接收调用中各一次。其次，若要避免死锁，需要对通信进行相当精确的编排；而使用异步调用的替代方案编程繁琐，需要程序管理大量缓冲区。最后，它要求接收处理器知道预期会有多少条传入消息，这在非规则应用中可能很棘手。如果能从另一处理器拉取数据，或反之将数据推送到另一处理器，而无需该处理器显式参与，事情会简单得多。

这种编程方式因某些硬件支持远程直接内存访问（*RDMA*）而得到进一步推广。早期的例子包括 *CrayT3E*。如今，单边通信通过 MPI-2 库的整合已广泛可用；参见章节 2.6.3.7。

让我们以数组值平均为例，简要了解一下 MPI-2 中的单边通信：

$$\forall_i : a_i \leftarrow (a_i + a_{i-1} + a_{i+1})/3.$$

MPI 并行代码将如下所示

```
// do data transfer
a_local = (a_local+left+right)/3
```

传输需要完成的任务很明确： `a_local` 变量需成为更高一级处理器上的 `left` 变量，以及更低一级处理器上的 `right` 变量。

首先，处理器需明确声明哪些内存区域可用于单边传输，即所谓的 ‘窗口’。本例中，这些区域包含处理器上的 `a_local`、 `left` 和 `right` 变量：

```
MPI_Win_create(&a_local,...,&data_window);
MPI_Win_create(&left,....,&left_window);
MPI_Win_create(&right,....,&right_window);
```

当前代码有两种选择：可以推送数据出去

```
target = my_tid-1;
MPI_Put(&a_local,...,target,right_window);
target = my_tid+1;
MPI_Put(&a_local,...,target,left_window);
```

或是拉取数据进来

```
data_window = a_local;
source = my_tid-1;
MPI_Get(&right,...,data_window);
source = my_tid+1;
MPI_Get(&left,...,data_window);
```

若 Put 与 Get 调用是阻塞式的，上述代码将具有正确的语义；参见章节 2.6.3.4。然而，单边通信的部分吸引力在于它更易于表达通信行为，为此通常采用非阻塞语义。

非阻塞单边调用的难点在于必须显式确保通信成功完成。例如，若某处理器对另一处理器执行单边 *put* 操作，接收方处理器无法验证数据是否已送达，甚至无法确认传输是否已启动。因此需要在程序中插入全局屏障，每个包都有其独立实现。MPI-2 中相关调用是 `MPI_Win_fence` 例程。这些屏障实际上将程序执行划分为超步；参见章节 2.6.8。

Charm++ 包中使用了另一种形式的单边通信；参见章节 2.6.7。

2.6.4 混合共享 / 分布式内存计算

现代架构通常是共享内存与分布式内存的混合体。例如，集群在节点层面是分布式的，但节点上的插槽和核心则共享内存。再往上一层，每个插槽可以拥有共享的 L3 缓存，但独立的 L2 和 L1 缓存。直观上看，混合使用共享和分布式编程技术似乎能产生与架构最优匹配的代码。本节将讨论此类混合编程模型及其效能。

集群的一种常见配置采用分布式内存节点，每个节点包含多个共享内存的插槽。这意味着使用 MPI 在节点间进行通信（节点间通信），而利用 OpenMP 实现节点内的并行处理（节点内通信）。

实践中具体实现方式如下：

- 每个节点上启动单个 MPI 进程（而非每个核心一个）；
- 该 MPI 进程随后通过 OpenMP （或其他线程协议）生成与节点上独立插槽或核心数量相等的线程。
- OpenMP 线程随后可访问该节点的共享内存。

另一种替代方案是在每个核心或插槽上运行一个 MPI 进程，并通过消息传递完成所有通信，即使是在能访问同一共享内存的进程之间。

备注 7 出于亲和性考虑，可能更倾向于每个插槽启动一个 *MPI* 进程而非每个节点。这实质上不会改变上述论点。

这种混合策略听起来不错，但实际情况更为复杂。

- MPI 进程间的消息传递看似比共享内存通信开销更大。然而优化版的 MPI 通常能检测到进程是否位于同一节点，并会用简单的数据拷贝替代消息传递。反对使用 MPI 的唯一理由是每个进程拥有独立数据空间，这会因缓冲区分配和数据副本导致内存开销。

- 线程方案更灵活：若代码某部分需要更多单进程内存，OpenMP 方法可限制该部分的线程数。但另一方面，线程的灵活调度会带来 MPI 固定进程所没有的操作系统开销。

- 共享内存编程在概念上简单，但可能存在意想不到的性能陷阱。例如，两个进程的性能现在可能因维护缓存一致性和伪共享的需求而受到阻碍。

另一方面，混合方法由于能捆绑消息而具有一定优势。例如，如果一个节点上的两个 MPI 进程向另一个节点上的两个进程各发送一条消息，总共会有四条消息；在混合模型中，这些消息会被捆绑成一条。

练习 2.26. 分析上述最后一项的讨论。假设两个节点之间的带宽仅够同时维持一条消息。混合模型相比纯分布式模型的成本节省是多少？提示：分别考虑带宽和延迟。

这种 MPI 进程的捆绑可能因更深层次的技术原因而具有优势。为了支持握手协议，每个 MPI 进程需要为其他每个进程预留少量缓冲区空间。当进程数量较多时，这可能成为限制，因此在高核数处理器（如 *IntelXeon Phi* ）上，捆绑显得尤为吸引人。

MPI 库明确声明了其所支持的线程类型：您可以查询是否支持多线程、是否所有 MPI 调用必须源自单个线程或每次仅允许一个线程调用，或者

是否允许线程完全自由地调用 MPI 接口。

2.6.5 并行语言

缓解并行编程困难的一种方法是设计显式支持并行性的语言。存在多种实现路径，我们将通过案例进行说明。

- 某些语言的设计理念源于科学计算中大量操作本质上是数据并行（参见章节 2.5.1 ）。例如高性能 *Fortran(HPF)* （章节 2.6.5.3 ）采用的数组语法，允许将数组加法等操作直接表示为 `A = B+C`。这种语法不仅简化了编程，更重要的是在抽象层面明确了操作定义，使底层能自主决定并行实现方案。但 HPF 仅能表达最简单的规则数组数据并行，非规则数据并行更为复杂；*Chapel* 语言（章节 2.6.5.5 ）尝试解决这一问题。

- 并行语言中的另一个概念（与前一个概念未必正交）是
分区全局地址空间（ PGAS ）模型：与 MPI 模型不同，该模型仅存在单一地址空间，但该地址空间被分区且每个分区与特定线程或进程具有亲和性。因此，该模型同时涵盖 SMP 和分布式共享内存。典型的 PGAS 语言 *Unified Parallel C (UPC)* 允许编写大部分代码与常规 C 程序相似的并行程序，只需通过声明主要数组在处理器间的分布方式即可实现并行执行。

2.6.5.1 讨论

P并行语言通过将通信操作简化为复制或算术运算，有望降低并行编程难度。
c但这种方式也容易导致用户编写出低效代码，例如因频繁发送小规模消息而影响性能。
u编写可能不够高效的代码，例如通过引发大量小消息。

例如，考虑数组 `a,b` 这些数组已在处理器间水平分区，并进行了移位（参见图 2.18 ）：

```
for (i=0; i<N; i++)
  for (j=0; j<N/np; j++)
    a[i][j+joffset] = b[i][j+1+joffset]
```

若此代码在共享内存机器上执行，将会很高效，但在分布式场景下的简单翻译会导致每次 `i` 循环迭代中仅有一个数字被通信。显然，这些操作可合并为单个缓冲区的发送 / 接收操作，但编译器通常无法完成此转换。因此，用户实际上被迫重新实现 MPI 实现中所需的阻塞操作：

```
for (i=0; i<N; i++)
  t[i] = b[i][N/np+joffset]
for (i=0; i<N; i++)
  for (j=0; j<N/np-1; j++) {
    a[i][j] = b[i][j+1]
```

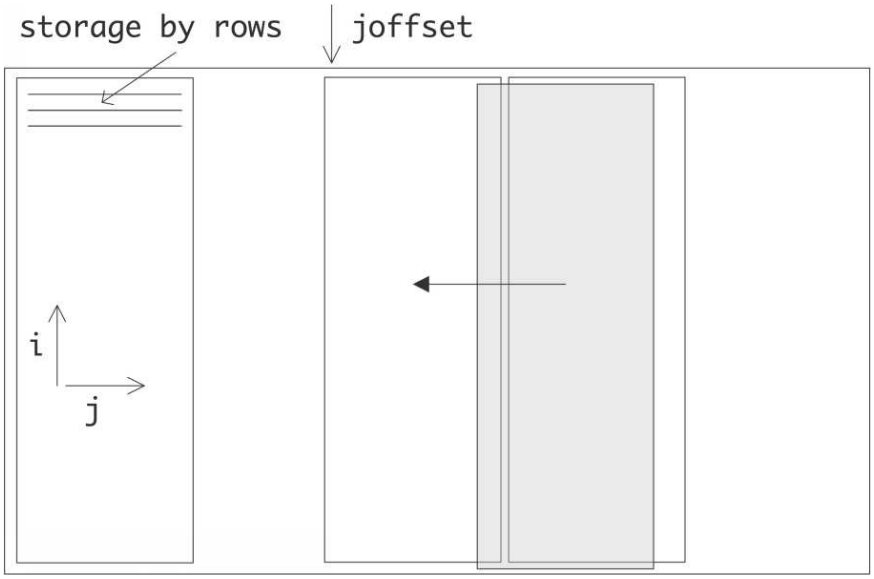


图 2.18：需要通信的数据移位。

```
    a[i][N/np] = t[i]
  }
```

另一方面，某些机器通过全局内存硬件支持直接内存拷贝。在这种情况下，即使物理内存是分布式分布的，PGAS 语言也可能比显式消息传递更高效。

2.6.5.2 统一并行 C 语言

统一并行 C 语言（ UPC ） [181] 是对 C 语言的扩展。其主要并行性来源是数据并行，编译器会发现对数组操作的独立性，并将它们分配给不同的处理器。该语言具有扩展的数组声明，允许用户指定数组是按块分区，还是以轮询方式分区。

以下 UPC 程序执行向量 - 向量加法。

```
//vect_add.c#include <upc_relaxed.h>
#define N 100*THREADSshared int v1[N],
 v2[N],  v1plusv2[N];void main() {
int i;
for(i=MYTHREAD; i<N; i+=THREADS)
v1plusv2[i]=v1[i]+v2[i];}
```

2. 并行计算

同一程序采用显式并行循环结构：

```
//vect_add.c#include <upc_relaxed.h>
#define N 100*THREADSshared int v1[N],
 v2[N],  v1plusv2[N];void main(){
int i;upc_forall(i=0; i<N; i++; i)
v1plusv2[i]=v1[i]+v2[i];}
```

在理念上与 UPC 类似，但基于 Java 而非 C 语言。

2.6.5.3 高性能 *Fortran*

High Performance Fortran[3] (HPF) 是 Fortran 90 的扩展，其构造支持并行
c由高性能 Fortran 论坛（ HPFF ）发布的《计算科学》。HPFF 由莱斯大学的 Ken Kennedy 召集并
c担任主席。HPF 报告的首个版本于 1993 年发布。

B基于 Fortran 90 引入的数组语法，HPF 采用数据并行计算模型，
s支持将单个数组计算的工作负载分摊到多个处理器上。这使得在
iSIMD 和 MIMD 架构上都能高效实现。HPF 的特性包括：

- 新增的 Fortran 语句，如 FORALL，以及创建 PURE （无副作用）过程的能力；
- 使用编译器指令来推荐数组数据的分布方式；
- 外部过程接口，用于与非 HPF 并行过程（如使用消息传递的过程）对接；
- 额外的库例程，包括环境查询、并行前缀 / 后缀（如 “ 扫描 ” ）、数据分散和排序操作。

Fortran 95 整合了多项 HPF 功能。尽管一些供应商在 1990 年代确实将 HPF 纳入其编译器中，但某些方面难以实现且实用性存疑。此后，大多数供应商和用户转向基于 OpenMP 的并行处理。然而，HPF 仍具影响力。例如，为即将发布的 Fortran-2008 标准提出的 BIT 数据类型直接采用了 HPF 中的多个新内置函数。

2.6.5.4 协同数组 *Fortran*

协同数组 Fortran （ CAF ）是对 Fortran 95/2003 语言的扩展。支持并行性的主要机制是对数组声明语法的扩展，通过额外维度指示并行分布。例如，在

```
Real,dimension(100),codimension[*] :: X
Real :: Y(100)[*]
```

3. 本节引自 Wikipedia

```
Real :: Z(100,200)[10,0:9,*]
```

数组 X 、 Y 在每个处理器上各有 100 个元素。数组 Z 的行为表现为可用处理器排列在三维网格上，其中两侧尺寸固定，第三侧可调整以适应可用处理器数量。

处理器间的通信现通过沿描述处理器网格的（共）维度进行数据拷贝实现。Fortran 2008 标准已包含共数组特性。

2.6.5.5 Chapel 语言

Chapel [31] 是一种新型并行编程语言 4，由 Cray 公司作为 DARPA 主导的高效能计算系统计划（ HPCS ）的一部分开发。该语言旨在提升高端计算机用户的生产力，同时也可作为可移植的并行编程模型，适用于商用集群或桌面多核系统。Chapel 致力于大幅提升大规模并行计算机的可编程性，并在性能与可移植性方面达到或超越当前 MPI 等编程模型的水平。

Chapel 通过高层次抽象支持数据并行、任务并行、并发及嵌套并行的多线程执行模型。其 locale 类型允许用户指定并推理数据与任务在目标架构上的分布，以优化局部性。Chapel 支持具有用户自定义实现的全局视图数据聚合，使得分布式数据结构的操作能以自然方式表达。与许多早期高级并行语言不同，Chapel 采用多分辨率设计理念，用户可先编写高度抽象的代码，再逐步添加细节直至满足机器级需求。该语言通过面向对象设计、类型推断和泛型编程特性，支持代码复用与快速原型开发。

Chapel 基于首创原则而非扩展现有语言设计而成。作为一种命令式块结构语言，它力求让 C 、 C++ 、 Fortran 、 Java 、 Perl 、 Matlab 等流行语言用户快速上手。尽管 Chapel 融合了多种前驱语言的概念与语法，其并行特性主要受 ZPL 、高性能 Fortran （ HPF ）以及 Cray MTA 对 C 和 Fortran 的扩展直接影响。

以下是 Chapel 中的向量 - 向量加法示例:

```
const BlockDist= newBlock1D(bbox=[1..m], tasksPerLocale=...);
const ProblemSpace: domain(1, 64)) distributed BlockDist = [1..m];
var A, B, C: [ProblemSpace] real;forall(a, b, c) in(A, B, C) do
a = b + alpha * c;
```

4. Thissection quoted fromthe Chapel homepage.

2.6.5.6 Fortress

Fortress [65] 是由 Sun Microsystems 开发的一种编程语言。Fortress[5] 旨在通过多种方式使并行性更易于处理。首先，并行性是默认设置。这旨在推动工具设计、库设计和程序员技能向并行化方向发展。其次，该语言设计得更适合并行性。副作用被不鼓励，因为副作用需要同步以避免错误。Fortress 提供了事务，因此程序员无需面对确定锁顺序或调整锁定代码的任务，以确保足够正确性，同时又不至于影响性能。Fortress 的循环结构与库一起，将 “ 迭代 ” 内外翻转；不是由循环指定如何访问数据，而是由数据结构指定如何运行循环，并且聚合数据结构被设计为可以分解为可有效调度并行执行的大块部分。Fortress 还包括来自其他语言的特性，旨在普遍提高生产力 —— 测试代码和方法，与被测试代码绑定；可以在代码运行时选择性地检查的契约；以及可能运行成本过高但可以提供给定理证明器或模型检查器的属性。此外，Fortress 还包括安全语言特性，如检查数组边界、类型检查和垃圾回收，这些特性在 Java 中已被证明是有用的。
Fortress 的语法设计尽可能类似于数学语法，以便任何在规范中使用数学解决问题的人都可以编写一个与其原始规范明显相关的程序。

2.6.5.7 X10

X10 是 IBM 与学术合作伙伴共同开发的一种实验性新语言，目前正处于研发阶段。X10 项目隶属于 IBM PERCS 计划（高效易用可靠计算机系统），该计划是 DARPA 高生产力计算机系统项目的一部分。PERCS 项目专注于硬件 - 软件协同设计方法，旨在整合芯片技术、架构、操作系统、编译器、编程语言及编程工具等领域的前沿成果，以构建新型可适应、可扩展的系统，目标是在 2010 年前将并行应用程序的开发效率提升一个数量级。

X10 旨在通过开发新的编程模型（结合集成到 Eclipse 中的工具集）及优化可扩展并行性的运行时管理实现技术，为提升开发效率做出贡献。作为一种类型安全、现代化、并行分布式的面向对象语言，X10 力求让 Java™ 程序员能够轻松上手。其目标平台是未来由多核 SMP 芯片（具有非均匀内存层次结构）节点构建的低端与高端系统，这些节点通过可扩展集群配置互联。作为分区全局地址空间（ PGAS ）语言家族成员，X10 的核心特性包括：通过 places 形式显式物化局部性；通过 async 、 future 、 foreach 和 ateach 结构实现轻量级活动；支持终止检测（ finish ）和分阶段计算（ clocks ）的结构；采用无锁同步（ atomic 块）机制；以及全局数组与数据结构的操作能力。

2.6.5.8 Linda

正如目前应当明确的那样，数据的处理是并行编程中最为关键的方面，其重要性远超算法考量。编程系统 *Linda* [72, 73], 同样

5. 本节引自 Fortress 主页。

称为协调语言，其设计目的是显式处理数据。Linda 本身并非一种语言，但可以且已被整合到其他语言中。

Linda 的基本概念是元组空间：通过为数据添加标签，将其加入全局可访问的信息池中。进程随后通过标签值检索数据，无需知晓是哪些进程将数据添加至元组空间。

Linda 主要针对与 *HPC* 不同的计算模型：它满足异步通信进程的需求。然而，它也被用于科学计算 [46]。例如，在热方程并行模拟中（第 4.3 节），处理器可将数据写入元组空间，相邻进程无需知晓来源即可检索其幽灵区域。因此，Linda 成为实现单边通信的一种方式。

2.6.5.9 Global Arrays 库

Global Arrays 库（`http://www.emsl.pnl.gov/docs/global/`）是单边通信的另一实例，事实上它早于 MPI 出现。该库以笛卡尔积数组 [6] 作为主要数据结构，分布在相同或更低维度的处理器网格上。通过库调用，任何处理器均能以 `put` 或 `get` 操作访问数组中的任意子块。这些操作是非集合式的。与任何单边协议一样，需通过屏障同步确保发送 / 接收操作完成。

2.6.6 基于操作系统的方法

可以设计一种具有共享地址空间的架构，并由操作系统处理数据移动。Kendall Square 计算机 [119] 采用了一种名为 ‘全缓存’ 的架构，其中没有任何数据直接与任何处理器关联。相反，所有数据都被视为缓存在处理器上，并根据需求通过网络移动，就像常规 CPU 中数据从主存移动到缓存一样。这一理念类似于当前 SGI 架构中的 NUMA 支持。

2.6.7 主动消息

MPI 范式（章节 2.6.3.3）传统上基于双向操作：每次数据传输都需要显式的发送和接收操作。这种方法对于相对简单的代码效果良好，但对于复杂问题，协调所有数据移动变得困难。简化方法之一是使用主动消息。这被用于包 *Charm++*[115] 中。

使用主动消息时，一个处理器可以向另一个处理器发送数据，而无需第二个处理器执行显式的接收操作。相反，接收方声明处理传入数据的代码，即面向对象术语中的 ‘方法’，发送处理器调用此方法并传递要发送的数据。由于发送处理器实际上激活了另一处理器上的代码，因此这也被称为远程方法调用。这种方法的一大优势是更容易实现通信与计算的重叠。

6. 这意味着如果数组是三维的，它可以用三个整数 n_1、n_2、n_3 来描述，每个点都有一个坐标 (i_1, i_2, i_3)，其中 $1 \le i_1 \le n_1$ 等等。

例如，考虑使用三对角矩阵进行矩阵 - 向量乘法

$$\forall_i : y_i \leftarrow 2x_i - x_{i+1} - x_{i-1}.$$

参见章节 4.2.2 了解该问题在偏微分方程中的起源。假设每个处理器恰好拥有一个索引 i，对应的 MPI 代码可能如下：

```
if ( /* I am the first or last processor */ )
   n_neighbors = 1;
else
   n_neighbors = 2;

/* do the MPI_Isend operations on my local data */

sum = 2*local_x_data;
received = 0;
for (neighbor=0; neighbor<n_neighbors; neighbor++) {
   MPI_WaitAny( /* wait for any incoming data */ )
   sum = sum - /* the element just received */
   received++
   if (received==n_neighbors)
      local_y_data = sum
}
```

使用活动消息时，代码呈现如下形式

```
void incorporate_neighbor_data(x) {sum = sum-x;
if (received==n_neighbors)local_y_data = sum}sum = 2*local_xdata;
received = 0;
all_processors[myid+1].incorporate_neighbor_data(local_x_data);
all_processors[myid-1].incorporate_neighbor_data(local_x_data);
```

2.6.8 批量同步并行

MPI 库（章节 2.6.3.3 ）可以生成非常高效的代码。其代价是程序员需要极其详细地指定通信细节。在光谱的另一端，PGAS 语言（章节 2.6.5 ）对程序员要求极低，但回报的性能也有限。一种寻求折中的尝试是批量同步并行（*BSP*）模型 [182, 173]。在此模型中，程序员需明确通信内容，但无需指定其顺序。

BSP 模型将程序组织为一系列超步，每个超步以屏障同步结束。在一个超步中启动的所有通信都是异步的，并依赖屏障来实现完成。这简化了编程并消除了死锁的可能性。此外，所有通信都属于单边通信类型。

练习 2.27. 考虑章节 2.1 中的并行求和示例。论证 BSP 实现需要 $\log_2 n$ 个超步。

由于 BSP 模型通过超级步结束时的屏障实现处理器同步，它能对并行算法进行简单的成本分析。

BSP 模型的另一个特点是采用过度分解策略 —— 为每个处理器分配多个进程，并采用随机分布方式分配数据和任务。这种设计基于统计学论证，可有效缓解负载不均问题。若有 p 个处理器，在某超级步中进行 p 次远程访问时，大概率会出现某个处理器需处理 $\log p / \log \log p$ 次访问，而其他处理器零负载的情况。这种负载失衡会随处理器数量增加而恶化。反之，若进行 $p \log p$ 次访问（例如每个处理器分配 $\log p$ 个进程），则最大访问次数大概率维持在 $3 \log p$ 次，这意味着负载均衡始终保持在理想状态的常数倍范围内。

BSP 模型通过 BSPlib 实现 [105]。其他系统若采用超级步概念也可视为类 BSP 系统，例如谷歌的 *Pregel* [143]。

2.6.9 数据依赖

若两条语句引用同一数据项，我们称这两条语句间存在数据依赖关系。此类依赖限制了语句执行顺序可被重排的程度。该领域的研究可能始于 1960 年代，当时处理器可通过乱序执行语句来提高吞吐量。语句重排的约束在于执行必须遵循程序顺序语义：最终结果必须与严格按程序书写顺序执行语句时的结果一致。

语句顺序问题（进而引发数据依赖问题）在以下场景中尤为突出：

- 一个并行化编译器必须分析源代码以确定允许的转换操作；
- 若使用 OpenMP 指令对串行代码进行并行化，您需要自行完成此类分析。

以下两类活动需要进行此类分析：

- 当循环被并行化时，迭代不再按照程序顺序执行，因此我们必须检查依赖关系。
- 任务的引入还意味着程序的某些部分可以按照与顺序执行中不同的顺序执行。

依赖分析中最简单的情况是检测循环迭代是否可以独立执行。如果同一数据项在两个不同迭代中被读取，迭代当然是独立的；但如果同一项在一个迭代中被读取而在另一个迭代中被写入，或在两个不同迭代中被写入，则需进一步分析。

数据依赖的分析可由编译器执行，但编译器出于必要性采取保守方法。这意味着迭代可能是独立的，但编译器无法识别这一点。因此，OpenMP 将这一责任转移给程序员。

接下来我们将详细讨论数据依赖问题。

2.6.9.1 数据依赖类型

三种依赖关系分别为 :

- 流依赖（ ‘读后写’ ）；
- 反依赖（ ‘读后写’ ）；以及
- 输出依赖（ ‘写后写’ ）。

这些依赖关系可以在标量代码中研究，事实上编译器会这样做以确定语句是否可以重新排列，但我们主要关注它们在循环中的出现，因为在科学计算中大部分工作都出现在那里。

流依赖流依赖，或者说写后读，如果读写发生在同一循环迭代中则不成问题：

```
for (i=0; i<N; i++) {
  x[i] = .... ;
  .... = ... x[i] ... ;
}
```

另一方面，如果读操作发生在后续迭代中，则没有简单的方法可以并行化或向量化该循环：

```
for (i=0; i<N; i++) {
  .... = ... x[i] ... ;
  x[i+1] = .... ;
}
```

这通常需要重写代码。

Exercise 2.28. Consider the code

`for (i=0; i<N; i++) {a[i] = f(x[i]);x[i+1] = g(b[i]);}` 其中 `f()` 和 `g()` 表示算术表达式，且不进一步依赖于 `x` 或 `i`。证明若允许使用临时数组，此循环可被并行化 / 向量化。

反依赖最简单的反依赖或写后读情况是归约操作：

```
for (i=0; i<N; i++) {
  t = t + .....
}
```

可通过显式声明该循环为归约操作来处理，或采用 7.1.2 节所述的其他策略。

若读写操作针对数组，情况则更为复杂。以下代码片段中的迭代：

```
for (i=0; i<N; i++) {
  x[i] = ... x[i+1] ... ;
}
```

不能随意按任意顺序执行。然而，从概念上讲并不存在依赖关系。我们可以通过引入临时数组来解决这个问题：

```
for (i=0; i<N; i++)
  xtmp[i] = x[i];
for (i=0; i<N; i++) {
  x[i] = ... xtmp[i+1] ... ;
}
```

这是一个编译器不太可能执行的转换示例，因为它会极大影响程序的内存需求。因此，这项工作留给了程序员。

输出依赖 关于输出依赖 或写后写的情况本身并不会发生：如果一个变量被连续写入两次而没有中间读取操作，第一次写入可以在不改变程序含义的情况下被移除。因此，这种情况可简化为流依赖。

其他输出依赖同样可被移除。在以下代码中， `t` 可声明为私有变量，从而消除该依赖。

```
for (i=0; i<N; i++) {
  t = f(i)
  s += t*t;
}
```

若需要获取 `t` 的最终值，可在 OpenMP 中使用 lastprivate 子句。

2.6.9.2 并行化嵌套循环

上述示例中，若循环迭代 i 中出现不同索引（如 i 和 $i+1$），则数据依赖关系非平凡。反之，类似以下形式的循环：

```
for (int i=0; i<N; i++)
  x[i] = x[i]+f(i);
```

易于并行化。然而，嵌套循环需要更多考量。OpenMP 为此类循环提供了 ‘collapse’ 指令，例如

```
for (int i=0; i<M; i++)
  for (int j=0; j<N; j++)
    x[i][j] = x[i][j] + y[i] + z[j];
```

此处，整个 i, j 迭代空间是并行的。

以下情况如何处理：

```
for (n = 0; n < NN; n++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i] += B[i][j]*c[j] + d[n];
```

练习 2.29. 对此循环进行重用分析。假设 `a`、`b`、`c` 无法同时存入缓存。现假设 `c` 和 `b` 的一行数据可存入缓存且略有剩余空间。能否找到可大幅提升性能的循环交换方案？编写测试验证。

分析该循环嵌套的并行性时，可见 `j` 循环是归约操作，而 `n` 循环存在数据流 w
d依赖：每个 `a[i]` 都在每次 `n` 迭代中被更新。结论是仅能合理
p并行化 `i` 循环。

练习 2.30. 此并行性分析与练习 2.29 的循环交换有何关联？交换后的循环是否仍可并行化？若熟悉 OpenMP，请编写对元素求和的代码验证结论。无论进行交换还是引入 OpenMP 并行，结果应保持一致。

2.6.10 并行程序设计

很久以前，人们曾认为编译器和运行时系统的某种神奇组合可以将现有的顺序程序转化为并行程序。这种希望早已破灭，因此如今的并行程序都是从一开始就作为并行程序编写的。当然，并行有不同的类型，每种类型对如何设计并行程序都有其特定的影响。在本节中，我们将简要探讨其中的一些问题。

2.6.10.1 Parallel data structures

并行程序设计中的一个问题是使用结构数组 *(AOS)* 还是数组结构 *(SOA)*。在常规程序设计中，你通常会定义一个结构体

```
struct { int number; double xcoord,ycoord; } _Node;
struct { double xtrans,ytrans} _Vector;
typedef struct _Node* Node;
typedef struct _Vector* Vector;
```

如果需要多个这样的结构体，你会创建一个结构体数组。

```
Node *nodes = (Node*) malloc( n_nodes*sizeof(struct _Node) );
```

这是 AOS 设计。

现在假设您想要并行化一个操作

```
void shift(Node the_point,Vector by) {
  the_point->xcoord += by->xtrans;
  the_point->ycoord += by->ytrans;
}
```

该操作在循环中完成

```
for (i=0; i<n_nodes; i++) {
  shift(nodes[i],shift_vector);
}
```

这段代码符合 MPI 编程的结构要求（章节 2.6.3.3），其中每个处理器都拥有自己的本地节点数组。该循环也易于通过 OpenMP 实现并行化（章节 2.6.2）。

然而在 1980 年代，当人们意识到 AOS 设计不适用于向量计算机时，代码必须进行大幅重写。这种情况下，操作数需要连续存储，因此代码必须转向 SOA 设计：

```
node_numbers = (int*) malloc( n_nodes*sizeof(int) );
node_xcoords = // et cetera
node_ycoords = // et cetera
```

然后进行迭代

```
for (i=0; i<n_nodes; i++) {
  node_xoords[i] += shift_vector->xtrans;
  node_yoords[i] += shift_vector->ytrans;
}
```

哦，我刚才是否提到最初的 SOA 设计最适合分布式内存编程？这意味着在向量计算机时代结束十年后，所有人又不得不为集群重写代码。当然如今，随着 *SIMD* 宽度不断增加，我们需要部分回归到 AOS 设计。（英特尔 *ispc* 项目中有一些实验性软件支持这种转换， http://ispc.github.io/ 它能将 *SPMD* 代码转换为 *SIMD*。）

2.6.10.2 延迟隐藏

处理器间的通信通常较慢，比单处理器从内存传输数据更慢，且远慢于对数据的操作。因此，在设计并行程序时，有必要权衡网络流量与 ‘有效’ 操作的相对比例。每个处理器必须有足够的工作量来抵消通信开销。

应对通信相对缓慢的另一种方法是合理安排程序，使得通信实际发生在计算进行的同时。这被称为计算与通信重叠或延迟隐藏。

例如，考虑矩阵 - 向量积的并行执行 $y = Ax$ （关于此操作的进一步讨论将在第 7.2.1 节展开）。假设向量是分布式存储的，因此每个处理器 p 执行

$$\forall_{i\in I_p} : y_i = \sum_j a_{ij}x_j.$$

由于 x 也是分布式存储的，我们可以将其表示为

$$\forall_{i\in I_p} : y_i = \left(\sum_{j \text{ local}} + \sum_{j \text{ not local}} \right) a_{ij}x_j.$$

Th该方案如图 2.19 所示。现在我们可以继续 eed as follows:

- Start the transfer of non-local elements of x;

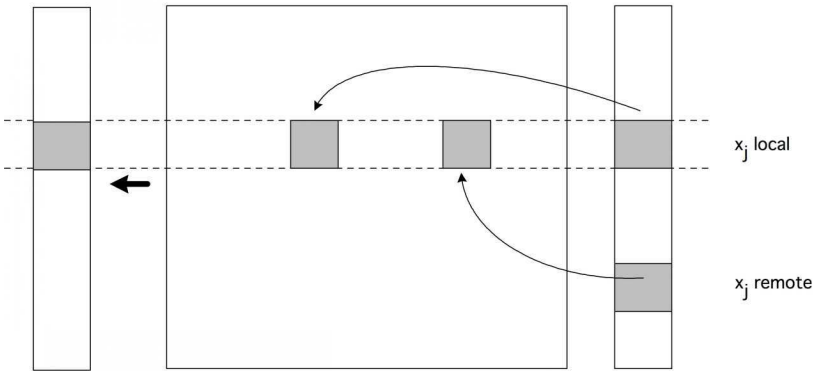


图 2.19：采用块行分布的并行矩阵 - 向量乘积示意图。

- 在数据传输进行时操作 x 的本地元素；
- 确保传输已完成；
- 操作 x 的非本地元素。

练习 2.31. 通过重叠计算与通信能获得多少收益？提示：考虑两种边界情况 —— 计算时间为零仅存在通信的情况，以及相反情况。再分析一般情况。

当然，这一场景的前提是存在支持这种重叠的软硬件支持。MPI 通过所谓的 2.6.3.6 节中提到的异步通信或非阻塞通信例程允许此类操作。但这并不直接意味着重叠实际会发生，因为硬件支持完全是另一个问题。

2.7 拓扑结构

若多个处理器协同处理同一任务，它们很可能需要进行数据通信。因此需要存在一种机制使数据能在任意处理器间传输。本节将讨论并行机中连接处理器的若干可能方案，此类方案称为（处理器）拓扑结构。

为理解此处确实存在实质性问题，请考虑两种无法 ' 扩展 ' 的简单方案：

- 以太网是一种将所有网络设备连接至单根电缆的通信方案（参见下文注释）。若一台设备通过线缆发送信号传递消息时，另一台设备也试图发送消息，后者将检测到唯一通信信道被占用，并等待一段时间后重试发送操作。以太网接收数据较为简单：消息包含目标接收者地址，处理器只需检查线缆信号是否指向自身即可。该方案的缺陷显而易见：通信信道容量有限，随着接入处理器数量增加，每个处理器可用带宽将递减。

由于冲突解决机制的存在，消息发送前的平均延迟时间也会随之增加。

- 在全连接配置中，每个处理器都有一根专线用于与其他处理器通信。这种方案在以下意义上是完美的：消息能以最短时间传输，且两条消息永远不会相互干扰。单个处理器可发送的数据量不再是处理器数量的递减函数；实际上它是递增函数。若网络控制器能处理，处理器甚至可进行多路并行通信。

  该方案的缺陷显然在于：处理器的网络接口设计不再固定 —— 随着并行计算机增加更多处理器，网络接口需要更多连接线。网络控制器同样会变得更复杂，整机成本的增速将超过处理器数量的线性增长。

备注 8 上文对以太网的描述基于原始设计。在使用交换机（尤其是 *HPC* 场景下）后，此描述已不再适用。

*I*最初人们认为消息碰撞意味着以太网将劣于其他解决方案
*s*例如 *IBM* 的令牌环 网络，它明确防止了碰撞。需要相当复杂的统计
*a*分析才能证明以太网的性能远超最初的天真预期。

本节我们将看到一些可以 扩展到大量处理器的方案。

2.7.1 图论基础

并行计算机中连接处理器的网络可以方便地用基本图论 概念来描述。我们用图来描述并行机器，其中每个处理器是一个节点，若两者间存在直接连接则连接两个节点。（假设连接是对称的，因此网络是一个无向图。）

接下来，我们可以分析该图的两个重要概念。

首先，图中节点的度是指其连接的其它节点数量。当节点代表处理器、边代表导线时，显然高度数不仅对计算效率有利，从工程角度看也意味着更高成本。我们假设所有处理器具有相同的度数。

其次，消息从一个处理器经一个或多个中间节点传输到另一个处理器时，很可能会在路径的每个阶段产生延迟。因此，图的直径至关重要。直径定义为任意两节点间最短路径（按链路数计算）的最大值：

$$d(G) = \text{最大值}_{i,j} \mid i \text{ 与 } j \text{ 之间的最短路径}$$

若 d 为直径，且通过一条导线发送消息耗时一个单位时间，则意味着消息最多在 d 时间内必定到达。

习题 2.32. 找出处理器数量、其度数以及连接图直径之间的关系。

除了关注 ‘从处理器 A 到处理器 B 的消息需要多长时间’ 这一问题外，我们还经常担忧两个同时传输的消息之间的冲突：是否存在这样一种可能性，即两条同时传输的消息需要使用同一网络链路？如图 2.20 所示，若每个处理器 p_i 向 $i < n/2$ 发送消息，则会出现 $p_{i+n/2}$ 条消息试图通过 $n/2$ 与 $p_{n/2-1}$ 之间的线路。这类冲突被称为**拥塞**或**争用**。显然，

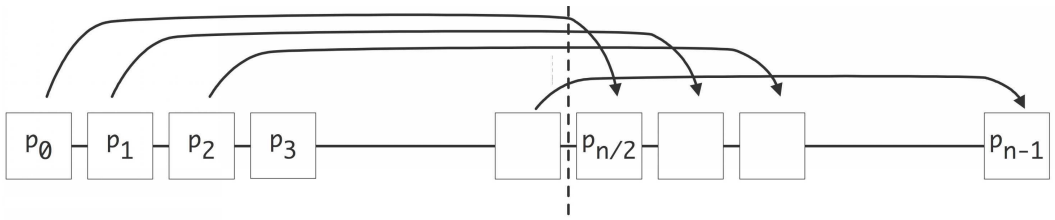


图 2.20：由于同时传输的消息导致的网络链路争用。

并行计算机拥有的链路越多，发生拥塞的可能性越小。

描述拥塞可能性的精确方法是考察**二分宽度**。其定义为：将处理器图分割成两个不连通子图所需移除的最小链路数量。例如，假设处理器以线性阵列方式连接，即处理器 P_i 连接到 P_{i-1} 和 P_{i+1}。此时二分宽度为 1。

二分宽度 w 描述了在并行计算机中，可以保证同时传输的消息数量。证明：取 w 个发送处理器和 w 个接收处理器。由此定义的 w 条路径是不相交的：如果它们相交，我们只需移除 $w-1$ 条链接即可将处理器分为两组。

实际上，当然可以有超过 w 条消息同时传输。例如，在线性阵列中（其具有 $w = 1$），如果所有通信仅在相邻处理器间进行，且一个处理器在同一时间只能发送或接收（不能同时进行），则 $P/2$ 条消息可以同时被发送和接收。若处理器能同时发送和接收，网络中 P 条消息可同时传输。

二分宽度也描述了网络中的**冗余性**：当一个或多个连接出现故障时，消息是否仍能从发送方传递到接收方？

虽然二分宽度是衡量导线数量的指标，但实际上我们更关注这些导线的传输能力。这里相关的概念是**二分带宽**：即跨越二分宽度的带宽，它是二分宽度与导线容量（以比特每秒计）的乘积。二分带宽可视为当任意半数处理器与另一半通信时可达带宽的衡量标准。二分带宽是比聚合带宽更现实的指标 —— 后者有时被引用，定义为当每个处理器都在发送时的总数据速率：处理器数量乘以单连接带宽再乘以单个处理器可同时发送的次数。这个数值可能非常高，通常无法代表实际应用中的通信速率。

2.7.2 总线结构

我们考虑的第一种互连设计是将所有处理器置于同一内存总线上。该设计将所有处理器直接连接到同一内存池，因此它提供了统一内存访问 *(UMA)* 或对称多处理 *(SMP)* 模型。

使用总线的主要缺点是扩展性有限，因为同一时间只能有一个处理器进行内存访问。为克服此限制，我们需要假设处理器速度慢于内存，或处理器配备缓存或其他本地内存进行操作。在后一种情况下，通过让处理器监听总线上的所有内存流量（这一过程称为侦听），可以轻松维护缓存一致性。

2.7.3 线性阵列与环形网络

连接多个处理器的一种简单方式是将其组成线性阵列：每个处理器有一个编号 i，处理器 P_i 与 P_{i-1} 和 P_{i+1} 相连。首尾处理器可能是例外：若它们彼此相连，我们称该架构为环形网络。

该解决方案要求每个处理器具备两个网络连接，因此设计相对简单。

习题 2.33. 线性阵列的二分宽度是多少？环形结构呢？

习题 2.34. 由于线性阵列的连接有限，在编写并行算法时可能需要巧妙设计。例如考虑一个 ‘广播’ 操作：处理器 0 需要将数据项发送给所有其他处理器。我们做出以下简化假设：

- 一个处理器可以同时发送任意数量的消息，
- 但一条线路同一时间只能承载一条消息；不过，
- 任意两个处理器之间的通信耗时固定为一个单位时间，与中间经过的处理器数量无关。

在完全连接网络或星型网络中，可以简单地针对 $i = 1 \ldots N-1$ 写出：

将消息发送至处理器 i

假设一个处理器能发送多条消息，这意味着该操作可一步完成。

现在考虑线性阵列。证明即使具备无限发送能力，上述算法仍会因拥塞出现问题。

寻找更好的发送操作组织方式。提示：假设您的处理器

以二叉树形式连接。假设共有 $N = 2^n - 1$ 个处理器。证明广播可在 $\log N$ 个阶段完成，且处理器只需具备同时发送单条消息的能力。

This exercise is an example of *embedding* a ‘logical’ communication pattern in a physical one.

2.7.4 二维与三维阵列

并行计算机的一种流行设计是将处理器组织成二维或三维的笛卡尔网格。这意味着每个处理器都有一个坐标 (i, j) 或 (i, j, k)，并且它在所有坐标方向上与相邻处理器相连。这种处理器设计仍然相当简单：

网络连接数（连通图的度数）是网络空间维度数（ 2 或 3 ）的两倍。

构建二维或三维网络是一个相当自然的想法，因为我们所处的世界是三维的，而计算机常被用于模拟现实现象。如果我们暂时接受物理模型需要最近邻类型的通信（这将在 4.2.3 节中看到），那么网格计算机就是运行物理模拟的天然候选方案。

习题 2.35. 一个由 $n \times n \times n$ 个处理器组成的 3D 立方体的直径是多少？其二分宽度是多少？若添加环绕环形连接，这些参数会如何变化？

习题 2.36. 你的并行计算机处理器以二维网格形式组织。芯片制造商推出了一款新芯片，时钟速度相同，但采用双核而非单核设计，且能适配现有插槽。请批判以下论点：‘不涉及通信的工作量每秒可完成量翻倍；由于网络保持不变，二分带宽也保持不变，因此可以合理预期新机器的速度会提升一倍’。

基于网格的设计通常具有所谓的环绕式或环形连接，这种连接方式将二维网格的左右两侧以及上下两端相连。如图 2.21 所示。

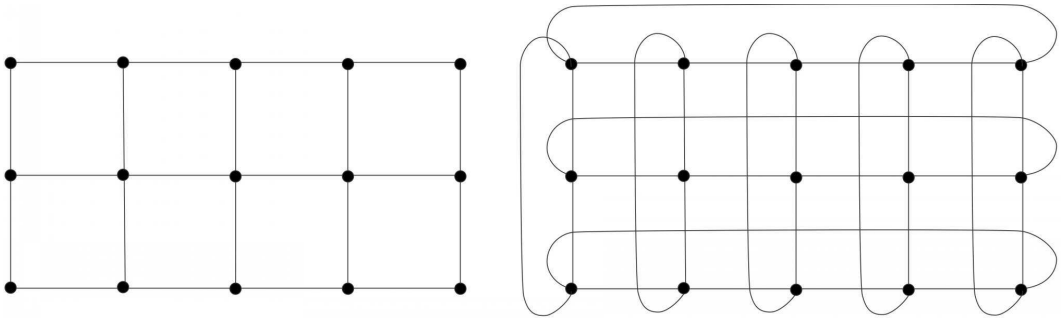


图 2.21：具有环形连接的二维网格。

某些计算机设计声称采用高维度网格结构（例如 5D ），但并非所有维度都具有同等地位。例如，一个三维网格中，若每个节点均为四插槽四核配置，则可视为五维网格。然而，最后两个维度是完全连接的。

2.7.5 超立方体

前文我们通过粗略论证说明了基于最近邻通信普遍性的网状结构处理器组织的适用性。然而，处理器间偶尔需要进行任意点对点的数据收发操作，前文提及的广播就是典型例子。因此，我们需要比网状结构直径更小的网络拓扑。另一方面，我们希望避免全连接网络那种复杂的设计。

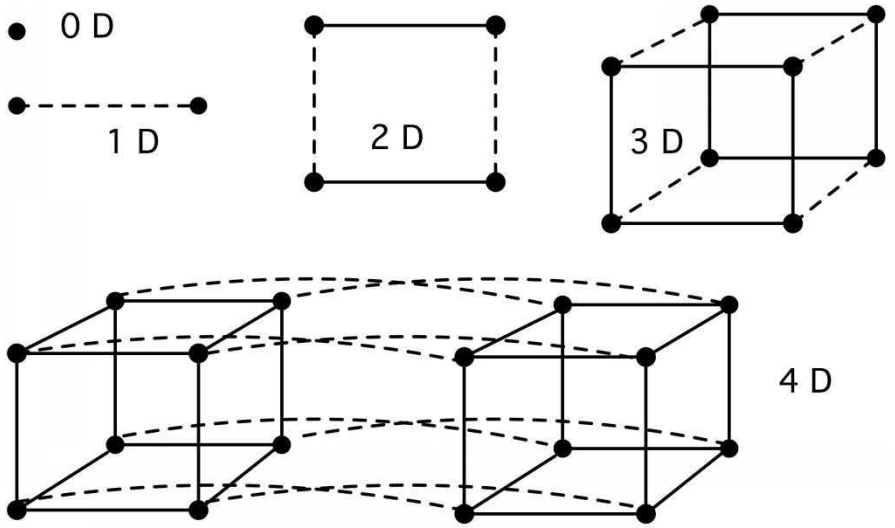


图 2.23：超立方体。

一种良好的折中方案是采用超立方体设计。一个 n 维超立方体计算机拥有 2^n 个处理器，每个处理器在每个维度上都与另一个处理器相连；参见图 2.23。

一种简单的描述方式是给每个处理器分配一个由 d 位组成的地址：我们将超立方体的每个节点编号为描述其在立方体中位置的比特模式；参见图 2.22。

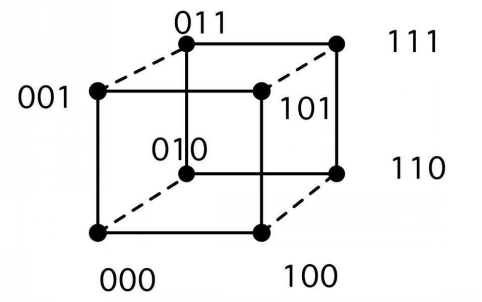


图 2.22：超立方体节点的编号方式。

采用这种编号方案后，一个处理器将与所有地址仅有一位之差的其它处理器相连。这意味着，与网格结构不同，处理器的相邻节点编号差值并非 1 或 $\sqrt{P}$，而是 1、 2、 4、 8、 ...。

超立方体设计的巨大优势在于其网络直径小且承载流量大的能力。

练习 2.37. 超立方体的直径是多少？其截面宽度又是多少？

一个缺点是处理器的设计依赖于机器整体规模。实际上，处理器会按照最大可能的连接数来设计，而购买较小规模机器的用户将不得不为未使用的容量付费。另一个缺点是扩充现有机器规模只能通过翻倍实现：除了 2^p 之外的尺寸都不可能实现。

练习 2.38. 参考 2.1 节中的并行求和示例，给出超立方体上并行实现的执行时间。证明该示例的理论加速比（至多相差一个常数因子）在超立方体实现中得以实现。

2.7.5.1 在超立方体中嵌入网格

上文我们论证了，对于许多模拟物理现象的应用而言，网状连接的处理器是一个合乎逻辑的选择。超立方体虽然看起来不像网状结构，但它们拥有足够的连接性，只需忽略某些连接，就能轻松模拟出网状结构。

假设我们需要一个一维数组的结构：我们希望处理器的编号方式能让处理器 i 能够直接发送数据给 $i-1$ 和 $i+1$。我们不能简单地采用如图 2.22 所示的节点编号方式。例如，节点 1 直接连接到节点 0，但与节点 2 的距离为 2。在环形结构中，节点 3 的右邻节点 4 在此超立方体中甚至存在最大距离 3。显然，我们需要以某种方式重新编号这些节点。

我们将证明，可以遍历超立方体的每一个顶点且仅访问一次，这等同于在超立方体中嵌入一维网格。

1D Gray code	0 1 一维代码及其反射： 0 1 ⋮ 1 0 附加
2D Gray code	0 和 1 位： 0 0 ⋮ 1 1 二维代码及其反射： 0 1 1 0 ⋮ 0 1 1 0
3D Gray code	0 0 1 1 ⋮ 1 1 0 0 附加 0 和 1 位： 0 0 0 0 ⋮ 1 1 1 1

图 2.24：格雷码。

这里的基本概念是（二进制反射）格雷码 [84]。这是一种排序二进制数的方式 $0\ldots 2^d-1$ 使得 $g_0, \ldots g_{2^d-1}$ 相邻的 g_i 和 g_{i+1} 仅有一位不同。显然，普通二进制数不满足这一点：1 和 2 的二进制表示已有两位不同。格雷码为何有用？因为 g_i 和 g_{i+1} 仅有一位不同，意味着它们在超立方体中对应的节点是直接相连的。

图 2.24 展示了如何构建格雷码。该过程是递归的，可以非正式地描述为 ‘将立方体划分为两个子立方体，对其中一个子立方体进行编号，交叉到另一个子立方体，并按照第一个子立方体的逆序对其节点进行编号’。二维立方体的结果如图 2.25 所示。

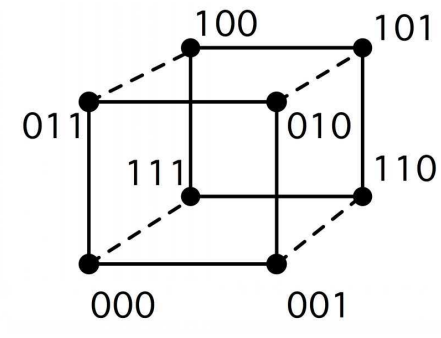


图 2.25：超立方体节点的格雷码编号。

既然格雷码为我们提供了一种将一维 ‘网格’ 嵌入超立方体的方法，我们现在可以逐步推进。

习题 2.39. 展示如何通过附加两个 2^d 节点立方体的嵌入位模式，将 2^{2d} 个节点的方形网格嵌入超立方体。你将如何适应 $2^{d_1+d_2}$ 个节点的网格？一个 $2^{d_1+d_2+d_3}$ 个节点的三维网格？

2.7.6 交换网络

前文简要讨论了全连接处理器。若通过在所有处理器之间布置大量线缆实现连接，这种方式并不实际。但存在另一种可能性：将所有处理器连接到交换机或交换网络。一些流行的网络设计包括交叉开关、蝶形交换和胖树。

交换网络由交换元件构成，每个元件具有少量（约十几个）入站和出站链接。通过将所有处理器连接到某个交换元件，并采用多级交换结构，即可实现任意两个处理器通过网络路径互联。

2.7.6.1 交叉开关

最简单的交换网络是交叉开关，一种由 n 水平与垂直线条组成的布局，每个交叉点设有开关元件以决定线路是否连接；参见图 2.26。若将水平线指定为输入、垂直线为输出，这显然是一种让 n 输入映射到 n 输出的方式。所有输入与输出的组合（有时称为 “ 排列 ” ）均被允许。

此类网络的一个优势是连接互不阻塞。主要缺点在于开关元件数量为 n^2，这是随处理器数量 n 快速增长的函数。

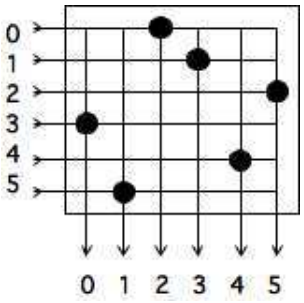


图 2.26：一个简单的交叉开关，将 6 个输入连接到 6 个输出。

2.7.6.2 蝶形交换

蝶形交换网络由小型交换元件构建而成
元件构成，且具有多级结构：随着处理器数量增加，级数也相应增长。图 2.27 展示了连接 2 、4 及 8 个处理器的蝶形网络，每个处理器均配有本地内存。（亦可将所有处理器置于网络一侧，所有内存置于另一侧。）

如图 2.28 所示，蝶形交换网络允许多个处理器同时访问内存。由于各处理器的访问时间相同，这种交换网络是实现 *UMA* 架构的一种方式（参见章节 2.4.1 ）。基于蝶形交换网络的典型计算机是 *BBNButterfly* （ http://en.wikipedia.org/wiki/BBN_Butterfly ）。在 2.7.7.1 节中，我们将看到这些理念如何在实际集群中实现。

习题 2.40. 无论是简单交叉开关还是蝶形交换网络，随着处理器数量增加都需要扩展网络结构。请分别计算连接 n 个处理器与内存所需的导线（单位长度）数量与交换元件数量。数据包从内存传输到处理器所需的时间如何表示？请用穿越单位长度导线的单位时间与穿越交换元件的时间来表达。

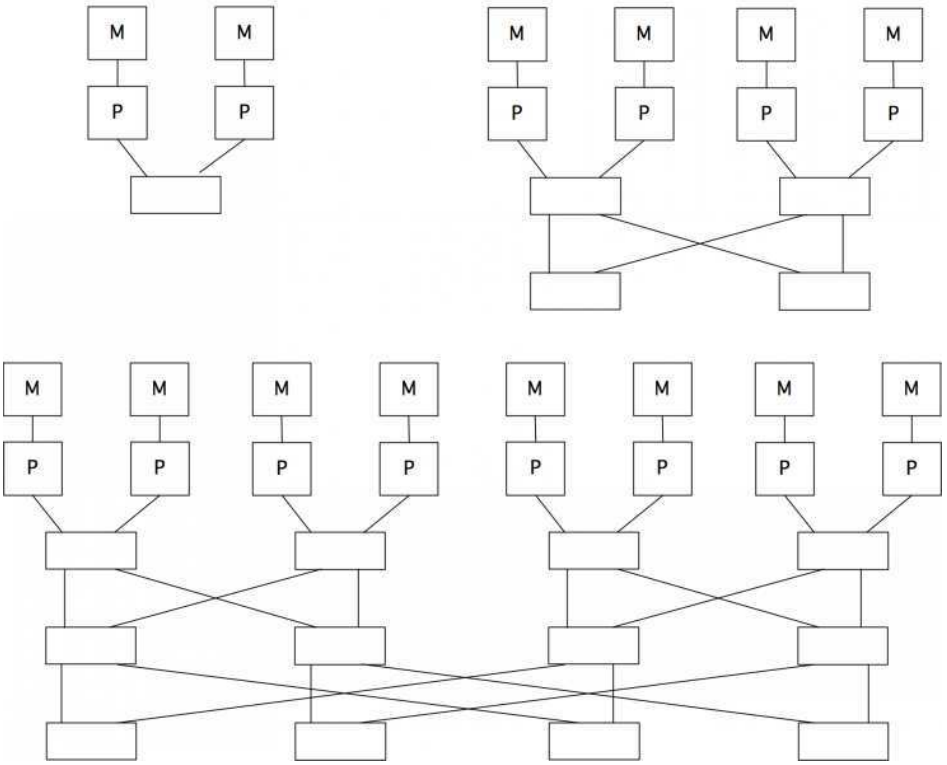


图 2.27: 为 2 、 4 、 8 个处理器设计的蝶形交换网络，每个处理器均配备本地内存。

*P*蝶形网络中的数据包路由基于目标地址的比特位进行决策。
在 i 层级时，检查目标地址的第 i 位：若为 1，则选择交换节点的左侧出口；若为 0，则选择右侧出口。如图 2.29 所示。若采用图 2.28 中将内存连接到处理器的方案，则仅需两位（至末级交换节点），但需额外三位描述反向路径。

2.7.6.3 胖树结构

若以树状结构连接交换节点，根节点附近将因仅有两根连线而面临严重拥塞问题。假设存在 k 层树结构，共包含 2^k 个叶节点。当左子树所有叶节点需与右子树节点通信时，将有 2^{k-1} 条消息通过根节点的单条入线，同理需通过单条出线传输。胖树网络通过使每层保持相同总带宽消除此拥塞问题 —— 根节点实际配备 2^{k-1} 条出入连线 [85]。图 2.30 左侧展示该结构；右侧为 Stampede 集群机柜示意图，其中叶交换机分别管理机柜上下半部。

首个基于胖树结构的成功计算机体系架构是 Connection Machines CM5。

在胖树结构中，与其他交换网络类似，每条消息都携带自身的路由信息。由于在胖树中路由选择仅限于向上一层级移动或在当前层级切换至另一子树，因此消息只需携带与层级数量相等的路由信息比特数，即对于 n 而言是 $\log_2 n$ 。

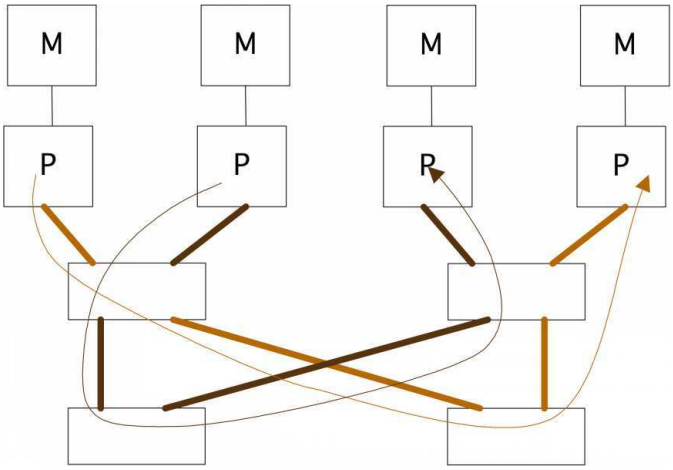


图 2.28：蝶形交换网络中两条独立路径。

处理器。

习题 2.41. 证明胖树结构的二分宽度为 $P/2$，其中 P 表示处理器叶节点数量。提示：证明将胖树连接的处理器集合分割为两个连通子集仅存在唯一方式。

关于胖树的理论阐述见 [135]，该研究表明胖树在某种意义上是最优的：其消息传递速度（在对数因子范围内）可与任何占用相同构建空间的网络相媲美。此结论的潜在假设是：越靠近根节点的交换机需连接更多线缆，因而需要更多组件，相应体积也更大。这一论点虽具理论价值，但无实际意义，因为在当前使用胖树互连的最大规模计算机中，网络的物理尺寸几乎不构成限制。例如，在 *TACC Frontera* 集群（德克萨斯大学奥斯汀分校）中，仅需 6 台核心交换机（即容纳胖树顶层结构的机柜）即可连接 91 个处理器机柜。

如上文所述的胖树结构在构建上成本高昂，因为每增加一级就需要设计更大规模的新交换机。因此在实际应用中，具有胖树特性的网络通常由简单的交换元件构成；参见图 2.31。该网络在带宽和路由可能性方面与胖树等效。路由算法会稍显复杂：在胖树中，数据包只能向上单一方向传输，而此处数据包需判断应路由至两个上级交换机中的哪一个。

这类交换网络是 *Clos* 网络 [35] 的一种特例 .

2.7.6.4 过载与竞争

实际应用中，胖树网络并不使用 2 进 2 出元件，而采用规模约为 20 进 20 出的交换机。这使得网络层级可控制在 3 至 4 级（顶层交换机称为脊柱卡）。

在这种情况下，网络分析的一个额外复杂性是可能存在过度订阅。网卡中的端口可配置为输入或输出，只有总数是固定的。因此，一个 40 端口的

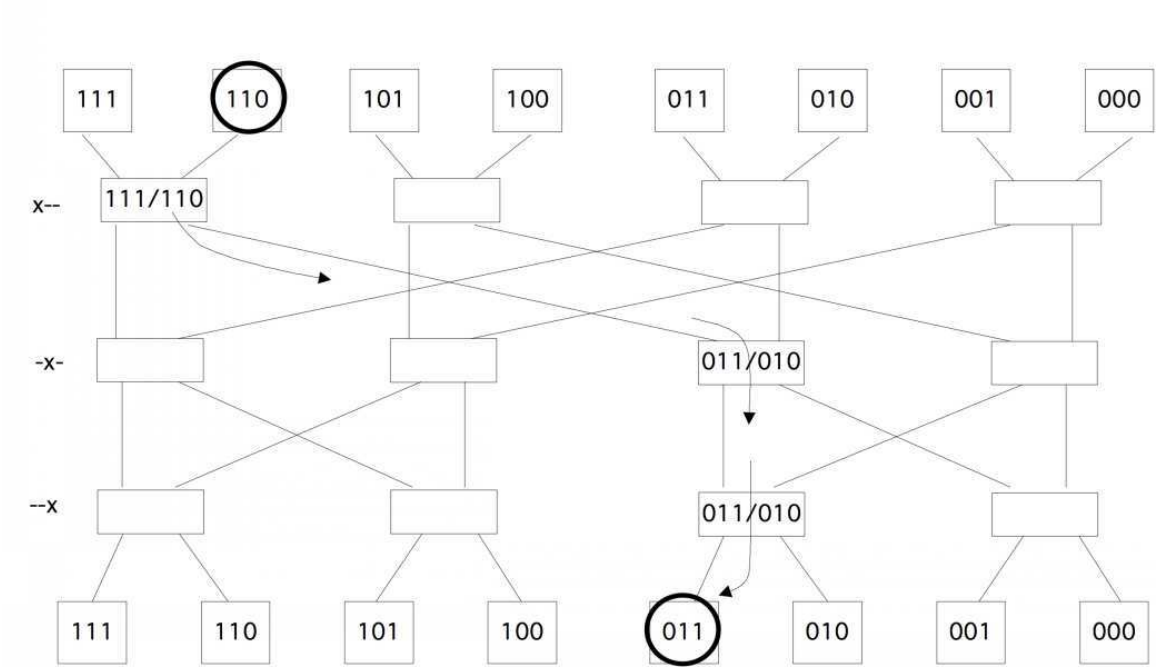


图 2.29：通过三级蝶形交换网络的路由。

交换机可配置为 20 进 20 出，或 21 进 19 出等。当然，若所有 21 个连接节点同时发送数据，19 个出端口将限制带宽。

还存在另一个问题。假设我们构建一个小型集群，交换机配置为 p 个输入端口和 w 个输出端口，即采用 $p+w$ 端口交换机。图 2.32 展示了两个此类交换结构，共连接 $2p$ 个节点。当节点通过交换机发送数据时，其从 w 条可用线路中选择路径由目标节点决定。这被称为输出路由。

C显然最多只能期望 w 个节点在避免消息冲突的情况下发送数据，因为这是
o交换机间可用线路的数量。然而对于 w 个目标节点的多数选择场景都会出现争用
f或无视线路的存在。这是生日悖论的一个例子。

习题 2.42. 考虑上述架构，其中 p 个节点通过交换之间的 w 条线路发送数据。编写一个模拟程序，其中 p 个节点中有 $w' \leq w$ 个向随机选择的目标节点发送消息。碰撞概率作为 w'、 w、 p 的函数是多少？找到一种制表或绘图的方法。加分项：对简单情况 $w' = 2$ 进行统计分析。

2.7.7 集群网络

上述讨论较为抽象，但在实际集群中，你可以看到网络设计的具体体现。例如，胖树集群网络会有一个中央机柜对应树状结构的顶层。图 2.33 展示了 TACC*Ranger*（已停用）和 *Stampede* 集群的交换机。第二张图中可见实际上存在多个冗余的胖树网络。

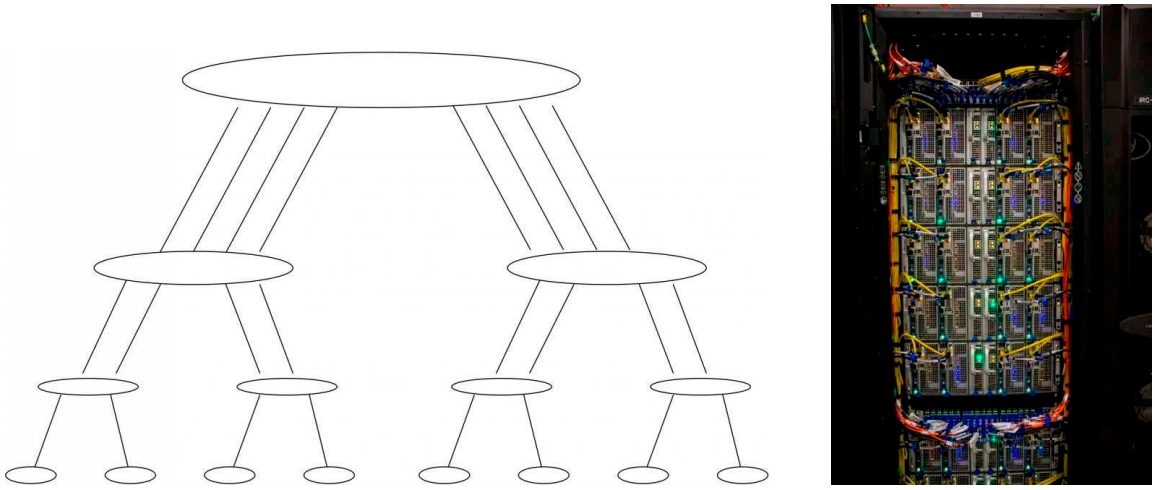


图 2.30：具有三级互连的胖树结构（左）；Stampede 集群机柜中的叶交换机（右）。

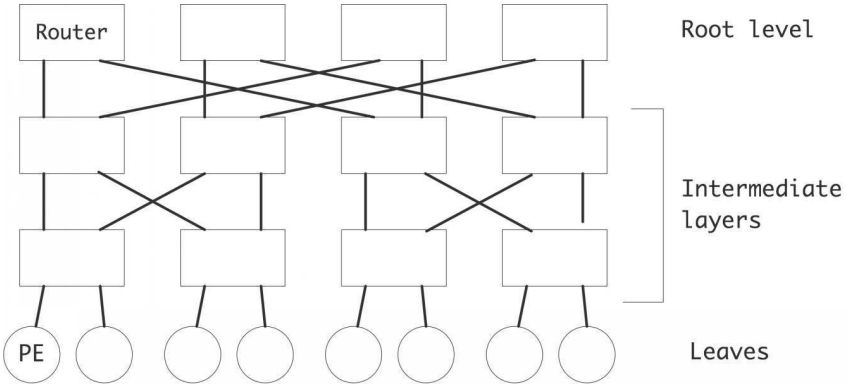


图 2.31：由简单交换元件构建的胖树结构。

另一方面，像 *IBM BlueGene* 这样的集群，基于环形拓扑集群，会呈现为一组相同的机柜，因为每个机柜包含网络中的相同部分；参见图 2.34。

2.7.7.1 案例研究：*Stampede*

作为实际网络应用的示例，我们来看德克萨斯高级计算中心的 *Stampede* 集群。它可以被描述为一个多根多级胖树结构。

- 每个机架包含 2 个机箱，每个机箱有 20 个节点。
- 每个机箱配备一个叶交换机，该交换机是内部交叉开关，可实现机箱内节点间的完美连接；
- 叶交换机拥有 36 个端口，其中 20 个连接节点，16 个对外连接。这种超额订阅意味着最多只有 16 个节点在与机箱外部通信时能获得完美带宽。

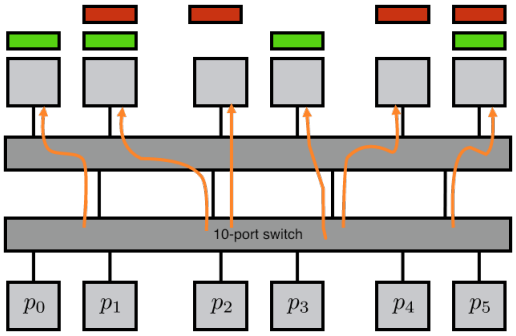


图 2.32：过载交换机中的端口争用情况。

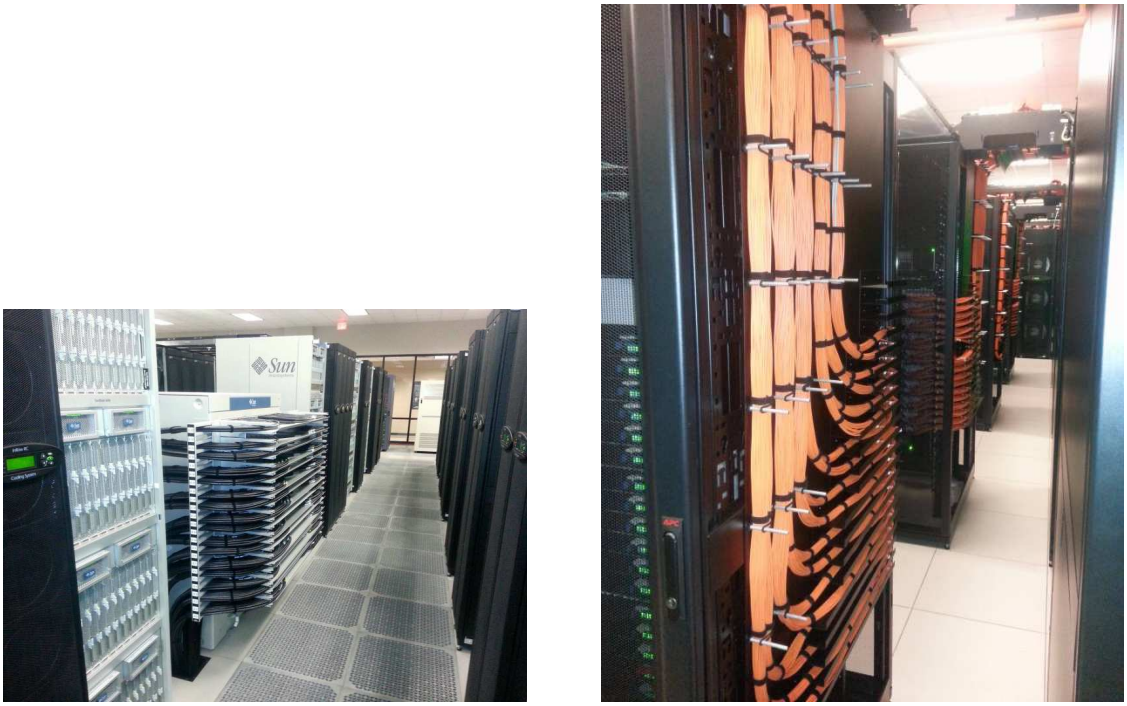


图 2.33：TACC Ranger 与 Stampede 集群的网络交换机。

- 共有 8 个中央交换机作为 8 个独立的胖树根节点运作。每个机箱通过两条连接分别接入各中央交换机的 ‘叶卡’，恰好占用 16 个出站端口。
- 每个中央交换机配备 18 块脊卡，每块脊卡含 36 个端口，每个端口连接至不同的叶卡。
- 每个中央交换机有 36 块叶卡，其中 18 个端口连接叶交换机，另 18 个端口连接脊卡。这意味着系统可支持 648 个机箱，实际使用 640 个。

该网络中的一项优化在于，连接到同一叶卡的两个通信无需经历更高树状层级的延迟。这意味着同一机箱内的 16 个节点与

图 2.34：一台 BlueGene 计算机。

另一种可能具备完美的连接性。

然而，采用静态路由（如 *Infiniband* 所用方案），每个目标节点会固定关联一个端口（该目标节点与端口的映射关系存储在每个交换机的路由表中）。因此，从 20 个可能目标节点中选取 16 个节点的某些子集时，将获得完美带宽；而其他子集则会因两个目标节点的流量经由同一端口传输而导致性能下降。

2.7.7.2 案例研究：*Cray* 蜻蜓网络

Cray 蜻蜓网络是一种颇具实用性的折中方案。前文已论证完全连接的网络因成本过高而难以扩展。但若将处理器数量控制在有限范围内，则可能实现全互连架构。蜻蜓设计采用小型全互连组，再将这些组构建为全连接图。

这引入了一种明显的不对称性，因为组内处理器之间的带宽大于组间带宽。然而，得益于动态路由，消息可以采取非最短路径，通过其他组进行路由。这可以缓解争用问题。

2.7.8 带宽与延迟

前文所述将发送消息视为单位时间操作的说法显然不切实际。长消息的传输时间必然短于短消息。要更真实地描述传输过程，需引入两个概念；我们已在 1.3.2 节关于处理器缓存层级间数据传输的讨论中提及。

延迟建立两个处理器间的通信需要一定时间，该时间与消息大小无关。此过程耗时称为消息的延迟。造成这种延迟的原因多种多样。

- 两个处理器会进行 ‘握手’ 协议，以确保接收方准备就绪，且有足够的缓冲区空间接收消息。

- 消息需要由发送方进行编码以便传输，并由接收方进行解码。
- 实际传输可能需要时间：并行计算机通常规模庞大，即使以光速传输，消息的第一个字节也可能需要数百个周期才能跨越两个处理器之间的距离。

带宽 在两个处理器之间启动传输后，主要关注的是每秒可通过通道的字节数，这被称为带宽。带宽通常由通道速率（物理链路传送比特的速率）和通道宽度（链路中物理导线的数量）决定。通道宽度通常是 16 的倍数，一般为 64 或 128。这也表现为通道可以同时发送一个或两个 8 字节的字。

带宽与延迟在以下表达式中被形式化

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

对于传输一个 n 字节消息的时间。这里，α 是延迟，而 β 是每字节时间，即带宽的倒数。有时我们会考虑涉及通信的数据传输，例如在集体操作的情况下；参见第 7.1 节。此时我们将传输时间公式扩展为

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

其中 γ 是每个操作的时间，即计算速率的倒数。

也可以将这些公式进一步细化为

$$T(n, p) = \alpha + \beta n + \delta p$$

其中 p 是穿越的网络 ‘跳数’。然而，在大多数网络中，δ 的值远低于 α，因此我们在此忽略它。此外，在胖树网络中（章节 2.7.6.3），跳数大约为 $\log P$，其中 P 是处理器的总数，因此无论如何它都不会很大。

2.7.9 并行计算中的局部性

在章节 1.6.2 中，你找到了关于单处理器计算中局部性概念的讨论。并行计算中的局部性概念包含了所有这些以及更多层次。

核心间；私有缓存 现代处理器上的核心拥有私有的一致性缓存。这意味着看起来你不需要担心局部性，因为无论数据在哪个缓存中都是可访问的。然而，维护一致性会消耗带宽，因此最好保持访问的局部性。

Between cores; shared cache 在核心之间共享的缓存是一个您无需担心局部性的地方：这是处理核心之间真正对称的内存。

在插槽之间 对于程序员而言，节点（或主板）上的插槽看似共享内存，但实际上这是 *NUMA* 访问（章节 2.4.2 ），因为内存与特定插槽相关联。

通过网络结构 某些网络具有明显的局部性效应。您已在章节 2.7.1 中看到一个简单示例，一般而言，任何网格型网络都会优先支持 “ 邻近 ” 处理器之间的通信。基于胖树的网络似乎不受此类竞争问题影响，但层级结构会引入另一种形式的局部性。比节点局部性更高一级的是，小型节点组通常通过叶交换机连接，这防止数据流向中央交换机。

2.8 多线程架构

现代 CPU 的架构主要由一个事实决定：从内存获取数据的速度远慢于处理数据的速度。因此，通过构建一个由更快、更小内存组成的层级结构，尽可能让数据靠近处理单元，以缓解主存的高延迟和低带宽问题。处理单元中的指令级并行（ ILP ）也有助于隐藏延迟，并更充分地利用可用带宽。

然而，寻找指令级并行性（ ILP ）是编译器的职责，且其实际能发现的并行性存在上限。另一方面，科学计算代码往往具有程序员显而易见（但对编译器未必明显）的数据并行特性。是否有可能让程序员显式指定这种并行性，并由处理器加以利用?

在 2.3.1 节中可见，SIMD 架构可通过显式数据并行方式编程。若我们拥有大量数据并行性但处理单元有限时，可将并行指令流转换为线程（参见 2.6.1 节），让每个处理单元执行多个线程。当某线程因未完成的存储器请求而停滞时，处理器可切换至输入数据已就绪的其他线程。此技术称为多线程。它虽看似用于避免处理器空等内存，亦可视为保持内存最大占用率的方法。

练习 2.43. 若将内存的长延迟和有限带宽视为两个独立问题，多线程技术是否能同时解决这两个问题?

问题在于大多数 CPU 不擅长快速切换线程。上下文切换（从一个线程切换到另一个线程）需要消耗大量时钟周期，其耗时与等待主内存数据相当。在所谓的多线程架构（ *MTA* ）中，上下文切换效率极高，有时仅需单个周期，这使得单个处理器能同时处理多个线程。

多线程概念最早在 *TeraComputer MTA* 计算机中实现，该架构后来演变为现款的 *Cray XMT*[7]。

另一个 MTA 实例是 GPU，其处理器作为 SIMD 单元运行的同时本身也具备多线程能力；详见章节 2.9.3。

7. Tera Computer 在从 *SGI* 收购 *CrayResearch* 后更名为 *CrayInc.*

2.9 协处理器，包括 GPU

当前的 CPU 设计旨在对各种可能的计算保持适度高效。这意味着通过限制处理器的功能范围，有可能提升其效率，或在保持相似效率的同时降低功耗。因此，连接至主处理器的协处理器概念已被多次探索。例如，驱动第一代 IBM 个人电脑的英特尔 8086 芯片可额外搭载数值协处理器 —— 英特尔 *8087* 协处理器。该协处理器在超越函数运算上极为高效，并集成了 SIMD 技术。针对图形处理的专用功能也广受欢迎，由此催生了 x86 处理器的 *SSE* 指令集，以及可连接 PCI-X 总线的独立 GPU 单元。

其他案例还包括用于数字信号处理（ DSP ）指令的协处理器，以及可重新配置以适应特定需求的 FPGA 板卡。早期的阵列处理器如 *ICL DAP* 同样属于协处理器范畴。

In this section we lo简要看看这一理念在现代的一些具体实现， particular GPUs.

2.9.1 简史

协处理器可通过两种不同方式进行编程：有时它会无缝集成，某些指令会自动在其上执行而非在‘主’ 处理器上运行；另一方面，也可能需要显式调用协处理器功能，甚至可能实现协处理器功能与主机功能的并行执行。后者从效率角度看或许颇具吸引力，但会引发严重的可编程性问题 —— 程序员现在需要明确识别两条工作流：一条针对主处理器，另一条针对协处理器。

一些配备协处理器的著名并行计算机包括：

- 1993 年的 *Intel Paragon* 每个节点配备两个处理器，分别负责通信与计算。它们实际是相同的 *Inteli860* 处理器。在后续版本中，实现了向通信处理器传递数据及函数指针的功能。
- 位于洛斯阿拉莫斯的 *IBM Roadrunner* 是首台达到 PetaFlop 级别的超级计算机。（ *Grape* 计算机虽更早达成此里程碑，但它是专用于分子动力学计算的特殊用途机器。）其通过采用 Cell 协处理器实现了这一速度。值得一提的是，Cell 处理器本质上是索尼 PlayStation3 的核心引擎，再次印证了超级计算机的平民化趋势（章节 2.3.3 ）。
- 中国的天河 *-1A* 于 2010 年登顶全球超级计算机 500 强榜单，借助 NVIDIA GPU 实现了约 2.5 PetaFlops 的运算能力。
- 天河 -2 与 TACC Stampede 集群采用 *Intel* 至强融核协处理器。

Roadrunner 和天河 -1A 是高性能协处理器的典型代表，它们需要独立于主 CPU 进行显式编程。例如，天河 -1A 的 GPU 上运行的代码需使用 *CUDA* 语言编写并单独编译。

在这两种情况下，由于协处理器无法直接与网络通信，可编程性问题进一步加剧。要将数据从一个协处理器发送到另一个协处理器，必须先将其传递至主机处理器，再经由网络传输到另一台主机处理器，最后才能移动到目标协处理器。

2.9.2 瓶颈

协处理器通常拥有独立内存，*Intel Xeon Phi* 可独立运行程序，但更常见的问题是如何访问主处理器的内存。主流解决方案是通过 *PCI* 总线连接协处理器。这种方式访问主内存的速度低于主处理器的直接连接。例如，*Intel Xeon Phi* 的带宽为 512 位宽（每秒 5.5GT，后文将解释 “GT” ），而其与主内存的连接速率虽为 5.0GT/s，但位宽仅为 16 位。

***GT* 度量** 我们习惯看到带宽以千兆比特每秒为单位。对于 *PCI* 总线，常见 *GT* 度量。GT 代表千兆传输（ giga-transfer ），用于测量总线在 0 和 1 之间切换状态的速度。通常每个状态转换对应一个比特，但总线需自行提供时钟信息，若发送连续相同比特流会导致时钟混乱。因此，每 8 比特需编码为 10 比特来避免此类问题。这意味着实际有效带宽会低于理论值，本例中降低系数为 4/5 。

由于制造商倾向于对事物进行正面宣传，他们报告的是更高的数值。

2.9.3 GPU 计算

图形处理器（ GPU ）（有时也称为通用图形处理器（ GPGPU ））是一种专用处理器，专为快速图形处理而设计。然而，由于图形处理所执行的操作属于算术运算的一种形式，*GPU* 逐渐演变为一种对非图形计算也很有用的设计。*GPU* 的总体设计理念源于 ‘ 图形管线 ’：对大量数据元素执行相同操作，形成数据并行（第 2.5.1 节），并且多个这样的数据并行块可以同时处于活跃状态。

CPU 的基本限制同样适用于 GPU：访问内存会产生较长的延迟。CPU 解决此问题的方法是引入多级缓存；而 GPU 则采取了不同的方法（另见第 2.8 节）。GPU 专注于吞吐量计算，即以高平均速率交付大量数据，而非尽快获得单个结果。这是通过支持多线程（第 2.6.1 节）并实现线程间的极速切换来实现的。当一个线程在等待内存数据时，另一个已获取数据的线程可以继续执行其计算。

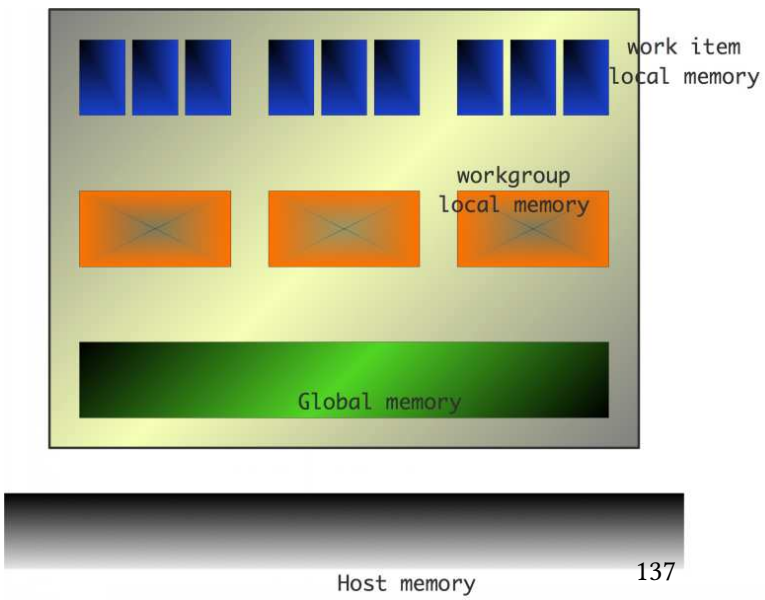


图 2.35：GPU 的内存结构。

2.9.3.1 基于内核的 *SIMD* 类型编程

现代 GPU[8] 采用了一种结合 SIMD 与
SPMD 并行的架构。线程并非完全独立，
而是被组织在线程块中，其中所有线程
执行相同的指令，
m这使得执行过程呈现 SIMD 特性。同样地，可以将同一指令流（在 Cuda 术语中称为 ‘内核’ ）
t调度到多个线程块上执行。这种情况下，各线程块之间可能不同步，类似于
p在 SPMD （单程序多数据）上下文中的进程。然而，由于此处我们处理的是线程而非进程，
t术语单指令多线程 （*SIMT*） 被用来描述。

这种软件设计在硬件上表现明显；例如，英伟达 GPU 拥有 16 至 30 个流式多处理器（ SM ），每个 SM 由 8 个流式处理器（ SP ）组成，对应处理器核心；参见图 2.36。SP 以真正的 SIMD 方式运作。GPU 中的核心数量通常多于传统多核处理器，但核心功能更为受限。因此，此处采用众核这一术语。

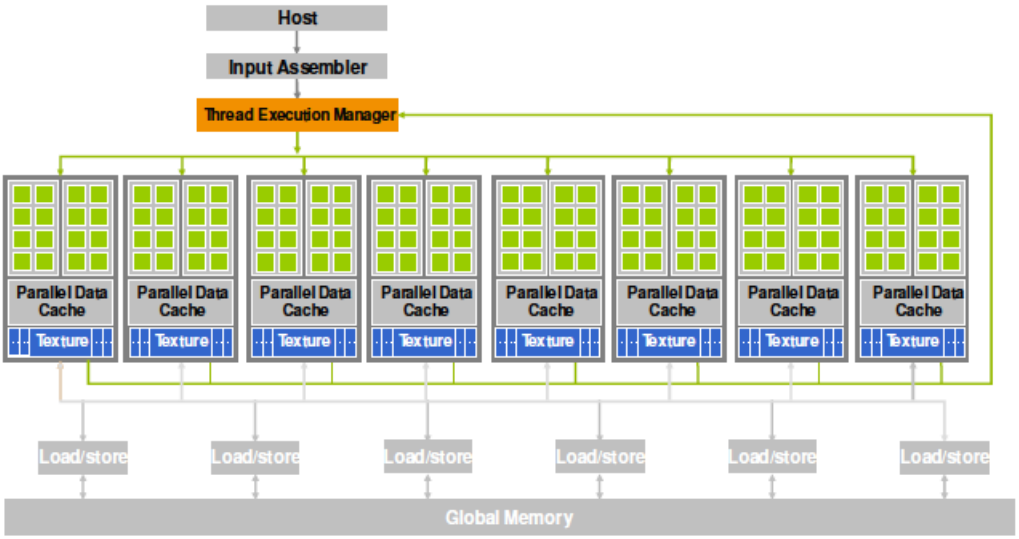


图 2.36：GPU 结构示意图

GPU 的 SIMD 或数据并行特性体现在 *CUDA* 启动进程的方式上。一个内核（即在 GPU 上执行的函数）通过以下方式在 *mn* 个核心上启动：

```
KernelProc<< m,n >>(args)
```

执行内核的 *mn* 核心集合被称为网格，其结构为 *m* 线程块，每个线程块包含 *n* 个线程。一个线程块最多可拥有 512 个线程。

8. The most popular GPUs today are made by NVidia, and are programmed in *CUDA*, an extension of the C language.

回顾线程共享地址空间的特性（参见章节 2.6.1 ），因此需要一种机制来标识每个线程操作的数据部分。为此，线程块通过 x, y 坐标进行编号，而块内的线程则通过 x, y, z 坐标标识。每个线程知晓自身在块内的坐标，以及所属块在网格中的坐标。

我们通过向量加法示例来说明：

```
// Each thread performs one addition
__global__ void vecAdd(float* A, float* B, float* C)
{int i = threadIdx.x + blockDim.x * blockIdx.x;
C[i] = A[i] + B[i];}int main(){
// Run grid of N/256 blocks of 256 threads each
vecAdd<<< N/256,  256>>>(d_A, d_B, d_C);}
```

这体现了 GPU 的 SIMD 特性：所有线程执行相同的标量程序，仅操作不同数据。

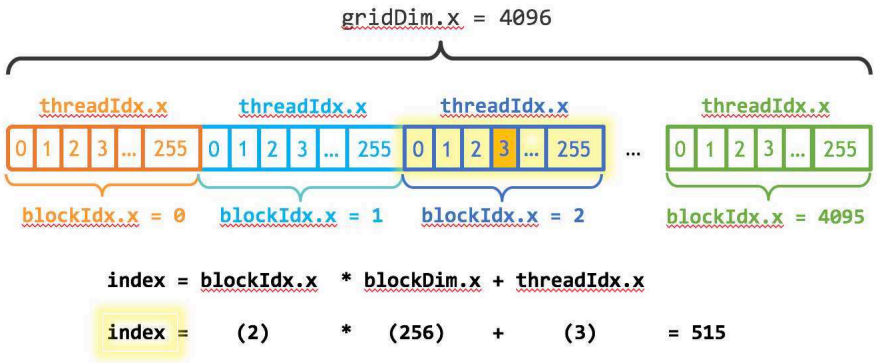


图 2.37：CUDA 中的线程索引。

线程块中的线程是真正的数据并行：如果存在条件分支，使得部分线程执行 *true* 分支而其他线程执行 *false* 分支，那么将先执行其中一个分支，同时暂停所有属于另一分支的线程。随后，而非同时，另一分支的线程才会执行其代码。这可能导致严重的性能损失。

GPU 依赖于大量数据并行性及快速上下文切换能力。这意味着它们在图形和科学计算应用中表现优异，因为这些领域存在大量数据并行性。然而，它们在 ‘商业应用’ 和操作系统上可能表现不佳，因为这些场景的并行性属于指令级并行（ ILP ）类型，通常较为有限。

2.9.3.2 GPU 与 CPU 的对比

以下是 GPU 与常规 CPU 之间的一些主要区别：

- 首先，截至本文撰写时（2010 年末），GPU 是通过诸如 ***PCI-X*** 总线等接口连接的附属处理器，因此其操作的所有数据必须从 CPU 传输而来。由于该传输过程的内存带宽较低（至少比 GPU 内部内存带宽低 10 倍），必须在 GPU 上执行足够多的计算任务才能抵消这一开销。
- 由于 GPU 本质是图形处理器，其设计重点在于单精度浮点运算。为满足科学计算需求，双精度支持正在逐步增强，但双精度运算速度通常仅为单精度浮点性能的一半。这种差异有望在下一代产品中得到改善。
- CPU 专为处理可能高度异构的单一指令流而优化；而 GPU 明确为数据并行设计，在传统代码上的表现会非常糟糕。
- CPU 设计用于处理单个线程，或至多少量线程。而 GPU 需要远超过计算核心数量的海量线程才能高效运行。

2.9.3.3 GPU 的预期效益

GPU 因能以极高性价比实现卓越性能而迅速赢得声誉。大量案例表明，代码只需经过最小改动移植到 CUDA 平台，有时就能获得高达 400 倍的加速效果。GPU 真的是这样的奇迹机器吗？原始代码是否编写得很糟糕？既然 GPU 如此强大，为何不将其用于所有场景？

真相具有多重维度。

首先，GPU 并不像常规 CPU 那样通用：GPU 极其擅长数据并行计算，而 CUDA 能优雅地表达这种细粒度并行性。换言之，GPU 仅适用于特定类型的计算，对许多其他类型则表现欠佳。

反观常规 CPU，其数据并行处理能力未必出色。除非代码经过精心编写，否则性能可能从最优水平下降约以下倍数：

- 除非使用指令或显式并行结构，编译后的代码仅会利用可用核心中的 1 个（例如 4 核中的 1 个）。
- 若指令未采用流水线技术，浮点运算流水线导致的延迟会额外增加 4 倍的影响因子。
- 若核心具备独立的加法与乘法流水线，但未能同时利用二者，则会再损失 2 倍的性能潜力。
- 未能使用 SIMD 寄存器会进一步加剧与峰值性能之间的差距。

W编写计算内核的最优 CPU 实现通常需要汇编语言编码，
w而直观的 CUDA 代码却能以相对较小的开发成本实现高性能，
v当然，前提是计算任务具备足够的数据并行性。

2.9.4 Intel 至强融核

Intel *Xeon Phi*（以其架构设计 Many Integrated Cores (MIC) 著称）是专为数值计算设计的处理器。初代产品 *Knights Corner* 采用协处理器形态，尽管

第二次迭代中，*Knights Landing* 实现了自托管运行。

作为协处理器，Xeon Phi 与 GPU 既有差异也有相似之处。

- 两者均通过 *PCI-X* 总线连接，这意味着设备上的操作在启动时存在显著延迟。
- Xeon Phi 拥有通用计算核心，因此能运行完整程序；而 GPU 仅具备有限程度的此能力（参见章节 2.9.3.1 ）。
- Xeon Phi 可直接运行标准 C 代码。
- 两种架构都需要大量的 SIMD 风格并行处理能力，对于至强融核而言，这是由于 8 字宽的 *AVX* 指令集要求。
- 两种设备都能（或可通过）从主机程序进行卸载计算 来工作。在至强融核上，这可通过 OpenMP 构造和 *MKL* 调用来实现。

2.10 负载均衡

本章大部分内容中，我们假设问题可以完美分配到各处理器上，即处理器始终在执行有效工作，仅因通信延迟而处于闲置 状态。但实际上，处理器可能因等待消息而闲置，而发送方处理器甚至尚未执行到其代码中的发送指令。这种一个处理器工作而另一个闲置的情况被称为负载不平衡：闲置处理器本无必然理由空闲，若我们采用不同的工作负载分配方式，它本可参与工作。

处理器工作过量与不足之间存在不对称性：与其让一个处理器超负荷工作导致其他所有处理器等待，不如让一个处理器提前完成任务。

练习 2.44. 请精确阐述这一概念。假设一个并行任务在所有处理器上耗时均为 1，但有一个处理器例外。

- 设 $0 < \alpha < 1$ 且让一个处理器耗时 $1 + \alpha$。加速比和效率如何随处理器数量变化？请分别从 Amdahl 和 Gustafsson 角度（章节 2.2.3 ）考虑。
- 若一个处理器耗时 $1 - \alpha$，请回答相同问题。

负载均衡通常代价高昂，因其需要移动大量数据。例如，章节 7.5 的分析表明，稀疏矩阵 - 向量乘积过程中的数据交换量低于处理器本地存储量级。但本文不探讨具体移动成本：我们主要关注工作负载均衡及保持原始负载分布中的局部性。

2.10.1 负载均衡与数据分布

工作与数据存在二元性：多数应用中数据分布决定工作分布，反之亦然。若应用需更新大型数组，数组各元素通常 ' 驻留 ' 于唯一确定的处理器，该处理器负责该元素的所有更新。此策略称为所有者计算。

因此，数据与工作量之间存在直接关联，相应地，数据分布与负载均衡也密不可分。例如，在 7.2 节中我们将讨论数据分布如何影响效率，但这立即转化为对负载分配的考量：

- 负载需要均匀分布。这通常可以通过均匀分配数据来实现，但有时这种关系并非线性。
- 任务需被放置以最小化它们之间的通信量。在矩阵 - 向量乘法案例中，这意味着二维分布优于一维分布；关于空间填充曲线的讨论同样基于此动机。

举个数据分布如何影响负载均衡的简单例子：假设一个线性数组中每个点执行相同计算且每次计算耗时相同。若数组长度 N 可被处理器数量 p 整除，则工作量被完美均分。若数据无法整除，则先为每个处理器分配 $\lfloor N/p \rfloor$ 个点，剩余 $N - p\lfloor N/p \rfloor$ 个点分配给最后的处理器。

练习 2.45. 在最坏情况下，这会使处理器的工作负载失衡到什么程度？将此方案与将 $\lceil N/p \rceil$ 个数据点分配给除一个处理器外的所有处理器（该处理器获得较少数据点）的方案进行比较；参见上文练习。

将盈余的 $r = N - p\lfloor N/p \rfloor$ 分散到 r 个处理器上比集中到一个更好。具体实现方式可以是给前或后 r 个处理器各分配一个额外数据点。这可通过为进程 p 分配以下范围来实现：

$$\big[p \times \lfloor (N+p-1)/p \rfloor, (p+1) \times \lfloor (N+p-1)/p \rfloor\big)$$

虽然该方案实现了较好的负载均衡，但计算特定数据点属于哪个处理器较为复杂。以下方案简化了此类计算：设 $f(i) = \lfloor iN/p \rfloor$，则处理器 i 将获得从 $f(i)$ 到 $f(i+1)$ 的数据点。

练习 2.46. 证明 $\lfloor N/p \rfloor \leq f(i+1) - f(i) \leq \lceil N/p \rceil$。
在此方案下，拥有索引 i 的处理器是 $\lfloor (p(i+1)-1)/N \rfloor$。

2.10.2 负载调度

在某些情况下，计算负载可以自由分配给处理器，例如在共享内存环境中，所有处理器均可访问全部数据。此时我们可以考虑静态调度（预先确定工作分配至各处理器）与动态调度（执行过程中动态决定分配）之间的差异。

为说明动态调度的优势，假设在 4 个线程上调度 8 个运行时递减的任务（图 2.38 ）。静态调度中，第一个线程获取任务 1 和 4，第二个获取 2 和 5，依此类推。动态调度中，任何完成当前任务的线程将自动获取下一个任务。显然在此特定示例中动态调度能获得更优的运行时间。但另一方面，动态调度可能产生更高的开销。

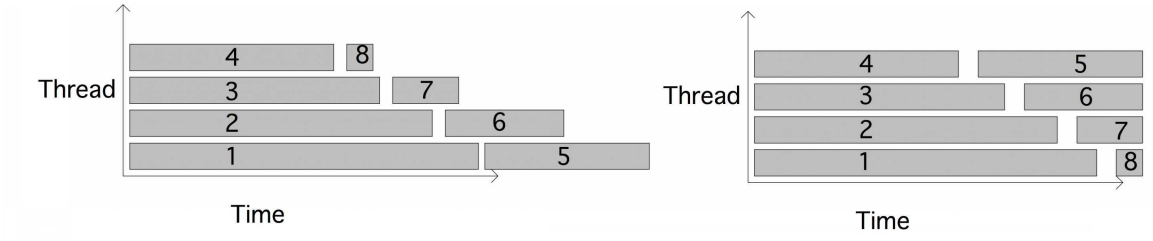


图 2.38：静态或轮询（左）与动态（右）线程调度对比；任务编号已标注。

2.10.3 独立任务的负载均衡

在其他情况下，工作负载并非直接由数据决定。这可能发生在存在待处理工作池，且每个工作项的处理时间难以从其描述中计算得出时。此类情况下，我们可能需要在将工作分配给进程时保留一定灵活性。

首先考虑可被划分为无通信独立任务的情况。例如计算 *Mandelbrot* 集图像的像素，其中每个像素根据不依赖周边像素的数学函数设定。若能预测绘制图像任意部分所需时间，就能实现工作的完美划分并分配给处理器。这种方法称为静态负载均衡。

更现实的情况是，我们无法完美预测部分作业的运行时间，因此采用工作过分解策略：将工作划分为比处理器数量更多的任务。这些任务随后被分配至工作池，处理器在完成当前作业后从池中获取下一个任务。这种方法被称为动态负载均衡。许多图论和组合问题均可采用此方法处理，详见章节 2.5.3。关于多核环境中的任务分配，参见章节 7.12。

有研究结果表明，任务的随机分配在统计意义上接近最优 [117]，，但这忽略了科学计算任务通常需要频繁通信的特性。

习题 2.47。假设存在任务 $\{T_i\}_{i=1}$ 、 … 、 N，其运行时间分别为 t_i，且处理器数量不受限。参考章节 *2.2.4* 中的 Brent 定理，可推导出最优任务执行方案的特征：存在一个专用处理器仅执行具有最大 t_i 值的任务。（本题灵感源自 [161]。）

2.10.4 负载均衡作为图问题

接下来让我们考虑一个各部分需要通信的并行作业。在这种情况下，我们需要同时平衡标量工作负载和通信。

并行计算可以被表述为一个图（关于图论的介绍，请参见附录 20），其中处理器是顶点，如果两个处理器在某个时刻需要通信，则它们对应的顶点之间就有一条边。这样的图通常源自所解决问题的底层图。例如，考虑矩阵 - 向量积 $y = Ax$ 其中 A 是一个稀疏矩阵，且

详细观察正在计算 y_i 的处理器，其中 i 为某些值。语句 $y_i \leftarrow y_i + A_{ij}x_j$ 意味着该处理器需要获取数值 x_j，因此，若该变量位于其他处理器上，则需进行数据传输。
我们可以将其形式化：设向量 x 和 y 以不相交方式分布在处理器上，并唯一定义 $P(i)$ 为拥有索引 i 的处理器。若存在非零元素 a_{ij} 且满足 $P = P(i)$ 与 $Q = P(j)$，则存在边 (P,Q)。对于结构对称矩阵（即 $a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$），该图是无向的。

处理器上的索引分布现可转化为顶点权重与边权重：处理器的顶点权重等于其拥有的索引数量；边 (P,Q) 的权重表示需要从 Q 向 P 传输的向量组件数量，如前所述。

负载均衡问题现在可以表述如下：

找到一个分区 $\mathbb{P} = \cup_i \mathbb{P}_i$，使得顶点权重的变化最小，同时边权重尽可能低。

最小化顶点权重的变化意味着所有处理器的工作量大致相同。保持边权重低意味着通信量较少。这两个目标可能无法同时满足：可能需要某种权衡。

练习 2.48。考虑极限情况，即处理器速度无限快且处理器间带宽无限。此时决定运行时间的唯一剩余因素是什么？为了找到最优负载均衡，你需要解决什么图问题？稀疏矩阵的什么属性会导致最坏情况行为？

负载均衡的一个有趣方法来自谱图理论（第 20.6 节）：如果 A_G 是无向图的邻接矩阵，$D_G - A_G$ 是图拉普拉斯矩阵，那么对应于最小特征值零的特征向量 u_1 为正，而对应于下一个特征值的特征向量 u_2 与之正交。因此 u_2 必须具有交替符号的元素；进一步分析表明，正号元素相互连接，负号元素亦然。这自然导致了图的对分。

2.10.5 负载重新分配

在某些应用中，初始负载分配是明确的，但后续需要调整。典型的例子是有限元方法（FEM）代码，其中负载可以通过物理域的分区进行分配；参见第 7.5.3 节。如果后续域的离散化发生变化，负载必须进行重新平衡或重新分配。在接下来的小节中，我们将看到旨在保持局部性的负载平衡和重新平衡技术。

2.10.5.1 扩散负载平衡

在许多实际情况下，我们可以将处理器图与问题关联起来：任何一对通过点对点通信直接交互的进程之间存在一个顶点。因此，很自然地想到在负载平衡中使用这个图，并且只将负载从一个处理器移动到图中的邻居处理器。

这就是扩散负载平衡 [38, 109] 的思想 .
虽然图本身没有方向性，但在负载平衡中，我们为边赋予任意方向。负载平衡的描述如下。

设 ℓ_i 为进程 i 上的负载，$\tau_i^{(j)}$ 为边 $j \to i$ 上的负载转移量，则有

$$\ell_i \leftarrow \ell_i + \sum_{j \to i} \tau_i^{(j)} - \sum_{i \to j} \tau_j^{(i)}$$

尽管我们仅使用了 i, j 条边，但在实践中会对边进行线性编号。由此得到一个系统

$$AT = \bar{L}$$

其中

- A 是大小为 $|N| \times |E|$ 的矩阵，描述节点输入 / 输出边的连接关系，其元素值根据 ± 1 而定；
- T 是大小为 $|E|$ 的转移向量；且
- $\bar{L}$ 是负载偏差向量，表示每个节点相对于平均负载的超出或不足程度。

在线性处理器阵列的情况下，该矩阵是欠定的，边数少于处理器数；但在大多数情况下，系统会是超定的，边数多于进程数。因此，我们求解

$$T = (A^t A)^{-1} A^t \bar{L} \qquad \text{or} T = A^t (A A^t)^{-1} \bar{L}.$$

由于 $A^t A$ 和 $A A^t$ 是正不定的，我们可以通过松弛法近似求解，仅需局部知识。当然，这种松弛法收敛缓慢，而全局方法（如共轭梯度法 /CG ）会更快 [109]。

2.10.5.2 基于空间填充曲线的负载均衡

前几节我们探讨了负载均衡的两个方面：确保所有处理器获得近似等量的工作负载，以及让分布反映问题结构以保持合理通信量。我们可以将第二点表述为：在并行机上分配问题时保持空间局部性 —— 空间中邻近的点更可能发生交互，因此应置于同一处理器或至少相距不远的处理器上。

致力于保持局部性并不明显是最佳策略。在 BSP （参见章节 2.6.8 ）中，有一个统计论点表明随机放置既能实现良好的负载均衡，也能平衡通信。

练习 2.49。考虑将进程分配到处理器上的情形，其中问题的结构使得每个进程仅与其最近邻通信，且处理器按二维网格排列。若直接将进程网格映射到处理器网格，则不会出现争用。现编写一个程序，将进程随机分配到处理器上，并评估由此产生的争用程度。

上一节中，你已了解图划分技术如何有助于实现保持问题局部性的第二点。本节将介绍另一种技术，它对于初始负载分配及后续的负载再平衡都具有吸引力。在后一种情况下，处理器的工作量可能增减，需要将部分负载迁移至其他处理器。

例如，某些问题会进行自适应细化 [9] 。如图 2.39 所示。若我们持续追踪

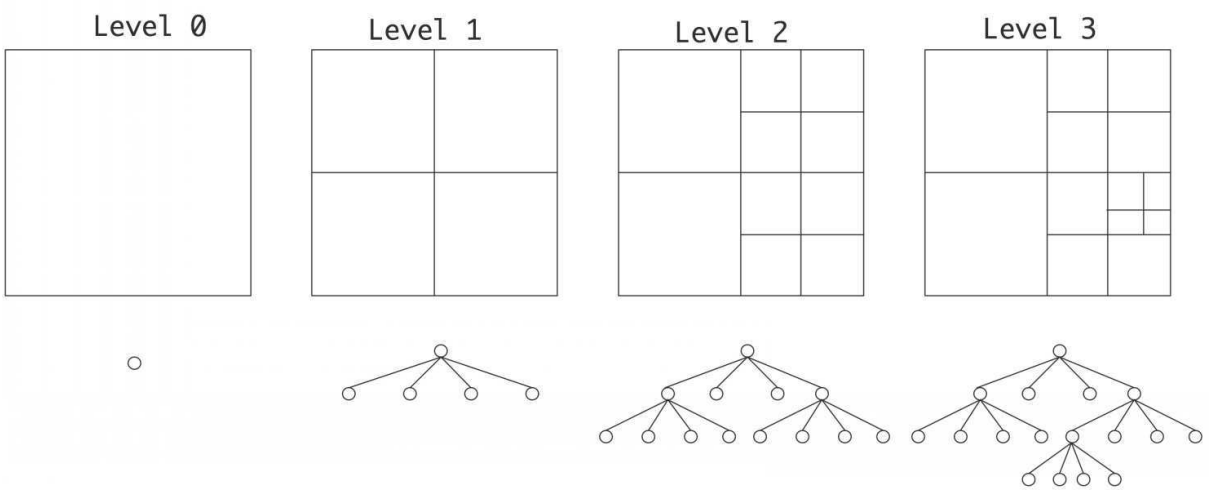


图 2.39：区域在后续层级中的自适应细化过程。

这些细化层级后，问题会形成树状结构，其中叶子节点包含所有工作负载。负载均衡转化为将树的叶子节点分配到各处理器上的问题；参见图 2.40。此时我们

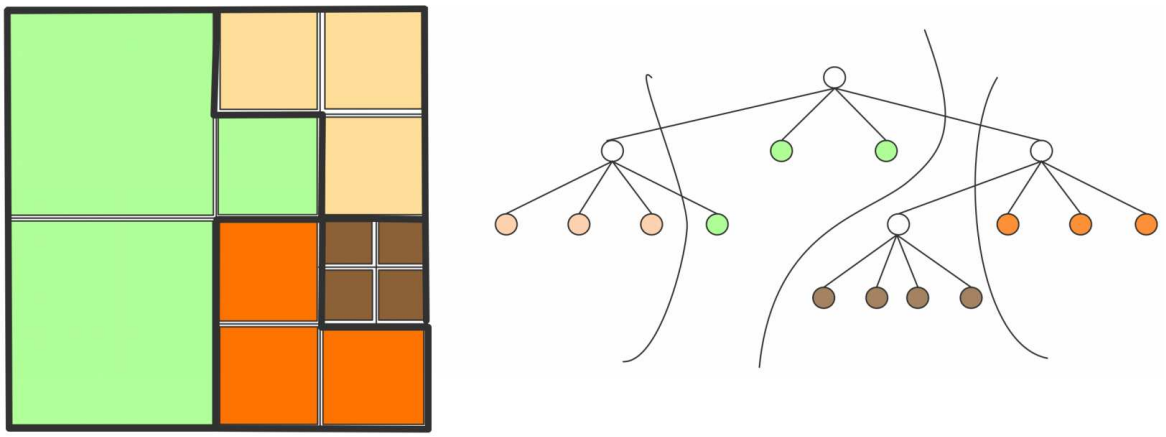


图 2.40：自适应细化区域的负载分布情况。

可以观察到该问题具有特定局部性：任何非叶子节点的子树在物理上相邻，因此它们之间很可能存在通信。

- 很可能子域的数量会超过处理器数量；为了最小化处理器间的通信，我们希望每个处理器包含一组简单连通的子域。此外，我们希望每个处理器覆盖的域部分是 ‘紧凑’ 的，即具有低纵横比和低表面积体积比。

9. 详细讨论参见 [28]。

- 当子域被进一步细分时，其处理器的部分负载可能需要转移到另一个处理器上。这种负载重新分配的过程应保持局部性。

为满足这些需求，我们采用空间填充曲线 (SFCs)。负载均衡树的空间填充曲线 (SFC) 如图 2.41 所示。我们不会对 SFC 进行正式讨论，而是让

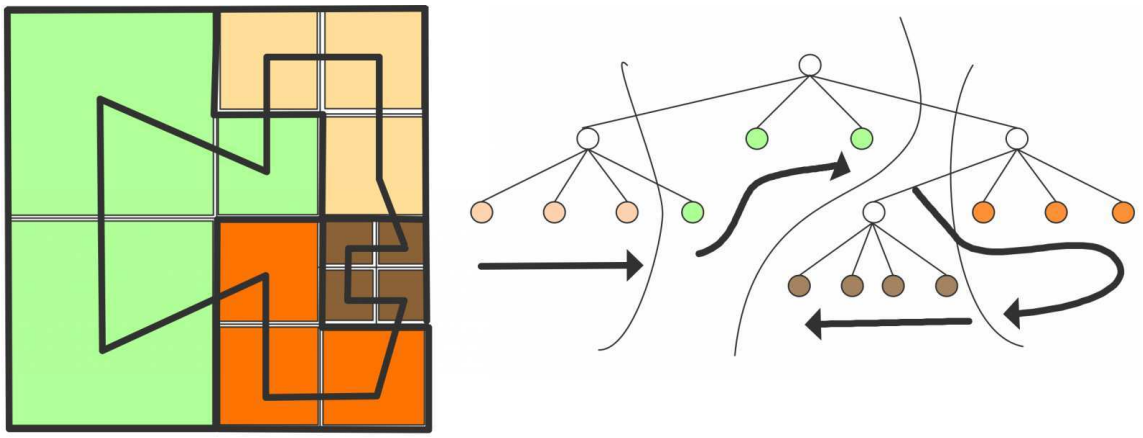


图 2.41：负载均衡树的空间填充曲线。

图 2.42 代表一个定义：SFC 是一种递归定义的曲线，每个子域仅被触及一次 10。该 SFC 具有这样的属性：物理上相邻的域元素在曲线上也会相邻

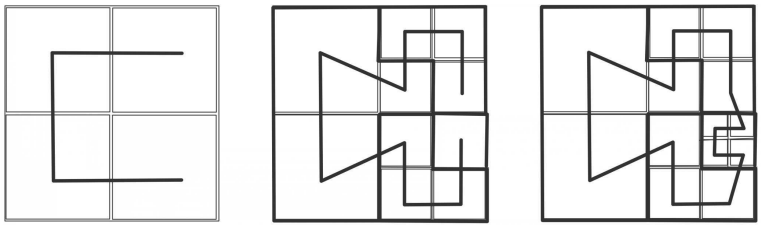


图 2.42：规则与不规则细化的空间填充曲线。

在曲线上，因此如果我们将空间填充曲线（ SFC ）映射到处理器的线性排序上，就能保持问题的局部性。

更重要的是，若通过另一层级对域进行细化，我们可相应调整曲线。随后负载可被重新分配到曲线上相邻的处理器，且仍能保持局部性。

(空间填充曲线（ SFCs ）在 N 体问题中的应用已在 [189] 和 [177] 中讨论过。)

10. 空间填充曲线（ SFCs ）由 Peano 引入，作为一种数学工具，用于构建从线段 $[0,1]$ 到高维立方体 $[0,1]^n$的连续满射函数。这颠覆了 ‘ 无法通过拉伸和折叠线段来填满正方形 ’ 的直观维度观念。后来 Brouwer 对维度概念给出了严谨的阐述。

2. 并行计算

2.11 剩余主题

2.11.1 分布式计算、网格计算与云计算

本节我们将简要探讨云计算及更早的术语分布式计算等概念。这些概念在科学意义上与并行计算存在关联，但在某些基础层面存在差异。

分布式计算可追溯至大型数据库服务器（如航空订票系统）的应用场景，这些系统需要被众多旅行社代理同时访问。当数据库访问量达到一定规模时，单一服务器无法满足需求，因此发明了远程过程调用（*RPC*）机制 —— 中央服务器可调用另一台（远程）机器上的代码（特定过程）。远程调用可能涉及数据传输（数据可能已存在于远程机器），或通过某种机制保持两台机器间的数据同步。这催生了存储区域网络（*SAN*）。在分布式数据库系统之后一代，Web 服务器也面临相同问题：需要让多台服务器像单一服务器那样处理大量并发访问。

我们已经看到分布式计算与高性能并行计算之间的一大区别。科学计算需要并行化，因为单个模拟任务对一台机器而言变得过于庞大或缓慢；而上述商业应用场景则涉及众多用户针对大型数据集执行小型程序（即数据库或网络查询）。对于科学需求，并行机器的处理器（集群中的节点）之间必须保持极高速的连接；而对于商业需求，只要中央数据集保持一致性，则无需此类高速网络。

无论是在 *HPC* 还是商业计算中，服务器都必须保持可用和运行状态，但在分布式计算中实现这一目标的方式有更大的自由度。对于连接到数据库等服务的用户而言，实际执行其请求的是哪台服务器并不重要。因此，分布式计算可以利用虚拟化技术：虚拟服务器可以在任何硬件设备上生成。

远程服务器（按需提供算力）与电网（按需供电）之间可以作类比。这催生了网格计算或效用计算的概念，例如美国国家科学基金会拥有的 Teragrid。网格计算最初旨在连接通过局域网 *(LAN)* 或广域网 *(WAN)*（通常是互联网）相连的计算机，这些机器本身可以是并行系统，且常隶属于不同机构。近年来，它被视为通过网络共享数据集、软件资源和科学仪器等资源的一种方式。

效用计算的概念作为一种使服务可用的方式，正如您从上述分布式计算的描述中所认识到的，随着谷歌搜索引擎的普及而成为主流，该引擎对整个互联网进行索引。另一个例子是安卓手机的 GPS 功能，它结合了地理信息系统（GIS）、全球定位系统（GPS）和混搭数据。谷歌收集和处理数据的计算模型已在 MapReduce [41] 中形式化。它结合了数据并行部分（“映射”阶段）和中心累积部分（“归约”阶段）。两者均不涉及科学计算中常见的紧耦合邻接通信。针对 MapReduce 计算的开源框架存在于 Hadoop [93] 中。亚马逊提供了商业化的 Hadoop 服务。

即使不涉及大型数据集，让远程计算机满足用户需求的概念也颇具吸引力，因为它免除了用户在本地机器上维护软件的需要。因此，

Google Docs 提供各种 ‘办公’ 应用程序，而用户无需实际安装任何软件。这一理念有时被称为软件即服务（*SAS*），用户通过连接到 ‘应用服务器’，并通过诸如网页浏览器等客户端进行访问。就 Google Docs 而言，不再有大型中央数据集，而是每个用户与存储在 Google 服务器上的个人数据进行交互。这当然具有巨大的优势，即用户从任何可以访问网页浏览器的地方都能获取数据。

SAS 概念与早期技术有若干联系。例如，在大型机和工作站时代之后，所谓的瘦客户端理念曾短暂流行。此时，用户将拥有工作站而非终端，但仍处理存储在中央服务器上的数据。Sun 公司的 *Sun Ray*（约 1999 年）便是此类产品之一，用户依靠智能卡在任意一台无状态的工作站上建立本地环境。

2.11.1.1 使用场景

按需提供服务的模式对企业极具吸引力，越来越多的企业正在采用云服务。其优势在于无需前期资金与时间投入，也无需决策设备类型和规模。目前，云服务主要集中于数据库和办公应用领域，但具备高性能互连的科学云正在开发中。

The following is a broad classification of usage scenarios of cloud resources[11] .

- 扩展性。此处云资源被用作可按用户需求扩展的平台。这可视为平台即服务 (PAS)：云提供软件和开发平台，为用户免除运维负担。我们可区分两种情况：若用户运行单个任务并主动等待输出，可增加资源以最小化任务等待时间（能力计算）；另一方面，若用户将任务提交至队列且单任务完成时间不关键（容量计算），则可随队列增长添加资源。在 HPC 应用中，用户可将云资源视作集群；这属于基础设施即服务 (IAS)：云服务作为计算平台，允许在操作系统级别进行定制。

- 多租户。同一软件被提供给多个用户，每位用户均可进行个性化定制。这属于软件即服务（ SAS ）范畴：软件按需提供；客户无需购买软件，仅需支付使用费用。
- 批处理。这是前述扩展场景之一的简化版本：用户需以批处理模式处理大量数据。此时云平台充当批量处理器。该模式非常适合 MapReduce 计算；章节 2.11.3。
- 存储。多数云服务商提供数据库服务，该模式免除了用户自行维护数据库的负担，正如扩展模式和批处理模式消除了用户维护集群硬件的后顾之忧。
- 同步。该模式在商业用户应用中颇受欢迎。Netflix 和亚马逊 Kindle 允许用户消费在线内容（分别为流媒体电影和电子书）；暂停内容后可从任何其他平台继续播放。苹果近期推出的 iCloud 提供

11. 基于 RickyHo 的博客文章： http://blogs.globallogic.com/five-cloud-computing-patterns。

为办公应用程序中的数据提供同步功能，但与 Google Docs 不同，这些应用程序并非 ‘在云端’，而是运行在用户本地机器上。

首个向公众开放的云计算平台是亚马逊的弹性计算云（EC2），于 2006 年推出。EC2 提供多种不同的计算平台和存储设施。如今，已有超过百家公司提供基于云的服务，远远超出了最初 ‘租用计算机’ 的概念范畴。

从计算机科学的角度来看，云计算的基础设施颇具研究价值，涉及分布式文件系统、任务调度、虚拟化以及确保高可靠性的机制。

一个结合网格与云计算特点的有趣项目是加拿大天文研究高级网络 [169]。该项目以网格形式为天文学家提供大型中央数据集，同时以类云方式提供计算资源用于分析。值得注意的是，这些云资源甚至支持用户配置虚拟集群。

2.11.1.2 特征描述

总结 12 我们有三种云计算服务模式：

软件即服务 (SaaS) 消费者通过浏览器等瘦客户端运行提供商的应用，且无需安装或管理软件。典型案例如 Google 文档

平台即服务 (PaaS) 向消费者提供运行其自主开发应用的能力，消费者无需管理底层处理平台或数据存储

I基础设施即服务 (IaaS) 提供商向消费者提供运行软件及管理存储和网络的能力，消费者可自主选择操作系统和防火墙等网络组件

这些服务可按以下方式部署：

私有云 云基础设施由单一组织独占管理**公有云** 云基础设施面向广泛客户群体共享管理

人们也可以定义混合模型，例如社区云。

云计算的特征如下：

按需自助服务 消费者无需与供应商进行人工交互，即可快速请求服务并变更服务级别。

快速弹性扩展 对消费者而言，存储容量或计算能力看似无限，仅受预算限制。请求额外资源的速度极快，某些情况下可自动完成。

资源池化 虚拟化机制使云表现为单一实体，无论其底层基础设施如何。某些情况下，云会记录用户访问的 “ 状态 ” ；例如亚马逊 Kindle 电子书允许用户在 PC 和智能手机上阅读同一本书，云端存储的书籍会 “ 记住 ” 读者的阅读进度，不受平台限制。

网络访问 云可通过多种网络机制访问，从网页浏览器到专用门户均可。

12. 本节剩余内容基于美国国家标准与技术研究院（NIST）对云计算的定义 http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf。

Measured service 云服务通常是 “ 按量计费 ” 的，消费者需为计算时间、存储和带宽付费。

2.11.2 能力计算与容量计算

大型并行计算机可通过两种不同方式使用。后续章节将展示科学问题如何几乎无限扩展。这意味着随着精度或规模需求的提升，需要越来越大的计算机。将整台机器用于单一问题，仅以求解时间为成功衡量标准的方式，称为 *capabilitycomputing*（能力计算）。

另一方面，许多问题的求解无需整台超级计算机，因此计算中心通常会将机器配置为持续处理用户问题流，每个问题规模都小于整机容量。此模式下，成功标准是单位成本的持续性能。这被称为 *capacity computing*（容量计算），需要精细调整的 *job scheduling*（作业调度）策略。

主流方案是 *fair-sharescheduling*（公平份额调度），其尝试在用户间而非进程间均衡分配资源。这意味着若用户近期有作业运行，其优先级会被降低，而短小作业会获得更高优先级。采用此原则的调度器实例包括 *SGE* 和 *Slurm*。

作业之间可能存在依赖关系，这使得调度变得更加困难。事实上，在许多现实条件下，调度问题属于 *NP* 完全问题，因此实践中会采用启发式算法。这一主题虽然有趣，但本书不再深入讨论。

2.11.3 MapReduce

MapReduce [41] 是一种面向特定并行操作的编程模型。其显著特征之一是采用函数式编程实现。该模型处理以下形式的计算：

- 遍历所有可用数据，筛选符合特定条件的项目；
- 并为这些项目生成键值对。此为映射阶段。
- 可选地可包含合并 / 排序阶段，将具有相同键值的所有键值对分组。
- 然后对键进行全局归约，生成一个或多个对应的值。此为归约阶段。

接下来我们将给出几个使用 MapReduce 的示例，并阐述支撑 MapReduce 抽象背后的函数式编程模型。

2.11.3.1 MapReduce 模型的表达能力

MapReduce 模型中的归约部分使其成为计算数据集全局统计量的理想选择。例如统计一组单词在若干文档中各自出现的次数：映射函数知晓单词集合，会为每个文档输出文档名称与单词出现次数的列表构成的键值对。随后归约操作会对这些出现次数执行组件式求和。

MapReduce 的 combine 阶段使得数据转换成为可能。一个例子是 ‘ 反向网页链接图 ’ ：map 函数为每个指向目标 URL 的链接输出目标 - 源对，该链接位于名为 “ 源 ” 的页面中。reduce 函数将关联到给定目标 URL 的所有源 URL 列表拼接起来，并发出目标 - 列表 (源) 对。

一个不太明显的例子是使用 MapReduce 计算 PageRank （章节 10.4 ）。这里我们利用 PageRank 计算依赖于分布式稀疏矩阵 - 向量乘积的特性。每个网页对应网络矩阵的一列 W；给定处于页面 p_j 的概率 j，该页面随后可计算元组 $\langle i, w_{ij} p_j \rangle$。MapReduce 的 combine 阶段随后将 $(Wp)_i = \sum_j w_{ij} p_j$ 求和。

数据库操作可通过 MapReduce 实现，但由于其延迟相对较高，不太可能与独立数据库竞争，后者针对快速处理单个查询而非批量统计进行了优化。

排序 与 MapReduce 相关的讨论参见章节 9.5.1。

其他应用场景请参阅 http://horicky.blogspot.com/2010/08/designing-algorithmis-for-map-reduce.html。

2.11.3.2 MapReduce 软件

谷歌实现的 MapReduce 以 *Hadoop* 为名发布。虽然它适合 G
oogle 单阶段数据读取与处理模型，但对许多其他用户而言存在显著缺陷：
o其他用户：

- Hadoop 会在每个 MapReduce 周期后将所有数据刷新回磁盘，因此对于需要超过单个周期的操作，文件系统和带宽需求会变得过高。
- 在计算中心环境中，用户数据并非持续在线，将数据加载到 *Hadoop File System (HDFS)* 所需的时间很可能会超过实际分析的时间。

F基于这些原因，后续项目如 Apache *Spark* (https://spark.apache.org/) 提供了缓存
o数据。

2.11.3.3 实现问题

I在分布式系统上实现 MapReduce 存在一个有趣的问题：键集合中的键
键值对的分配是动态决定的。例如，在上述 “ 词频统计 ” 类应用中，我们并不 *a priori* 知晓单词集合，因此无法明确应将键值对发送至哪个归约器进程处理。

例如，我们可以使用哈希函数来确定这一点。由于每个进程都使用相同的函数，因此不会产生分歧。这遗留的问题是进程不知道需要接收多少条包含键值对的消息。此问题的解决方案已在 7.5.6 节中描述。

2.11.3.4 函数式编程

映射（ mapping ）和归约（ reduction ）操作能够轻松地在任何类型的并行架构上实现，通过结合线程与消息传递机制即可完成。然而，在开发此模型的谷歌环境中，传统并行方案并不受欢迎，原因有二：其一，处理器可能在运行过程中发生故障，

传统并行计算模型需增强容错机制以适应云计算环境。其次，硬件可能已有负载，部分计算需迁移，且任务间任何形式的同步都将极为困难。

MapReduce 是通过采用函数式编程模型来抽象并行计算细节的一种方式。该模型中唯一操作是函数求值：将函数应用于某些参数（这些参数本身也是函数应用的结果），计算结果再作为另一函数应用的输入。严格函数式模型中不存在变量，因而也没有静态数据。

以 *Lisp* 风格书写的函数应用 `(f a b)`（表示函数 `f` 应用于参数 `a` 和 `b`），其执行过程是：从数据所在处收集输入，并将其传递至执行函数 `f` 的处理器。MapReduce 流程的映射阶段标记为

```
(map f (some list of arguments))
```

结果是一个列表，包含将 `f` 应用于输入列表后得到的函数结果。所有并行处理的细节以及确保计算成功完成的保障都由 `map` 函数处理。

现在我们仅缺还原阶段，这一阶段同样简单：

```
(reduce g (map f (the list of inputs)))
```

`reduce` 函数接收一个输入列表并对其执行归约操作。

这种函数式模型的吸引力在于函数不会产生副作用：因为它们只能产生单一输出结果，无法改变其运行环境，因此不存在多任务访问同一数据的协调问题。

T因此，MapReduce 对于处理海量数据的程序员而言是一种实用的抽象工具。当然，在实现层面，MapReduce 软件采用了诸如数据空间分解、维护工作列表、任务分配至处理器、失败操作重试等常见概念。

2.11.4 TOP500 榜单

衡量计算机 “ 规模 ” 有几种非正式方法，其中最流行的是 TOP500 榜单，该榜单由 http://www.top500.org/ 维护，记录计算机在 ***Linpack*** 基准测试上的性能表现。*Linpack* 是一个用于线性代数运算的包，现已不再使用，因其被共享内存领域的 *Lapack* 及分布式内存计算机领域的 *Scalapack* 所取代。基准测试操作是通过部分主元 LU 分解求解（方形、非奇异、稠密）线性方程组，随后进行前向和后向求解。

LU 分解操作具有极高的缓存复用潜力，因其基于 1.6.1 节讨论的矩阵 - 矩阵乘法内核。该操作还具有计算量远超通信量的属性：$O(n^3)$ 对比 $O(n^2)$。因此，Linpack 基准测试很可能以机器峰值速度的显著比例运行。另一种表述方式是：Linpack 基准测试属于 ***CPU*** 密集型或计算密集型算法。

典型的效率数值介于 60% 至 90% 之间。但需注意，许多科学代码并不包含密集线性求解核心，因此该基准测试的性能并不能代表典型代码的性能表现。例如，通过迭代方法（第 5.5 节）求解线性系统时，在每秒浮点运算次数的意义上效率要低得多，其性能主要受 CPU 与内存之间的带宽限制（属于带宽受限型算法）。

常用的 Linpack 基准测试实现之一是 “高性能 LINPACK”（http://www.netlib.org/benchmark/hpl/），它包含区块大小等多个可调参数以优化性能。

2.11.4.1 Top500 榜单：超级计算的近代史

Top500 榜单记录了近 20 年的超级计算发展史。本节我们将简要回顾历史演进过程 [13]。首先，图 2.43 展示了架构类型的演变趋势：

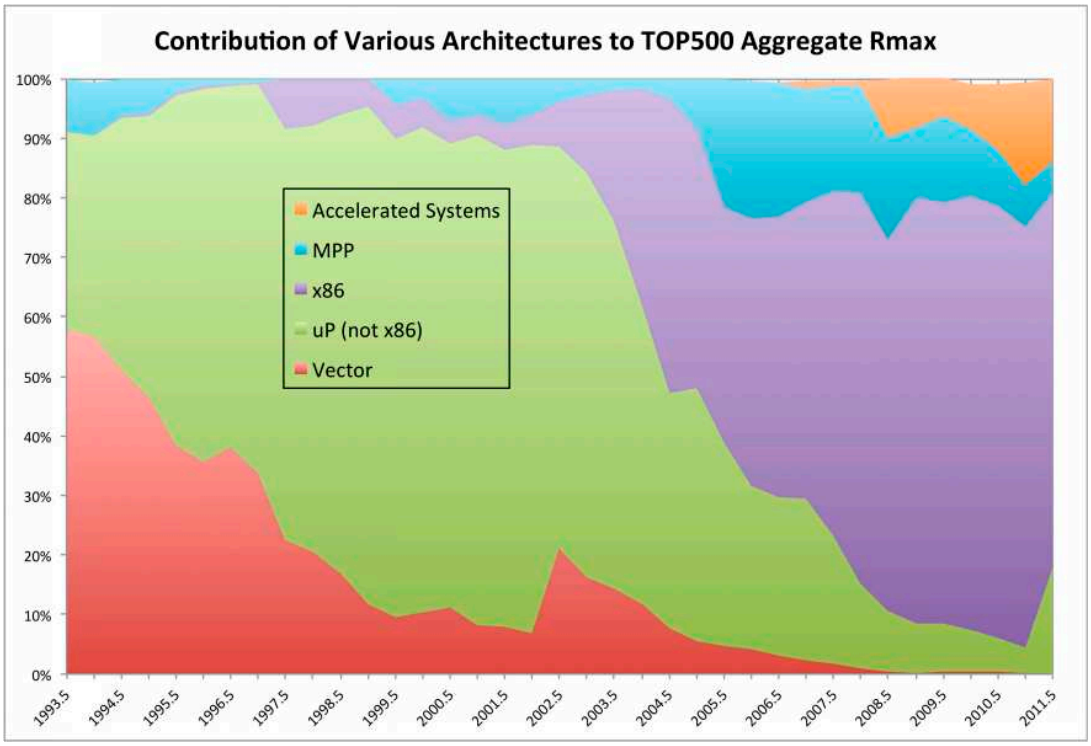


图 2.43：top500 榜单上架构类型的演变历程。

图表展示了各类架构对榜单总峰值性能的贡献比例。

- 向量机采用少量但性能极强的向量流水线处理器（章节 2.3.1.1）。此类架构已基本消失，最后一台著名机型是日本的地球模拟器，对应图中 2002 年左右的峰值数据，该机器曾连续两年蝉联榜首。

13. 图表数据基于 John McCalpin 对 top500 榜单的分析。

- 基于微处理器的架构通过单台机器中大量处理器的数量获得其性能优势。图中区分了 *x86*（*Intel* 和 *AMD* 处理器，但 *IntelItanium* 除外）处理器与其他类型；另见下一图表。
- 若干系统被设计为高度可扩展的架构：这些被称为 MPP（‘大规模并行处理器’）。在时间线早期，这包括诸如 *Connection Machine* 的架构，后期则几乎完全是 *IBMBlueGene*。
- 近年来，‘加速系统’ 成为新兴趋势。在此类系统中，诸如 *GPU* 的处理单元被附加到联网的主处理器上。

接下来，图 2.44 展示了 *x86* 处理器类型相对于其他微处理器的统治地位。（自

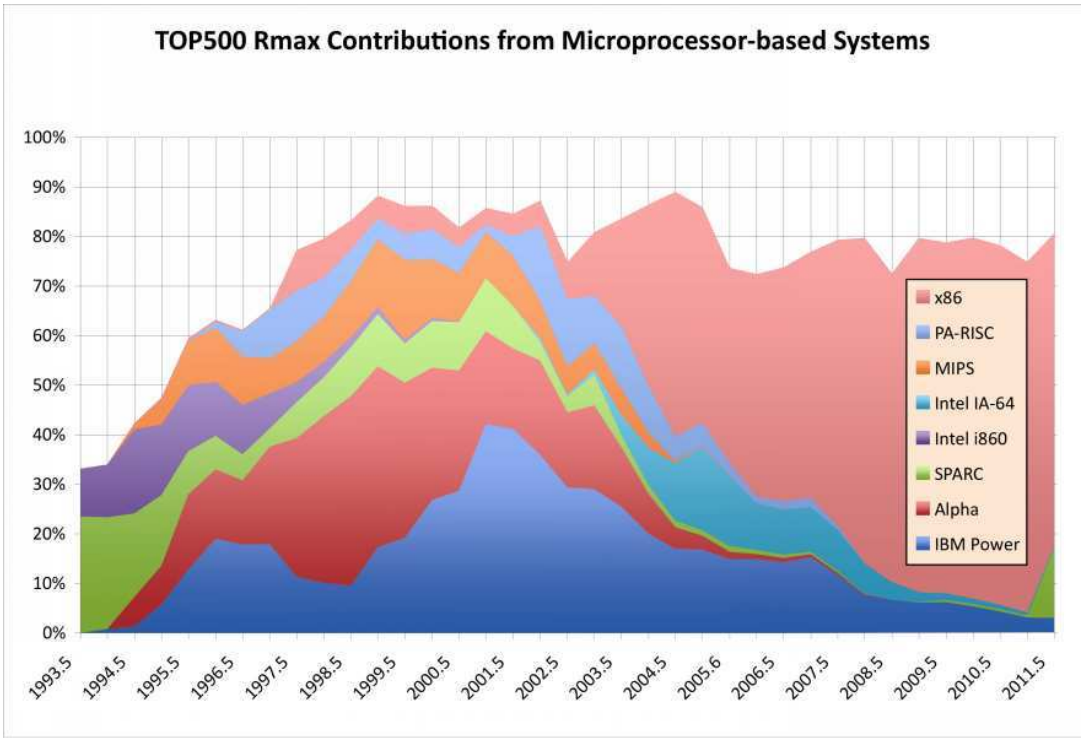


图 2.44：top500 列表中架构类型的演变。

我们将 *IBM BlueGene* 归类为 MPP，但其处理器在此不属于 ‘Power’ 类别。

最后，图 2.45 展示了核心数量的逐步增长。从中我们可以得出以下观察结果：

- 在 20 世纪 90 年代，许多处理器由多个芯片组成。在图表其余部分，我们统计的是每个 ‘包’（即每个插槽）的核心数量。某些情况下，一个插槽实际上可能包含两个独立的晶片。
- 随着多核处理器的出现，图表中该部分的曲线近乎垂直，这一现象十分显著。这意味着新型处理器被极其迅速地采用，而较低核心数的产品同样快速地彻底消失。

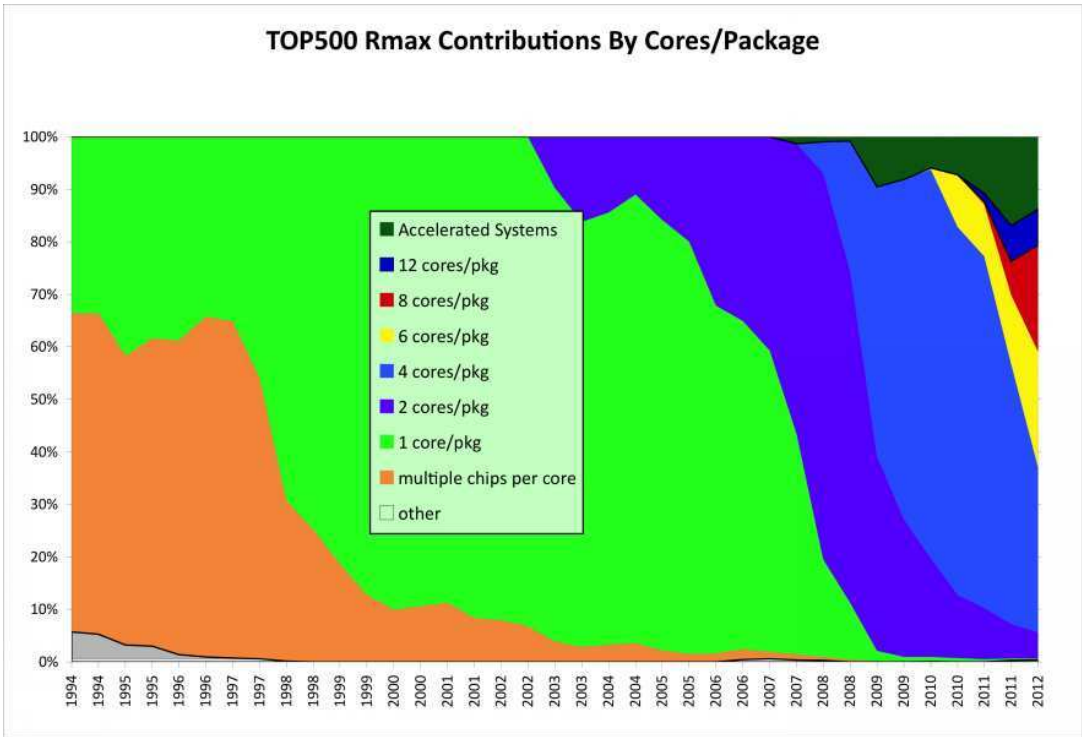


图 2.45：TOP500 榜单上架构类型的演变历程。

- 对于加速系统（主要是配备 GPU 的系统），' 核心数量 ' 的概念较难界定；该图表仅展示了此类架构日益增长的重要性。

2.11.5 异构计算

至此您已接触过多种计算模型：单核、共享内存多核、分布式内存集群、 GPU。这些模型的共性是 —— 若存在多个活跃指令流，所有流均可互换。关于 GPU 需修正此表述：*GPU* 上的所有指令流可互换。但 GPU 并非独立设备，可视为主处理器的协处理器。

若要在协处理器运行期间使主机执行有效工作，就会存在两种不同的指令流或流类型。这种情况称为异构计算。以 GPU 为例，这些指令流甚至通过略有差异的机制编程 —— 使用 *CUDA* 编写 GPU 代码 —— 但并非必须如此：英特尔众核架构（ MIC ）就是用普通 C 语言编程的。

第三章

Computer Arithmetic

在常见的数据类型中，科学计算领域主要关注数值类型：整数（或称整数） $\ldots$、 -2、 -1、 0、 1、 2、 $\ldots$；实数 0、 1、 -1.5、 $2/3$、 $\sqrt{2}$、 $\log 10$、 $\ldots$；以及复数 $1+2i$、 $\sqrt{3}-\sqrt{5}i$、 $\ldots$。计算机硬件设计为每个数字分配固定存储空间（以字节为单位，每字节 8 位），典型配置为：整数占 4 字节，实数占 4 或 8 字节，复数占 8 或 16 字节。

由于存储数字的内存空间有限，显然无法存储某类数字的全部可能值。例如整数仅能存储特定范围（如 *Python* 等语言支持任意长度的大整数，但这并无硬件支持）。对于实数，由于任何区间 $[a,b]$ 都包含无限多个数，连存储范围也无法实现。因此，任何实数表示法都会导致存储数值之间存在间隙。计算机运算常被称为有限精度算术，以反映通常无法以无限精度存储数字的特性。

由于许多结果无法精确表示，任何产生此类数值的计算都需通过报错或近似处理来解决。本章将探讨计算机如何实现有限精度，以及这种对数值计算 ‘ 真实 ’ 结果的近似所带来的影响。

For detailed discussions,

- 浮点运算的终极参考是 *IEEE 754* 标准 [1]；
- 延伸阅读可参考 Overton 的著作 [155]；Goldberg 的论文 [78] 在线副本易于获取；
- 关于算法中舍入误差分析的深入讨论，请参阅 Higham [104] 和 Wilkinson [192] 的专著。

3.1 比特

A在最低层级，计算机存储与数值以比特为单位组织。比特（ bit ）是 ‘ 二进制数字 ’ 可以取值为零和一。利用比特，我们就能用二进制表示法来表达数字：

$$10010_2 \equiv 18_{10} \tag{3.1}$$

其中下标表示数字系统的基数，且在这两种情况下，最右边的数字都是最低有效位。

内存的下一个组织层级是字节：一个字节由八位组成，因此可以表示 $0 \cdots 255$ 这些值。

Exercise 3.1. Use bit operations to test whether a number is odd or even.
Can you think of more than one way?

3.2 整数

在科学计算中，大多数运算都是针对实数进行的。整数运算很少会累积成严重的计算负担，除非是在诸如密码学等应用中。还有一些应用如 ‘粒子网格法’ 可以通过位操作实现。然而，在索引计算中仍然会遇到整数。

整数通常以 16 位、 32 位或 64 位存储，其中 16 位逐渐减少使用，而 64 位越来越普遍。这种增长的主要原因并非计算性质的变化，而是整数被用于数组索引的事实。随着数据集规模的增大（特别是在并行计算中），需要更大的索引。例如，32 位可以存储从零到 $2^{32}-1 \approx 4 \cdot 10^9$的数字。换句话说，32 位索引可以寻址 4GB 的内存。直到最近，这对大多数用途来说已经足够；如今，对更大数据集的需求使得 64 位索引成为必要。

当我们对数组进行索引时，只需要正整数。当然，在一般的整数运算中，我们也需要容纳负整数。现在我们将讨论几种实现负整数的策略。我们的动机在于，正负整数的算术运算应当与仅使用正整数时同样简单：我们用于比较和操作比特串的电路应当同样适用于（有符号）整数。

实现负整数有几种方法。最简单的解决方案是保留一个比特作为符号位，并使用剩余的 31 位（或 15 位、 63 位；从现在起我们将 32 位视为标准）来存储绝对数值。相比之下，我们会将比特串直接解释为无符号整数。

比特串	$00\cdots0$	$\ldots$	$01\cdots1$	$10\cdots0$	$\ldots$	$11\cdots1$
解释为无符号整数	0	$\ldots$	$2^{31}\ \underline{31}1$	2	$\ldots$	$2^{32}-1$
作为有符号整数的解释	0	$\cdots$	$2^{31}-1$	-0	$\cdots$	$-(2^{31}-1)$

(3.2)

该方案存在一些缺点，其中之一是同时存在正零和负零。这意味着相等性测试变得比简单地按位串测试相等性更为复杂。更重要的是，在位串的后半部分，作为有符号整数的解释会向右递减。这意味着大于测试变得复杂；此外，将正数加到负数上现在必须与将其加到正数上区别对待。

另一种解决方案是让无符号数 n 被解释为 $n-B$，其中 B 是某个合理的偏置值，例如 2_{31}。

位串	$00\cdots0$	...	$01\cdots1$	$10\cdots0$	...	$11\cdots1$
解释为无符号整型	0	...	$2^{31}-1$	2^{31}	...	$2^{32}-1$
作为偏移整数的解释	-2^{31}	...	-1	0	...	$2^{31}-1$

(3.3)

这种偏移方案不会受到 ±0 问题的影响，且数字的排序始终一致。然而，如果我们通过操作表示 n 的位串来计算 $n-n$，得到的并非表示零的位串。

为确保这一预期行为，我们改为旋转数轴，将正负数重新排列以使零的模式回归零点。最终形成的方案 —— 最广泛使用的方案 —— 被称为 *2* 的补码。采用此方案后，整数的表示形式定义如下。

定义 2 设 n 为整数，则其二进制补码 $'$ 位模式 $'$ $\beta(n)$ 是一个非负整数，定义如下：

- 若 $0\le n\le 2^{31}-1$，则使用 n 的标准位模式，即

$$0\le n\le 2^{31}-1\Rightarrow \beta(n)=n. \tag{3.4}$$

- 对于 $-2^{31}\le n\le -1$， n 由 $2^{32}-|n|$的位模式表示：

$$-2^{31}\le n\le -1\Rightarrow \beta(n)=2^{32}-|n|. \tag{3.5}$$

我们用 $\eta=\beta^{-1}$表示将位模式解析为整数的逆函数。

The following diagram shows the correspondence between bitstrings and their interpretation as 2's complement integer:

位串 n	$00\cdots0$	...	$01\cdots1$	$10\cdots0$	...	$11\cdots1$
解释为无符号整数	0	...	$2^{31}-1$	2^{31}	...	$2^{32}-1$
解释 $\beta(n)$ 为 2 的补码整数	0	$\cdots$	$2^{31}-1$	-2^{31}	$\cdots$	-1

(3.6)

Some observations:

- 正负整数的位模式之间没有重叠，特别是
  零值仅对应唯一一种位模式。
- 正数的首位是零，而负数的首位是一。这使得首位如同符号位；但请注意上述讨论。

- 对于一个正数 n，可以通过翻转所有位并加一来得到 $-n$。

练习 3.2. 针对负数的 ' 原码 ' 表示法和 2 的补码表示法，给出比较测试 $m<n$ 的伪代码，其中 m 和 n 是整数。注意区分所有情况，包括 m、 n 为正数、零或负数的情况。

3.2.1 整数溢出

将两个同符号数相加，或任意符号的两个数相乘，可能导致结果过大或过小而无法表示。这种现象称为溢出；关于浮点数的类似现象，请参阅章节 3.3.4 。以下练习可让您观察实际程序的行为。

练习 3.3. 探究执行此类计算时会发生什么。如果您尝试显式写出一个无法表示的数字（例如在赋值语句中），您的编译器会提示什么信息？

若用 C 语言实现此操作，值得注意的是：虽然您可能会得到一个可理解的结果，但根据 C 语言标准，有符号数溢出的行为实际上是未定义的。

3.2.2 二进制补码加法

L让我们考虑对补码整数进行一些简单的算术运算。我们首先假设我们
h拥有处理无符号整数的硬件。目标是证明我们可以利用此硬件来
c执行以补码表示的有符号整数运算。

We consider the 计算 $m+n$，其中 m,n 可表示为 n umbers:

$$0 \leq |m|, |n| < 2^{31}. \tag{3.7}$$

我们区分不同情况。

- 简单的情况是 $0 < m, n$。此时我们执行常规加法，只要结果保持在 2_{31} 以下，就能得到正确结果。若结果达到或超过 2_{31}，则发生整数溢出，对此我们无能为力。

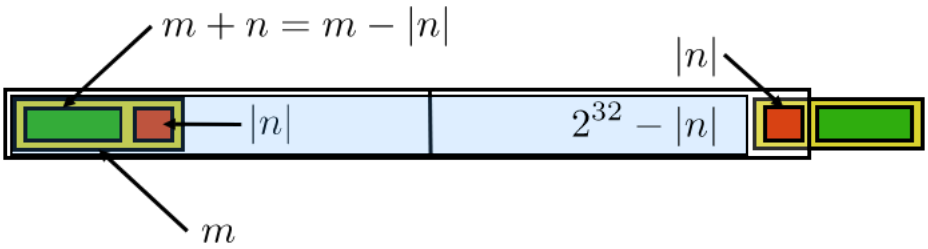


图 3.1：二进制补码表示中一正一负两数相加 .

- 情况 $m > 0$、 $n < 0$ 和 $m+n > 0$。此时 $\beta(m) = m$ 且 $\beta(n) = 2^{32} - |n|$，因此无符号加法运算结果为

$$\beta(m) + \beta(n) = m + (2^{32} - |n|) = 2^{32} + m - |n|. \tag{3.8}$$

由于 $m - |n| > 0$，该结果为 $> 2^{32}$。（参见图 3.1。）但我们注意到这本质上是 $m+n$ 且第 33 位被置位。若忽略该溢出位，即可得到正确结果。

- 情况 $m > 0$ 和 $n < 0$，但 $m+n < 0$。此时

$$\beta(m) + \beta(n) = m + (2^{32} - |n|) = 2^{32} - (|n| - m). \tag{3.9}$$

Since $|n| - m > 0$ we get

$$\eta\big(2^{32} - (|n| - m)\big) = -|(|n| - m)| = m - |n| = m + n. \tag{3.10}$$

3.2.3 二进制补码减法

在练习 3.2 中，你探索了如何比较两个整数。现在让我们探讨二进制补码减法是如何实现的。考虑 $0 \le m \le 2^{31}-1$ 和 $1 \le n \le 2^{31}$，看看在计算 $m-n$ 时会发生什么。

假设我们有一个用于加减无符号 32 位数的算法。能否用它来执行二进制补码整数的减法？我们首先注意到整数减法 $m-n$ 等价于无符号加法 $m+(2^{32}-n)$。

- 情况：$m < |n|$。此时，$m-n$ 为负数且 $1 \le |m-n| \le 2^{31}$，因此 $m-n$ 的位模式为

$$\beta(m-n) = 2^{32} - (n-m). \tag{3.11}$$

  现在，$2^{32}-(n-m) = m+(2^{32}-n)$，因此我们可以通过将 m 和 $-n$ 的位模式作为无符号整数相加来计算 $m-n$ 的二进制补码：

$$\eta\big(\beta(m)+\beta(-n)\big) = \eta\big(m+(2^{32}-|n|)\big) = \eta\big(2^{32}+(m-|n|)\big) = \eta\left(2^{32}-\big|m-|n|\big|\right) = m-|n| = m+n. \tag{3.12}$$

- 案例：$m > n$。此处我们观察到 $m+(2^{32}-n) = 2^{32}+m-n$。由于 $m-n>0$，这是一个数字 $> 2^{32}$，因此不能合法表示负数。但若将此数存储在 33 位中，会发现它是正确结果 $m-n$ 加上第 33 位的一个单独比特。因此，通过执行无符号加法并忽略溢出位，我们再次得到了正确结果。

两种情况下我们均可得出结论：可以通过将表示 m 和 $-n$ 的无符号数相加并忽略可能发生的溢出，来完成减法 $m-n$。

3.2.4 二进制编码的十进制

十进制数在科学计算中并不重要，但在金融计算中非常有用，因为涉及货币的计算必须绝对精确。二进制算术在此处于劣势，因为诸如 $1/10$ 这样的数字在二进制中是循环小数。由于尾数位数有限，这意味着数字 $1/10$ 无法用二进制精确表示。因此，**二进制编码的十进制**方案曾被用于旧式 IBM 大型机，且事实上正被纳入 *IEEE 754* [1, 110] 的修订标准中；另见章节 3.4。

在 BCD （二进制编码十进制）方案中，一个或多个十进制数字通过若干位二进制数进行编码。最简单的方案是将数字 $0\ldots9$ 用 4 位二进制编码。这种方案的优点是 BCD 数字中的每个数字都能被直接识别；缺点是约 $1/3$ 的二进制位被浪费，因为 4 位二进制可编码的数字范围为 $0\ldots15$。更高效的编码方式会用 10 位二进制来编码 $0\ldots999$，理论上可存储的数字范围为 $0\ldots1023$。虽然这种编码在减少位浪费方面效率较高，但要识别此类数字中的单个数字需进行解码操作。因此，BCD 运算需要处理器提供硬件支持，这在当今已较为罕见；IBM Power 架构（从 *IBM Power6* 开始）是其中一例。英特尔曾拥有 ‘BCD 操作码’，用 4 位存储一个十进制数字，因此一个字节可表示的数字范围为 $0\cdots99$。这些操作码在 64 位模式下已被移除。

3.3 实数

本节将探讨计算机中实数表示的原理及各种方案的局限性。章节 3.4 将讨论 IEEE 754 的具体解决方案，而章节 3.5 则会深入探讨该方案对计算机数字运算的影响。

3.3.1 它们并非真正的实数

在数学科学中，我们通常使用实数进行计算，因此很容易假设计算机也能做到这一点。然而，由于计算机中的数字仅由有限位数表示，大多数实数无法被精确表示。事实上，甚至许多分数也无法精确表示，因为它们会无限循环；例如，$1/3 = 0.333\ldots$，这在十进制或二进制中都无法精确表示。下面的练习 3.6 将对此进行说明。

练习 3.4. 某些编程语言允许在循环中使用实数而非仅整数作为“计数器”。解释为何这是一个糟糕的主意。提示：上限何时达到?

一个分数是否循环取决于所使用的数制系统。（如何用三进制或基数为 3 的算术表示 $1/3$？）在二进制计算机中，这意味着诸如 $1/10$ 这样的分数，在十进制算术中是终止的，而在二进制中却是循环的。由于十进制算术在金融计算中至关重要，部分人关注此类算术的精确性；详见章节 3.2.4，简要探讨了计算机硬件如何实现这一点。

习题 3.5. 证明每个二进制分数（即形如 0.01010111001_2 的数）均可精确表示为终止的十进制分数。为何并非所有十进制分数都能表示为二进制分数?

习题 3.6. 前文提到许多分数无法用二进制精确表示。请通过以下操作演示：先将一个数除以 7，再乘以 7，最后除以 49。尝试以 3.5 和 3.6 作为输入值。

3.3.2 实数表示法

实数采用类似于‘科学记数法’的方案存储，其中数字由有效数和指数表示，例如 $6.022 \cdot 10^{23}$，其有效数为 6022，第一个数字后有一个小数点，指数为 23。这个数字代表

$$6.022 \cdot 10^{23} = \left[6 \times 10^0 + 0 \times 10^{-1} + 2 \times 10^{-2} + 2 \times 10^{-3}\right] \cdot 10^{23}. \tag{3.13}$$

为了一般化处理，我们引入一个基数 β，它是一个小的整数，在前例中为 10，在计算机数字中为 2。基于此，我们将数字表示为 t 项的和：

$$\begin{aligned} x &= \pm 1 \times \left[d_1\beta^0 + d_2\beta^{-1} + d_3\beta^{-2} + \cdots + d_t\beta^{-t+1}b\right] \times \beta^e \\ &= \pm \Sigma_{i=1}^{t} d_i\beta^{1-i} \times \beta^e \end{aligned} \tag{3.14}$$

其中各组件为

- 符号位：一个单独的位，存储数字是正还是负；

- β 是数字系统的基数；
- $0 \le d_i \le \beta - 1$ 尾数 *mantissa* 或 *significand* 的各位数字 —— 基数点（十进制数中的小数点）的位置被隐式假定为紧跟在第一个数字 d_1 之后；
- t 是尾数的长度；
- $e \in [L,U]$ 指数；通常为 $L < 0 < U$ 和 $L \approx -U$。

请注意，整个数有一个显式的符号位，但指数的符号处理方式不同。出于效率考虑，e 不是一个有符号数；而是被视为一个无符号数，其值为某个最小值 *excess* 的偏移量。例如，用 8 位表示指数时，我们可以表示数字 $0 \cdots 256$，但偏移量为 128 时，这被解释为 $-127 \cdots 128$。此外，最高和最低的指数值可用于表示特殊值，如无穷大。

3.3.2.1 一些示例

让我们来看一些具体的浮点数表示法示例。基数为 10 对人类而言是最合乎逻辑的选择，但计算机是二进制的，因此基数 2 在那里占主导地位。老式 IBM 大型机将比特分组以实现基数为 16 的表示。

	β	t	L	U
IEEE 单精度（32 位）	2	23	−126	127
IEEE 双精度（64 位）	2	53	−1022	1023
老式 Cray 64 位	2	48	−16383	16384
IBM 大型机 32 位 16		6	−64	63
压缩十进制 10 50			−999	999

(3.15)

其中，单精度和双精度格式是最常见的。我们将在 3.4 节进一步讨论；关于压缩十进制，请参阅 3.2.4 节。

3.3.3 Normalized and unnormalized numbers

浮点数的一般定义，即公式 (3.14)，带来了一个问题：数字可能有多种表示形式。例如，$.5 \times 10^2 = .05 \times 10^3$。由于这会使计算机算术运算变得不必要的复杂，比如在测试数字相等性时，我们使用规范化浮点数。一个数字如果其首位数字非零，则称为规范化的。这意味着尾数部分为 $1 \le x_m < \beta$。

对于二进制数的一个实际影响是，首位数字始终为 1，因此我们无需显式存储它。在 *IEEE 754* 标准中，这意味着每个规范化浮点数都具有以下形式

$$1.d_1 d_2 \ldots d_t \times 2^e \tag{3.16}$$

并且仅存储数字 $d_1 d_2 \ldots d_t$。

习题 3.7. 对于量 t 和 e，最小的正规格化数是多少？

非规格化数确实存在，但仅适用于比最小正规格化数更小的数值。详见章节 3.3.4.3。

3.3.4 局限性：溢出与下溢

由于我们仅用有限位数存储浮点数，并非所有数字都能被精确表示。无法表示的数字可分为两类：数值过大或过小（在某种意义上的），以及落在表示间隙中的数值。

第二类情况 —— 计算结果需经舍入或截断才能被表示 —— 构成了舍入误差分析领域的基础。我们将在下文对此进行详细研究。

数值可能在以下方面表现得过大或过小 .

3.3.4.1 溢出

我们能存储的最大数字其每一位都等于 β:

	unit	小数部分			指数部分
数字	$\beta-1$	$\beta-1$	$\cdots$	$\beta-1$	
值	1	β^{-1}	$\cdots$	$\beta^{-(t-1)}$	U

(3.17)

总计为

$$(\beta-1)\cdot 1+(\beta-1)\cdot\beta^{-1}+\cdots+(\beta-1)\cdot\beta^{-(t-1)}=\beta-\beta^{-(t-1)}, \tag{3.18}$$

而最小的数（即最负的）是 $-\left(\beta-\beta^{-(t-1)}\right)$；任何大于前者或小于后者的情况都会引发一种称为溢出的状态。

3.3.4.2 下溢

最接近零的数是 $\beta^{-(t-1)}\cdot\beta^{L}$。

	unit	小数部分			指数
数字	0	0	$\cdots$	$0,\beta-1$	
值	0	0	$\cdots$	$0,\beta^{-(t-1)}$	L

(3.19)

计算结果（绝对值）小于该值的运算会导致一种称为下溢的情况。

若使用规范化数值，上述 ‘最小’ 数字实际上不可能存在。因此，我们还需考察 ‘渐进式下溢’。

3.3.4.3 渐进式下溢

The normalized number closest to zero is $1\cdot\beta^{L}$.

$$\begin{array}{lccccc} & \text{单位} & \text{分数} & & \text{指数} & \\ \text{数字} & 1 & 0 & \cdots & 0 & \\ \text{值} & 1 & 0 & \cdots & 0 & L \end{array} \tag{3.20}$$

尝试计算绝对值小于该值的数字时，有时会通过使用次正规（或非规范化或非正规化）数来处理，这一过程称为渐进式下溢。此时，指数的特殊值表明该数字不再规范化。在 IEEE 标准算术（第 3.4.1 节）中，这是通过零指数字段实现的。

然而，这种计算通常比常规浮点数运算慢数十至数百倍 [1]。截至本文撰写时，仅 IBM Power6（及更高版本）硬件支持渐进式下溢。6.3.1 节探讨了性能影响。

3.3.4.4 上溢 / 下溢时会发生什么？

上溢或下溢的出现意味着您的计算从该点开始将 ‘出错’。下溢会使计算在应为非零值的位置继续使用零值；上溢则表示为 `Inf`，即 ‘无限’ 的缩写。

习题 3.8. 对于实数 x、 y，量 $g = \sqrt{(x^2 + y^2)/2}$ 满足

$$g \leq \max\{|x|, |y|\} \tag{3.21}$$

因此，若 x 和 y 可表示，则该量亦可表示。若使用上述公式计算 g 可能出现什么问题？你能想到更好的方法吗？

使用 `Inf` 进行计算在一定范围内可行：将两个此类量相加仍会得到 `Inf`。然而，相减则会产生 `NaN`：‘非数值’（参见章节 3.4.2.1）。

这些情况下计算均不会自行终止：除非收到指令，处理器将持续运行。‘指令’ 是指要求编译器生成一个中断，该中断会停止计算并显示错误信息（参见章节 3.7.2.4）。

3.3.4.5 渐进下溢

归一化方案的另一个含义是需修正下溢定义（参见前文章节 3.3.4）：任何小于 $1 \cdot \beta^L$ 的数值现在都会引发下溢。

3.3.5 表示误差

让我们考虑一个在计算机数字系统中无法精确表示的实数。

不可表示的数字通常通过常规的四舍五入、向上取整或向下取整，或是截断来近似表示。这意味着机器数 x 代表了其周围区间内所有 x 的数值。在 t 的情况下

1. 在音频处理等实时应用中，这种现象尤为明显；参见 http://phonophunk.com/articles/pentium4-denormalization.php?pg=3。

尾数中的数字，这是与 x 在第 $t+1$ 位数字上不同的数字区间。对于尾数部分，我们得到：

$$\begin{cases} x \in \left[\tilde{x}, \tilde{x} + \beta^{-t+1}\right) & \text{truncation} \\ x \in \left[\tilde{x} - \frac{1}{2}\beta^{-t+1}, \tilde{x} + \frac{1}{2}\beta^{-t+1}\right) & \text{rounding} \end{cases} \tag{3.22}$$

如果 x 是一个数字，$\tilde{x}$ 是其在计算机中的表示，我们称 $x - \tilde{x}$ 为表示误差或绝对表示误差，而 $\frac{x-\tilde{x}}{x}$ 为相对表示误差。

通常我们对误差的符号不感兴趣，因此可以分别将术语 “ 误差 ” 和 “ 相对误差 ” 应用于 $|x - \tilde{x}|$ 和 $|\frac{x-\tilde{x}}{x}|$。

通常我们只关心误差的界限。如果 ϵ 是误差的一个界限，我们将写作

$$\tilde{x} = x \pm \epsilon \underset{\text{D}}{\equiv} |x - \tilde{x}| \leq \epsilon \Leftrightarrow \tilde{x} \in [x - \epsilon, x + \epsilon] \tag{3.23}$$

对于相对误差，我们注意到

$$\tilde{x} = x(1 + \epsilon) \Leftrightarrow \left|\frac{\tilde{x} - x}{x}\right| \leq \epsilon \tag{3.24}$$

让我们以十进制算术为例，即 $\beta = 10$，并采用 3 位尾数：$t = 3$。数字 $x = 1.256$ 的表示取决于我们采用四舍五入还是截断方式：x 四舍五入得 $= 1.26$，x $= 1.25$ 4 $\epsilon = x - x$ $|\epsilon| < \beta^{-(t-1)}$ ~ 截断得。误差出现在第位：若则。 ~

习题 3.9。此例中的数字无指数部分。若存在指数部分，其绝对误差与相对误差分别是多少？

习题 3.10。如前所述，在二进制算术中首位数字恒为 1。这会如何改变表示误差？

3.3.6 机器精度

通常我们只关心表示误差的数量级，故记作 $x = x(1 + \epsilon)$，其中 $|\epsilon| \leq \beta^{-t}$。此最大相对表示误差称为机器精度，有时亦称机器 *epsilon*。典型值为：

$$\begin{cases} \epsilon \approx 10^{-7} & \text{32-bit single precision} \\ \epsilon \approx 10^{-16} & \text{64-bit double precision} \end{cases} \tag{3.25}$$

(在您学习了关于 IEEE 754 标准的章节后，能否根据尾数的位数推导出这些值？)

机器精度还可以这样定义：ϵ 是能够加到 1 上的最小数值，使得 $1 + \epsilon$ 的表示与 1 不同。一个小例子展示了指数对齐如何使过小的操作数发生位移，从而在加法运算中被实际忽略：

$$\begin{array}{rlll} & 1.0000 & \times 10^{0} \\ + & 1.0000 & \times 10^{-5} \end{array} \quad \Rightarrow \quad \begin{array}{rlll} & 1.0000 & \times 10^{0} \\ + & 0.00001 & \times 10^{0} \\ = & 1.0000 & \times 10^{0} \end{array} \tag{3.26}$$

另一种理解方式是注意到，在加法 $x + y$ 中，如果 x 与 y 的比值过大，结果将与 x 完全相同。

机器精度是计算可达到的最高精度：在单精度下要求超过约 6 位数字的精度，或在双精度下要求超过 15 位数字的精度，都是没有意义的。

练习 3.11. 编写一个小程序计算机器 epsilon。设置编译器优化级别高低是否会产生差异？（如果你懂 C++，能否用单一模板代码解决此题？）

练习 3.12. 自然对数的底数 $e \approx 2.72$ 有多种定义方式，其中之一是

$$e = \lim_{n\to\infty} (1 + 1/n)^n. \tag{3.27}$$

编写一个单精度程序尝试通过此方法计算 e。（不要使用 `pow` 函数：显式编写幂运算代码。）对上限 $n = 10^k$（其中 k）求值表达式。（你让 k 的范围有多大？）解释大 n 时的输出结果，并对误差行为进行评论。

练习 3.13. 指数函数 e^x 可通过以下方式计算：

$$e = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{3.28}$$

编写代码并尝试对一些正数 x 进行计算，例如 $x = 1$、3、10、50。你需要计算多少项？
你计算了多少项？
现在计算 e^{-x} 并从中得到 e^x。例如使用与 e^{-x} 相同的 x 值和迭代次数。
迭代次数与 e^x 相同。
你观察到关于 e^x 与 e^{-x} 计算的哪些现象？请解释。

3.4 IEEE 754 浮点数标准

数十年前，尾数长度和运算舍入行为等问题在不同计算机制造商之间，甚至同一制造商的不同型号之间都可能存在差异。从代码可移植性和结果可复现性的角度来看，这显然是个糟糕的状况。*IEEE754* 标准 [1] 于 1985 年将这一切规范化，例如规定单精度和双精度运算中尾数（归一化前）分别为 24 位和 53 位，采用符号位、指数、尾数的存储顺序；参见图 3.2。

注 9 *754* 标准的全称为《*IEEE* 二进制浮点算术标准（*ANSI/IEEE Std 754-1985*）》。该标准也与 *IEC 559*《微处理器系统二进制浮点算术》相同，后由 *ISO/IEC/IEEE 60559:2011* 取代。

IEEE 754 是二进制算术标准；另有一项 IEEE854 标准允许十进制算术运算。

符号位	指数位	尾数	符号位	指数位	尾数位
p	$e = e_1 \cdots e_8$	$s = s_1 \ldots s_{23}$	s	$e_1 \cdots e_{11}$	$s_1 \ldots s_{52}$
31	$30 \cdots 23$	$22 \cdots 0$	63	$62 \cdots 52$	$51 \cdots 0$
$\pm$	2^{e-127} （除 $e = 0$ 、 255 外）	$2^{-s_1} + \cdots + 2^{-s_{23}}$	$\pm$	2^{e-1023} （除 $e = 0$ 外，2047）	$2^{-s_1} + \cdots + 2^{-s_{52}}$

(3.29)

图 3.2：单精度与双精度浮点数定义。

备注 10 ‘令人瞩目的是，当时众多硬件领域人士明知 *p754* 标准推行之艰难，仍一致认同它应当惠及整个计算社区。若能促进浮点软件生态繁荣、提升可靠软件开发效率，终将扩大所有硬件厂商的市场规模。这种利他精神如此震撼人心，以至于 *MATLAB* 创始人克里夫·莫勒尔博士曾建议外国访客务必观摩美国两大奇观：大峡谷，以及 *IEEE p754* 标准会议。’—— 威廉·卡汉，*http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html*。

该标准同时将舍入行为定义为正确舍入：运算结果应为精确值的舍入版本。详见章节 3.5.1。

3.4.1 浮点数格式定义

IEEE 754 单精度浮点数所有位模式的含义清单如图 3.3 所示。回顾上文第 3.3.3 节可知，对于规范化数，首个非零数字是隐含存储的 1，因此位模式 $d_1 d_2 \ldots d_t$ 会被解析为 $1.d_1 d_2 \ldots d_t$。

最高指数位用于表示 `Inf` 和 `Nan`，详见第 3.4.2.1 节。最低指数位用于表示非规范化数，详见第 3.3.4.3 节。

习题 3.14. 每个程序员在其职业生涯中，都曾犯过将实数存入整数变量或反之的错误。例如当调用函数时实参与形参类型不匹配时就会发生这种情况。

```
void a(double x) {....}
int main() {
  int i;
  .... a(i) ....
}
```

当你在函数中打印 `x` 会发生什么？考虑一个小整数的位模式，并参考图 3.3 的表格将其解释为浮点数。说明这将是一个非规范化数。

（这是那种一旦犯过就永生难忘的错误。今后，每当你看到 10^{-305} 量级的数字时，就会意识到自己很可能又犯了这个错误。）

$$
\begin{array}{rll}
\text{指数} & \text{数值范围}_{-126} & \\
(0\cdots0)=0 & \pm 0.s_1\cdots s_{23}\times 2 & s=0\cdots01 \Rightarrow 2^{-23}\cdot 2^{-126}=2^{-149}\approx 10^{-50} \\
 & & s=1\cdots11 \Rightarrow (1-2^{-23})\cdot 2^{-126} \\
(0\cdots01)=1 & \pm 1.s_1\cdots s_{23}\times 2^{-126} & s=0\cdots00 \Rightarrow 1\cdot 2^{-126}\approx 10^{-37} \\
(0\cdots010)=2 & \pm 1.s_1\cdots s_{23}\times 2^{-125} & \\
\cdots & & \\
(01111111)=127 & \pm 1.s_1\cdots s_{23}\times 2^{0} & s=0\cdots00 \Rightarrow 1\cdot 2^{0}=1 \\
 & & s=0\cdots01 \Rightarrow 1+2^{-23}\cdot 2^{0}=1+\epsilon \\
 & & s=1\cdots11 \Rightarrow (2-2^{-23})\cdot 2^{0}=2-\epsilon \\
(10000000)=128 & \pm 1.s_1\cdots s_{23}\times 2^{1} & \\
\cdots & & \\
(11111110)=254 & \pm 1.s_1\cdots s_{23}\times 2^{127} & \\
(11111111)=255 & s_1\cdots s_{23}=0 \Rightarrow \pm\infty & \\
 & s_1\cdots s_{23}\neq 0 \Rightarrow \texttt{NaN} &
\end{array}
\tag{3.30}
$$

图 3.3：根据指数位模式对单精度数的解释。

如今，几乎所有处理器都遵循 IEEE 754 标准。早期几代 *NVIDIA Tesla GPU* 在单精度运算上并不符合标准。其理由是单精度更可能用于图形处理，而精确合规性在此领域重要性较低。对于许多科学计算而言，双精度是必要的，因为随着问题规模或运行时间的增加，计算精度会变差。这对于有限差分（ FD ）计算（第 4 章）成立，但对于其他如格子玻尔兹曼方法（ *LBM* ）则不然。

3.4.2 浮点异常

多种运算可能产生无法表示为浮点数的结果。这种情况称为异常，我们称异常被引发。（注意：这些并非 C++ 或 Python 异常。）结果取决于错误类型，计算会正常继续。（也可使程序中断：参见第 3.7.4 节。）

3.4.2.1 非数值

除了溢出和下溢，还存在其他异常情况。例如，当程序请求非法操作（如 $\sqrt{-4}$ ）时应返回什么结果？IEEE 754 标准为此定义了两个特殊量： `Inf` （表示 ‘无穷大’ ）和 `NaN` （表示 ‘非数值’ ）。无穷大是溢出或除以零的结果，而非数值则是诸如从无穷大中减去无穷大这类运算的结果。

准确地说，处理器会将以下运算结果表示为 `NaN` （ ‘非数值’ ）：

- 加法 `Inf-Inf`，但需注意 `Inf+Inf` 会得到 `Inf`；
- 乘法 $0 \times$ `Inf`；
- 除法 $0/0$ 或 `Inf/Inf`；
- 取余 `Inf`$\operatorname{rem} x$ 或 $x \operatorname{rem}$ `Inf`；
- $\sqrt{x}$ 对于 $x < 0$；
- 比较 $x < y$ 或 $x > y$ 其中任一操作数为 `Nan`。

由于处理器可以继续用此类数值进行计算，它被称为静默 `NaN`。相反，某些 NaN 值可能导致处理器生成中断或异常，这类情况被称为信号 `NaN`。

如果 `NaN` 出现在表达式中，整个表达式将计算为该值。处理 `Inf` 的运算规则则稍显复杂 [78]。

T信号 NaN （ Not-a-Number ）有其特定用途。例如，您可以用此类值填充已分配的内存，以便i表明该值在计算目的上是未初始化的。任何对此类值的使用都将被视为p程序错误，将引发异常。

IEEE 754 标准的 *2008* 年修订版建议使用 NaN 的最高有效位作为 `is_quiet` 位来区分静默 NaN 与信号 NaN。

参见 https://www.gnu.org/software/libc/manual/html_node/Infinity-and-NaN.html 关于 GNU `Nan` 编译器中的处理方法

备注 11 *NaN* 与 *Inf* 并不总是传播：

- 有限数除以 *Inf* 结果为零；
- 诸如 $\tan^{-1}(\text{Inf}) = \pi/2$ 等各类超越函数能正确处理 *Inf*。

2008 版标准将静默 NaN 视为‘缺失数据’，这意味着 *1* 与静默 *NaN* 的最小值为 *1*。这导致 *min/max* 运算不满足结合律。该问题在 *2019* 版标准中得以修正，规定 *NaN* 始终传播。

3.4.2.2 除零错误

除以零会导致 `Inf`。

3.4.2.3 溢出

当结果无法表示为有限数时，会引发此异常。

3.4.2.4 下溢

This exception is raised if a number is too small to be represented.

3.4.2.5 不精确

此异常会在出现不精确结果（如平方根）或未捕获的溢出时触发。

3.5 舍入误差分析

数值过大或过小导致无法表示的情况（即溢出和下溢）并不常见：通常可以通过调整计算方式避免此类问题。相比之下，计算结果（即便是简单的加法运算）无法精确表示的情形则极为普遍。因此，在分析算法实现时，我们需要研究这类微小误差在计算过程中传播的影响。这一分析通常被称为舍入误差分析。

3.5.1 正确舍入

IEEE 754 标准（见第 3.4 节所述）不仅规定了浮点数的存储方式，还对加法、减法、乘法、除法等运算的精度提出了标准化要求。该标准的算术模型基于正确舍入原则：运算结果应遵循以下流程得出：

- 首先精确计算运算结果，无论该结果是否可表示；
- 该结果随后被舍入至最接近的计算机可表示数字。

简而言之：运算结果的表示形式是该运算精确结果的舍入值。当然，经过两次运算后，计算结果不再需要保证是精确结果的舍入版本。

I如果这个陈述听起来琐碎或不言自明，请以减法为例。在一个十进制数
s系统中，若尾数保留两位数字，计算

$$\begin{aligned} 1.0 - 9.4 \cdot 10^{-1} = \\ 1.0 - 0.94 &= 0.06 \\ &= 0.6 \cdot 10^{-2} \end{aligned} \tag{3.31}$$

注意在中间步骤中出现了尾数 .094，这比我们数系声明的两位数多了一位。这个额外数字被称为 *guarddigit*（保护位）。

若无保护位，该运算将按 $1.0 - 9.4 \cdot 10^{-1}$进行，其中 $9.4 \cdot 10^{-1}$ 会被标准化为 0.9，最终结果为 0.1—— 几乎是正确结果的两倍误差。

习题 3.15. 考虑计算 $1.0 - 9.5 \cdot 10^{-1}$，并再次假设数字被舍入以适应 2 位尾数。为何该计算在某种程度上比示例糟糕得多?

单一保护位不足以保证正确舍入。我们此处不再复现的分析表明，需要额外三位二进制位 [77]。

3.5.1.1 乘加运算

2008 年，IEEE 754 标准经过修订，纳入了融合乘加 *(FMA)* 运算的行为规范，即形如以下形式的运算

$$c \leftarrow a * b + c. \tag{3.32}$$

该运算具有双重动机。

首先，FMA （乘加运算）比独立的乘法和加法运算可能更精确，因为它能以更高精度处理中间结果，例如使用 80 位扩展精度格式（详见 3.8.3 节）。

该标准将正确舍入定义为：这个组合运算的结果应当是经过舍入的正确结果。此运算的简单实现会涉及两次舍入：乘法后一次，加法后一次 [2]。

习题 3.16. 能否举例说明 FMA 的正确舍入比分别对乘法和加法进行舍入显著更精确？提示：让 `c` 项与 `a*b` 符号相反，并尝试在减法中迫使抵消发生。

其次，FMA 指令是实现更高性能的途径：通过流水线技术，我们渐近地每周期可执行两个操作。因此构建 FMA 单元比独立的加法和乘法单元成本更低。幸运的是，FMA 在实际计算中频繁出现。

习题 3.17. 你能想到哪些线性代数运算会用到 FMA 操作吗？参见章节 1.2.1.2 了解处理器中 FMA 的历史应用。

3.5.2 加法运算

两个浮点数的加法运算通过以下几个步骤完成。

1. 首先对齐指数：将较小的数调整为与较大数相同的指数。2. 然后相加尾数。3. 最后调整结果使其重新成为规范化表示。

以 $1.00+2.00\times10^{-2}$为例。指数对齐后变为 $1.00+0.02=1.02$，该结果无需最终调整。我们注意到这个计算是精确的，但求和 $1.00+2.55\times10^{-2}$会得到相同结果，此时计算显然不精确：精确结果应为 1.0255，而该值无法用三位尾数表示。

在示例 $6.15\times10^{1}+3.98\times10^{1}=10.13\times10^{1}=1.013\times10^{2}\rightarrow1.01\times10^{2}$ 中，我们可以看到尾数相加后需要对指数进行调整。该误差同样源于截断或舍入结果中超出尾数容纳范围的第一个数字：若 x 为真实和而 $\tilde{x}$ 为计算和，则 $\tilde{x}=x(1+\epsilon)$，其中对于一个 3 位尾数 $|\epsilon|<10^{-3}$。

形式上，让我们考虑 $s=x_1+x_2$ 的计算，并假设数字 x_i 表示为 $\tilde{x}_i=x_i(1+\epsilon_i)$。那么和 $\tilde{s}$ 被表示为

$$\begin{aligned}\tilde{s} &= (\tilde{x}_1+\tilde{x}_2)(1+\epsilon_3)\\ &= x_1(1+\epsilon_1)(1+\epsilon_3)+x_2(1+\epsilon_2)(1+\epsilon_3)\\ &\approx x_1(1+\epsilon_1+\epsilon_3)+x_2(1+\epsilon_2+\epsilon_3)\\ &\approx s(1+2\epsilon)\end{aligned} \tag{3.33}$$

在假设所有 ϵ_i 都很小且大小大致相等，且两者 $x_i>0$ 情况下，我们发现相对误差会在加法过程中累加。

2. 另一方面，如果应用程序的行为是通过非 FMA 架构 ‘认证’ 的，那么增加的精度会破坏认证。已知芯片制造商收到过关于 ‘双舍入’ FMA 的请求，以抵消数值行为的这种变化。

3.5.3 乘法运算

浮点数乘法与加法类似，需经过多个步骤。要将两个数

$m_1 \times$ 相乘，需执行以下步骤。

β^{e_1} m_2 与 β^{e_2}

- 指数相加： $e \leftarrow e_1 + e_2$。
- 尾数相乘： $m \leftarrow m_1 \times m_2$。
- 对尾数进行规范化处理，并相应调整指数。

例如： $1.23 \cdot 10^0 \times 5.67 \cdot 10^1 = 0.69741 \cdot 10^1 \rightarrow 6.9741 \cdot 10^0 \rightarrow 6.97 \cdot 10^0$。

练习 3.18. 分析乘法运算的相对误差。

3.5.4 减法

减法的表现与加法截然不同。加法运算中误差会累积，导致整体舍入误差逐步增加；而减法运算则可能在单次操作中引发误差的急剧放大。

例如，考虑尾数为 3 位的减法运算： $1.24 - 1.23 = 0.01 \rightarrow 1.00 \cdot 10^{-2}$。虽然结果精确，但其有效数字仅剩一位 [3]。假设第一个操作数 1.24 实际上是某次计算结果（本应为 1.235 ）的舍入值：

$$\begin{aligned} .5 \times 2.47 - 1.23 \quad &= 1.235 - 1.23 && = 0.005 \quad \text{exact} \\ \downarrow \qquad\qquad\quad & && = 5.0 \cdot 10^{-3} \\ 1.24 - 1.23 \quad &= 0.01 = 1. \cdot 10^{-2} \quad \text{compute} \end{aligned} \tag{3.34}$$

此时即使输入值的相对误差已小到预期极限，最终误差仍高达 100%。显然，后续基于该减法结果的操作也将不准确。由此我们得出结论：对近似相等的数进行相减是引发数值舍入误差的常见原因。

关于此示例存在一些精妙之处。几乎相等数字的减法运算是精确的，且我们遵循了 IEEE 算术的正确舍入行为。然而，单一运算的正确性并不意味着包含该运算的序列操作就必然准确。虽然加法示例仅显示出数值精度的轻微下降，但本例中的抵消效应可能引发灾难性后果。您将在 3.6.1 节中看到相关示例。

习题 3.19. 考虑以下迭代

$$x_{n+1} = f(x_n) = \begin{cases} 2x_n & \text{if } 2x_n < 1 \\ 2x_n - 1 & \text{if } 2x_n \ 1 \end{cases} \tag{3.35}$$

该函数是否存在不动点 $x_0 \equiv f(x_0)$，抑或存在循环 $x_1 = f(x_0)$、 $x_0 \equiv x_2 = f(x_1)$ 等？现在编写此函数代码。能否复现这些不动点？不同起始点 x_0 会导致何种现象？你能解释其原因吗？

3. 通常，尾数部分具有 3 位数字的数表明其误差对应于第四位数字的舍入或截断。我们称此类数字具有 3 有效数字。本例中，最后两位数字因归一化过程而无实际意义。

3.5.5 结合律

浮点数的实现意味着简单的算术运算（如加法或乘法）不再像数学中那样具有结合性。

让我们通过一个简单示例，展示浮点数舍入行为如何影响结合律。假设浮点数存储为尾数一位、指数一位及一个保护位；现考虑计算 $4+6+7$。从左到右求值得出：

$$\begin{array}{lll} (4\cdot 10^0 + 6\cdot 10^0) + 7\cdot 10^0 \Rightarrow 10\cdot 10^0 + 7\cdot 10^0 & & \text{addition} \\ \Rightarrow 1\cdot 10^1 + 7\cdot 10^0 & & \text{rounding} \\ \Rightarrow 1.0\cdot 10^1 + 0.7\cdot 10^1 & & \text{using guard digit} \\ \Rightarrow 1.7\cdot 10^1 & & \\ \Rightarrow 2\cdot 10^1 & & \text{rounding} \end{array} \tag{3.36}$$

另一方面，从右到左求值得出：

$$\begin{array}{lll} 4\cdot 10^0 + (6\cdot 10^0 + 7\cdot 10^0) \Rightarrow 4\cdot 10^0 + 13\cdot 10^0 & & \text{addition} \\ \Rightarrow 4\cdot 10^0 + 1\cdot 10^1 & & \text{rounding} \\ \Rightarrow 0.4\cdot 10^1 + 1.0\cdot 10^1 & & \text{using guard digit} \\ \Rightarrow 1.4\cdot 10^1 & & \\ \Rightarrow 1\cdot 10^1 & & \text{rounding} \end{array} \tag{3.37}$$

结论是：对中间结果应用舍入和截断的顺序会产生不同结果。

练习 3.20. 上述例子使用了四舍五入。你能在采用截断法的算术系统中提出一个类似的例子吗?

U通常，表达式的求值顺序由编程语言的定义决定，
或至少由编译器决定。在章节 3.6.5 中我们将看到，在并行计算中结合律并不总是唯一确定的。

3.6 舍入误差示例

通过以上内容，读者可能会认为舍入误差仅在特殊情况下才会导致严重问题。本节我们将讨论一些非常实际的例子，在这些例子中计算机算术的不精确性会在计算结果中显现。这些例子相对简单；更复杂的例子超出了本书范围，例如矩阵求逆的不稳定性。感兴趣的读者可参阅 [104, 192]。

3.6.1 相消误差：‘abc 公式’

作为一个实际例子，考虑二次方程 $ax^2+bx+c=0$，其解为 $x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$。假设 $b>0$ 且 $b^2\gg 4ac$，则 $\sqrt{b^2-4ac}\approx b$，此时 ‘+’ 解将不准确。这种情况下，更好的做法是计算 $x_-=\frac{-b-\sqrt{b^2-4ac}}{2a}$ 并使用 $x_+\cdot x_-=c/a$。

为说明问题，我们考察方程

$$f(x)=\epsilon x^2-(1+\epsilon^2)x+\epsilon,$$

当 ϵ 较小时，其根为

$$x_+\approx\epsilon^{-1},\qquad x_-\approx\epsilon.$$

这里存在两个问题。首先，当 ϵ 小于机器精度时，$b\approx 1$、$b^2-4ac=1$和 $x_-=0$ 会导致该解完全错误。即使判别式 $b^2-4ac>1$，仍会出现相消误差问题，函数值严重失准：教科书解法得到 $f(x_-)\approx\epsilon$，而无相消误差的解法则得到 $f(x_-)\approx\epsilon^3$。

	textbook		*accurate*	
ϵ	x_-	$f(x_-)$	x_-	$f(x_-)$
10^{-3}	$1.000\cdot 10^{-03}$	$-2.876\cdot 10^{-14}$	$1.000\cdot 10^{-03}$	$-2.168\cdot 10^{-19}$
10^{-4}	$1.000\cdot 10^{-04}$	$5.264\cdot 10^{-14}$	$1.000\cdot 10^{-04}$	0.000
10^{-5}	$1.000\cdot 10^{-05}$	$-8.274\cdot 10^{-13}$	$1.000\cdot 10^{-05}$	$-1.694\cdot 10^{-21}$
10^{-6}	$1.000\cdot 10^{-06}$	$-3.339\cdot 10^{-11}$	$1.000\cdot 10^{-06}$	$-2.118\cdot 10^{-22}$
10^{-7}	$9.992\cdot 10^{-08}$	$7.993\cdot 10^{-11}$	$1.000\cdot 10^{-07}$	$1.323\cdot 10^{-23}$
10^{-8}	$1.110\cdot 10^{-08}$	$-1.102\cdot 10^{-09}$	$1.000\cdot 10^{-08}$	0.000
10^{-9}	0.000	$1.000\cdot 10^{-09}$	$1.000\cdot 10^{-09}$	$-2.068\cdot 10^{-25}$
10^{-10}	0.000	$1.000\cdot 10^{-10}$	$1.000\cdot 10^{-10}$	0.000

(3.38)

Exercise3.21. 编写一个程序，分别用 ‘教科书’ 方法和上述方法计算二次方程的根。

- 设 $b=-1$ 和 $a=-c$，并通过逐步减小 a 和 c 的值来 $4ac\downarrow 0$。
- 输出计算得到的根、使用稳定计算法得到的根及其值 of $f(x)=ax^2+bx+c$ in the computed root.

现在假设你并不太关心根的精确值：你希望确保计算根时的残差 $f(x)$ 很小。设 x^* 为精确根，则

$$f(x^*+h)\approx f(x^*)+hf'(x^*)=hf'(x^*). \tag{3.39}$$

现在分别研究情况 $a\downarrow 0$、$c=-1$ 和 $a=-1$、$c\downarrow 0$。你能解释其中的差异吗?

练习 3.22. 考虑以下函数

$$\begin{cases} f(x) = \sqrt{x+1} - \sqrt{x} \\ g(x) = 1/\left(\sqrt{x+1} + \sqrt{x}\right) \end{cases} \tag{3.40}$$

- 证明它们在精确算术中是相同的；然而：
- 证明 f 会出现相消情况，而 g 则不存在此问题。
- 编写代码展示 f 与 g 之间的差异。可能需要为 x 使用较大数值。

- 从 x 和机器精度的角度分析相消现象。当 $\sqrt{x+1}$ 与 $\sqrt{x}$ 的差值小于 ϵ 时会发生什么？（更精确的分析：当它们相差 $\sqrt{\epsilon}$ 时，会如何表现？）
- $y = f(x)$ 的反函数是

$$x = (y^2 - 1)^2/(4y^2) \tag{3.41}$$

将此添加到你的代码中。这能否说明计算的准确性？

务必在单精度和双精度下测试你的代码。若使用 Python，可尝试 `bigfloat` 包。

3.6.2 级数求和

前例展示了如何防止单次运算中的大范围舍入误差。本例则说明即使舍入误差逐渐累积，也有多种处理方式。

考虑求和式 $\sum_{n=1}^{10000} \frac{1}{n^2} = 1.644834\ldots$，并假设我们使用单精度运算，在多数计算机上这意味着机器精度为 10^{-7}。此示例的问题在于两项之间的比率，以及项与部分和的比率，都在不断增大。在 3.3.6 节中我们观察到，过大的比率会导致加法运算中的一个操作数实际上被忽略。

若按原顺序对该级数求和，可以观察到首项为 1，因此所有部分和（$\sum_{n=1}^{N}$ 其中 $N < 10000$）至少为 1。这意味着当 $1/n^2 < 10^{-7}$ 时，任何项都会被忽略，因其小于机器精度。具体而言，最后 7000 项被忽略，计算所得和为 1.644725，前 4 位数字是正确的。

然而，若以相反顺序对该级数求和，则可以在单精度下获得精确结果。虽然我们仍将小数累加到大数上，但此时比率永远不会达到一比 ϵ 的程度，因此较小数不会被忽略。为了理解这一点，请考虑相邻两项的比率：

$$\frac{n^2}{(n-1)^2} = \frac{n^2}{n^2 - 2n + 1} = \frac{1}{1 - 2/n + 1/n^2} \approx 1 + \frac{2}{n} \tag{3.42}$$

由于我们仅求和 10_5 项且机器精度为 10^{-7}，在加法运算 $1/n^2 + 1/(n-1)^2$ 中，第二项不会像从大到小求和时那样被完全忽略。

练习 3.23. 我们的推理仍缺失一个步骤。我们已证明在相加两个连续项时，较小项不会被忽略。但在计算过程中，我们会将部分和与序列中的下一项相加。请证明这不会使情况恶化。

这里的经验是：单调（或接近单调）的级数应当从小到大求和，因为当相加的量级相近时误差最小。请注意这与减法情形相反 —— 涉及相似量级的运算会导致更大误差。这意味着如果应用需要交替加减数列，且我们事先知道各项的正负性，据此重新安排算法可能获得更好效果。

练习 3.24. 正弦函数的定义为

$$\begin{aligned} \sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \\ &= \sum_{i\geq 0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}. \end{aligned} \tag{3.43}$$

以下是计算该和的两个代码片段（假设已给定 `x` 和 `nterms` ）：

```
double term = x, sum = term;
for (int i=1; i<=nterms; i+=2) {
  term *=
    - x*x / (double)((i+1)*(i+2));
  sum += term;
}
printf("Sum: %e\n\n",sum);
```

```
double term = x, sum = term;
double power = x, factorial = 1.,
    factor = 1.;
for (int i=1; i<=nterms; i+=2) {
  power *= -x*x;
  factorial *= (factor+1)*(factor+2);
  term = power / factorial;
  sum += term; factor += 2;
}
printf("Sum: %e\n\n",sum);
```

- 解释如果计算 $x > 1$ 的大量项会发生什么。
- 对于大量项的计算，其中任一代码是否有意义？
- 是否可以从最小的项开始求和？这会是个好主意吗？
- 你能提出其他方案来改进 $\sin(x)$ 的计算吗？

3.6.3 不稳定算法

W我们将通过一个例子直接论证该算法无法胜任的情况
w由于实数表示不精确导致的第 i 个问题。
考虑单调递减的递推关系 $y_n = \int_0^1 \frac{x^n}{x-5} dx = \frac{1}{n} - 5y_{n-1}$，其首项可计算为 $y_0 = \ln 6 - \ln 5$。

以 3 位小数进行运算，我们得到：

计算 正确结果 $y_0 = \quad 6 - \quad 5 = .182|322 \times 10^1 \ldots$ 1 82 ln ln . .
$y_1 = .900 \times 10^{-1}$
884 .
$y_2 = .500 \times 10^{-1}$
0580 上升? $.y_4 = -.165$ 负数?
$y_3 = .830 \times 10^{-1}$ 0431
.0343

可见计算结果不仅迅速变得不准确，实际上已毫无意义。我们可以分析其原因。

若将第 n 步的误差 ϵ_n 定义为

$$\tilde{y}_n - y_n = \epsilon_n, \tag{3.44}$$

then

$$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5n_{n-1} + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1} \tag{3.45}$$

因此 $\epsilon_n \geq 5\epsilon_{n-1}$。该计算产生的误差呈现指数级增长。

3.6.4 线性方程组求解

有时我们无需指定具体算法，也能对问题的数值精度作出论断。假设我们需要求解一个线性系统，即给定一个 $n \times n$ 矩阵 A 和一个大小为 n 的向量 b，要求计算向量 x 使得 $Ax = b$ 成立。（我们将在第 5 章具体讨论该问题的求解算法。）由于向量 b 是某种计算或测量的结果，实际上我们处理的是向量 b，即理想解 b 的扰动版本：

$$\tilde{b} = b + \Delta b. \tag{3.46}$$

扰动向量 Δb 的量级可能达到机器精度（若仅由表示误差引起），也可能更大 —— 这取决于生成 b 的具体计算过程。

现在我们要探讨：在理想情况下（即用精确的 A 和 b 进行不可能实现的精确计算）得到的精确解 x，与通过 A 和 b 计算得到的数值解 x 之间存在何种关系。（为简化讨论，此处假设 A 本身是精确值。）

代入 $x \doteq x + \Delta x$，我们的计算结果现为

$$A\tilde{x} = \tilde{b} \tag{3.47}$$

or

$$A(x + \Delta x) = b + \Delta b. \tag{3.48}$$

由于 $Ax = b$，可得 $A\Delta x = \Delta b$。由此得出（详见附录 14 ）

$$\left\{ \begin{array}{ll} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{ll} \|A\|\|x\| & \geq \|b\| \\ \|\Delta x\| & \leq \|A^{-1}\|\|\Delta b\| \end{array} \right. \Rightarrow \frac{\|\Delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\| \frac{\|\Delta b\|}{\|b\|} \tag{3.49}$$

量 $\|A\|\|A^{-1}\|$ 被称为矩阵的条件数。界 (3.49) 表明，右端项的任何扰动可能导致解的扰动最多放大矩阵条件数 A 倍。需注意，这并不意味着 x 的扰动必然达到该量级，但我们不能排除这种可能性，某些情况下该上界确实会被达到。

假设 b 精确到机器精度，且 A 的条件数为 10_4。界 (3.49) 常被解读为： x 的最后 4 位数字不可靠，或计算 ' 损失了 4 位精度 '。

方程 (3.49) 还可解释为：求解线性系统 $Ax = b$ 时，得到的近似解 $x + \Delta x$ 实际上是扰动系统的精确解 $A(x + \Delta x) = b + \Delta b$。解的扰动与系统扰动相关这一特性，表明该算法具有后向稳定性。

线性代数算法精度的分析本身就是一个研究领域；例如可参考 Higham 的著作 [104]。

3.6.5 并行计算中的舍入误差

正如我们在 3.5.5 节讨论的，以及你在上述级数求和示例中所见，计算机算术中的加法并不具有结合律。类似的性质也适用于乘法。这对并行计算产生了一个有趣的后果：计算任务在并行处理器间的分配方式会影响最终结果。

以一个非常简单的例子为例，考虑计算四个数 $a+b+c+d$ 的和。在单处理器上，常规执行对应于以下结合方式：

$$((a + b) + c) + d. \tag{3.50}$$

另一方面，若将此计算分配到两个处理器上，其中处理器 0 处理 a, b ，处理器 1 处理 c, d，则对应于

$$((a + b) + (c + d)). \tag{3.51}$$

推广来看，我们会发现归约操作很可能在不同数量的处理器上给出不同结果。（ MPI 标准规定，同一组处理器上的两次程序运行应给出相同结果。）可以通过用聚集操作替代归约操作来规避此问题，即所有处理器先执行聚集操作，随后进行本地归约。然而，这会增加处理器的内存需求。

并行求和问题存在另一种引人入胜的解决方案。若我们使用 4000 位尾数存储浮点数，则无需指数位，所有以此形式存储的数字计算都是精确的，因为它们本质上是定点运算的一种形式 [122, 123]。虽然整个应用都采用此类数字会极其浪费资源，但仅将其保留用于偶尔的内积计算，或许是解决可重现性问题的良方。

3.7 编程语言中的计算机算术

不同语言对整数和浮点数的声明方式各有不同。此处我们将探讨其中的一些问题。

3.7.1 C/C++ 数据模型

C/C++ 语言具有 `short`、 `int`、 `float` 类型（复数类型参见 3.8.6 节），以及针对这些类型的 `long` 和 `unsigned` 修饰符。（单独使用 `long` 等同于 `long int`。）语言标准未对这些类型给出明确定义，仅设定了以下限制：

- short int 类型至少为 16 位；
- 整数类型至少为 16 位，这在过去 *DEC PDP-11* 时代确实如此，但如今通常为 32 位；
- 长整型至少为 32 位，但通常为 64 位；
- `long long` 整数类型至少为 64 位。
- 若仅需一个字节存储整数，可使用 `char` 类型。

此外，指针与无符号整数相关；详见章节 3.7.2.3 .

针对不同整数类型的实际大小，存在多种称为数据模型的约定规范。

- *LP32* (或称 ‘2/4/4’) 采用 16 位 `int`，以及 32 位 `long int` 和指针。该模型用于 *Win16*API。
- *ILP32* (或称 ‘4/4/4’) 采用 32 位 `int`, `long`和指针。该模型用于 *Win32* API 及大多数 32 位 Unix 或类 Unix 系统。
- *LLP64* (或称 ‘4/4/8’) 采用 32 位 `int` 和 `long` (!!!)，指针为 64 位。该模型用于 *Win64* API。
- *LP64* (或称 ‘4/8/8’) 采用 32 位 `int`， `long` 和指针为 64 位。
- *ILP64* （或称 ‘8/8/8’ 模式）对 `int`、 `long` 和指针均采用 64 位表示。该模式仅存在于早期 Unix 系统如 *CrayUNICOS* 上。

3.7.2 C/C++

C 与 C++ 语言共享某些类型处理机制；关于 C++ 特有的机制请参阅下文说明。

3.7.2.1 位操作

C 语言的逻辑运算符及其位操作变体如下：

	boolean	位运算
and	`&&`	`&`
or	`||`	`|`
not	`!`	
xor		`^`

The following *bit shift* operations are available:

left shift	<<
right shift	>>

您可以通过位运算进行算术操作：

- 左移即乘以 2：

```
i_times_2 = i<<1;
```

- 提取位：`i_mod_8 = i & 7`
（最后一个是如何工作的？）

练习 3.25. 位移操作通常应用于无符号数。当您在 2 的补码中使用位移来乘以或除以 2 时，是否存在额外的复杂性？

以下代码片段对于打印数字的位模式非常有用：

```
// printbits.c
void printBits(size_t const size, void const * const ptr)
{unsigned char *b = (unsigned char*) ptr;
unsigned char byte;int i,  j;for (i=size-1;i>=0;i--)
for (j=7;j>=0;j--) {byte = (b[i] >> j) & 1;
printf("%u", byte);}}
```

示例用法：`// bits.cint five = 5;`

```
printf("Five=%d, in bits: ",five);
printBits(sizeof(five),&five);
printf("\n");
```

3.7.2.2 打印位模式

虽然 C++ 提供了 `hexfloat` 格式，但这并非直观展示二进制数位模式的方式。这里有一个便捷的例程用于显示实际的位模式。

Code:

```
    void format(const std::string  &s)
    {
      // sign bit
      std::cout << s.substr(0,1)  << ' ';
      // exponent
      std::cout << s.substr(1,8);
      // mantissa in groups of 4
      for(int walk=9;walk<32;walk+=4)
        std::cout << ' ' << s.substr(walk,4);
      // newline
      std::cout << "\n";
    }
      uint32_t    u;
      std::memcpy(&u,&d,sizeof(u));
      std::bitset<32> b{u};
      std::stringstream s;
      s << std::hexfloat << b << '\n';
      format(s.str());
```

Output

```
[code/754] bitprint:

missing snippet
code/754/bitprint.runout : looking
in codedir=code missing snippet
code/754/bitprint.runout : looking
in codedir=code
```

3.7.2.3 整数与浮点数

`sizeof()` 运算符给出用于存储数据类型的字节数 pe.

浮点类型在 `float.h` 中指定。

存在具有指定存储的 C 整数类型：诸如 `int64_t` 的常量由 `typedef` 在 *stdint.h* 中定义。

常量 `NAN` 在 `math.h` 中声明。要检查某个值是否为 `NaN`，请使用 `isnan()`。

3.7.2.4 改变舍入行为

根据 754 标准的规定，舍入行为应当是可控制的。在 C99 中，相关 API 包含于 *fenv.h* (或对于 C++ *cfenv*)：

```
#include <fenv.h>

int roundings[] =
  {FE_TONEAREST, FE_UPWARD, FE_DOWNWARD, FE_TOWARDZERO};
rchoice = ....
int status = fesetround(roundings[rchoice]);
```

设置舍入行为可作为算法稳定性的快速测试：若两种不同舍入策略下的结果差异显著，则该算法很可能不稳定。

3.7.3 限制

对于 C 语言，数值范围 的 C 整数在 *limits.h* 中定义，通常给出上限或下限。例如， `INT_MAX` 被定义为 32767 或更大。

在 C 语言中 ++ 仍可使用 C 头文件 `limits.h` 或 *climits*，但更推荐使用 *std::numeric_limits*，其类型已模板化。（详见《科学编程入门》一书第 *24.2* 节。）

例如

```
std::numerical_limits<int>.max();
```

3.7.4 异常

IEEE 754 标准与 C++ 语言均定义了异常概念，但二者存在差异。

- 浮点异常是指出现 ‘无效数字’ 的情况，例如溢出或除以零（参见章节 3.4.2.1 ）。从技术上讲，它们表示发生了 ‘没有适合所有合理应用场景结果’ 的操作。
- 编程语言可以 ‘抛出异常’，即在发生任何类型的意外事件时中断常规程序控制流。

3.7.4.1 启用

有时可以在浮点异常时生成语言异常。

在溢出时的行为可设置为生成一个异常。在 C 语言中，您可以通过库调用来指定：

```
#include <fenv.h>int main() {...
feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW);
```

3.7.4.2 定义的异常

定义的异常：

- *FE_DIVBYZERO* 极错误发生在先前的浮点运算中。
- *FE_INEXACT* 不精确结果：需要舍入才能存储先前浮点运算的结果。
- *FE_INVALID* 定义域错误发生在先前的浮点运算中。
- *FE_OVERFLOW* 先前浮点运算的结果过大，无法表示。
- *FE_UNDERFLOW* 之前的浮点运算结果是一个次正规数且存在精度损失。
- *FE_ALL_EXCEPT* 所有支持的浮点异常的按位或。

用法：

```
std::feclearexcept(FE_ALL_EXCEPT);
if(std::fetestexcept(FE_UNDERFLOW)) { /* ... */ }
```

在 C 语言中 ++, `std::numeric_limits<double>::quiet_NaN()` 声明于 `limits`, 其意义在于当 `std::numeric_limits::has_quiet_NaN` 为真时成立，这种情况发生在 `std::numeric_limits::is_iec559` 为真时。（ ICE 559 本质上等同于 IEEE 754 ； 参见第 3.4 节。）

同一模块还包含 `infinity()` 和 `signaling_NaN()`。

要检查某个值是否为 NaN，在 C++ 中使用 cmath 的 std::isnan()。

参见 http://en.cppreference.com/w/cpp/numeric/math/nan。

3.7.4.3 编译器特定行为

捕获异常有时可由编译器指定。例如，*gcc* 编译器可通过标志 -ffpe-trap=list 捕获异常；参见 https://gcc.gnu.org/onlinedocs/gfortran/Debugging-Options.html。

3.7.5 快速数学的编译器标志

多种编译器提供了快速数学优化的选项。

- GCC 和 Clang：-ffast-math
- Intel：-fp-model=fast （默认）
- MSVC：/fp:fast

这通常涵盖以下情况：

- 有限数学：假设 Inf 和 Nan 不会发生。这意味着测试 x==x 始终为真。
- 结合律数学：这允许重新排列算术表达式中的运算顺序。这被称为重结合，例如对向量化有益。然而，正如你在 3.5.5 节中所见，这会改变计算结果。此外，它使得补偿求和无法实现；参见 3.8.1 节。
- 将次正规数刷新为零。

详细讨论：https://simonbyrne.github.io/notes/fastmath/

3.7.6 Fortran

3.7.6.1 变量 *'kind'* 类型

Fortran 有几种机制用于指示数值类型的精度。

```
integer(2) :: i2
integer(4) :: i4
integer(8) :: i8

real(4) :: r4
real(8) :: r8
real(16) :: r16

complex(8) :: c8
complex(16) :: c16
complex*32 :: c32
```

这通常对应于使用的字节数，**但并非总是如此**。从技术上讲，它是一个数值型的种类选择器，仅作为特定类型的标识符存在。

```
integer, parameter :: k9 = selected_real_kind(9)
real(kind=k9) :: r
r = 2._k9; print *, sqrt(r) ! prints 1.4142135623730
```

'kind' 的取值通常为 4 、 8 、 16，但这取决于具体的编译器。

3.7.6.2 舍入行为

在 Fortran2003 中，函数 `IEEE_SET_ROUNDING_MODE` 可通过 `IEEE_ARITHMETIC` 模块调用。

3.7.6.3 C99 与 Fortran2003 的互操作性

C 和 Fortran 语言的最新标准引入了 C/Fortran 互操作性标准，可用于在一种语言中声明某种类型，使其与另一种语言中的特定类型兼容。

3.7.7 编程中的舍入行为

从上述讨论中可以清楚地看出，一些对数学实数成立的简单陈述在浮点数中并不成立。例如，在浮点算术中

$$(a+b)+c \neq a+(b+c). \tag{3.52}$$

这意味着编译器无法在不影响舍入行为的情况下执行某些优化 [4]。在某些代码中，这种微小差异是可以容忍的，例如因为该方法内置了安全措施。例如，第 5.5 节的静态迭代方法会抑制引入的任何误差。

另一方面，如果程序员编写的代码考虑了舍入行为，编译器就没有这样的自由。这一点在之前的练习 3.11 中有所暗示。我们使用*值安全性* 的概念来描述编译器被允许如何改变计算的解释。在最严格的情况下，编译器不允许做出任何影响计算结果的更改。

编译器通常有一个选项控制是否允许可能改变数值行为的优化。对于 Intel 编译器，该选项是 `-fp-model=...`。另一方面，诸如 `-Ofast` 等选项仅针对性能提升，可能会严重影响数值行为。对于 Gnu 编译器，完全符合 754 标准的选项是 `-frounding-math`，而 `-ffast-math` 则允许违反 754 和 / 或语言标准的以性能为导向的编译器转换。

这些问题同样重要，如果你关心结果的*可复现性*。如果一段代码用两种不同的编译器编译，相同的输入是否应产生相同的输出？如果代码在两个不同的处理器配置上并行运行呢？这些问题非常微妙。在第一种情况下，人们有时坚持比特级可复现性，而在第二种情况下，只要结果保持 ‘科学上’ 等效，允许存在一些差异。当然，这一概念很难严格界定。

以下是考虑编译器对代码行为及可复现性影响时相关的一些问题。

4. 本节内容参考了微软 http://msdn.microsoft.com/en-us/library/aa289157(vs.71).aspx 和英特尔 http://software.intel.com/sites/default/files/article/164389/fp-consistency-122712_1.pdf 的文档；详细讨论请参阅这些资料。

重关联 编译器对计算所能做的最重要的改变之一是重关联，这是将 $a+b+c$ 分组为 $a+(b+c)$ 的技术术语。*C 语言标准* 和 *C++ 语言标准* 规定了严格的从左到右的无括号表达式求值顺序，因此重关联实际上是不被标准允许的。*Fortran 语言标准* 没有这样的规定，但在那里编译器必须尊重括号所隐含的求值顺序。

重关联的一个常见来源是 **循环展开**；参见章节 6.3.2。在严格的值安全性下，编译器在如何展开循环方面受到限制，这对性能有影响。循环展开的数量，以及是否执行展开，取决于编译器的优化级别、编译器的选择以及目标平台。

一个更微妙的重新关联来源是并行执行；参见章节 3.6.5。这意味着代码的输出在不同的并行配置之间运行两次时，不必严格可重现。

常量表达式 在编译时计算常量表达式是一种常见的编译器优化。例如，在

```
float one = 1.;...
x = 2. + y + one;
```

编译器将赋值改为 `x = y+3.`。然而，这违反了上述重关联规则，并且忽略了任何动态设置的舍入行为。

表达式求值 在计算表达式 $a+(b+c)$ 时，处理器会为 $b+c$ 生成一个未被赋值给任何变量的中间结果。许多处理器能够为中间结果分配更高精度。编译器可以通过标志位决定是否使用此功能。

浮点单元行为 舍入行为（截断与就近舍入）及渐进下溢处理可通过库函数或编译器选项控制。

库函数 IEEE 754 标准仅规定了基本运算；目前尚无针对正弦或对数函数的标准。因此，它们的实现可能是差异性的来源。

更多讨论，请参阅 [137]。

3.8 关于浮点运算的更多内容

3.8.1 Kahan 求和法

第 3.5.5 节的示例揭示了计算机算术运算中的部分问题：舍入可能导致结果严重错误，且高度依赖于计算顺序。存在若干算法试图弥补这些问题，特别是在加法运算中。我们将简要讨论

Kahan summation[114]，以 *WilliamKahan* 命名，这是补偿求和的一个例子
算法。

```
sum ← 0correction ← 0 当 存在另一个
 input 时执行 oldsum ← sum
input ← input − correction
sum ← oldsum + input
correction ← (sum − oldsum) − input
```

练习 3.26. 回顾第 3.5.5 节的示例，添加最后一项 3；即在原示例条件下计算 $4 + 6 + 7 + 3$ 和 $6 + 7 + 4 + 3$。证明修正值恰好是当 17 舍入为 20 时的 3 不足误差，或当 14 舍入为 10 时的 4 过剩误差；两种情况下均计算出正确结果 20。

3.8.2 其他计算机算术系统

已有其他系统被提出用于解决计算机上不精确算术运算的问题。一种解决方案是扩展精度算术，其中数字以比通常更多的位数存储。其常见用途是在向量内积计算中：累加过程在内部以扩展精度执行，但结果以常规浮点数返回。此外，还有如 GMP 库 [76] 这样的库，允许任何计算以更高精度执行。

应对计算机算术不精确性的另一种方案是 ‘区间算术’ [111],，其中每个计算都维护区间边界。尽管该领域已研究相当长时间，但除通过专门库 [21] 外，实际应用仍较少。

曾进行过一些关于三值算术的实验（参见 http://en.wikipedia.org/wiki/Ternary_computer 及 http://www.computer-museum.ru/english/setun.htm），但目前尚无实用硬件实现。

3.8.3 扩展精度

IEEE 754 标准制定时，曾设想处理器可支持多种精度级别。实践中仅单精度和双精度被广泛采用。但扩展精度的一个实例仍存在：Intel 处理器使用 80 位寄存器存储中间结果（这可追溯至 *Intel 8087* 协处理器）。该策略在 *FMA* 指令及内积累加运算中具有实际意义。

这些 80 位寄存器具有一种奇特的结构，包含一个有效位整数位，可能产生不属于任何已定义数字有效表示的位模式 [162]。

3.8.4 降低精度

你可能会问 ‘双精度是否总是优于单精度’，但答案并非总是 ‘是的’，而是：‘视情况而定’。

3.8.4.1 迭代求精中的低精度运算

在迭代线性系统求解（章节 5.5 中，精度由残差计算的精确度决定，而非求解步骤的精度。因此，可以采用诸如在降低精度下应用预条件子（章节 5.5.6 ）等操作 [26]。这是迭代求精的一种形式；参见章节 5.5.6。

3.8.4.2 深度学习中的低精度运算

IEEE 754-2008 标准定义了 *binary16* 半精度格式，该格式采用 5 位指数和 11 位尾数。

在深度学习（ *DL* ）中，表达数值范围比精确表示具体数值更为重要。（这与传统科学应用相反，后者需要精确区分相近数值。）这促使了 *bfloat16* “ 脑浮点 ” 格式 `https://en.wikipedia.org/wiki/Bfloat16_floating-point_format` 的定义，这是一种 16 位浮点格式。它使用 8 位指数和 7 位尾数，这意味着其指数范围与 IEEE 单精度格式相同；参见图 3.4。

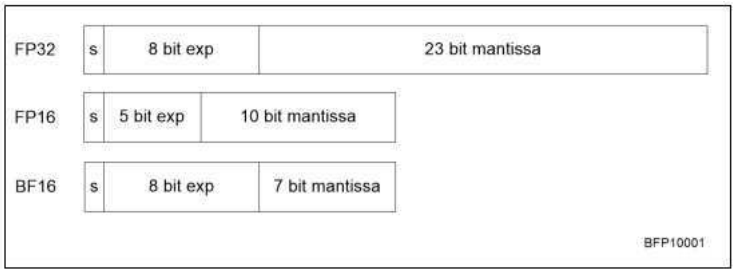


图 3.4：fp32 、 fp16 与 bfloat16 格式对比。（插图源自 [36]。）

- 由于 bfloat16 与 fp32 在前两个字节结构相同，可通过截断 fp32 数值的第三、第四字节得到 bfloat16 数值。但实践中采用舍入法可能获得更优结果。
- 反之，将 bloat16 转换为 fp32 仅需在末两字节补零即可。

bfloat16 的有限精度可能足以表征深度学习应用中的量值，但为避免进一步损失精度，设想 FMA 硬件在内部使用 32 位数：两个 bfloat16 数值的乘积是常规 32 位数。为计算内积（这在深度学习的矩阵乘法过程中发生），我们需要如图 3.5 所示的 FMA 单元。

- *IntelKnights Mill* 处理器基于 *Intel Knights Landing* 架构，支持降低精度运算。
- 英特尔 *Cooper Lake* 架构实现了 *bfloat16* 浮点格式 [36]。

Even further reduction to 8-bit was discussed in [47].

3.8.5 定点运算

定点数（更深入的讨论可参考 [196] ）可表示为 $\langle N, F\rangle$ 其中 $N \geq \beta^0$ 为整数部分， $F < 1$ 为小数部分。另一种理解方式是，

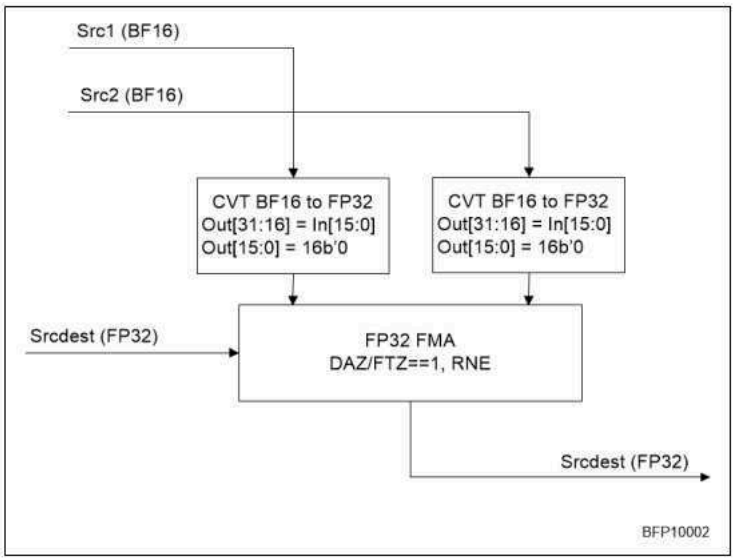


图 3.5: 一个接收两个 bloat16 和一个 fp32 数值的 FMA 单元。（插图来自 [36]。）

定点数是以 $N+F$ 位数字存储的整数，隐含的小数点位于前 N 位之后。

定点计算可能发生溢出，且无法通过调整指数来避免。以乘法 $\langle N_1, F_1\rangle \times \langle N_2, F_2\rangle$ 为例，其中 $N_1 \geq \beta^{n_1}$ 且 $N_2 \geq \beta^{n_2}$。若 n_1+n_2 超过整数部分的可用位数就会溢出。（通俗地说，乘积的位数等于操作数位数之和。）这意味着在使用定点数的程序中，若需进行乘法运算，数字必须预留前导零位，这会降低数值精度。同时也要求程序员更谨慎地设计计算流程，通过合理安排防止溢出，并尽可能保持合理的数值精度。

那么为何人们会使用定点数？一个重要应用场景是低功耗嵌入式设备，例如电池供电的数字温度计。由于定点运算本质上与整数运算相同，它们无需浮点运算单元，从而缩小芯片尺寸并降低功耗需求。此外，许多早期游戏主机的处理器要么没有浮点单元，要么其整数单元速度远快于浮点单元。在这两种情况下，通过整数单元实现非整数计算的定点化处理，是实现高吞吐量的关键。

定点运算至今仍在使用的另一个领域是信号处理。现代 CPU 中，整数与浮点运算速度已基本相当，但二者间的转换却相对耗时。若正弦函数通过查表法实现，这意味着在 $\sin(\sin x)$ 中，函数的输出被用作下一次函数调用的索引。显然，以定点数形式输出正弦函数可避免实数与整数间的转换需求，这不仅简化了所需芯片逻辑，还加速了计算过程。

3.8.6 复数

某些编程语言将复数作为内置数据类型，而其他语言则不支持，还有些语言介于两者之间。例如，在 Fortran 中你可以声明

```
COMPLEX z1,z2, z(32)
COMPLEX*16 zz1, zz2, zz(36)
```

复数由一对实数（实部与虚部）组成，在内存中相邻存储。第一种声明使用 8 字节存储 `REAL*4` 个数字，第二种声明中实部和虚部各占 `REAL*8` 字节。（也可用 `DOUBLE COMPLEX` 或在 Fortran 中使用 90 `COMPLEX(KIND=2)` 替代第二行声明。）

相比之下， `C` 语言本身并不直接支持复数，但 `C99` 和 `C++` 都提供了 `complex.h` 头文件[5]，该文件采用与 Fortran 相同的复数定义方式 —— 即用两个实数表示复数。

以这种方式存储复数虽然简单，但在计算上并非最优解。当我们处理复数数组时，这一点尤为明显。若计算过程频繁需要单独访问复数的实部（或虚部），遍历复数数组的步长会变为两倍，这显然不利于性能（参见章节 1.3.5.7 ）。此时，更优的做法是为实部和虚部分别分配独立的数组。

习题 3.27. 假设复数数组以 Fortran 方式存储。分析数组两两相乘时的内存访问模式，即 $\forall_i : c_i \leftarrow a_i \cdot b_i$，其中 `a()` 、 `b()` 、 `c()` 均为复数数组。

习题 3.28. 证明复数域上的 $n \times n$ 线性方程组 $Ax = b$ 可转化为实数域上的 $2n \times 2n$ 方程组。提示：将矩阵和向量拆分为实部与虚部。论述将复数数组存储为实部和虚部分离的独立数组在效率上的优势。

3.9 结论

计算机上进行的计算总是伴随着数值误差。从某种意义上说，误差的根源在于计算机算术的不完美性：如果我们能用实际的实数进行计算，就不会有问题。（即便如此，数据中的测量误差和数值方法中的近似处理仍会存在；详见下一章。）然而，若我们将舍入误差视为不可避免的事实，则可得出以下观察结论：

- 数学上等价的操作在稳定性方面可能表现不同；参见 ‘abc 公式’ 示例。
- 即便是相同计算的不同重组方式也会表现各异；参见求和示例。

因此，分析计算机算法关于其舍入误差行为变得至关重要：舍入误差是否会随着问题参数（如计算的项数）缓慢增长而增加，还是可能出现更糟糕的行为？本书将不再深入探讨此类问题。

5. These two header files are not identical, and in fact not compatible. Beware, if you compile C code with a C++ compiler [52].

3.10 复习题

Exercise 3.29. 判断正误?

- 对于整数类型，‘最负’ 的整数是 ‘最正’ 整数的相反数。
- 对于浮点类型，‘最负’ 的数是 ‘最正’ 数的相反数。
  正’ 的那个。
- 对于浮点类型而言，最小的正数是最大正数的倒数。

第 4 章

微分方程的数值处理

本章将探讨常微分方程（ ODEs ）和偏微分方程（ PDEs ）的数值解法。这些方程在物理学中常被用于描述现象，如飞机周围的气流，或桥梁在各种应力下的弯曲。虽然这些方程通常相当简单，但从中获取具体数值（ ' 如果桥上有百辆汽车，桥会下垂多少 ' ）则更为复杂，往往需要大型计算机才能得出所需结果。这里我们将描述将 ODEs 和 PDEs 转化为可计算问题（即线性代数）的技术。第 5 章将探讨这些线性代数问题的计算层面。

首先，在 4.1 节中，我们将研究初值问题（ IVPs ），它描述了随时间发展的过程。这里我们考虑 ODEs：仅依赖于时间的标量函数。其名称源于通常会在初始时间点指定函数值这一事实。

接下来，在章节 4.2 中，我们将探讨边界值问题（ BVPs ），它描述了空间中的过程。在实际情况下，这会涉及多个空间变量，因此我们面对的是偏微分方程（ PDE ）。BVP 这一名称的由来是因为解是在定义域的边界上指定的。

最后，在章节 4.3 中，我们将讨论 ‘ 热方程 ’ ，这是一个初始边界值问题（ IBVP ），它结合了初值问题（ IVPs ）和边界值问题（ BVPs ）的特点：它描述了热量通过物理物体（如一根杆）的扩散过程。初始值描述了初始温度，而边界值给出了杆两端的设定温度。

本章的目标是展示一类重要计算问题的起源。因此，我们不会深入探讨解的存在性、唯一性或条件性等理论问题。关于这些内容，请参阅 [97] 或任何专门讨论常微分方程（ ODEs ）或偏微分方程（ PDEs ）的书籍。为了便于分析，我们还将假设所有涉及的函数都具有足够多的高阶导数，且每个导数都足够平滑。

4.1 初值问题

许多物理现象会随时间变化，而物理定律通常描述的是这种变化本身，而非我们关注的量。例如，牛顿第二定律

$$F = ma \tag{4.1}$$

是关于质点位置变化的陈述：表述为

$$a(t) = \frac{d^2}{dt^2} x(t) = F/m \tag{4.2}$$

它指出加速度线性依赖于施加在质点上的力。质点的位置 $x(t) = \ldots$ 有时可以通过解析方法推导出闭式描述，但在许多情况下需要某种近似，通常通过数值计算来实现。这也被称为 ‘数值积分’。

牛顿第二定律（4.1）是一个常微分方程（ODE），因为它描述了一个变量（时间）的函数。它是一个初值问题（IVP），因为它描述了从某些初始条件开始的时间演化过程。作为一个 ODE，它是 ‘二阶’ 的，因为它涉及二阶导数。如果我们允许向量量，可以将其降为一阶，仅涉及一阶导数。我们引入一个二组件向量 u，它结合了位置 x 和速度 x'，其中我们使用撇号表示单变量函数的微分：

$$u(t) = (x(t), x'(t))^t. \tag{4.3}$$

牛顿方程用 u 表示后变为：

$$u' = Au + B, \qquad A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ F/a \end{pmatrix}. \tag{4.4}$$

为简化起见，本课程中我们仅考虑一阶标量方程；此时我们的参考方程为

$$u'(t) = f(t, u(t)), \qquad u(0) = u_0, \qquad t > 0. \tag{4.5}$$

方程 (4.5) 允许过程存在显式时间依赖性，但通常我们只考虑不含这种显式依赖的方程，即所谓 ' 自治 ' ODE，其形式为

$$u'(t) = f(u(t)) \tag{4.6}$$

其中右侧不显式依赖于 t。

注 12 非自治 *ODE* 可转化为自治 *ODE*，故该限制不影响一般性。若 $u = u(t)$ 为标量函数且 $f = f(t, u)$，我们定义 $u_2(t) = t$ 并考虑等效自治系统 $\begin{pmatrix} u' \\ u_2' \end{pmatrix} = \begin{pmatrix} f(u_2, u) \\ 1 \end{pmatrix}$。

通常给定某起始点（常取 $t = 0$）的初值：$u(0) = u_0$ 对应某值 u_0，我们关注 u 随 $t \to \infty$ 的变化行为。例如 $f(x) = x$ 将方程 (4.5) 简化为 $u'(t) = u(t)$。这是种群增长的简单模型：该方程表明增长率等于种群规模。现我们讨论该过程的数值解法及其精度。

在数值方法中，我们选择离散的时间步长来近似连续时间依赖过程的解。由于这会引入一定量的误差，我们将分析每个时间步中产生的误差，以及这些误差如何累积成全局误差。在某些情况下，限制全局误差的需求会对数值方案施加约束。

4.1.1 误差与稳定性

由于数值计算总会涉及机器算术带来的不精确性，我们需要避免初始值的微小扰动导致解的剧烈变化。因此，当不同（但足够接近）初始值 u_0 对应的解随着 $t \to \infty$ 收敛于同一解时，我们称该微分方程是 ‘稳定’ 的。

定理 2 稳定性的一个充分判据为：

$$\frac{\partial}{\partial u} f(u) = \begin{cases} > 0 & unstable \\ = 0 & neutrally\ stable \\ < 0 & stable \end{cases} \tag{4.7}$$

证明。若 u^* 是 f 的零点，即 $f(u^*) = 0$，则常函数 $u(t) \equiv u^*$ 是 $u' = f(u)$的解，即所谓的 ‘平衡’ 解。现考察微小扰动偏离平衡态的行为。设 u 为偏微分方程的解，记 $u(t) = u^* + \eta(t)$，则有

$$\begin{aligned} \eta' = u' = f(u) = f(u^* + \eta) = f(u^*) + \eta f'(u^*) + O(\eta^2) \\ = \eta f'(u^*) + O(\eta^2) \end{aligned} \tag{4.8}$$

忽略二阶项后，其解为

$$\eta(t) = e^{f'(x^*)t} \tag{4.9}$$

这意味着若 $f'(x^*) < 0$，扰动将衰减；若 $f'(x^*) > 0$，扰动将放大。 .

我们将频繁引用简单示例 $f(u) = -\lambda u$ 及其解 $u(t) = u_0 e^{-\lambda t}$。当 $\lambda > 0$ 时，该问题是稳定的。

4.1.2 有限差分近似：显式与隐式欧拉方法

为了数值求解常微分方程，我们通过有限时间 / 空间步长将连续问题转化为离散问题。假设所有函数均充分光滑，直接对泰勒级数展开 u 在 t 处（参见附录 17 ）可得：

$$u(t + \Delta t) = u(t) + u'(t)\Delta t + u''(t)\frac{\Delta t^2}{2!} + u'''(t)\frac{\Delta t^3}{3!} + \cdots \tag{4.10}$$

由此得到 u'的表达式：

$$\begin{aligned} u'(t) &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + \frac{1}{\Delta t}\left(u''(t)\frac{\Delta t^2}{2!} + u'''(t)\frac{\Delta t^3}{3!} + \cdots\right) \\ &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + \frac{1}{\Delta t} O(\Delta t^2) \\ &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + O(\Delta t) \end{aligned} \tag{4.11}$$

若所有导数有界，我们可用单个 $O(\Delta t^2)$ 近似无穷高阶导数之和；或可证明该和等于 $\Delta t^2 u''(t + \alpha\Delta t)$，其中 $0 < \alpha < 1$。

可以看出，我们可以用有限差分来近似微分算子，其误差作为时间步长的函数在数量级上是已知的。

将此代入 $u' = f(t,u)$ 将方程 4.6 转化为

$$\frac{u(t+\Delta t)-u(t)}{\Delta t} = f(t,u(t)) + O(\Delta t) \tag{4.12}$$

or

$$u(t+\Delta t) = u(t) + \Delta t\, f(t,u(t)) + O(\Delta t^2). \tag{4.13}$$

备注 13 前两个等式是数学上的等式，不应将其解释为计算给定函数 u 的 u' 的方法。回顾第 *3.5.4* 节的讨论，你会发现这样的公式会很快导致小的 Δt 时的抵消。关于数值微分的进一步讨论超出了本书的范围；请参阅任何标准的数值分析教材。

我们现在用方程 (4.13) 推导一个数值方案：对于 $t_0 = 0, t_{k+1} = t_k + \Delta t = \cdots = (k+1)\Delta t$，我们得到一个差分方程

$$u_{k+1} = u_k + \Delta t\, f(t_k, u_k) \tag{4.14}$$

对于 u_k 量，我们希望 u_k 能很好地近似 $u(t_k)$。这被称为显式欧拉方法，或欧拉前向方法。

从微分方程转化为差分方程的过程通常被称为离散化，因为我们仅在离散的点集上计算函数值。计算得到的值本身仍是实数。另一种表述方式是：若我们计算 k 个时间步长，数值解存在于有限维空间 $\mathbb{R}^k$ 中。而原问题的解则位于无限维的函数空间 $\mathbb{R} \to \mathbb{R}$ 内。

在方程 (4.11) 中，我们用一个算子近似另一个算子，在此过程中产生了截断误差，其阶数为 $O(\Delta t)$ 当 $\Delta t \downarrow 0$ 时（关于该阶数符号更正式的介绍参见附录 15 ）。这并不直接意味着差分方程计算的解接近真实解。为此需要进一步分析。

W我们首先分析 ‘局部误差’：假设在步骤 k 处计算解是精确的，即 $u_k = u(t_k)$，那么在步骤 $k+1$ 时我们会产生多大误差？我们有

$$\begin{aligned} u(t_{k+1}) &= u(t_k) + u'(t_k)\Delta t + u''(t_k)\frac{\Delta t^2}{2!} + \cdots \\ &= u(t_k) + f(t_k, u(t_k))\Delta t + u''(t_k)\frac{\Delta t^2}{2!} + \cdots \end{aligned} \tag{4.15}$$

and

$$u_{k+1} = u_k + f(t_k u_k)\Delta t \tag{4.16}$$

So

$$\begin{aligned} L_{k+1} &= u_{k+1} - u(t_{k+1}) = u_k - u(t_k) + f(t_k, u_k) - f(t_k, u(t_k)) - u''(t_k)\frac{\Delta t^2}{2!} + \cdots \\ &= -u''(t_k)\frac{\Delta t^2}{2!} + \cdots \end{aligned} \tag{4.17}$$

这表明在每一步中我们都会产生 $O(\Delta t^2)$的误差。若假设这些误差可累加，则全局误差为

$$E_k \approx \Sigma_k L_k = k\Delta t \frac{\Delta t^2}{2!} = O(\Delta t) \tag{4.18}$$

由于全局误差在 Δt 中为一阶，我们称此为 ‘一阶方法’。需注意，该误差（衡量真实解与计算解之间的差异）与截断误差（算子近似产生的误差）同属 $O(\Delta t)$ 阶。

4.1.2.1 显式欧拉方法的稳定性

考虑初值问题 $u' = f(t,u)$（针对 $t \geq 0$），其中给定 $f(t,u) = -\lambda u$ 及初始值 $u(0) = u_0$。其精确解为 $u(t) = u_0 e^{-\lambda t}$。根据前述讨论，我们得出该问题是稳定的 —— 即若 $\lambda > 0$，解的微小扰动最终会衰减。现在我们将探究数值解是否与精确解表现一致，即数值解是否同样收敛于零。

针对此问题的欧拉前向（显式欧拉）格式为

$$u_{k+1} = u_k - \Delta t \lambda u_k = (1 - \lambda\Delta t)u_k \Rightarrow u_k = (1 - \lambda\Delta t)^k u_0. \tag{4.19}$$

为保证稳定性，我们要求 $u_k \to 0$ 当 $k \to \infty$ 时。这等价于

$$\begin{aligned} u_k \downarrow 0 &\Leftrightarrow |1 - \lambda\Delta t| < 1 \\ &\Leftrightarrow -1 < 1 - \lambda\Delta t < 1 \\ &\Leftrightarrow -2 < -\lambda\Delta t < 0 \\ &\Leftrightarrow 0 < \lambda\Delta t < 2 \\ &\Leftrightarrow \Delta t < 2/\lambda \end{aligned}$$

可见数值解法的稳定性取决于 Δt 的值：仅当 Δt 足够小时该格式才稳定。因此，我们称显式欧拉方法为条件稳定。需注意微分方程的稳定性与数值格式的稳定性是两个不同问题。连续问题在 $\lambda > 0$ 时稳定；而数值问题还需额外满足取决于所用离散化格式的条件。

请注意我们刚进行的稳定性分析仅针对微分方程 $u' = -\lambda u$。若处理不同的初值问题 (IVP)，需另行分析。但你会发现显式方法通常具有条件稳定性。

4.1.2.2 欧拉隐式方法

您刚才看到的显式方法易于计算，但条件稳定性是一个潜在问题。例如，这可能意味着时间步数将成为限制因素。有一种替代显式方法的方法不会受到同样的限制。

不展开 $u(t+\Delta t)$，而是考虑 $u(t-\Delta t)$ 的以下展开：

$$u(t-\Delta t) = u(t) - u'(t)\Delta t + u''(t)\frac{\Delta t^2}{2!} + \cdots \tag{4.20}$$

这意味着

$$u'(t) = \frac{u(t) - u(t-\Delta t)}{\Delta t} + u''(t)\Delta t/2 + \cdots \tag{4.21}$$

与之前一样，我们取方程 $u'(t) = f(t,u(t))$ 并用差分公式近似 $u'(t)$：

$$\frac{u(t) - u(t-\Delta t)}{\Delta t} = f(t,u(t)) + O(\Delta t) \Rightarrow u(t) = u(t-\Delta t) + \Delta t f(t,u(t)) + O(\Delta t^2) \tag{4.22}$$

A增益我们定义固定点 $t_k = kt$，并定义一个数值方案：

$$u_{k+1} = u_k + \Delta t f(t_{k+1}, u_{k+1}) \tag{4.23}$$

其中 u_k 是 $u(t_k)$ 的近似值。

与显式方案的一个重要区别在于，u_{k+1} 现在也出现在方程的右侧。也就是说，u_{k+1} 的计算现在是隐式的。例如，设 $f(t,u) = -u^3$，则 $u_{k+1} = u_k - \Delta t u_{k+1}^3$。换言之，$u_{k+1}$ 是方程 $\Delta t x^3 + x = u_k$ 关于 x 的解。这是一个非线性方程，通常可以使用牛顿方法求解。

4.1.2.3 隐式欧拉方法的稳定性

让我们再看一下示例 $f(t,u) = -\lambda u$。构建隐式方法得到

$$u_{k+1} = u_k - \lambda\Delta t u_{k+1} \Leftrightarrow (1+\lambda\Delta t)u_{k+1} = u_k \tag{4.24}$$

so

$$u_{k+1} = \left(\frac{1}{1+\lambda\Delta t}\right)u_k \Rightarrow u_k = \left(\frac{1}{1+\lambda\Delta t}\right)^k u_0. \tag{4.25}$$

如果 $\lambda > 0$，这是稳定方程的条件，我们发现 $u_k \to 0$ 对于所有 λ 和 Δt 的值都成立。这种方法被称为无条件稳定。隐式方法相对于显式方法的一个明显优势在于稳定性：可以采取更大的时间步长而无需担心非物理行为。当然，大的时间步长可能会使收敛到稳态（见附录 16.4 ）变慢，但至少不会出现发散。

另一方面，隐式方法更为复杂。正如上文所述，它们可能涉及在每个时间步长中求解非线性系统。当 u 为向量值时（如下文讨论的热方程所示），您会发现隐式方法需要求解方程组。

习题 4.1. 分析以下针对初值问题 $u'(x) = f(x)$的数值格式的精度与计算特性:

$$u_{i+1} = u_i + h(f(x_i) + f(x_{i+1}))/2 \tag{4.26}$$

该格式相当于将显式欧拉与隐式欧拉格式相加。你无需分析此格式的稳定性。

习题 4.2. 考虑初值问题 $y'(t) = y(t)(1 - y(t))$。注意到 $y \equiv 0$ 和 $y \equiv 1$ 是解，称为 ‘平衡解’。1. 若扰动 ‘收敛回解’，即对于足够小的 ϵ，则称解是稳定的。

$$\text{if } y(t) = \epsilon \text{ for some } t, \text{ then } \lim_{t\to\infty} y(t) = 0 \tag{4.27}$$

and

$$\text{if } y(t) = 1 + \epsilon \text{ for some } t, \text{ then } \lim_{t\to\infty} y(t) = 1 \tag{4.28}$$

这需要满足例如

$$y(t) = \epsilon \Rightarrow y'(t) < 0. \tag{4.29}$$

研究该行为。零解是稳定的吗? 值为 1 的解呢?

2. 考虑显式方法

$$y_{k+1} = y_k + \Delta t y_k(1 - y_k) \tag{4.30}$$

用于计算微分方程的数值解。证明

$$y_k \in (0, 1) \Rightarrow y_{k+1} > y_k, \qquad y_k > 1 \Rightarrow y_{k+1} < y_k \tag{4.31}$$

3. 编写一个小程序来研究数值解在不同 Δt 选择下的行为。作业提交中需包含程序清单及若干次运行结果。

4. 通过运行程序可见数值解可能出现振荡。推导一个使数值解单调的 Δt 条件。仅需证明 $y_k < 1 \Rightarrow y_{k+1} < 1$ 及 $y_k > 1 \Rightarrow y_{k+1} > 1$ 即可。

5. 现在考虑隐式方法

$$y_{k+1} - \Delta t y_{k+1}(1 - y_{k+1}) = y_k \tag{4.32}$$

并证明 y_{k+1} 可从 y_k 计算得出。编写程序，研究数值解在不同 Δt 选择下的行为。

6. 证明隐式格式的数值解对所有 Δt 的选择都是单调的。

4.2 边值问题

在前一节中，您了解了初值问题，它们模拟随时间演化的现象。现在我们将转向 ‘边值问题’，这类问题通常在时间上是静止的，但描述的现象与空间位置相关。例如桥梁在荷载作用下的形状，或窗玻璃中的温度分布（当内外温度不同时）。

（二阶一维）边值问题的一般形式为

$$u''(x) = f(x, u, u') \text{ for } x \in [a, b] \text{ where } u(a) = u_a, u(b) = u_b \tag{4.33}$$

但此处我们仅考虑简单形式（参见附录 16 ）

$$-u''(x) = f(x) \text{ for } x \in [0, 1] \text{ with } u(0) = u_0, u(1) = u_1. \tag{4.34}$$

在一维空间中，或

$$-u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) \text{ for } \bar{x} \in \Omega = [0, 1]^2 \text{ with } u(\bar{x}) = u_0 \text{ on } \delta\Omega. \tag{4.35}$$

在二维空间中。此处， $\delta\Omega$ 是区域 Ω 的边界。由于我们在边界上规定了 u 的值，此类问题被称为边值问题（ BVP ）。

备注 14 边界条件可以更为广义，涉及区间端点的导数。此处我们仅考察狄利克雷边界条件，即在区域边界上规定函数值。与诺伊曼边界条件的差异更多体现在数学层面而非计算层面。

4.2.1 偏微分方程通论

偏微分方程有若干类型，各自具有不同的数学属性。最重要的属性是影响区域：如果我们调整问题使得解在某一点发生变化，其他哪些点会受到影响。

4.2.1.1 双曲型方程

偏微分方程的形式为

$$Au_{xx} + Bu_{yy} + \text{lower order terms} = 0 \tag{4.36}$$

当 A 与 B 符号相反时。这类方程描述的是保守的波动或更一般的对流现象，不会趋于稳态。

直观而言，改变波动方程在任意点的解只会影响特定的未来点，因为波具有传播速度，使得某一点无法影响空间上过于遥远且时间上临近的未来点。本书将不讨论此类偏微分方程。

4.2.1.2 抛物型方程

PDEs are of the form

$$Au_x + Bu_{yy} + \text{no higher order terms in } x = 0 \tag{4.37}$$

它们描述了类似扩散的现象；这些现象通常会趋向于一个稳态。最佳的表征方式是认为空间和时间中每一点的解都受到先前空间各点某个有限区域的影响。

备注 15 这导致在初边值问题中存在一个限制时间步长的条件，称为 Courant-Friedrichs-Lewy 条件。它描述了精确问题中 $u(x,t)$ 依赖于 $u(x',t-\Delta t)$ 值范围的概念；数值方法的时间步长必须足够小，以便数值解能考虑到所有这些点。

热传导方程 (章节 4.3) 是抛物型的标准示例。

4.2.1.3 椭圆型方程

偏微分方程具有如下形式

$$Au_{xx} + Bu_{yy} + \text{lower order terms} = 0 \tag{4.38}$$

其中 $A, B > 0$ ；这类方程通常描述已达到稳态的过程，例如抛物问题中的 $t \to \infty$ 。其特征是所有点之间相互影响。这类方程常用于描述结构力学现象，如梁或膜的行为。直观上很容易理解：按压膜上任意一点都会改变其他所有点的高度，无论变化多么微小。拉普拉斯方程（章节 4.2.2 ）是此类方程的典型代表。

4.2.2 一维空间中的拉普拉斯方程

我们将由下式定义的算子 Δ 称为

$$\Delta u = u_{xx} + u_{yy}, \tag{4.39}$$

二阶微分算子，而方程（ 4.35 ）称为二阶偏微分方程。具体而言，问题

$$-\Delta u = -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) \text{ for } \bar{x} \in \Omega = [0,1]^2 \text{ with } u(\bar{x}) = u_0 \text{ on } \delta\Omega. \tag{4.40}$$

被称为泊松方程（此处定义在单位正方形上）。当 $f \equiv 0$ 时则称为拉普拉斯方程。二阶偏微分方程十分常见，可描述流体力学、热传导及结构力学中的诸多现象。

首先，为简化起见，我们考虑一维泊松方程

$$-u_{xx} = f(x). \tag{4.41}$$

我们将在下文考虑二维情况；而向三维的扩展将变得显而易见。

为了找到数值方案，我们像之前一样使用泰勒级数，用 u 及其在 x 处的导数表示 $u(x+h)$ 和 $u(x-h)$。设 $h>0$，则有

$$u(x+h)=u(x)+u'(x)h+u''(x)\frac{h^2}{2!}+u'''(x)\frac{h^3}{3!}+u^{(4)}(x)\frac{h^4}{4!}+u^{(5)}(x)\frac{h^5}{5!}+\cdots \tag{4.42}$$

and

$$u(x-h)=u(x)-u'(x)h+u''(x)\frac{h^2}{2!}-u'''(x)\frac{h^3}{3!}+u^{(4)}(x)\frac{h^4}{4!}-u^{(5)}(x)\frac{h^5}{5!}+\cdots \tag{4.43}$$

我们的目标是近似 $u''(x)$。可以看到，这些方程中的 u' 项在相加时会相互抵消，剩下 $2u(x)$：

$$u(x+h)+u(x-h)=2u(x)+u''(x)h^2+u^{(4)}(x)\frac{h^4}{12}+\cdots \tag{4.44}$$

so

$$-u''(x)=\frac{2u(x)-u(x+h)-u(x-h)}{h^2}+u^{(4)}(x)\frac{h^2}{12}+\cdots \tag{4.45}$$

数值方案的基础是观察到 (4.34)

$$\frac{2u(x)-u(x+h)-u(x-h)}{h^2}=f(x,u(x),u'(x))+O(h^2), \tag{4.46}$$

这表明我们可以用差分算子近似微分算子，其 $O(h^2)$截断误差为 $h\downarrow 0$。

为了推导数值方法，我们将区间 $[0,1]$ 划分为等距点： $x_k=kh$ 其中 $h=1/(n+1)$ 且 $k=0\ldots n+1$。利用这些点，有限差分公式 (4.45) 导出一个形成方程组数值方案：

$$-u_{k+1}+2u_k-u_{k-1}=h^2f(x_k)\quad \text{for } k=1,\ldots,n \tag{4.47}$$

使用有限差分公式（ 4.45 ）近似求解偏微分方程的过程被称为有限差分方法（ *FDM* ）。

对于 k 的大多数值，该方程将 u_k 未知量与未知量 u_{k-1} 和 u_{k+1} 相关联。异常情况是 $k=1$ 和 $k=n$。此时我们记得 u_0 和 u_{n+1} 是已知的边界条件，并将方程改写为未知量在左侧、已知量在右侧的形式：

$$\begin{cases}-u_{i-1}+2u_i-u_{i+1}=h^2f(x_i)\\ 2u_1-u_2=h^2f(x_1)+u_0\\ 2u_n-u_{n-1}=h^2f(x_n)+u_{n+1}.\end{cases} \tag{4.48}$$

现在我们可以将这些关于 u_k、 $k=1\ldots n-1$ 的方程总结为一个矩阵方程：

$$\begin{pmatrix}2&-1&&&\\-1&2&-1&&\\&\ddots&\ddots&\ddots&\\&&&-1&2\end{pmatrix}\begin{pmatrix}u_1\\u_2\\\vdots\\u_n\end{pmatrix}=\begin{pmatrix}h^2f_1+u_0\\h^2f_2\\\vdots\\h^2f_n+u_{n+1}\end{pmatrix} \tag{4.49}$$

其形式为 $Au = f$，其中 A 为完全已知的矩阵，f 为完全已知的向量，u 为未知向量。注意右侧向量在首尾位置包含了问题的边界值。这意味着，若要在不同边界条件下求解同一微分方程，仅需改变向量 f。

习题 4.3. 形如 $u(0) = u_0$ 的条件称为狄利克雷边界条件。物理上，这相当于已知杆件端点的温度。其他边界条件也存在。若指定导数值 $u'(0) = u'_0$而非函数值，则适用于模拟流体流动且已知 $x = 0$ 处流出速率的情形，这被称为诺伊曼边界条件。诺伊曼边界条件 $u'(0) = u'_0$ 可通过声明来建模

$$\frac{u_0 - u_1}{h} = u'_0. \tag{4.50}$$

证明与 Dirichlet 边界条件的情况不同，这会影响线性系统的矩阵。

证明在两端均采用 Neumann 边界条件会导致矩阵奇异，从而使线性系统无唯一解。（提示：猜测特征值为零的向量。）从物理角度这是合理的。例如，在弹性问题中，Dirichlet 边界

条件表明杆件在某一高度被固定；而 Neumann 边界条件仅规定其端点的角度，这使得其高度无法确定。

让我们列出 A 的一些性质，稍后你会看到这些性质与求解此类方程组相关：

- 该矩阵非常稀疏：非零元素占比很低。非零元素并非随机分布，而是集中在主对角线周围的带状区域。我们通常称其为带状矩阵，在此特定情况下则称为三对角矩阵。这种带状结构是偏微分方程的典型特征，但不同应用中的稀疏矩阵可能规则性较低。

- 该矩阵是对称的。这一属性并非所有来源于离散化边值问题的矩阵都具备，但当不存在奇数阶（如一阶、三阶、五阶等）导数时（例如 u_x），该性质成立。

  ，

  u_{xxx}, u_{xyy}。

- 矩阵元素在各对角线（即每组点 $\{(i, j) : i - j = c\}$，其中 c）上保持恒定。此性质仅适用于非常简单的问题。若微分方程含有位置相关项（如 $\frac{d}{dx}(a(x)\frac{d}{dx}u(x))$），或使 h 变量在区间内变化（例如需对左端点附近行为进行更精细建模时），该性质将不再成立。

- 矩阵元素符合以下符号模式：对角线元素为正，非对角线元素为非正。该属性取决于所用数值格式 —— 例如对于不含一阶项的二阶方程二阶格式成立。结合下文的正定性，此类矩阵被称为 M 矩阵。关于这类矩阵存在完整的数学理论 [13]。

- 该矩阵是正定的：$x^t Ax > 0$ 对于所有非零向量 x。这一属性继承自原始连续问题，前提是数值方案经过谨慎选择。尽管

目前看来这一用途可能并不明确，但稍后你将看到依赖于它的求解线性系统的方法。

严格来说，方程 (4.49) 的解很简单：$u = A^{-1} f$。然而，计算 A^{-1} 并非找到 u 的最佳方式。如前所述，矩阵 A 仅需存储 $3N$ 个非零元素。而其逆矩阵却连一个非零元素都没有。虽然我们不会在此证明，但这类结论对大多数稀疏矩阵都成立。因此，我们需要以不依赖 $O(n^2)$ 存储的方式求解 $Au = f$。

习题 4.4. 你会如何求解三对角方程组？证明系数矩阵的 LU 分解得到的因子具有双对角矩阵形式：它们有一条非零对角线及恰好一条非零次对角线或超对角线。求解三对角方程组的操作总数是多少？用此类矩阵乘以向量的操作计数又是多少？这种关系并不典型！

4.2.3 二维空间中的拉普拉斯方程

前述一维边值问题在多个方面具有非典型性，尤其是与由此产生的线性代数问题相关。本节我们将探讨二维泊松 / 拉普拉斯问题。您将发现这是对一维问题的非平凡推广。三维情况与二维非常相似，因此不再赘述。（关于一个本质差异，请参阅 7.8.1 节的讨论。）

前述一维问题中的函数 $u = u(x)$ 在二维情况下变为 $u = u(x, y)$。我们关注的二维问题可表述为

$$-u_{xx} - u_{yy} = f, \quad (x, y) \in [0, 1]^2, \tag{4.51}$$

其中边界值和给定。在 x, y 方向上通过应用方程 (4.45

$$\begin{aligned} -u_{xx}(x, y) &= \frac{2u(x,y) - u(x+h,y) - u(x-h,y)}{h^2} + u^{(4)}(x, y)\frac{h^2}{12} + \cdots \\ -u_{yy}(x, y) &= \frac{2u(x,y) - u(x,y+h) - u(x,y-h)}{h^2} + u^{(4)}(x, y)\frac{h^2}{12} + \cdots \end{aligned} \tag{4.52}$$

或合并表示为：

$$4u(x, y) - u(x + h, y) - u(x - h, y) - u(x, y + h) - u(x, y - h) = 1/h^2\, f(x, y) + O(h^2) \tag{4.53}$$

再次令 $h = 1/(n + 1)$ 并定义 $x_i = ih$ 和 $y_j = jh$；设 u_{ij} 为 $u(x_i, y_j)$ 的近似值，则我们的离散方程变为

$$4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = h^{-2} f_{ij}. \tag{4.54}$$

现在我们得到 $n \times n$ 个未知数 u_{ij}。为了将其渲染为线性系统，如同之前一样，我们需要将它们按线性顺序排列，这通过定义 $I = I_{ij} = j + i \times n$ 实现。这被称为字典序，因为它将坐标 (i, j) 视为字符串进行排序。

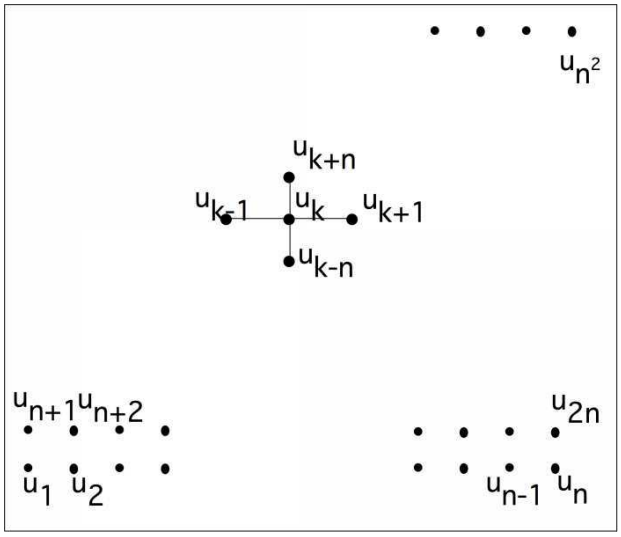


图 4.1：应用于二维方形区域的差分格式。

采用这种排序方式可得到 $N = n^2$ 个方程

$$4u_I - u_{I+1} - u_{I-1} - u_{I+n} - u_{I-n} = h^{-2} f_I, \qquad I = 1, \ldots, N \tag{4.55}$$

线性方程组的形式如下

$$A = \left(\begin{array}{ccccc|ccccc|ccc}
4 & -1 & & & \emptyset & -1 & & & & \emptyset & & & \\
-1 & 4 & -1 & & & & -1 & & & & & & \\
 & \ddots & \ddots & \ddots & & & & \ddots & & & & & \\
 & & \ddots & \ddots & -1 & & & & \ddots & & & & \\
\emptyset & & & -1 & 4 & \emptyset & & & & -1 & & & \\
-1 & & & & \emptyset & 4 & -1 & & & & -1 & & \\
 & -1 & & & & -1 & 4 & -1 & & & & -1 & \\
 & \uparrow & \ddots & & & \uparrow & \uparrow & \uparrow & & & & \uparrow & \\
 & k-n & & & & k-1 & k & k+1 & & -1 & & k+n & \\
 & & & & -1 & & & & -1 & 4 & & & \\
 & & & & & \ddots & & & & & \ddots & &
\end{array}\right) \tag{4.56}$$

其中矩阵大小为 $N \times N$，且 $N = n^2$。与一维情况类似，我们观察到边值问题会生成一个稀疏矩阵。

以矩阵形式考虑该线性方程组看似自然，但通过渲染这些方程以明确未知量间的二维关联可能更具启发性。为此，图 4.1 展示了区域内的变量分布，以及方程 (4.54) 如何通过有限差分格式建立它们之间的联系（关于格式的更多讨论见第 4.2.4 节）。此后绘制区域示意图时，我们将仅使用变量的下标索引，并省略 ‘u’ 标识符。

方程 4.56 的矩阵在一维情况下呈带状分布，但与一维情况不同的是，带状内部存在零元素。（这在尝试求解时会产生一些重要影响）

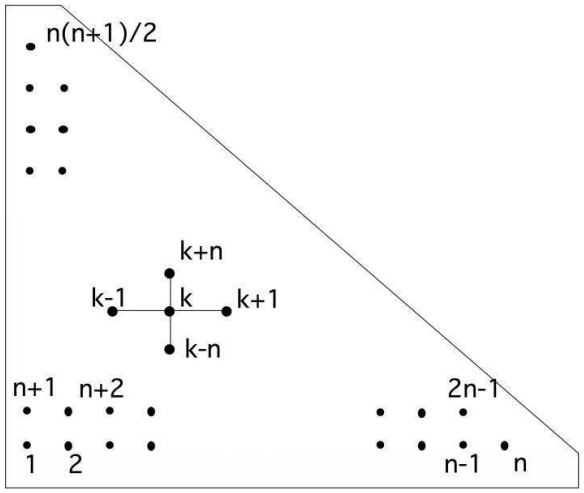


图 4.2：泊松方程定义的三角区域。

线性系统；参见章节 5.4.3。）由于矩阵有五条非零对角线，因此被称为具有五对角结构。

您还可以通过将同一行区域内的未知量分组，对矩阵施加块结构。这被称为块矩阵，在块级别上，它具有三对角矩阵结构，因此我们称之为块三对角矩阵。注意对角块本身是三对角的；非对角块是负单位矩阵。

与上述一维示例类似，该矩阵具有恒定的对角线，但这同样归因于问题的简单性。在实际问题中，情况并非如此。尽管如此，此类 ‘常系数’ 问题确实存在，并且当它们位于矩形区域时，存在非常高效的求解线性系统的方法，其时间复杂度为 $N \log N$。

习题 4.5. 矩阵的分块结构，其中所有对角块具有相同的大小，
这是由于我们在方形区域上定义了边界值问题。请绘制由离散化方程 (4.51) 产生的矩阵结构，同样使用中心差分法，但这次定义在三角形区域上；参见图 4.2。展示此时矩阵仍具有块三对角结构，但各块大小不一。提示：可先绘制一个小规模示例。对于 $n = 4$，应得到一个 10×10 矩阵，其具有 4×4 块结构。

对于更加不规则的区域，矩阵结构也将变得不规则。

规则的块结构还源于我们按行和列排序未知数的决策。这被称为自然排序或字典序；还存在其他多种排序方式。一种常见的未知数排序方法是红黑排序或棋盘排序，这对并行计算具有优势。这将在第 7.7 节讨论。

备注 16 对于这个简单的矩阵，可以通过傅里叶分析来确定特征值和特征向量。在更复杂的情况下，比如算子 $\nabla a(x,y)\nabla x$，这是不可能的，但 Gershgorin

定理 仍会告诉我们该矩阵是正定的，或至少是半正定的。这对应于连续问题具有强制性。

关于边值问题（ BVP ）的解析性质还有更多可探讨（例如，解的光滑性如何？其与边界条件的关系如何？），但这些问题超出了本课程范围。此处我们仅聚焦于矩阵的数值特性。在线性代数章节，特别是 5.4 与 5.5 节，我们将讨论如何求解边值问题产生的线性系统。

4.2.4 差分格式

离散化（ 4.53 ）常表述为对函数应用差分格式

$$\begin{array}{ccc} \cdot & -1 & \cdot \\ -1 & 4 & -1 \\ \cdot & -1 & \cdot \end{array} \tag{4.57}$$

于函数 u。给定物理域后，我们将该格式应用于域内每个点以导出该点的方程。图 4.1 展示了具有 $n \times n$ 个点的方形域情况。将此图与方程（ 4.56 ）关联可见：同一行的连接产生主对角线及首次上下副对角线；与相邻行的连接则形成非对角块中的非零元素。

这种特定模板常被称为 ‘ 五点星形 ’ 或五点差分模板。还存在其他差分模板；其中部分结构如图 4.3 所示。仅含水平或垂直方向连接的模板被称为

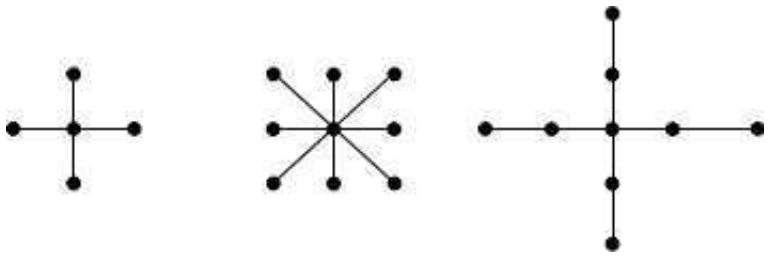


图 4.3：二维空间中几种差分模板的结构示意图。

‘ 星形模板 ’，而含有交叉连接的模板（如图 4.3 中第二个示例）则称为 ‘ 盒式模板 ’ 。

习题 4.6 考虑图 4.3 中第二和第三个模板，用于方形区域上的边值问题。若我们仍按行列顺序排列变量，所得矩阵的稀疏结构会呈现何种形态?

除五点星形外，其他模板可用于实现更高精度，例如使截断误差达到 $O(h^4)$。它们也适用于上文所述之外的其他微分方程。例如，不难证明对于方程 $u_{xxxx} + u_{yyyy} = f$，我们需要包含 x、$y \pm h$ 和 x、 $y \pm 2h$ 连接的模板（如图中第三个模板）。反之，使用五点

模板系数值无法以低于 $O(1)$ 截断误差的方式对四阶问题进行离散化。

尽管目前讨论集中在二维问题上，但可将其推广至高维方程如 $-u_{xx} - u_{yy} - u_{zz} = f$。例如，五点模板的直接推广在三维中会变为七点模板。

习题 4.7. 三维空间中中心差分模板的矩阵结构如何？请描述其分块结构。

4.2.5 其他离散化技术

前文我们使用有限差分法 (FDM) 求解微分方程的数值解。实际上还存在多种其他技术，尤其在边界值问题中，这些方法通常比有限差分更受青睐。最流行的两种方法是有限元法 ***(FEM)*** 和有限体积法。有限元法因其能更灵活处理不规则几何形状，且更便于进行近似误差分析而尤为突出。不过，在本文讨论的简单问题上，它产生的线性方程组与有限差分法相似甚至相同。鉴于我们主要关注线性系统的计算特性，故将讨论限定在有限差分法范畴。

关于有限元矩阵的简要讨论将在第 7.6.2 节进行。

4.3 初始边界值问题

接下来我们将讨论初始边界值问题（ IBVP ），从名称可以推断，它结合了初值问题（ IVP ）和边值问题（ BVP ）的特点：同时包含空间和时间导数。此处我们仅限一维空间情形。

我们考虑的问题是杆件中的热传导，其中 $T(x,t)$ 描述位置 x 在时间 t 的温度，其中 $x \in [a,b]$，$t > 0$。所谓的热方程（简短推导见第 16.3 节）为：

$$\frac{\partial}{\partial t} T(x,t) - \alpha \frac{\partial^2}{\partial x^2} T(x,t) = q(x,t) \tag{4.58}$$

其中

- 初始条件 $T(x,0) = T_0(x)$ 描述了初始温度分布。
- 边界条件 $T(a,t) = T_a(t)$, $T(b,t) = T_b(t)$ 描述了杆的端点情况，例如可以固定在一个已知温度的物体上。
- 构成杆的材料由单一参数 $\alpha > 0$ （热扩散率）建模，该参数描述了热量在材料中的扩散速度。
- 外力函数 $q(x,t)$ 描述了外部施加的加热情况，作为时间和位置的函数。
- 空间导数是我们之前研究过的相同算子。

初边值问题（IBVP）与边值问题（BVP）之间存在简单关联：若边界函数 T_a 和 T_b 为常数，且 q 不随时间变化仅取决于空间位置，则直观上 T 将收敛至稳态。其对应方程为 $-\alpha u''(x) = q$，这正是我们先前研究过的情形。

接下来我们将讨论一维空间中的热传导方程。基于前文对二维及三维泊松方程的探讨，将热传导方程扩展至高维空间应无特殊困难。

4.3.1 离散化

现通过 $x_{j+1} = x_j + \Delta x$ 和 $t_{k+1} = t_k + \Delta t$ 对空间与时间进行离散化处理，边界条件为 $x_0 = a$、$x_n = b$ 及 $t_0 = 0$。记 T_j^k 为 $x = x_j$、$t = t_k$ 处的数值解；若近似得当，该解将逼近精确解 $T(x_j, t_k)$。

对于空间离散化，我们采用中心差分公式（4.47）：

$$\left.\frac{\partial^2}{\partial x^2} T(x, t_k)\right|_{x=x_j} \Rightarrow \frac{T_{j-1}^k - 2T_j^k + T_{j+1}^k}{\Delta x^2}. \tag{4.59}$$

对于时间离散化，我们可以使用 4.1.2 节中的任一方案。我们将再次研究显式和隐式方案，并得出关于结果稳定性的类似结论。

4.3.1.1 显式方案

采用显式时间步进时，我们将时间导数近似为

$$\left.\frac{\partial}{\partial t} T(x_j, t)\right|_{t=t_k} \Rightarrow \frac{T_j^{k+1} - T_j^k}{\Delta t}. \tag{4.60}$$

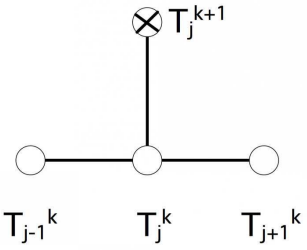


图 4.4：欧拉前向方法针对热传导方程的差分模板。

将此与空间中的中心差分结合，我们现在有

$$\frac{T_j^{k+1} - T_j^k}{\Delta t} - \alpha \frac{T_{j-1}^k - 2T_j^k + T_{j+1}^k}{\Delta x^2} = q_j^k \tag{4.61}$$

我们将其重写为

$$T_j^{k+1} = T_j^k + \frac{\alpha \Delta t}{\Delta x^2}(T_{j-1}^k - 2T_j^k + T_{j+1}^k) + \Delta t q_j^k \tag{4.62}$$

从图示角度，我们将其渲染为图 4.4 中的差分模板。这表明每个点的函数值由前一时间层上若干点的组合决定。

将方程组 (4.62) 针对给定 k 和所有 j 值以向量形式总结如下

$$T^{k+1} = \left(I - \frac{\alpha \Delta t}{\Delta x^2} K\right) T^k + \Delta t q^k \tag{4.63}$$

其中

$$K = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix}, \qquad T^k = \begin{pmatrix} T_1^k \\ \vdots \\ T_n^k \end{pmatrix}. \tag{4.64}$$

此处的重要观察是，从 T^k推导向量 T^{k+1}的主要计算是一个简单的矩阵 - 向量乘法：

$$T^{k+1} \leftarrow AT^k + \Delta t q^k \tag{4.65}$$

其中 $A = I - \frac{\alpha \Delta t}{\Delta x^2} K$。这初步表明稀疏矩阵 - 向量乘积是一个重要运算；参见章节 5.4 与 7.5。实际使用显式方法的计算机程序通常不会构建矩阵，而是直接计算方程 (4.62)。然而，线性代数表述 (4.63) 对于分析目的更具启发性。

后续章节我们将讨论操作的并行执行。目前我们注意到，显式格式天然可并行化：每个点只需利用周围少数几个点的信息即可更新。

4.3.1.2 隐式格式

在方程 (4.60) 中，我们令 T^{k+1} 由 T^k定义。我们可以反过来通过 T^k 从 T^{k-1}定义，如我们在第 4.1.2.2 节对初值问题所做的那样。对于时间离散化，这给出了

$$\frac{\partial}{\partial t} T(x,t)\Big|_{t=t_k} \Rightarrow \frac{T_j^k - T_j^{k-1}}{\Delta t} \qquad \text{or} \qquad \frac{\partial}{\partial t} T(x,t)\Big|_{t=t_{k+1}} \Rightarrow \frac{T_j^{k+1} - T_j^k}{\Delta t}. \tag{4.66}$$

整个热传导方程的隐式时间步离散化，在 t_{k+1} 处求值，现可表示为：

$$\frac{T_j^{k+1} - T_j^k}{\Delta t} - \alpha \frac{T_{j-1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}}{\Delta x^2} = q_j^{k+1} \tag{4.67}$$

or

$$T_j^{k+1} - \frac{\alpha \Delta t}{\Delta x^2}(T_{j-1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}) = T_j^k + \Delta t q_j^{k+1} \tag{4.68}$$

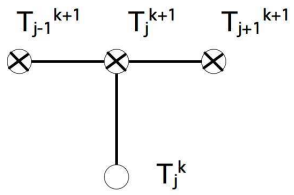


图 4.5：欧拉后向方法求解热传导方程的差分格式图示。

图 4.5 将此渲染为一个模板；这表明当前时间层上的每个点都会影响下一时间层上多个点的组合。我们再次用向量形式表示：

$$\left(I + \frac{\alpha \Delta t}{\Delta x^2} K\right) T^{k+1} = T^k + \Delta t q^{k+1} \tag{4.69}$$

与显式方法（仅需矩阵 - 向量乘法）不同，现在从 T^k 推导向量 T^{k+1} 需要求解线性方程组：

$$T^{k+1} \leftarrow A^{-1}(T^k + \Delta t q^{k+1}) \tag{4.70}$$

其中 $A = I + \frac{\alpha \Delta t}{\Delta x^2} K$。求解此线性系统比矩阵 - 向量乘法更为复杂。与方程 (4.70) 所暗示的不同，采用隐式方法的代码实际上并不构造逆矩阵，而是直接求解系统 (4.69)。线性系统的求解将成为第 5 章和第 7 章的重点内容。

与显式格式不同，我们现在没有明显的并行化策略。线性系统的并行求解将在 7.6 节及后续部分讨论。

习题 4.8. 证明隐式方法单时间步的浮点运算量与显式方法同阶。(此结论仅适用于一维空间问题。) 至少给出一个理由说明为何我们认为隐式方法在计算上更 ‘ 困难 ’ 。

此处采用的数值格式在时间上为一阶精度，空间上为二阶精度：截断误差 (4.1.2 节) 为 $O(\Delta t + \Delta x^2)$。通过在时间上也使用中心差分，可采用时间二阶精度的格式。另见习题 4.10。

4.3.2 稳定性分析

我们现在分析 IBVP 显式和隐式格式在简单情况下的稳定性。该讨论将部分参照 4.1.2.1 和 4.1.2.3 章节的内容，但由于空间组件的存在，会增加一些复杂性。

设 $q \equiv 0$，并假设 $T_j^k = \beta^k e^{i\ell x_j}$ 对于某些 ℓ[1] 成立。这一假设在直觉上是合理的：由于微分方程不会 ‘ 混合 ’ x 和 t 坐标，我们推测解将是 $T(x,t) = v(x) \cdot w(t)$ 各自解的乘积

$$\begin{cases} v_{xx} = c_1 v, & v(0) = 0,\ v(1) = 0 \\ w_t = c_2 w & w(0) = 1 \end{cases} \tag{4.71}$$

唯一有意义的解出现在 $c_1, c_2 < 0$ 的情况下，此时我们发现：

$$v_{xx} = -c^2 v \Rightarrow v(x) = e^{-icx} = e^{-i\ell\pi x} \tag{4.72}$$

其中我们代入 $c = \ell\pi$ 以考虑边界条件，且

$$w(t) = e^{-ct} = e^{-ck\Delta t} = \beta^k, \quad \beta = e^{-ck}. \tag{4.73}$$

如果关于解的这种形式的假设成立，我们需要 $|\beta| < 1$ 以保证稳定性。

1. 实际上，β 也依赖于 ℓ，但我们将省略一系列下标，因为不同的 β 值永远不会同时出现在一个公式中。

将 T_j^k的假设形式代入显式格式后得到

$$\begin{aligned} T_j^{k+1} &= T_j^k + \frac{\alpha\Delta t}{\Delta x^2}(T_{j-1}^k - 2T_j^k + T_{j+1}^k) \\ \Rightarrow \beta^{k+1}e^{i\ell x_j} &= \beta^k e^{i\ell x_j} + \frac{\alpha\Delta t}{\Delta x^2}(\beta^k e^{i\ell x_{j-1}} - 2\beta^k e^{i\ell x_j} + \beta^k e^{i\ell x_{j+1}}) \\ &= \beta^k e^{i\ell x_j}\left[1 + \frac{\alpha\Delta t}{\Delta x^2}\left[e^{-i\ell\Delta x} - 2 + e^{i\ell\Delta x}\right]\right] \\ \Rightarrow \beta &= 1 + 2\frac{\alpha\Delta t}{\Delta x^2}[\frac{1}{2}(e^{i\ell\Delta x} + e^{-\ell\Delta x}) - 1] \\ &= 1 + 2\frac{\alpha\Delta t}{\Delta x^2}(\cos(\ell\Delta x) - 1) \end{aligned}$$

为了稳定性，我们需要 $|\beta| < 1$:

- $\beta < 1 \Leftrightarrow 2\frac{\alpha\Delta t}{\Delta x^2}(\cos(\ell\Delta x) - 1) < 0$: this is true for any ℓ and any choice of $\Delta x, \Delta t$.
- $\beta > -1 \Leftrightarrow 2\frac{\alpha\Delta t}{\Delta x^2}(\cos(\ell\Delta x) - 1) > -2$: this is true for all ℓ only if $2\frac{\alpha\Delta t}{\Delta x^2} < 1$, that is

$$\Delta t < \frac{\Delta x^2}{2\alpha} \tag{4.74}$$

后一条件对允许的时间步长提出了重大限制：时间步长必须足够小才能保证方法的稳定性。这与初值问题显式方法的稳定性分析类似；但此时时间步长还与空间离散化相关。这意味着，如果我们决定需要更高的空间精度并将空间离散化 Δx 减半，时间步数将增加四倍。

现在让我们考虑隐式格式的稳定性。将解 $T_j^k = \beta^k e^{i\ell x_j}$ 的形式代入数值格式后得到

$$\begin{aligned} T_j^{k+1} - T_j^k &= \frac{\alpha\Delta t}{\Delta x^2}(T_{j_1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}) \\ \Rightarrow \beta^{k+1}e^{i\ell\Delta x} - \beta^k e^{i\ell x_j} &= \frac{\alpha\Delta t}{\Delta x^2}(\beta^{k+1}e^{i\ell x_{j-1}} - 2\beta^{k+1}e^{i\ell x_j} + \beta^{k+1}e^{i\ell x_{j+1}}) \end{aligned}$$

约去 $e^{i\ell x_j}\beta^{k+1}$ 后得到

$$\begin{aligned} 1 &= \beta^{-1} + 2\alpha\frac{\Delta t}{\Delta x^2}(\cos\ell\Delta x - 1) \\ \beta &= \frac{1}{1 + 2\alpha\frac{\Delta t}{\Delta x^2}(1 - \cos\ell\Delta x)} \end{aligned}$$

由于 $1 - \cos\ell\Delta x > 0$，因此条件 $|\beta| < 1$ 始终满足，无论 ℓ 的值以及 Δx 和 Δt 的选择如何：该方法始终稳定。

练习 4.9. 将此分析推广至二维和三维空间。是否存在质的差异?

练习 4.10. 我们此处讨论的格式在时间上为一阶、空间上为二阶：其离散化阶数为 $O(\Delta t) + O(\Delta x^2)$。推导通过显式与隐式格式平均得到的 *Crank-Nicolson* 方法，证明其无条件稳定且时间上为二阶精度。

第 5 章

数值线性代数

在第 4 章中，你已经看到偏微分方程的数值解如何引出了线性代数问题。有时这是一个简单的问题 —— 例如前向欧拉方法中的矩阵 - 向量乘法 —— 但有时则更为复杂，比如后向欧拉方法中需要求解线性方程组。求解线性系统将是本章的重点；在其他应用中（此处不作讨论），还需要解决特征值问题。

Y你可能已经学过求解线性方程组的简单算法：消元法，
a也称为高斯消元法。这一方法仍可使用，但我们需要对其效率进行细致讨论。
i此外还有其他算法，即所谓的迭代解法，它们通过
g逐步逼近线性方程组的解。它们本身值得单独讨论。

由于偏微分方程的背景，我们仅考虑方形非奇异的线性系统。矩形（尤其是超定）方程组在数值分析的优化理论领域也有重要应用，但本书不会涉及这部分内容。

数值线性代数计算的标准著作是 Golub 和 Van Loan 的《矩阵计算》 [80]。该书涵盖算法、误差分析及计算细节。Heath 的《科学计算》 [97] 则涉及科学计算中最常见的计算类型；本书包含大量优质习题与实践项目。

5.1 未知量消元法

本节我们将深入探讨高斯消元法（或称未知量消元法），这是求解线性方程组的常用技术之一

$$Ax = b$$

对于 x，给定一个系数矩阵 A 和已知的右侧向量 b。您可能已接触过此方法（若未接触，下文将作解释），但此处我们将更系统地分析其各个方面。

备注 17 同样可以通过计算逆矩阵 A^{-1}来求解方程 $Ax = y$ 中的 x，例如执行高斯 - 约当算法并乘以 $x \leftarrow A^{-1}x$。不采用此方法的主要原因在于数值精度问题，这已超出本书讨论范围。

本章的一个核心脉络是讨论各种算法的效率问题。如果你在基础线性代数课程中学过如何通过逐步消元法求解未知数系统，很可能从未将该方法应用于大于 4×4 的矩阵。偏微分方程求解中出现的线性系统可能庞大数千倍，因此计算其所需操作次数及内存占用量变得至关重要。

让我们通过一个例子说明选择正确算法时效率的重要性。线性系统的解可以用行列式写出相当简洁的显式公式，即所谓的 ' 克拉默法则 '。这种解法在数学上很优雅，但对于我们的实际需求却完全不适用。

若给定矩阵 A 和向量 b，且需要找到满足 $Ax = b$ 的向量 x，则可记 $|A|$ 为 e
行列式，

$$x_i = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1n} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2n} \\ \vdots & & & & \vdots & & & \vdots \\ a_{n1} & & \dots & & b_n & & \dots & a_{nn} \end{vmatrix} / |A|$$

对于任意矩阵 M，行列式递归定义为

$$|M| = \sum_i (-1)^i m_{1i} |M^{[1,i]}|$$

其中 $M^{[1,i]}$ 表示通过从 M 中删除第 1 行和第 i 列得到的矩阵。这意味着计算一个 n 维矩阵的行列式需要进行 n 次 $n-1$ 维行列式的计算。每一次又需要计算 $n-1$ 个 $n-2$ 维行列式，因此计算行列式所需的操作次数随矩阵规模呈阶乘级增长。即使不考虑数值稳定性问题，这一计算量也会迅速变得不可行。本章后续将展示其他求解线性方程组方法的复杂度估计，这些方法更为合理。

现在让我们通过消元法看一个求解线性方程组的简单示例。考虑方程组

$$\begin{aligned} 6x_1 \quad -2x_2 \quad +2x_3 &= \quad 16 \\ 12x_1 \quad -8x_2 \quad +6x_3 &= \quad 26 \\ 3x_1 \quad -13x_2 \quad +3x_3 &= -19 \end{aligned} \tag{5.1}$$

我们通过以下方式从第二和第三个方程中消去 x_1：

- 将第一个方程 $\times 2$ 相乘，并将结果从第二个方程中减去，然后
- 将第一个方程 $\times 1/2$ 相乘，并将结果从第三个方程中减去。

The linear system then becomes

$$\begin{aligned} 6x_1 \quad -2x_2 \quad +2x_3 &= \quad 16 \\ 0x_1 \quad -4x_2 \quad +2x_3 &= \quad -6 \\ 0x_1 \quad -12x_2 \quad +2x_3 &= -27 \end{aligned}$$

请确认此方程组仍与原方程组同解。

最后，我们通过将第二个方程乘以 3 并从第三个方程中减去结果，消去第三个方程中的 x_2：

$$\begin{array}{rrrl} 6x_1 & -2x_2 & +2x_3 &= 16 \\ 0x_1 & -4x_2 & +2x_3 &= -6 \\ 0x_1 & +0x_2 & -4x_3 &= -9 \end{array}$$

现在我们可以从最后一个方程解出 $x_3 = 9/4$。将其代入第二个方程，得到 $-4x_2 = -6 - 2x_2 = -21/2$，因此 $x_2 = 21/8$。最后，由第一个方程 $6x_1 = 16 + 2x_2 - 2x_3 = 16 + 21/4 - 9/2 = 76/4$ 可得 $x_1 = 19/6$。

我们可以通过省略 x_i 系数更简洁地描述消元过程。记作

$$\begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 16 \\ 26 \\ -19 \end{pmatrix}$$

as

$$\left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 12 & -8 & 6 & 26 \\ 3 & -13 & 3 & -19 \end{array}\right] \tag{5.2}$$

则消元过程为

$$\left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 12 & -8 & 6 & 26 \\ 3 & -13 & 3 & -19 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & -12 & 2 & -27 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & 0 & -4 & -9 \end{array}\right].$$

在上述示例中，矩阵系数可以是任何实数（或复数）系数，您可以机械地遵循消元步骤。但存在以下异常情况：在计算的某个阶段，我们需要除以位于最后消元步骤矩阵对角线上的数字 6、-4、-4。这些数值被称为主元，显然它们必须是非零的。

Exercise 5.1. The system

$$\begin{array}{rrrl} 6x_1 & -2x_2 & +2x_3 &= 16 \\ 12x_1 & -4x_2 & +6x_3 &= 26 \\ 3x_1 & -13x_2 & +3x_3 &= -19 \end{array}$$

与我们刚才在方程 (5.1) 中研究的情况相同，除了 $(2,2)$ 元素。
确认在第二步会出现零主元。

第一个主元是原始矩阵的元素。正如您在前述练习中所见，其他主元必须通过实际消元才能确定。特别是，无法通过观察方程组来轻易预测零主元的出现。

若主元为零，计算并非无药可救：我们总可以交换矩阵的两行，此操作称为主元置换。不难证明（任何基础线性代数教材中均可找到），对于非奇异矩阵，总能通过行交换将非零元素置于主元位置。

习题 5.2. 假设你需要交换方程组 (5.2) 中矩阵的第 2 行与第 3 行。为确保仍能计算出正确解，还需进行哪些调整？请通过交换第 2 、 3 行继续前一道习题的方程组求解，并验证所得答案的正确性。

习题 5.3. 重新审视习题 5.1。不交换第 2 、 3 行，改为交换第 2 、 3 列。这对线性方程组意味着什么？请继续完成方程组求解过程，并验证所得解是否与之前一致。

一般而言，由于浮点数和舍入误差的存在，矩阵元素在计算过程中几乎不可能精确变为零。此外，在偏微分方程（ PDE ）的背景下，对角线元素通常非零。这是否意味着选主元在实际操作中几乎总是不必要的？答案是否定的：从数值稳定性的角度来看，选主元仍是必要的。下一节将通过具体案例阐明这一观点。

5.2 计算机算术中的线性代数

本章大部分内容将假设所有运算均可在精确算术下完成。但有必要认识到有限精度计算机算术可能引发的潜在问题，这有助于我们设计能最小化舍入误差影响的算法。更严谨的数值线性代数方法会对所讨论算法进行全面误差分析，但这已超出本课程范畴。计算机算术运算的误差分析是威尔金森经典著作《代数过程中的舍入误差》 [192] 与海厄姆较新著作《数值算法的准确性与稳定性》 [104] 的核心内容。

在此，我们仅列举几个计算机算术中可能出现的典型问题范例：我们将说明为何 LU 分解过程中的主元选取不仅是一种理论工具，并给出两个由于计算机算术有限精度导致特征值计算问题的实例。

5.2.1 消元过程中的舍入控制

前文已述，当某行某列消元过程中对角线元素为零时，必须进行行交换（即 ' 主元选取 ' ）。现在让我们观察当主元非零但接近零时会发生什么情况。

考虑线性方程组

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1+\epsilon \\ 2 \end{pmatrix}$$

其解为 $x = (1,1)^t$。利用 $(1,1)$ 元素消去第一列剩余部分可得：

$$\begin{pmatrix} \epsilon & 1 \\ 0 & 1-\frac{1}{\epsilon} \end{pmatrix} x = \begin{pmatrix} 1+\epsilon \\ 2-\frac{1+\epsilon}{\epsilon} \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 1-\frac{1}{\epsilon} \end{pmatrix}.$$

我们现在可以求解 x_2 并由此得到 x_1：

$$\begin{cases} x_2 &= (1-\epsilon^{-1})/(1-\epsilon^{-1}) = 1 \\ x_1 &= \epsilon^{-1}(1+\epsilon-x_2) = 1. \end{cases}$$

若 ϵ 很小（例如 $\epsilon < \epsilon$ 机器精度），则右侧的 $1+\epsilon$ 项将为 1：我们的线性系统将变为

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

但解 $(1,1)^t$ 在机器算术中仍能满足该系统。

在第一步消元过程中，$1/\epsilon$ 会非常大，因此右侧的第二组件在 r
消元结果将为 $2-\frac{1}{\epsilon} = -1/\epsilon$，且矩阵的 $(2,2)$ 元素此时为 $-1/\epsilon$ 而非 $1-1/\epsilon$：

$$\begin{pmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{pmatrix} x = \begin{pmatrix} 1 \\ 2-\epsilon^{-1} \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{pmatrix} x = \begin{pmatrix} 1 \\ -\epsilon^{-1} \end{pmatrix}$$

先求解 x_2，再求解 x_1，得到：

$$\begin{cases} x_2 &= \epsilon^{-1}/\epsilon^{-1} = 1 \\ x_1 &= \epsilon^{-1}(1-1\cdot x_2) = \epsilon^{-1}\cdot 0 = 0, \end{cases}$$

因此 x_2 是正确的，但 x_1 完全错误。

备注 18 在此例中，计算机算术所处理的问题数值与精确算术中的数值略有偏差，但计算结果却可能出现极大错误。对此现象的分析应纳入任何优秀的数值分析课程。具体可参阅 *[97]* 第 *1* 章。

若按上述方法进行主元交换会发生什么？我们交换矩阵行后得到

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} x = \begin{pmatrix} 2 \\ 1+\epsilon \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix} x = \begin{pmatrix} 2 \\ 1-\epsilon \end{pmatrix}$$

现在无论 epsilon 大小如何，我们都将得到：

$$x_2 = \frac{1-\epsilon}{1-\epsilon} = 1, \qquad x_1 = 2 - x_2 = 1$$

在此示例中，我们使用了极小的 ϵ 值；更精细的分析表明，即使 ϵ 大于机器精度，主元选取仍有意义。一般经验法则是：始终执行

通过行交换将当前列中剩余的最大元素移至主元位置。在第 4 章中，你看到了某些实际应用中出现的矩阵；可以证明对于这些矩阵而言，主元选取从不需要；参见习题 5.14。

上述讨论的主元选取策略亦称为部分主元法，因其仅基于行交换。另一种选择是完全主元法，通过行列交换组合在剩余子块中寻找最大元素作为主元。最后，对角主元法对行和列施加相同的交换操作（这相当于对问题中的未知量重新编号，我们将在 7.8 节讨论该策略以增强问题的并行性）。这意味着主元仅在对角线上搜索。此后我们将仅考虑部分主元法。

5.2.2 舍入误差对特征值计算的影响

考虑矩阵

$$A = \begin{pmatrix} 1 & \epsilon \\ \epsilon & 1 \end{pmatrix}$$

其中 $\epsilon\text{mach} < |\epsilon| < \sqrt{\epsilon}\text{mach}$，其特征值为 $1+\epsilon$ 和 $1-\epsilon$。若在计算机算术中计算其特征多项式

$$\begin{vmatrix} 1-\lambda & \epsilon \\ \epsilon & 1-\lambda \end{vmatrix} = \lambda^2 - 2\lambda + (1-\epsilon^2) \to \lambda^2 - 2\lambda + 1.$$

我们会得到一个二重特征值 1。需注意精确特征值本身是可以用工作精度表示的；导致误差的是算法本身。显然，即使对于性质良好的对称正定矩阵，使用特征多项式也不是计算特征值的正确方法。

非对称示例：设 A 为 20 阶矩阵

$$A = \begin{pmatrix} 20 & 20 & & & \emptyset \\ & 19 & 20 & & \\ & & \ddots & \ddots & \\ & & & 2 & 20 \\ \emptyset & & & & 1 \end{pmatrix}.$$

S由于这是三角矩阵，其特征值即为对角元素。若对该矩阵施加扰动
s令 $A_{20,1} = 10^{-6}$ 可发现特征值的扰动幅度远大于元素本身的扰动：

$$\lambda = 20.6 \pm 1.9i,\ 20.0 \pm 3.8i,\ 21.2,\ 16.6 \pm 5.4i,\ \ldots$$

此外，多个计算得到的特征值含有虚数组件，而精确特征值并不具备这一特性。

5.3 LU 分解

到目前为止，我们已经在求解单一线性方程组的背景下探讨了消元法，即在将矩阵化为上三角形式的同时更新右侧向量。假设你需要用相同的矩阵但不同的右侧向量求解多个方程组。例如，在隐式欧拉方法（章节 4.1.2.2 ）中进行多时间步长计算时就会出现这种情况。能否利用第一个方程组中的部分计算成果来简化后续方程组的求解过程?

答案是肯定的。你可以将求解过程拆分为仅涉及矩阵的部分和专门处理右侧向量的部分。当需要求解一系列方程组时，第一部分只需执行一次，而幸运的是，这部分通常占据了大部分工作量。

让我们再次观察章节 5.1 中的相同示例。

$$A = \begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix}$$

在消元过程中，我们用第二行减去 $2\times$ 第一行，第三行减去 $1/2\times$ 第一行。请确认这种行的组合操作可以通过从左侧乘以 A 来实现。

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix},$$

该矩阵是单位矩阵，其消元系数位于对角线下方第一列。变量消元的第一步等价于将系统 $Ax = b$ 转换为 $L_1Ax = L_1b$。

习题 5.4：你能找到另一个 L_1 也能实现清除第一列效果的矩阵吗？唯一性问题将在下文 5.3.2 节讨论。

在下一步中，你用第三行减去 $3\times$ 第二行。请确认这对应于将当前矩阵 L_1A 从左侧乘以

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

至此我们已将系统 $Ax = b$ 转换为 $L_2L_1Ax = L_2L_1b$，且 L_2L_1A 呈 ' 上三角 ' 形式。若定义 $U = L_2L_1A$，则 $A = L_1^{-1}L_2^{-1}U$。计算诸如 L_2^{-1}的矩阵有多难？事实证明，这异常简单。

我们得出以下观察结果：

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix} \qquad L_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 0 & 1 \end{pmatrix}$$

同理可得

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix} \qquad L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix}$$

更令人称奇的是：

$$L_1^{-1}L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 3 & 1 \end{pmatrix},$$

也就是说， $L_1^{-1}L_2^{-1}$包含了 L_1^{-1}的非对角元素， L_2^{-1}保持不变，而这些元素本身又包含了消元系数，只是符号相反。（这是 *Householder* 反射器的一个特例；参见 14.6 节。）**练习** 5.5. 证明即使对角线上方存在元素，类似的结论依然成立。

若定义 $L = L_1^{-1}L_2^{-1}$，则现在有 $A = LU$；这被称为 LU 分解。可见 L 对角线下方系数正是消元过程中所用系数的相反数。更妙的是，在消去 A 第一列的同时即可写出 L 的第一列，因此计算 L 和 U 无需额外存储空间 —— 前提是我们能接受丢失 A。

5.3.1 LU 分解算法

让我们以较为正式的代码形式写出 LU 分解算法。
⟨LU 分解⟩：对于 $k = 1, n-1$：⟨消除第
k⟩列的值⟨消除第 k⟩列的值：对于
$i = k+1$ 到 n：⟨计算第 i⟩行的乘数⟨更
新行 i⟩⟨计算第 i⟩行的乘数 $a_{ik} \leftarrow a_{ik}/a_{kk}$⟨
更新行 i⟩：对于 $j = k+1$ 到 n：
$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$ 或者，将所有内容整合
到图 5.1 中。⟨LU 分解⟩：

对于 $k = 1, n-1$：对于
$i = k+1$ 到 n：
$a_{ik} \leftarrow a_{ik}/a_{kk}$ 对于 $j = k+1$
到 n：$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

图 5.1：LU 分解算法

这是呈现 LU 分解最常见的方式。然而，也存在其他计算相同结果的方法。诸如 LU 分解之类的算法可以通过多种数学上等效但具有不同计算行为的方式进行编码。在稠密矩阵的背景下，这一问题正是 van de Geijn 和 Quintana 所著的《编程科学之矩阵计算》 [183]。

5.3.1.1 计算变体

上述算法仅是计算 $A = LU$ 结果的多种方法之一。该特定公式是通过形式化行消元过程推导得出的。若观察最终结果并将 a_{ij} 元素表示为 L 和 U 元素的组合，则可得到另一种算法。

特别地，设 $j \geq k$，则

$$a_{kj} = u_{kj} + \sum_{i<k} \ell_{ki} u_{ij},$$

由此我们得到第 u_{kj} 行（对应 $j \geq k$）以先前计算的 U 项表示：

$$\forall_{j \geq k} : u_{kj} = a_{kj} - \sum_{i<k} \ell_{ki} u_{ij}. \tag{5.3}$$

类似地，对于 $i > k$，

$$a_{ik} = \ell_{ik} u_{kk} + \sum_{j<k} \ell_{ij} u_{jk}$$

由此我们得到列 ℓ_{ik} （对应 $i > k$）为

$$\forall_{i>k} : \ell_{ik} = u_{kk}^{-1}(a_{ik} - \sum_{j<k} \ell_{ij} u_{jk}). \tag{5.4}$$

（这被称为 *Doolittle* 算法，用于 LU 分解。）

在经典分析中，该算法执行相同数量的操作，但计算分析显示存在差异。例如，前一个算法会为每个 k 值更新区块 $i, j > k$，这意味着这些元素被反复读写，导致难以将这些元素保留在缓存中。

另一方面，Doolittle 算法为每个 k 仅计算 U 的一行和 L 的一列。所用元素仅需读取，因此理论上可将其保留在缓存中。

若考虑并行执行，则出现另一差异。假设在第一个算法中计算完主元后，我们尽快计算下一个主元，并让其他线程执行 $i, j > k$ 区块的更新。此时可能有多个线程同时更新剩余区块，这需要同步机制。（该论点将在 7.12 节详细阐述。）

另一方面，新算法不存在此类冲突写入：通过计算 U 的行和 L 的列即可获得足够的并行性。

5.3.1.2 *Cholesky* 分解

对称矩阵的 LU 分解不会产生互为转置的 L 和 U：L 的对角线元素为 1，而 U 包含主元。但可以将对称矩阵 A 分解为 $A = LL^t$的形式。这种分解的优势在于其计算量保持不变，

空间与原矩阵相同，即 $n(n+1)/2$ 个元素。若稍有运气，我们就能像 LU 情况那样，用分解结果覆盖原矩阵。

我们通过归纳推理来推导该算法。将 $A = LL^t$ 写成块形式：

$$A = \begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t = \begin{pmatrix} \ell_{11} & 0 \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \ell_{11} & \ell_{21}^t \\ 0 & L_{22}^t \end{pmatrix}$$

于是 $\ell_{11}^2 = a_{11}$，由此可得 ℓ_{11}。我们还发现 $\ell_{11}(L^t)_{1j} = \ell_{j1} = a_{1j}$，因此可以计算 L 的整个第一列。最后，$A_{22} = L_{22}L_{22}^t + \ell_{12}\ell_{12}^t$，故

$$L_{22}L_{22}^t = A_{22} - \ell_{12}\ell_{12}^t,$$

这表明 L_{22} 是更新后的 A_{22} 块的 Cholesky 因子。通过递归，该算法现已定义完成。

5.3.2 唯一性

在讨论数值算法时，思考不同的计算方式是否会导致相同的结果总是一个好主意。这被称为结果的 ‘唯一性’，并且具有实际用途：如果计算结果唯一，那么更换不同的软件库不会对计算产生任何影响。

让我们考虑 LU 分解的唯一性。LU 分解算法（不进行主元选取）的定义是，给定一个非奇异矩阵 A，它将生成一个下三角矩阵 L 和一个上三角矩阵 U，使得 $A = LU$。上述计算 LU 分解的算法是确定性的（它不包含 ‘选择任意满足条件的行 …’ 这类指令），因此给定相同的输入，它总是会计算出相同的输出。然而，其他算法也是可能的，因此我们需要担心它们是否给出相同的结果。

让我们假设 $A = L_1U_1 = L_2U_2$，其中 L_1 和 L_2 是下三角矩阵，而 U_1 和 U_2 是上三角矩阵。那么，$L_2^{-1}L_1 = U_2U_1^{-1}$。在这个等式中，左边是下三角矩阵的乘积，而右边仅包含上三角矩阵。

习题 5.6. 证明下三角矩阵的乘积仍是下三角矩阵，上三角矩阵的乘积仍是上三角矩阵。对于非奇异三角矩阵的逆矩阵，类似命题是否成立?

乘积 $L_2^{-1}L_1$ 显然同时是下三角和上三角矩阵，因此必为对角矩阵。记作 D，则有 $L_1 = L_2D$ 和 $U_2 = DU_1$。结论是 LU 分解不唯一，但可以 “ 在对角缩放意义下 ” 唯一确定。

习题 5.7. 第 5.3.1 节算法得到的下三角因子 L 对角元均为 1。证明该额外条件可保证分解的唯一性。

习题 5.8. 证明若改取 U 对角元为 1 的条件，同样能保证分解的唯一性。

既然我们可以要求 L 或 U 具有单位对角线，您可能会好奇是否有可能同时满足两者。（请给出一个简单论证说明为何这在严格意义上不可行。）我们可以采取以下方法：假设 $A = LU$ 其中 L 和 U 是非奇异的下三角和上三角矩阵，但未进行任何归一化处理。设

$$L = (I + L')D_L, \qquad U = D_U(I + U'), \qquad D = D_L D_U.$$

经过一些重命名后，我们现在得到一个分解式

$$A = (I + L)D(I + U) \tag{5.5}$$

其中 D 是一个包含主元的对角矩阵。

练习 5.9. 证明你也可以将分解形式归一化为

$$A = (D + L)D^{-1}(D + U). \tag{5.6}$$

这个 D 与之前的有什么关联?

习题 5.10. 考虑形如 5.6 的三对角矩阵分解。生成的 L 与 U 如何关联到 L_A, U_A 的三角部分？推导 A 与 D 之间的关系，并证明该方程生成主元。

5.3.3 主元选取

前文展示了在分解过程中必须进行主元选取（即行交换）的案例，这既是为了确保非零主元的存在，也是出于数值稳定性考量。现在我们将主元选取整合到 LU 分解中。

首先注意行交换可通过矩阵乘法描述。设

$$P^{(i,j)} = \begin{array}{c} \\ i \\ \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccc} i & & & & j & & & \end{array} \\ \begin{pmatrix} 1 & 0 & & & & & & \\ 0 & \ddots & \ddots & & & & & \\ & & 0 & & 1 & & & \\ & & & I & & & & \\ & & 1 & & 0 & & & \\ & & & & & I & & \\ & & & & & & \ddots & \end{pmatrix} \end{array}$$

则 $P^{(i,j)}A$ 是交换了 i 行与 j 行的矩阵 A。由于在分解过程的每次迭代 i 中都可能需要选取主元，我们引入序列 $p(\cdot)$，其中 $p(i)$ 表示第 i 行所交换行的 j 值。简记为 $P^{(i)} \equiv P^{(i,p(i))}$。

练习 5.11. 证明 $P^{(i)}$ 是其自身的逆元。

采用部分主元消去法的分解过程可描述如下：

- 设 $A^{(i)}$ 为经过 $1 \ldots i-1$ 消元并应用部分主元法以在 (i,i) 位置获得所需元素的列矩阵。

- 设 $\ell^{(i)}$ 为第 i 步消元过程中的乘数向量（即该步消元矩阵 L_i 为单位矩阵加上第 i 列的 $\ell^{(i)}$）。
- 设 $P^{(i+1)}$（其中 $j \geq i+1$）为上述描述中为下一步消元执行部分主元置换的矩阵。
- 于是 $A^{(i+1)} = P^{(i+1)} L_i A^{(i)}$。

由此我们得到如下形式的分解

$$L_{n-1} P^{(n-2)} L_{n-2} \cdots L_1 P^{(1)} A = U.$$

突然无法写出 $A = LU$，转而写成

$$A = P^{(1)} L_1^{-1} \cdots P^{(n-2)} L_{n-1}^{-1} U. \tag{5.7}$$

习题 5.12. 回顾第 6.10 节和 6.8 节可知，从性能角度看，分块算法通常更可取。为何方程 (LU) 中 ‘带交错主元矩阵的分解’ 会对性能造成不利影响?

幸运的是，方程 (5.7) 可被简化：矩阵 P 与 $\tilde{L}$ ‘几乎可交换’。我们通过示例说明：$P^{(2)} L_1 = L_1 P^{(2)}$ 其中 L_1 极接近于 L_1。

$$\begin{pmatrix} 1 & & & \\ & 0 & & 1 \\ & & I & \\ & 1 & & 0 \\ & & & & I \end{pmatrix} \begin{pmatrix} 1 & & \varnothing \\ \vdots & 1 & \\ \ell^{(1)} & & \ddots \\ \vdots & & & 1 \end{pmatrix} = \begin{pmatrix} 1 & & \varnothing & \\ \vdots & 0 & & 1 \\ \tilde{\ell}^{(1)} & & & \\ \vdots & 1 & & 0 \\ \vdots & & & & I \end{pmatrix} = \begin{pmatrix} 1 & & \varnothing \\ \vdots & 1 & \\ \tilde{\ell}^{(1)} & & \ddots \\ \vdots & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & \\ & 0 & & 1 \\ & & I & \\ & 1 & & 0 \\ & & & & I \end{pmatrix}$$

其中 $\ell^{(1)}$ 与 $\ell^{(1)}$相同，只是元素 i 和 $p(i)$ 被交换了位置。现在你可以理解，类似地 $P^{(2)}$ 等也可以被 ‘传递’ 通过 L_1。

最终我们得到

$$P^{(n-2)} \cdots P^{(1)} A = \tilde{L}_1^{-1} \cdots L_{n-1}^{-1} U = \tilde{L} U. \tag{5.8}$$

这意味着我们可以像之前那样构造矩阵 L，只是每次进行主元置换时，需要更新已计算出的 L 的列。

习题 5.13. 若将方程 (5.8) 写作 $PA = LU$，可得 $A = P^{-1} LU$。你能推导出仅用 P 表示 P^{-1} 的简明公式吗? 提示：每个 $P^{(i)}$都是对称矩阵且自逆，参见上题。

练习 5.14 前文中你已了解到，二维边值问题（章节 4.2.3）会生成特定类型的矩阵。我们曾未经证明地指出，这类矩阵无需选主元。现在我们可以正式证明这一点，关键在于其 对角占优属性：

$$\forall_i a_{ii} \geq \sum_{j \neq i} |a_{ij}|.$$

假设矩阵 A 满足 $\forall_{j \neq i} : a_{ij} \leq 0$。

- 证明矩阵是对角占优的当且仅当存在向量 u、 $v \geq 0$ （即每个组件均为非负）使得 $Au = v$。
- 证明在消去一个变量后，对于剩余矩阵 A，仍存在向量 u、 $v \geq 0$ 使得 $Au = v$。
- 现在完成论证：若 A 对称且对角占优，则（部分）选主元是不必要的。

实际上可以证明，对于任何对称正定（*SPD*）矩阵，选主元都是不必要的，而对角占优是比 SPD 性质更强的条件。

5.3.4 求解方程组

既然我们已获得因式分解 $A = LU$，便可利用它来解线性方程组 $Ax = LUx = b$。若引入临时向量 $y = Ux$，可见此过程分为两步：

$$Ly = b, \qquad Ux = z.$$

第一部分 $Ly = b$ 被称为 ‘下三角求解’，因其涉及下三角矩阵 L：

$$\begin{pmatrix} 1 & & & \emptyset \\ \ell_{21} & 1 & & \\ \ell_{31} & \ell_{32} & 1 & \\ \vdots & & \ddots & \\ \ell_{n1} & \ell_{n2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \\ \vdots \\ b_n \end{pmatrix}$$

在第一行中，可见 $y_1 = b_1$。接着在第二行 $\ell_{21}y_1 + y_2 = b_2$，因此 $y_2 = b_2 - \ell_{21}y_1$。可以想象后续过程：在每一个 i 行中，都能根据前 y 个值计算出 y_i：

$$y_i = b_i - \sum_{j<i} \ell_{ij} y_j.$$

由于我们按递增顺序计算 y_i，此过程亦称为前向替换、前向求解或前向扫描。

求解过程的第二阶段 —— 上三角求解、后向替换或后向扫描 —— 通过 $Ux = y$ 计算 x：

$$\begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & \ddots & \vdots \\ \emptyset & & & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

现在观察最后一行，它直接表明 $x_n = u_{nn}^{-1} y_n$。由此，倒数第二行表明 $u_{n-1n-1}x_{n-1} + u_{n-1n}x_n = y_{n-1}$，从而得到 $x_{n-1} = u_{n-1n-1}^{-1}(y_{n-1} - u_{n-1n}x_n)$。一般而言，我们可以计算

$$x_i = u_{ii}^{-1}(y_i - \sum_{j>i} u_{ij} y_j)$$

对于 i 递减的值。

练习 5.15. 在反向扫描过程中，你需要除以数值 u_{ii}。如果其中任何一个为零，则无法进行。将此问题与上述讨论联系起来。

5.3.5 复杂度

在本章开头我们提到，并非所有求解线性方程组的方法都执行相同次数的运算。因此让我们更详细地考察复杂度（参阅附录 15 对复杂度的简要介绍），即运算次数随问题规模变化的函数关系，尤其是在使用 LU 分解求解线性系统时的复杂度。

首先考察在给定分解结果 x 的情况下从 $LUx = b$ 计算数值的过程（‘求解线性方程组’；见第 5.3.4 节）。观察下三角和上三角部分，你会发现需要对所有非对角元素（即满足 ℓ_{ij} 或 u_{ij} 且 $i \neq j$ 的元素）执行乘法运算。此外，上三角求解过程中还涉及对 u_{ii} 元素的除法运算。由于除法运算通常比乘法昂贵得多，实践中会预先计算 $1/u_{ii}$ 的值并存储以供后续使用。

练习 5.16. 观察因式分解算法，论证存储主元的倒数不会增加计算复杂度。

S总结来看，对于一个规模为 $n \times n$ 的系统，你需要执行 n^2 次乘法和大致相同
s数量的加法。这表明，给定一个因式分解后，求解线性系统的复杂度与
c简单的矩阵 - 向量乘法相同，即计算 Ax 给定 A 和 x 的复杂度。

构建 LU 因式分解的复杂度计算稍显复杂。参考第 5.3.1 节的算法。可以看到在第 k 步会发生两件事：乘数的计算和行的更新。需要计算 $n-k$ 个乘数，每个都涉及一次除法。之后，更新过程需要 $(n-k)^2$ 次加法和乘法。如果暂时忽略较少的除法操作，我们发现 LU 因式分解需要 $\sum_{k=1}^{n-1} 2(n-k)^2$ 次运算。如果逆序排列这个求和式中的项，可以得到

$$\#\text{ops} = \sum_{k=1}^{n-1} 2k^2.$$

由于我们可以用积分来近似求和，因此发现结果是 $2/3n^3$ 加上一些低阶项。这比求解线性系统的阶数更高：随着系统规模增大，构建 LU 分解的代价将完全占据主导地位。

当然，算法分析远不止运算计数这么简单。虽然求解线性系统与矩阵 - 向量乘法具有相同的复杂度，但两者的性质截然不同。一个重要区别在于，分解过程与前向 / 回代求解都涉及递归，因此不易并行化。我们将在后续详细讨论这一点。

5.3.6 分块算法

许多线性代数矩阵运算可以基于子块而非基本元素来表述。这类分块有时会自然地从应用特性中产生，例如二维边值问题的情况（见章节 4.2.3）。但更多时候，我们纯粹出于性能考虑对矩阵施加分块结构。

使用 分块矩阵 算法相比传统的标量矩阵视角具有多项优势。例如，它能提升 缓存分块 效率（参见第 6.7 节）；同时也便于在多核架构上调度线性代数算法（第 多核 架构（参见第 7.12 节）。

对于分块算法，我们将矩阵表示为

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & & \vdots \\ A_{M1} & \cdots & A_{MN} \end{pmatrix}$$

其中 M、 N 为分块维度，即以子块为单位的维度表达。通常我们会选择使 $M = N$ 且对角块为方阵的分块方式。

举个简单例子，考虑用分块形式表达的矩阵 - 向量积 $y = Ax$。

$$\begin{pmatrix} Y_1 \\ \vdots \\ Y_M \end{pmatrix} = \begin{pmatrix} A_{11} & \cdots & A_{1M} \\ \vdots & & \vdots \\ A_{M1} & \cdots & A_{MM} \end{pmatrix} \begin{pmatrix} X_1 \\ \vdots \\ X_M \end{pmatrix}$$

为验证分块算法与原有标量算法计算结果的一致性，我们观察组件 Y_{i_k}—— 即第 i 个分块中第 k 个标量分量。首先，

$$Y_i = \sum_j A_{ij} X_j$$

so

$$Y_{i_k} = \Big(\sum_j A_{ij} X_j\Big)_k = \sum_j \big(A_{ij} X_j\big)_k = \sum_j \sum_\ell A_{ij_{k\ell}} X_{j_\ell}$$

即 A 的第 i 个块行中第 k 行与整个 X 的乘积。

更有趣的算法是分块版本的 LU 分解。算法 (5.1) 于是变为

$$\begin{array}{l} \langle LU \text{ factorization} \rangle: \\ \quad \text{for } k = 1, n-1: \\ \quad\quad \text{for } i = k+1 \text{ to } n: \\ \quad\quad\quad A_{ik} \leftarrow A_{ik} A_{kk}^{-1} \\ \quad\quad\quad \text{for } j = k+1 \text{ to } n: \\ \quad\quad\quad\quad A_{ij} \leftarrow A_{ij} - A_{ik} \cdot A_{kj} \end{array} \tag{5.9}$$

与前一个算法的主要区别在于：除以 a_{kk} 的操作被替换为乘以 A_{kk}^{-1}。此外， U 因子现在对角线上将是主元块而非主元元素，因此 U 仅是 ' 分块上三角 ' 而非严格上三角。

习题 5.17. 我们将证明此处的分块算法计算结果与标量算法相同。通过显式考察计算元素来实现这一证明是

繁琐，因此我们采用另一种方法。首先，回顾第 5.3.2 节可知 LU 分解在施加某些归一化条件下是唯一的：若 $A = L_1U_1 = L_2U_2$ 与 L_1 具有单位对角线，则 $L_2, L_1 = L_2, U_1 = U_2$。

接下来，考虑计算 A^{-1} $_{kk}$。证明这可以通过先计算 A_{kk} 的 LU 分解来实现。利用此证明分块 LU 分解可得到严格三角化的 L 和 U 因子。LU 分解的唯一性由此证明该分块算法计算的是标量结果。

实际应用中需注意，这些矩阵块通常仅为概念性存在：矩阵仍以传统的行或列方式存储。三参数 `M,N,LDA` 矩阵描述法（见于 *BLAS*，参考教程手册第 6 节）使得提取子矩阵成为可能。

5.4 稀疏矩阵

在章节 4.2.3 中你已看到，边界值问题（及初边值问题）的离散化可能会产生稀疏矩阵。由于这类矩阵拥有 N^2 个元素但仅有 $O(N)$ 个非零元，若将其存储为二维数组将造成巨大的空间浪费。此外，我们希望能避免对零元素进行操作。

本节我们将探讨稀疏矩阵的高效存储方案，以及采用稀疏存储时常见线性代数运算的表现形式。

5.4.1 稀疏矩阵的存储

为稀疏矩阵寻找精确定义并无意义，但一个操作性定义是：若矩阵中存在足够多的零元素使得专用存储方案可行，则称其为“稀疏”。我们将简要讨论最流行的稀疏矩阵存储方案。由于矩阵不再以简单的二维数组形式存储，使用此类存储方案的算法也需要重新设计。

5.4.1.1 Band storage and diagonal storage

在章节 4.2.2 中，你已见过带状稀疏矩阵的示例。实际上，它们的非零元素恰好分布在若干条次对角线上。对于此类矩阵，可采用专用存储方案。

以一维边值问题（BVP）的矩阵为例（见章节 4.2.2）。其元素分布在三条对角线上：主对角线及其上下第一条副对角线。采用带状存储方式时，我们仅将包含非零元素的带状区域存入内存。此类矩阵最经济的存储方案是将 $2n-2$ 个元素连续存储。但出于多种考虑，通常会牺牲少量存储空间（如图 5.2 所示）。

因此，对于尺寸为 $n \times n$ 且矩阵带宽 p 的矩阵，我们需要一个大小为 $n \times p$ 的矩形数组来存储该矩阵。方程（4.49）的矩阵将被存储为：

$$\left|\begin{array}{c|c|c} \star & 2 & -1 \\ -1 & 2 & -1 \\ \vdots & \vdots & \vdots \\ -1 & 2 & \star \end{array}\right| \tag{5.10}$$

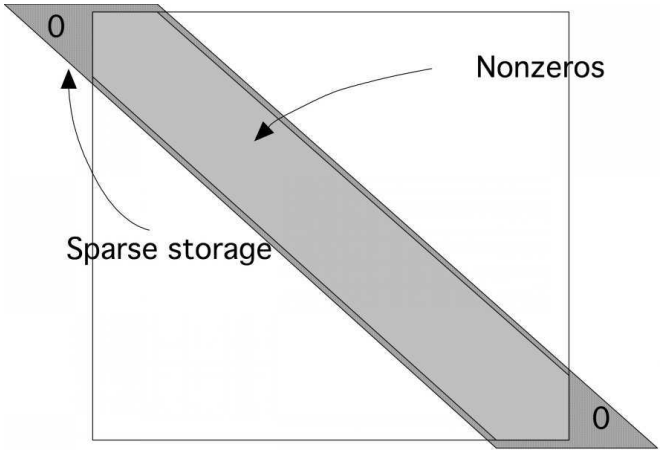


图 5.2：带状矩阵的对角线存储方式。

其中星号表示不对应矩阵元素的数组元素：即图 5.2 中左上和右下角的三角形区域。

当然，现在我们需要考虑数组元素 `A(i,j)` 与矩阵元素 A_{ij} 之间的转换。我们首先在 Fortran 语言中实现，采用列优先存储方式。若通过以下方式分配数组

```
dimension A(n,-1:1)
```

那么主对角线 A_{ii} 存储在 `A(*,0)` 中。例如，`A(1,0)` $\sim A_{11}$。矩阵同一行中的下一个位置 A，`A(1,1)` $\sim A_{12}$。显然，我们可得到以下转换关系

$$\texttt{A(i,j)} \sim A_{i,i+j}. \tag{5.11}$$

备注 19 *BLAS/* 线性代数包（*LAPACK*）库也支持带状存储格式，但采用的是基于列的存储方式，而非我们此处讨论的基于对角线的方式。

习题 5.18. 如何进行逆向转换？即矩阵元素 A_{ij} 对应数组 `A(?,?)` 中的哪个存储位置？

习题 5.19. 若您使用 C 语言编程，请推导矩阵元素 A_{ij} 与数组元素 `A[i][j]` 之间的转换关系。

若将此矩阵存储方案应用于二维边值问题矩阵（见章节 4.2.3）的数组，由于需要存储带状区域内的许多零元素，将造成存储浪费。因此，在对角线存储方案中，我们仅存储非零对角线：若矩阵有条非零对角线，则需要数组。对于方程 (4.56) 的矩阵

这意味着：

$$\begin{vmatrix} \star & \star & 4 & -1 & -1 \\ \vdots & \vdots & 4 & -1 & -1 \\ \vdots & -1 & 4 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & -1 & 4 & \star & \star \end{vmatrix}$$

当然，我们还需要一个额外的整数数组来告诉我们这些非零对角线的位置。

练习 5.20. 对于 $d = 1$、2、3 维空间中的中心差分矩阵，带宽作为 N 的阶数是多少？作为离散化参数 h 的阶数又是多少？

在前面的例子中，矩阵在主对角线上下有相同数量的非零对角线。一般来说，这并不一定成立。为此，我们引入了以下概念：

- 左半带宽：如果 A 的左半带宽为 p，则 $A_{ij} = 0$ 对于 $i > j + p$，且
- 右半带宽：如果 A 的右半带宽为 p，则 $A_{ij} = 0$ 对于 $j > i + p$。

若左半带宽与右半带宽相同，我们简称为半带宽。

5.4.1.2 对角存储操作

在显式时间步进（第 4.3.1.1 节）和线性系统迭代解法（第 5.5 节）等场景中，稀疏矩阵最重要的操作是矩阵 - 向量乘积。本节将探讨如何根据不同存储方案重新设计该操作。

对于上述对角存储的矩阵，仍可通过转换公式（ 5.11 ）执行常规行式或列式乘积 —— 事实上这正是 Lapack 带状矩阵例程的工作原理。但由于带宽较小时向量长度短且循环开销较高，此方法效率低下。我们完全可以实现更优方案。

若观察矩阵元素在矩阵 - 向量乘积中的使用方式，可见主对角线被用作

$$y_i \leftarrow y_i + A_{ii} x_i,$$

第一超对角线被用作

$$y_i \leftarrow y_i + A_{ii+1} x_{i+1} \quad \text{for } i < n,$$

and the first subdiagonal as

$$y_i \leftarrow y_i + A_{ii-1} x_{i-1} \quad \text{for } i > 1.$$

换言之，整个矩阵 - 向量乘积仅需执行三个长度为 n（或 $n-1$）的向量运算即可完成，而非 n 个长度为 3（或 2）的内积运算。

```
for diag = -diag_left 、  diag_right
for loc = max(1,1-diag) 、  min(n,n-diag)
y(loc) = y(loc) + val(loc,diag) * x(loc+diag)endend
```

练习 5.21. 编写一个通过对角线计算 $y \leftarrow A^t x$ 的例程。用你熟悉的语言实现它，并在随机矩阵上进行测试。

练习 5.22. 若矩阵在带内稠密，上述代码片段是高效的。但对于例如二维边值问题的矩阵（参见章节 4.2.3 特别是方程 (4.56)），情况并非如此。编写仅使用非零对角线的对角线法矩阵 - 向量乘积代码。

练习 5.23. 矩阵相乘比矩阵乘向量更复杂。若矩阵 A 的左半带宽为 p_A、右半带宽为 q_Q，矩阵 B 的左半带宽为 p_B、右半带宽为 q_B，则 $C = AB$ 的左、右半带宽分别是多少？假设已为 C 分配足够大小的数组，编写计算 $C \leftarrow AB$ 的例程。

5.4.1.3 压缩行存储

若稀疏矩阵不具备简单带状结构，或非零对角线数量庞大到不切实际时，我们采用更通用的压缩行存储（CRS）方案。顾名思义，该方案通过压缩所有行来消除零元素，如图 5.3 所示。由于这会丢失非零元素原始列位置信息，我们需要额外存储这些数据

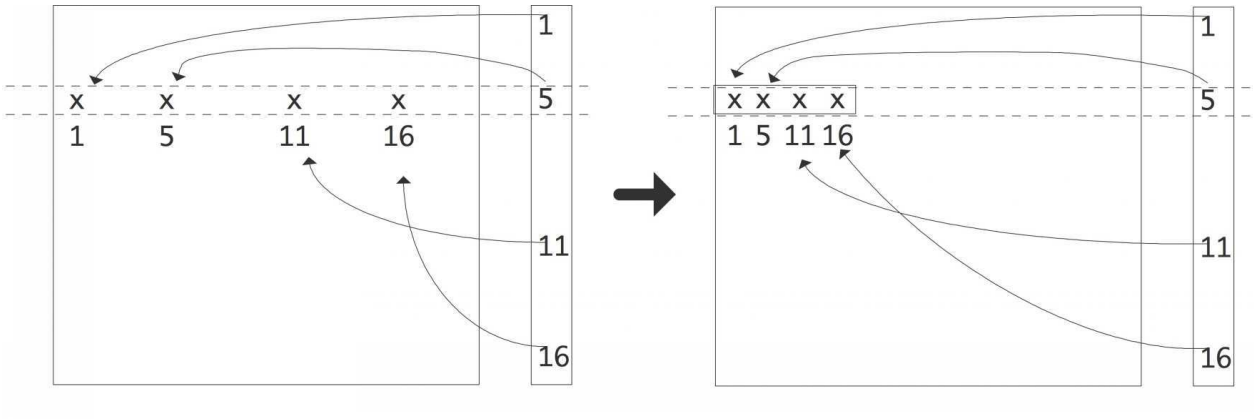


图 5.3：CRS 格式下稀疏矩阵单行压缩示意图

具体来看一个稀疏矩阵的例子：

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}. \tag{5.12}$$

压缩所有行后，我们将所有非零元素存储在一个实数数组中。列索引同样存储在一个整数数组中，并保存指向列起始位置的指针。使用基于 0 的索引，结果如下：

val	10	−2	3	9	3	7	8	7	3 ··· 9	13	4	2	−1
colind	0	4	0	1	5	1	2	3	0 ··· 4	5	1	4	5
rowptr	0	2	5	8	12	16	19						

CRS 的一种简单变体是压缩列存储（*CCS*），其中列中的元素被连续存储。这也被称为 *Harwell-Boeing* 矩阵格式 [54]。您可能遇到的另一种存储方案是坐标存储，其中矩阵被存储为三元组列表 $\langle i, j, a_{ij} \rangle$。流行的 *Matrix Market* 网站 [151] 使用了这种方案的一个变体。

5.4.1.4 压缩行存储上的算法

在本节中，我们将探讨一些算法在 CRS 中的形式。

首先我们考虑稀疏矩阵 - 向量积的实现。

```
for (row=0; row<nrows; row++) {
      s = 0;
      for (icol=ptr[row]; icol<ptr[row+1]; icol++) {
         int col = ind[icol];
         s += a[icol] * x[col];
      }
      y[row] = s;
   }
```

您能识别出针对 $y = Ax$ 的标准矩阵 - 向量乘积算法，其中对每行 A_{i*} 与输入向量 x 取内积。然而，内层循环不再以列号作为索引，而是以该数字所在位置为索引。这一额外步骤被称为间接寻址。

习题 5.24. 比较按行执行的稠密矩阵 - 向量乘积与前述稀疏乘积的数据局部性。证明对于一般稀疏矩阵，寻址输入向量 x 的空间局部性现已消失。是否存在仍能保留部分空间局部性的矩阵结构?

现在，假设您需要计算乘积 $y = A^t x$会怎样? 这种情况下您需要 A^t的行，或等价地 A 的列。寻找 A 的任意列非常困难，需要大量搜索，因此您可能认为该算法相应地难以计算。幸运的是，事实并非如此。

若在标准算法中交换 i 和 j 循环以计算 $y = Ax$，我们将得到

Original:

$y \leftarrow 0$
for i:
  for j:
    $y_i \leftarrow y_i + a_{ij}x_j$

Indices reversed:

$y \leftarrow 0$
for j:
  for i:
    $y_i \leftarrow y_i + a_{ji}x_j$

可见在第二种变体中，访问的是 A 的列而非行。这意味着我们可以使用第二种算法按行计算 $A^t x$ 乘积。

练习 5.25. 写出转置乘积 $y = A^t x$ 的代码，其中 A 以 CRS 格式存储。编写一个简单的测试程序并验证代码计算结果正确。

练习 5.26. 若需同时访问行和列该怎么办？实现一个算法来测试 CRS 格式存储的矩阵是否对称。提示：为每行维护一个指针数组，跟踪该行的处理进度。

练习 5.27. 目前描述的操作相对简单，因为它们从不改变矩阵的稀疏结构。上述 CRS 格式不允许添加新的非零元素，但扩展该格式以实现此功能并不困难。

> 设数值 $p_i i = 1 \ldots n$，描述第 i 行中非零元的数量，已知。
> 设计一种 CRS 的扩展方案，为每行预留 q 个额外元素的空间。实现该方案并进行测试：构建一个在第 i 行有 p_i 个非零元的矩阵，并在添加新元素（每行最多 q 个）前后验证矩阵 - 向量乘积的正确性。
> 现假设矩阵的非零元总数永远不会超过 qn 个。修改
> 你的代码，使其能够从空矩阵开始，逐步在随机位置添加非零元。再次验证正确性。

我们将在 7.5.5 节的共享内存并行上下文中重新讨论转置乘积算法。

5.4.1.5 Ellpack 存储变体

前文已介绍对角线存储（生成长向量但仅适用于确定带宽的矩阵）和压缩行存储（通用性更强但具有不同的向量 / 并行特性）。具体而言，在 CRS 中，唯一的向量操作是单行与输入向量的归约运算。这类运算通常较短且涉及间接寻址，因此会因双重原因导致效率低下。

练习 5.28. 论证若将矩阵 - 向量乘积视为并行操作则不存在效率问题。

向量运算具有吸引力 —— 不仅对 GPU 如此，随着现代处理器向量位宽的提升（例如现代英特尔处理器的指令集）更是如此。那么是否存在一种方法能兼顾 CRS 的灵活性与对角线存储的高效性?

此处采用的方法是使用 *Ellpack* 存储的一种变体。该方式通过锯齿对角线存储矩阵：第一条锯齿对角线由每行最左侧元素组成，第二条对角线由次左侧元素组成，以此类推。此时我们可以按对角线进行乘积运算，但需注意输入数据需要间接寻址。

该方案的第二个问题是最后若干条对角线不会完全填满。目前至少有两种解决方案被提出：

- 可按行长度对行进行排序。由于不同行长度的数量可能较少，这意味着我们只需对这些长度进行短循环处理，同时围绕对角线执行长向量化循环。在向量架构上，这种方法能带来巨大的性能提升 [39]。
- 使用向量指令或在 GPU 上运行时，仅需少量规则性处理，因此可选取 8 或 32 行的数据块，并用零值进行 ‘填充’。

5.4.2 稀疏矩阵与图论 y

关于稀疏矩阵的许多论证可以用图论的语言来表述。为了理解这种表述的可行性，考虑一个大小为 A 的矩阵 n，并注意到我们可以定义一个图 $\langle E,V\rangle$ 通过 $V=\{1,\dots,n\}$,$E=\{(i,j):a_{ij}\neq 0\}$。这被称为矩阵的邻接图。为简化起见，我们假设 A 具有非零对角线。如有必要，可以为该图附加权重，定义为 $w_{ij}=a_{ij}$。此时该图记为 $\langle E,V,W\rangle$。（若您不熟悉图论基础知识，请参阅附录 20。）

图的属性现在对应矩阵的属性；例如，图的度是每行非零元素的最大数量（不计对角元素）。再举一例，若矩阵对应的图是无向图，则意味着 $a_{ij}\neq 0\Leftrightarrow a_{ji}\neq 0$。我们称此类矩阵为结构对称：它并非真正意义上的对称（即 $\forall_{ij}:a_{ij}=a_{ij}$），但其上三角部分的每个非零元素都对应下三角部分的一个非零元素，反之亦然。

5.4.2.1 置换下的图性质

考虑矩阵的图结构有一个优势，即图的性质不依赖于节点的排序方式，也就是说，它们在矩阵置换下保持不变。

练习 5.29. 让我们看看当矩阵 A 的图节点 $G=\langle V、 E、 W\rangle$ 被重新编号时会发生什么。
举个简单的例子，我们将节点反向编号；即，设 n 为节点总数，将节点 i 映射到 $n+1-i$。
相应地，我们得到一个新的图 $G'=\langle V、 E'、 W'\rangle$，其中

$$(i,j)\in E'\Leftrightarrow(n+1-i,n+1-j)\in E,\qquad w'_{ij}=w_{n+1-i,n+1-j}.$$

这种重新编号对与 G' 对应的矩阵 A' 意味着什么？如果你交换两个节点上的标签 i、 j，对矩阵 A 会产生什么影响？

练习 5.30. 某些矩阵性质在置换下保持不变。请验证置换不会改变矩阵的特征值。

某些图的性质可能难以从矩阵的稀疏模式中直接观察，但通过图结构则更易推导得出。

习题 5.31. 设 A 为一维边值问题（参见章节 4.2.2 ）的尺寸为 n 且 n 为奇数的三对角矩阵。该矩阵对应的图结构呈现何种形态？考虑将节点按以下序列排列所产生的置换：

$$1, 3, 5, \ldots, n, 2, 4, 6, \ldots, n-1.$$

置换后的矩阵稀疏模式呈现何种形态？此类重编号策略将在章节 7.8.2 中详细讨论。现在将该矩阵靠近 “ 中部 ” 的次对角线元素置零：设

$$a_{(n+1)/2,(n+1)/2+1} = a_{(n+1)/2+1,(n+1)/2} = 0.$$

描述此操作对 A 对应图结构的影响。此类图被称为可约图。接着应用前一习题的置换操作并绘制所得稀疏模式。注意此时从稀疏模式中更难辨识图的可约性。

5.4.3 LU factorizations of sparse matrices

在章节 4.2.2 中，一维边值问题导出了具有三对角系数矩阵的线性系统。若进行一步高斯消元，唯一需要消去的元素位于第二行：

$$\begin{pmatrix} 2 & -1 & 0 & \ldots & \\ -1 & 2 & -1 & & \\ 0 & -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots & \ddots \end{pmatrix} \Rightarrow \left(\begin{array}{c|cccc} 2 & -1 & 0 & \ldots & \\ 0 & 2-\frac{1}{2} & -1 & & \\ 0 & -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots & \ddots \end{array}\right)$$

有两个重要的观察点需要指出：一是该消元步骤不会将任何零元素变为非零元素。另一个观察是，矩阵剩余待消元部分仍保持三对角形态。通过归纳可知，消元过程中零元素不会转为非零： $L+U$ 的稀疏模式与 A 相同，因此因式分解所需的存储空间与原矩阵相同。

遗憾的是，三对角矩阵的情况并不典型，我们将在二维问题案例中很快看到这一点。但首先我们将把第 5.4.2 节的图论讨论延伸至因式分解领域。

5.4.3.1 稀疏 *LU* 分解的图论原理

在讨论稀疏矩阵的 LU 分解时，图论往往非常有用。让我们从图论角度探究消除第一个未知数（或扫描第一列）的含义。此处我们假设矩阵具有结构对称性。

我们将消元一个未知量的过程视为：将图 $G = \langle V, E\rangle$ 转化为图 $G' = \langle V', E'\rangle$的操作。这些图之间的首要关系是某个顶点（假设为 k ）已从顶点集 $k \notin V', V' \cup \{k\} = V$中被移除。

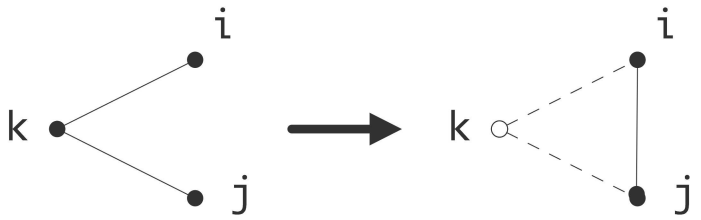


图 5.4：消除顶点会在商图中引入一条新边。

E 与 E' 之间的关系更为复杂。在高斯消元算法中，消除变量 k 的结果是执行语句

$$a_{ij} \leftarrow a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}$$

对所有 i，$j \neq k$ 执行。如果 $a_{ij} \neq 0$ 最初存在，即 $(i,j) \in E$，那么 a_{ij} 的值仅被修改，这不会改变邻接图。若原矩阵中 $a_{ij} = 0$，意味着 $(i,j) \notin E$，则在消除 k 未知数后会生成一个非零元素，称为**填充元**，其位置为 [159]：

$$(i,j) \notin E \quad \text{but} \quad (i,j) \in E'.$$

如图 5.4 所示。

总结而言，消除一个未知数会得到一个顶点数减一的图，并且对于所有满足 i、j 与已消除变量 k 之间存在边的 i 或 j，都会新增相应的边。

图 5.5 展示了小型矩阵的完整示意图。

练习 5.32. 回到练习 5.31。使用图论论证确定奇数变量被消除后的稀疏模式。

练习 5.33. 证明上述关于消除单个顶点的论证可推广。设 $I \subset V$ 为任意顶点集，J 是与 I 相连的顶点：

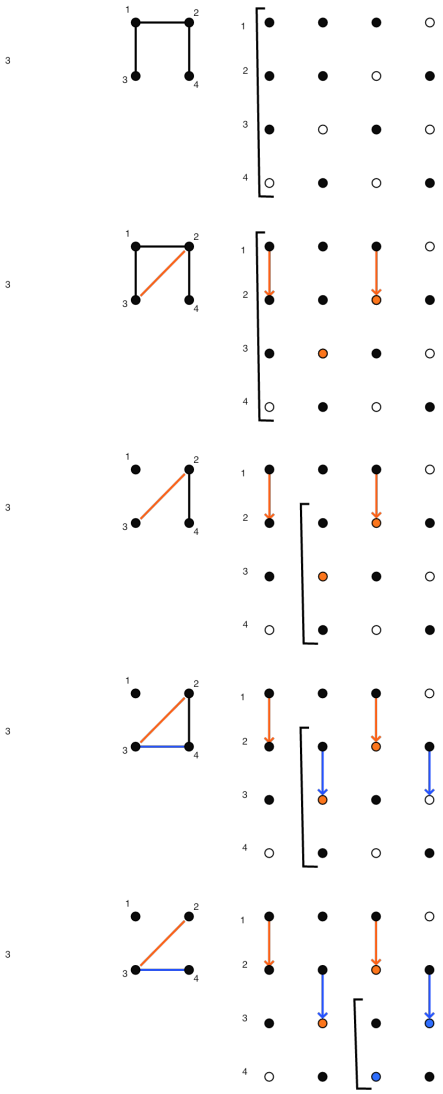
$$J \cap I = \emptyset, \quad \forall_{i \in I} \exists_{j \in J} : (i,j) \in E.$$

现在证明消除 I 中的变量会导致图 $\langle V', E' \rangle$ 中所有 J 节点在剩余图中相互连接（若它们原先通过 I 存在路径）：

$$\forall_{j_1, j_2 \in J} : \text{there is a path } j_i \to j_2 \text{ through } I \text{ in } E \Rightarrow (j_1, j_2) \in E'.$$

5.4.3.2 填充

现在我们回到二维问题矩阵的因式分解。我们将大小为 $N \times N$ 的此类矩阵写作块维度为 n 的分块矩阵，每块大小为 n。（复习问题：这些分块从何而来？）在第一步消元中，我们需要将两个元素 a_{21} 和 $a_{n+1,1}$ 归零。



Original matrix.

Eliminating $(2, 1)$ causes fill-in at $(2, 3)$.

Remaining matrix when step 1 finished.

Eliminating $(3, 2)$ fills $(3, 4)$

After step 2

Figure 5.5: Gaussian elimination of a sparse matrix.

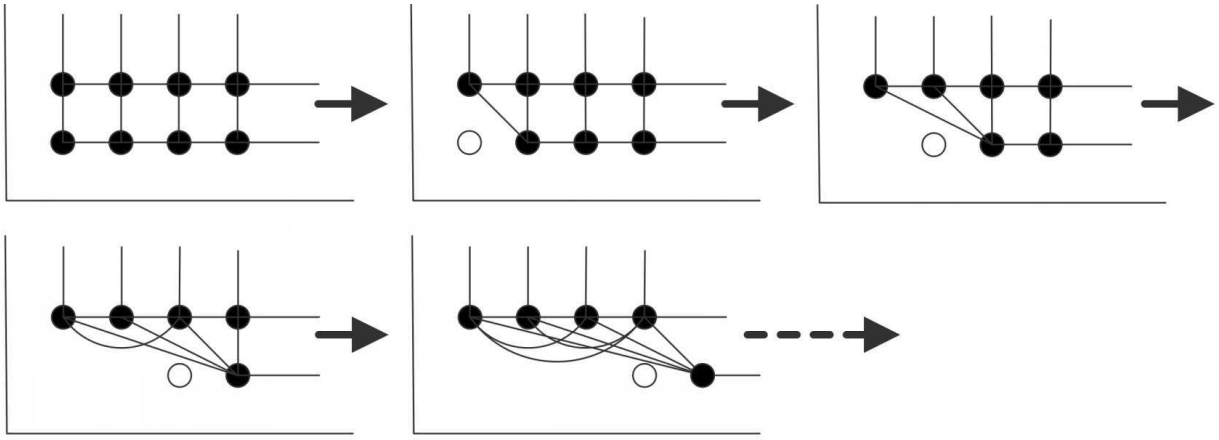


图 5.6：矩阵图中填充连接的创建过程。

$$\left(\begin{array}{ccccc|ccc} 4 & -1 & 0 & \dots & & -1 & & \\ -1 & 4 & -1 & 0 & \dots & 0 & -1 & \\ & \ddots & \ddots & \ddots & & & \ddots & \\ -1 & 0 & \dots & & & 4 & -1 & \\ 0 & -1 & 0 & \dots & & -1 & 4 & -1 \end{array}\right) \Rightarrow \left(\begin{array}{c|cccc|ccc} 4 & -1 & 0 & \dots & & -1 & & \\ & 4-\frac{1}{4} & -1 & 0 & \dots & -1/4 & -1 & \\ & \ddots & \ddots & \ddots & & & \ddots & \ddots \\ & -1/4 & & & & 4-\frac{1}{4} & -1 & \\ & -1 & 0 & & & -1 & 4 & -1 \end{array}\right)$$

可见，消去 a_{21} 和 $a_{n+1,1}$ 会导致两个填充元素出现：在原矩阵中 $a_{2,n+1}$ 和 $a_{n+1,2}$ 处为零，但在修改后的矩阵中这些位置变为非零。我们将填充位置定义为 (i,j) 处，其中 $a_{ij}=0$，但 $(L+U)_{ij}\neq 0$。

显然，矩阵在分解过程中会发生填充。稍加想象还可发现，带状区域中第一对角块外的每个元素都将被填充。不过，借助 5.4.3.1 节的图论方法，可以轻松可视化所创建的填充连接。

图 5.6 以二维边值问题的图为例展示了这一过程。（此处未绘制对角元素对应的边。）第一行中每个被消去的变量会在下一变量与第二行之间、以及第二行变量之间建立连接。通过归纳可知，第一行消去后第二行将完全连通。（将此与习题 5.33 关联。）

Exercise5.34. 完成论证。第二行变量完全连接这一事实对矩阵结构意味着什么？用图示说明第二行第一个变量被消除后会发生什么。

Exercise5.35. LAPACK 稠密线性代数软件中的 LU 分解例程会直接用因子覆盖输入矩阵。前文已说明这种操作可行的原因 —— L 的列正是在 A 的列被消除时生成的。若矩阵以稀疏格式存储，为何无法实现此类算法？

5.4.3.3 填充量估计

在上述示例中，您已经看到稀疏矩阵的分解可能比矩阵本身占用更多空间，但仍比存储整个矩阵维度的方形数组要少。现在我们将给出分解的空间复杂度的一些界限，即执行分解算法所需的空间量。

练习 5.36. 证明以下命题。

1. 假设矩阵 A 具有半带宽 p，即 $a_{ij}=0$ 如果 $|i-j|>p$。证明在不进行旋转的情况下分解后，$L+U$ 具有相同的半带宽。2. 证明在部分旋转的情况下分解后，L 的左半带宽为 p，而 U 的右半带宽为 $2p$。3. 假设不进行旋转，证明填充量可以如下表征：考虑行 i。设 $j\min$ 为行 i 中最左边的非零元素，即 $a_{ij}=0$ 对于 $j<j\min$。那么在行 i 中，列 $j\min$ 的左侧不会有填充。同样，如果 $i\min$ 是列 j 中最顶部的非零元素，那么在列 j 中，行 $i\min$ 的上方不会有填充。因此，L 和 U 具有“天际线”轮廓。给定一个稀疏矩阵，现在很容易分配足够的存储空间以适应不进行旋转的分解：这被称为天际线存储。

Exercise5.37. 考虑矩阵

$$A=\begin{pmatrix} a_{11} & 0 & a_{13} & & & & & \emptyset \\ 0 & a_{22} & 0 & a_{24} & & & & \\ a_{31} & 0 & a_{33} & 0 & a_{35} & & & \\ & a_{42} & 0 & a_{44} & 0 & a_{46} & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ \emptyset & & & & & a_{n,n-1} & 0 & a_{nn} \end{pmatrix}$$

你之前已经证明，进行 LU 分解时产生的任何填充元仅限于包含原始矩阵元素的带状区域内。本例中不存在填充元。请用归纳法证明这一点。观察邻接图（这类图有个特定名称，是什么？）。能否基于该图给出不存在填充元的证明?

Exercise 5.36 表明我们可以为带状矩阵的分解分配足够的存储空间：

- 对于带宽为 p 的矩阵进行无主元分解时，大小为 $N\times p$ 的数组即可满足存储需求；
- 对左半带宽 p 、右半带宽 q 的矩阵进行部分主元分解时，其存储空间可容纳于 $N\times(p+2q+1)$。
- 可根据具体矩阵构建天际线轮廓（ skyline profile ），该结构足以存储分解结果。

我们可以将此估计应用于二维边值问题中的矩阵，章节 4.2.3。

习题 5.38. 证明在方程 (4.56) 中，原始矩阵有 $O(N)=O(n^2)$ 个非零元素，$O(N^2)=O(n^4)$ 个元素总数，而分解后有 $O(nN)=O(n^3)=O(N^{3/2})$ 个非零元素。

这些估算表明，LU 分解所需的存储空间可能超过 A 所需，且差异并非恒定比例，而是与矩阵尺寸相关。无需证明即可断言：截至目前所见稀疏矩阵的逆矩阵均为完全稠密，因此存储它们需要更多空间。这是实践中不通过计算 A^{-1} 再乘以 $x = A^{-1}y$来求解线性系统 $Ax = y$ 的重要原因（数值稳定性是另一个原因）。甚至分解过程本身也可能占用大量空间，这成为考虑迭代方法（如第 5.5 节所述）的关键因素。

上文提到，一个大小为 $n \times n$ 的稠密矩阵分解需要 $O(n^3)$次运算。那么对于稀疏矩阵又是怎样的情形呢？让我们考虑一个半带宽为 p的矩阵，并假设原始矩阵在该带宽内是稠密的。主元 a_{11} 用于消去第一列中的 p 个元素，且每次操作需将首行加到目标行上，涉及 p 次乘法和加法。总体而言，我们发现运算次数大约为

$$\sum_{i=1}^{n} p^2 = p^2 \cdot n$$

加减低阶项。

习题 5.39. 初始密集带的假设对于二维边值问题的矩阵并不成立。为何上述估计仍能成立（忽略某些低阶项）？

在习题 5.36 中，您推导了一个易于应用的填充量估计方法。然而，这可能存在显著高估。理想情况是以少于实际分解的计算量来估算填充量。下面我们将概述一种算法，用于精确计算 $L+U$中的非零元数量，其计算成本与该数量呈线性关系。我们将在（结构）对称情形下进行说明。关键观察如下：假设第 i 列在对角线下有多个非零元：

$$\begin{pmatrix} \ddots & & & & \\ & a_{ii} & & a_{ij} & & a_{ik} \\ & & \ddots & & & \\ & a_{ji} & & a_{jj} & & \\ & & & & \ddots & \\ & a_{ki} & & ?a_{kj}? & & a_{kk} \end{pmatrix}$$

在第 i步消去 a_{ki} 会导致 a_{kj}的更新，若原位置 $a_{kj} = 0$ 为空则产生填充元。但我们可以推断该非零值的存在：消去 a_{ji} 会在位置 (j,k) 产生填充元，且已知结构对称性得以保持。换言之，若仅统计非零元数量，只需观察消去 (j,i) 位置（或枢轴下方首个非零元）产生的影响。由此推演，我们仅需记录每个枢轴对应一行的非零元，整个过程的时间复杂度与分解中的非零元数量呈线性关系。

5.4.3.4 填充减少

矩阵的图属性（如度和直径）在变量重新编号下保持不变。而其他属性（如因式分解过程中的填充）则会受到重新编号的影响。事实上，这是值得的

研究是否可能通过重新编号矩阵图的节点（或等价地，对线性系统应用置换）来减少填充量。

练习 5.40. 考虑仅在首行、首列及对角线上有非零元素的 ‘箭头’ 矩阵：

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & & \emptyset \\ \vdots & & \ddots & \\ * & \emptyset & & * \end{pmatrix}$$

假设加法运算永远不会产生零值，该矩阵及其分解中的非零元素数量是多少？能否找到问题变量的对称置换，使得新矩阵无填充?

此示例并不典型，但确实可以通过巧妙的矩阵置换（例如参见章节 7.8.1 ）有时改善填充估计。即便如此，通常而言，稀疏矩阵的 LU 分解所需存储空间远大于矩阵本身。这是下一节迭代方法的主要动机之一。

5.4.3.5 填充减少排序

某些矩阵性质在对称置换下保持不变 .

习题 5.41. 在线性代数课程中，通常会考察矩阵性质及其在基变换下的不变性，特别是酉基变换下的表现：

$$B = VAV^t, \quad \text{where } VV^t = I.$$

证明对称置换是一种特殊的基变换。列举若干在酉变换下保持不变的矩阵性质。

其他性质则不然：前一节已说明填充量就是其中之一。因此，人们可能想知道何种排序能最优减少给定矩阵分解时的填充量。该问题在实际中难以处理，但存在多种启发式方法。其中部分方法还可从并行计算角度进行论证；事实上，嵌套剖分排序将仅在讨论并行计算的 7.8.1 章节中展开。此处我们简要介绍两种早于并行需求出现的启发式方法。

首先考察 *Cuthill-McKee* 排序，该方法直接最小化置换后矩阵的带宽。由于填充量可通过带宽界定，我们期望这种带宽缩减排序也能减少填充量。

其次，我们将考虑最小度排序方法，该方法更直接地以减少填充量为目标。

5.4.3.5.1 Cuthill-McKee 排序法

该 *Cuthill-McKee* 排序法 [37] 是一种 带宽缩减排序法 ，其工作原理是通过对 层级集合 进行排序（参见图 5.7 ）。该方法基于邻接图 ，具体步骤如下：

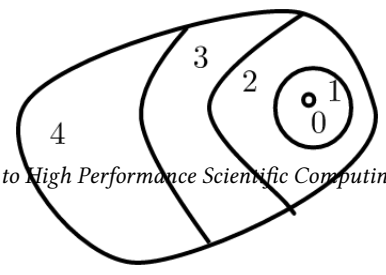


图 5.7：层级集合示意图。

1. 选取任意节点，将其称为 “ 零级 ”。2. 给定层级 n，将所有连接到层级 n 且尚未分配层级的节点分配到层级 $n+1$。3. 对于所谓的 ‘ 反向 Cuthill-McKee 排序 ’，反转层级的编号顺序。

练习 5.42. 证明根据 Cuthill-McKee 排序对矩阵进行置换后，会呈现块三对角结构。

我们将在 7.10.1 节讨论并行性时重新审视该算法。

当然，人们可能会好奇带宽究竟能缩减到什么程度。

练习 5.43. 图的直径定义为两个节点之间最短路径的最大值。

1. 论证在二维椭圆问题的图中，该直径为 $O(N^{1/2})$。2. 用带宽表示节点 1 与 N 之间的路径长度。3. 论证这给出了直径的下界，并利用此推导出带宽的下界。

5.4.3.5.2 **最小度排序** 另一种排序方法的动机源于观察到填充量与节点的度数相关。

习题 5.44. 证明消除一个度数为 d 的节点最多会产生 $2d$ 个填充元素

所谓的最小度排序 按以下步骤进行：

- 找到度数最低的节点；
- 消除该节点并更新剩余节点的度信息；
- 从第一步开始重复，使用更新后的矩阵图。

练习 5.45. 指出上述两种方法之间的一个区别。两者都基于对矩阵图的检查；然而，最小度方法在所使用的数据结构上需要更大的灵活性。解释原因并详细讨论两个方面。

5.5 迭代方法

高斯消元法，即使用 LU 分解，是求解线性系统的一种简单方法，但正如我们上面所看到的，在离散化偏微分方程产生的问题中，它可能会产生大量填充。在本节中，我们将探讨一种完全不同的方法 —— 迭代解法，即通过一系列近似来找到系统的解。

计算方案大致如下：

$$\begin{cases} \text{任选初始向量 } x_0\text{，并重复以下步骤 } i \geq 0\text{：} \\ x_{i+1} = Bx_i + c \\ \text{直至满足某个停止条件。} \end{cases}$$

此处的重要特性在于，无需使用原始系数矩阵求解任何系统；相反，每次迭代仅涉及矩阵 - 向量乘法或更简单系统的求解。因此，我们用一个重复的、更简单且成本更低的操作替代了复杂操作 —— 构建 LU 分解并用其求解系统。这使得迭代方法更易于编码，并可能更高效。

让我们通过一个简单示例来引出迭代方法的精确定义。假设需要求解方程组

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

其解为 $(2,1,1)$。假设已知（例如根据物理特性）解的各组件大小大致相当。观察对角线的主导性后可判定

$$\begin{pmatrix} 10 & & \\ & 7 & \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

可能是个良好的近似解。该方程组的解为 $(2.1, 9/7, 8/6)$。显然，仅涉及原系统对角线的方程组既易于求解，且至少在本例中具有相当高的准确性。

对原系统的另一种近似方法是采用下三角矩阵。该系统

$$\begin{pmatrix} 10 & & \\ 1/2 & 7 & \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

的解为 $(2.1, 7.95/7, 5.9/6)$。求解三角系统比对角系统稍复杂，但仍比计算 LU 分解容易得多。此外，在寻找此近似解的过程中我们未产生任何填充元。

由此可见，存在易于计算的方法能相当接近真实解。我们能否以某种方式重复这一技巧?

稍微抽象地表述，我们所做的不是求解 $Ax = b$ 而是求解 $Lx = b$。现定义 Δx 为真实解的偏差距离：$x = x + \Delta x$。由此得到 $A\Delta x = Ax - b \equiv r$，其中 r 是残差。接着我们再次求解 $L\widetilde{\Delta x} = \tilde{r}$ 并更新 $x = x - \widetilde{\Delta x}$。

迭代	1	2	3
x_1	2.1000	2.0017	2.000028
x_2	1.1357	1.0023	1.000038
x_3	0.9833	0.9997	0.999995

在此情况下，我们每次迭代获得两位小数精度的提升，这并非典型情况。

现在可以清楚地理解为何迭代方法具有吸引力。通过高斯消元法求解方程组需要 $O(n^3)$ 次运算，如前所述。而采用上述方案时，单次迭代仅需 $O(n^2)$ 次运算（若满足条件）。

矩阵是稠密的，对于稀疏矩阵可能低至 $O(n)$。如果迭代次数较少，这使得迭代方法具有竞争力。

练习 5.46. 在比较迭代方法和直接方法时，浮点运算次数并非唯一相关指标。概述与两种情况下代码效率相关的一些问题。同时比较求解单个线性系统和多个线性系统的情况。

5.5.1 抽象表述

现在需要对上述示例的迭代方案进行正式表述。假设我们需要求解 $Ax = b$，而直接求解成本过高，但乘以 A 是可行的。进一步假设我们有一个矩阵 $K \approx A$，使得求解 $Kx = b$ 可以低成本完成。

我们不直接求解 $Ax = b$，而是求解 $Kx = b$，并将 x_0 定义为解：$Kx_0 = b$。这会留下一个误差 $e_0 = x_0 - x$，对此我们有方程 $A(x_0 - e_0) = b$ 或 $Ae_0 = Ax_0 - b$。我们称 $r_0 \equiv Ax_0 - b$ 为残差；误差则满足 $Ae_0 = r_0$。

若能求解方程 $Ae_0 = r_0$ 中的误差，问题便迎刃而解：真实解随即通过 $x = x_0 - e_0$ 得出。然而，由于最初采用 A 求解成本过高，此次同样无法实现，因此我们采用近似方法确定误差修正量。通过求解 $Ke_0 = r_0$ 并设定 $x_1 := x_0 - e_0$；后续流程可继续基于 $e_1 = x_1 - x$、$r_1 = Ax_1 - b$、$Ke_1 = r_1$、$x_2 = x_1 - e_1$ 等参数展开。

迭代方案如下：

> 给定 x_0，对于 $i \geq 0$：令
> $r_i = Ax_i - b$ 根据 $Ke_i = r_i$ 计算出 e_i，更新 $x_{i+1} = x_i - e_i$

我们称该基础方案为

$$x_{i+1} = x_i - K^{-1}r_i \tag{5.13}$$

stationary iteration（静态迭代）。其静态性源于每次更新方式相同，与迭代次数无关。该方案分析简单，但适用性有限。

关于迭代方案，我们需要回答以下几个问题：

- 该方案是否总能引导我们找到解?
- 若方案收敛，其收敛速度如何?
- 何时停止迭代?
- 如何选择 K?

我们现在将对这些事项给予一定关注，尽管完整讨论已超出本书范围。

5.5.2 收敛性与误差分析

我们首先探讨迭代方案是否收敛以及收敛速度的问题。考虑单次迭代步骤：

$$\begin{aligned} r_1 &= Ax_1 - b = A(x_0 - \tilde{e}_0) - b \\ &= r_0 - AK^{-1}r_0 \\ &= (I - AK^{-1})r_0 \end{aligned} \tag{5.14}$$

通过归纳可得 $r_n = (I - AK^{-1})^n r_0$，因此 $r_n \downarrow 0$ 若所有特征值满足 $|\lambda(I - AK^{-1})| < 1$[1]。

最后这个结论通过关联 K 与 A，既给出了收敛条件，也提供了当 K 足够接近时的几何收敛速率。

习题 5.47. 推导 e_n 的类似归纳关系式。

很难通过计算实际特征值来判断条件 $|\lambda(I - AK^{-1})| < 1$ 是否满足。然而，有时 *Gershgorin* 定理（附录 14.5 ）能提供足够的信息。

习题 5.48. 考虑矩阵 A，其来自方程（ 4.56 ）中对二维边值问题的离散化。设 K 为包含 A 对角线的矩阵，即 $k_{ii} = a_{ii}$ 且 $k_{ij} = 0$ 对于 $i \neq j$。利用 Gershgorin 定理证明 $|\lambda(I - AK^{-1})| < 1$。

本习题中的论证难以推广到更复杂的 K 选择，如下文所示。这里我们仅指出，对于某些矩阵 A，这些 K 选择总能保证收敛，且收敛速度随矩阵规模增大而降低。我们不会深入细节，仅说明对于 M 矩阵（参见第 4.2.2 节），这些迭代方法收敛。关于静态迭代方法收敛理论的更多细节，请参阅 [186]

5.5.3 计算形式

上文第 5.5.1 节中，我们将稳态迭代推导为涉及乘以 A 并通过 K 求解的过程。但在某些情况下，可能有更简单的实现方式。考虑当 $A = K - N$，且我们已知 K 和 N 时，可将 $Ax = b$ 表示为

$$Kx = Nx + b \tag{5.15}$$

我们观察到满足 (x)5.15 式的是迭代的不动点 n

$$Kx^{(n+1)} = Nx^{(i)} + b.$$

显然这是一个稳态迭代过程：

$$\begin{aligned} Kx^{(n+1)} &= Nx^{(i)} + b \\ &= Kx^{(n)} - Ax^{(n)} + b \\ &= Kx^{(n)} - r^{(n)} \\ \Rightarrow x^{(n+1)} &= x^{(n)} - K^{-1}r^{(n)}. \end{aligned}$$

1. 当矩阵可对角化且具有完备特征向量基时，这一点相当明显。但这一结论在一般情况下同样成立。

这是方程的基本形式（ 5.13 ）。收敛准则 $|\lambda(I - AK^{-1})| < 1$ （见上文）现在简化为 $|\lambda(NK^{-1})| < 1$。

让我们考虑一些特殊情况。首先，设 $K = D_A$，即包含 A 对角部分的矩阵： $k_{ii} = a_{ii}$ 且 $k_{ij} = 0$ 对所有 $i \neq j$ 成立。同理， $n_{ii} = 0$ 且 $n_{ij} = -a_{ij}$ 对所有 $i \neq j$ 成立。
这被称为雅可比迭代。矩阵形式 $Kx^{(n+1)} = Nx^{(n)} + b$ 可以逐点表示为

$$\forall_i : a_{ii}x_i^{(t+1)} = \sum_{j \neq i} a_{ij}x_j^{(t)} + b_i \tag{5.16}$$

其变为

for $t = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $x_i^{(t+1)} = a_{ii}^{-1}(\sum_{j \neq i} a_{ij}x_j^{(t)} + b_i)$

（鉴于成本高昂，参见第 1.2 节，我们实际上会显式存储 a_{ii}^{-1} 量，并用乘法替代除法。）

这要求我们拥有一个向量 x 用于当前迭代 $x^{(t)}$，以及一个临时向量 u 用于下一个向量 $x^{(t+1)}$。
最简单的写法是：

for $t = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $u_i = a_{ii}^{-1}(-\sum_{j \neq i} a_{ij}x_j + b_i)$
  copy $x \leftarrow u$

对于一维问题的简单情况，如图 5.8 所示：在每个 x_i 点上的值

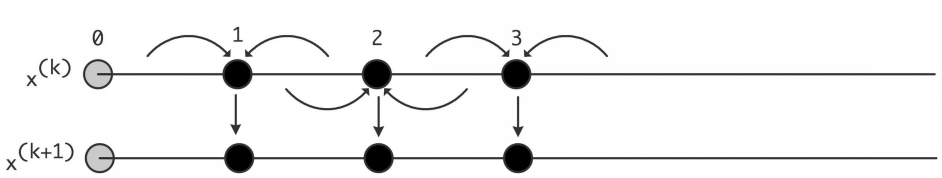


图 5.8：一维问题上雅可比迭代法的数据移动模式。

两个相邻点的值与当前值结合生成新值。由于所有 x_i 点上的计算相互独立，可在并行计算机上并行完成。
但你可能认为，在求和 $\sum_{j \neq i} a_{ij}x_j$ 时为何不使用已计算出的 $x^{(t+1)}$ 值？就向量 $x^{(t)}$ 而言，这意味着

for $k = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $x_i^{(t+1)} = a_{ii}^{-1}(-\sum_{j<i} a_{ij}x_j^{(t+1)} - \sum_{j>i} a_{ij}x_j^{(t)} + b_i)$

令人惊讶的是，其实现比雅可比方法更为简单：

对于 $t = 1$、 ... 直至收敛，执行：
  for $i = 1 \ldots n$:
    $x_i = a_{ii}^{-1}(-\sum_{j \neq i} a_{ij} x_j + b_i)$

若将其写成矩阵方程形式，可见新计算出的元素 $x_i^{(t+1)}$与 $D_A + L_A$ 的元素相乘，而旧元素 $x_j^{(t)}$则与 U_A 相乘，从而得到

$$(D_A + L_A)x^{(k+1)} = -U_A x^{(k)} + b$$

这被称为高斯 - 赛德尔方法。

在一维情况下，高斯 - 赛德尔方法如图 5.9 所示；每个 x_i 点依然

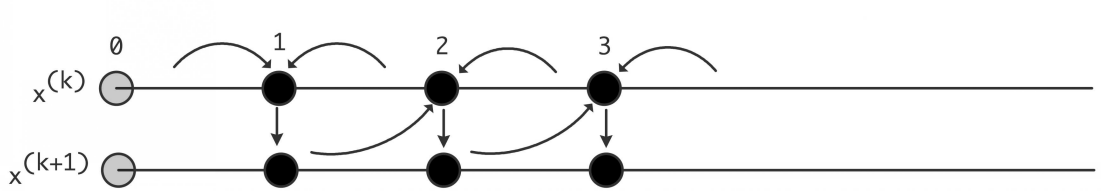


图 5.9：一维问题中高斯 - 赛德尔迭代的数据移动模式。

结合了其相邻节点的值，但此时左侧的值实际上来自下一次外部迭代。

最后，我们可以在高斯 - 赛德尔迭代法中引入一个阻尼参数，从而得到逐次超松弛 *(SOR)* 方法：

for $t = 1, \ldots$ until convergence, do:
  for $i = 1 \ldots n$:
    $x_i^{(t+1)} = \omega a_{ii}^{-1}(-\sum_{j<i} a_{ij} x_j^{(t+1)} - \sum_{j>i} a_{ij} x_j^{(t)} + b_i) + (1 - \omega)x^{(t)}$

令人惊讶的是，对于看似插值的操作，该方法实际上适用于 ω 范围内的值，其中最优值大于 1 [94]。计算最优的 ω 并不简单。

5.5.4 方法的收敛性

我们关注两个问题：一是迭代方法是否收敛，二是若收敛，其速度如何。相关理论远超本书范围。前文提到，对于 M- 矩阵通常能保证收敛；关于收敛速度的完整分析通常仅适用于模型案例。对于如第 4.2.3 节所述的边值问题矩阵，我们不加证明地指出系数矩阵的最小特征值为 $O(h^2)$。上文推导的几何收敛比 $|\lambda(I - AK^{-1})|$ 可表示为：

- 对于雅可比方法，该比率为 $1 - O(h^2)$；
- 对于高斯 - 赛德尔迭代法，该比率同样为 $1 - O(h^2)$，但该方法的收敛速度会快一倍；
- 对于 SOR 方法，最优 omega 值可将收敛因子提升至 $1 - O(h)$。

5.5.5 Jacobi 对比 Gauss-Seidel 及并行性

前文主要从数学角度探讨了 Jacobi 、 Gauss-Seidel 和 SOR 方法。然而在现代计算机上，这些考量很大程度上已被并行化问题所取代。

首先我们注意到，Jacobi 方法单次迭代中的所有计算完全独立，因此可以直接向量化或并行执行。而 Gauss-Seidel 方法则不同（从现在起我们忽略 SOR 方法，因其仅通过阻尼参数与 Gauss-Seidel 有所区别）：由于 x_i 点的计算在迭代中存在依赖关系，这类迭代难以简单向量化或在并行计算机上实现。

许多情况下，这两种方法已被共轭梯度法 (CG) 或广义最小残差法 (GMRES) 取代（参见章节 5.5.11 及 5.5.13 ）。Jacobi 方法有时会作为这些方法的预处理器使用。Gauss-Seidel 仍广受欢迎的一个场景是作为多重网格平滑器。此时常通过采用红黑排序变量来实现并行化，详见章节 7.8.2.1。

关于预条件子中并行化问题的进一步讨论可参阅章节 7.7。

5.5.6 K 的选择

上述收敛性和误差分析表明， K 越接近 A，收敛速度越快。在初始示例中，我们已经看到了 K 的对角线和下三角选择。我们可以通过将 A 分解为对角部分、下三角部分和上三角部分 $A = D_A + L_A + U_A$ 来形式化描述这些选择。以下是一些具有传统名称的方法：

- Richardson 迭代法： $K = \alpha I$。
- Jacobi 方法： $K = D_A$ （对角部分），
- Gauss-Seidel 方法： $K = D_A + L_A$ （下三角部分，包括对角线）
- $K = \omega^{-1} D_A + L_A$ SOR 方法：对称 SOR （ SSOR ）方法：
- $K = (D_A + L_A) D_A^{-1} (D_A + U_A)$。 • 在迭代精化中，我们令 $K = LU$ 为 A 的真实分解。在精确算术中，求解系统 $LUx = y$ 会给出精确解，因此在迭代方法中使用 $K = LU$ 将一步收敛。实际上，舍入误差会使解不精确，因此有时会迭代几步以获得更高精度。

习题 5.49. 对于一个稠密系统，相较于单次系统求解，进行几步迭代精化的额外成本是多少?

Exercise 5.50. The Jacobi iteration for the linear system $Ax = b$ is defined as

$$x_{i+1} = x_i - K^{-1}(Ax_i - b)$$

其中 K 是 A 的对角线。证明你可以通过变换线性系统（即找到一个不同的系数矩阵和右侧向量，但仍保持相同解）来计算相同的 x_i 向量，但使用 $K = I$ （单位矩阵）。

这一策略在存储需求和运算次数方面有何影响? 如果 A 是稀疏矩阵，是否存在特殊影响?

假设 A 对称。请给出一个简单例子说明 $K^{-1}A$ 不必对称。你能想出另一种系统变换方式，既能保持系数矩阵的对称性，又具有与上述变换相同的优势吗？可以假设矩阵具有正对角元素。

练习 5.51. 证明前一个练习的变换也可应用于高斯 - 赛德尔方法。列举若干理由说明为何这不是一个好主意。

备注 20 稳态迭代可视为不精确牛顿法的一种形式，其中每次迭代使用相同的导数逆矩阵近似值。标准泛函分析结果 *[116]* 阐明了该近似值与精确逆矩阵的最大允许偏差。

一个特例是迭代精化，牛顿法理论上应一步收敛，但实践中因计算机算术舍入误差需多步完成。只要函数（或残差）计算足够精确，牛顿法就能收敛这一特性，可通过以较低精度进行 LU 分解来提升性能 *[26]*。

选择预条件矩阵 K 有多种不同方法。其中一些是基于代数定义的，例如下文讨论的不完全分解法。其他选择则受到微分方程的启发。例如，若算子为

$$\frac{\delta}{\delta x}(a(x,y)\frac{\delta}{\delta x}u(x,y)) + \frac{\delta}{\delta y}(b(x,y)\frac{\delta}{\delta y}u(x,y)) = f(x,y)$$

那么矩阵 K 可以从算子推导得出

$$\frac{\delta}{\delta x}(\tilde{a}(x)\frac{\delta}{\delta x}u(x,y)) + \frac{\delta}{\delta y}(\tilde{b}(y)\frac{\delta}{\delta y}u(x,y)) = f(x,y)$$

对于某些 a、b 的选择。第二组方程被称为可分离问题，且存在快速求解器，意味着它们具有 $O(N$ 对数 $N)$ 时间复杂度；参见 [191]。

5.5.6.1 将 K 构造为不完全 LU 分解

我们简要提及另一种受高斯消元法启发的 K 选择。如同高斯消元法，我们设 $K = LU$，但此处采用不完全 LU （ILU）分解。需注意常规 LU 分解因填充现象而计算昂贵。在不完全分解中，我们人为限制填充量。

若将高斯消元法表示为

```
for k,i,j:
   a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

我们通过以下方式定义一个不完全变体

```
for k,i,j:
  if a[i,j] not zero:
    a[i,j] = a[i,j] - a[i,k] * a[k,j] / a[k,k]
```

- 由此得到的因式分解不再精确：$LU \approx A$，因此被称为不完全 LU 分解（ILU）。
- ILU 分解占用的空间远小于完全分解：$L+U$ 的稀疏性与 A 相同。

上述算法被称为 ‘ILU(0)’，其中 ‘0’ 表示在不完全分解过程中绝对不允许任何填充。其他允许有限填充的方案也存在。关于此方法还有更多可探讨的内容；我们仅指出对于 M 矩阵，此方案通常会给出一个收敛的方法 [146]。

练习 5.52. 矩阵 - 向量乘积的操作计数与使用 ILU 分解求解系统的操作计数如何比较？

回顾雅可比方法与高斯 - 赛德尔方法中关于并行性的讨论；参见第 5.5.5 节。ILU 的情况如何？

你已经了解到，稀疏矩阵的完全分解可能需要更高阶的存储空间（分解需 $N^{3/2}$，而矩阵本身仅需 N），但不完全分解仅需 $O(N)$，与矩阵相同。因此，令人惊讶的是误差矩阵 $R = A - LU$ 并非稠密，其本身也是稀疏的。

Exercise 5.53. Let A be the matrix of the *Laplacian*, LU an incomplete factorization, and $R = A - LU$。证明 R 是一个双对角矩阵：

- 考虑到 R 由分解过程中被丢弃的元素组成。它们在矩阵中的位置在哪里？
- 或者，写出乘积 LU 的稀疏模式，并将其与 A 的稀疏模式进行比较。

5.5.6.2 构造预处理器的成本

在热方程的例子中（见章节 4.3），你看到每个时间步都涉及求解一个线性系统。一个重要的实际后果是，任何用于求解线性系统的设置成本，例如构建预处理器，将在需要求解的一系列系统中分摊。类似的论点也适用于非线性方程的背景下，这一主题我们不会专门讨论。非线性方程通过迭代过程求解，例如牛顿法，其多维形式会导致一系列线性系统。尽管这些系统的系数矩阵不同，但通过为多个牛顿步骤重用预处理器，仍可分摊设置成本。

5.5.6.3 并行预处理器

构建和使用预处理器需要权衡多种因素：更精确的预处理器能以更少的迭代次数实现收敛，但每次迭代的计算成本可能更高；此外，构建更精确的预处理器的代价也可能更大。在并行计算环境中，这一问题更为复杂，因为某些预处理器本身就不太适合并行化。因此，我们可能会接受一个并行化程度高但迭代次数多于串行预处理器的方案。更多讨论详见章节 7.7。

5.5.7 停止测试

接下来我们需要解决的问题是何时停止迭代。前文已说明误差呈几何级数递减，因此显然我们永远无法精确获得解 —— 即便计算机在理论上能够实现这一点。

算术运算。由于我们仅有这种相对收敛行为，如何判断何时足够接近?

我们希望误差 $e_i = x - x_i$ 足够小，但直接测量是不可能的。前文观察到 $Ae_i = r_i$，因此

$$\|e_i\| \leq \|A^{-1}\|\|r_i\| \leq \lambda_{\max}(A^{-1})\|r_i\|$$

若我们对 A 的特征值有所了解，则可据此得到误差界限。（ A 的范数仅对对称 A 而言是最大特征值。一般情况下，此处需考虑奇异值。）

另一种方法是监测计算解的变化。若变化微小:

$$\|x_{n+1} - x_n\|/\|x_n\| < \epsilon$$

我们亦可判定已接近解 on.

Exercise5.54. 证明迭代向量间距离与真实解距离之间的解析关系。若方程中含有常数，这些常数能否从理论上或实践中确定?

Exercise 5.55. 编写一个简单程序进行线性方程组求解实验。取一维边值问题的矩阵（采用高效存储方案），并使用 $K = D_A$ 选择编程实现迭代方法。对残差和迭代向量间距离的停止测试进行实验。迭代次数如何随矩阵规模变化?

> 修改矩阵构造方式，使得某个特定量被添加到对角线上，即将 αI 加到原始矩阵中。当 $\alpha > 0$ 时会发生什么? 当 $\alpha < 0$ 时又会如何? 能否找到行为发生变化的临界值? 该值是否与矩阵规模有关?

5.5.8 通用多项式迭代方法理论

前文已介绍形如 $x_{i+1} = x_i - K^{-1}r_i$的迭代方法，接下来我们将探讨更一般形式的迭代方法

$$x_{i+1} = x_i + \sum_{j \leq i} K^{-1}r_j\alpha_{ji}, \tag{5.17}$$

即利用所有先前的残差来更新迭代值。有人可能会问，‘为什么不引入一个额外参数并写成 $x_{i+1} = \alpha_i x_i + \cdots$? ’在此我们简要论证前一种方案描述了一大类方法。事实上，当前作者尚未发现超出此方案范畴的方法。

我们定义残差，给定一个近似解 x，为 $r = Ax - b$。在此通用讨论中，我们将系统预条件化为 $K^{-1}Ax = K^{-1}b$。（参见 5.5.6 节，其中我们讨论了线性系统的变换。）初始猜测对应的残差为

$$\tilde{r} = K^{-1}A\tilde{x} - K^{-1}b.$$

我们现在发现

$$x = A^{-1}b = \tilde{x} - A^{-1}K\tilde{r} = \tilde{x} - (K^{-1}A)^{-1}\tilde{r}.$$

根据凯莱 - 哈密顿定理，对于每个 A 都存在一个多项式 $\phi(x)$ （即特征多项式），使得

$$\phi(A) = 0.$$

我们注意到可以将该多项式 ϕ 表示为

$$\phi(x) = 1 + x\pi(x)$$

其中 π 是另一个多项式。将其应用于 $K^{-1}A$，我们得到

$$0 = \phi(K^{-1}A) = I + K^{-1}A\pi(K^{-1}A) \Rightarrow (K^{-1}A)^{-1} = -\pi(K^{-1}A)$$

因此 $x = x + \pi(K^{-1}A)r_0$。现在，若令 $x_0 = x$，则 $r = K^{-1}r_0$，从而得到方程

$$x = x_0 + \pi(K^{-1}A)K^{-1}r_0.$$

该方程暗示了一种迭代方案：若能找到一系列 i 次多项式 $\pi^{(i)}$ 来逼近 π，将生成一个迭代序列

$$x_{i+1} = x_0 + \pi^{(i)}(K^{-1}A)K^{-1}r_0 = x_0 + K^{-1}\pi^{(i)}(AK^{-1})r_0 \tag{5.18}$$

最终收敛至真实解。基于这种在迭代过程中使用多项式的特性，此类方法被称为多项式迭代方法。

习题 5.56. 定常迭代方法是否属于多项式方法？能否与霍纳法则建立关联？

将方程 (5.18) 乘以 A 并两边减去 b 可得

$$r_{i+1} = r_0 + \tilde{\pi}^{(i)}(AK^{-1})r_0$$

其中 $\pi^{(i)}(x) = x\pi^{(i)}(x)$。由此可直接得到

$$r_i = \hat{\pi}^{(i)}(AK^{-1})r_0 \tag{5.19}$$

其中 $\pi^{(i)}$ 是一个 i 次多项式，且满足 $\pi^{(i)}(0) = 1$。这一陈述可作为迭代方法收敛性理论的基础。然而，这已超出本书的讨论范围。

让我们来看几个方程（ 5.19 ）的实例。对于 $i = 1$ ，我们有

$$r_1 = (\alpha_1 AK^{-1} + \alpha_2 I)r_0 \Rightarrow AK^{-1}r_0 = \beta_1 r_1 + \beta_0 r_0$$

对于某些值 α_i 、 β_i。对于 $i = 2$

$$r_2 = (\alpha_2(AK^{-1})^2 + \alpha_1 AK^{-1} + \alpha_0)r_0$$

对于不同的值 α_i。但我们已确定 AK_0^{-1} 是 r_1 和 r_0 的组合，因此现在我们有

$$(AK^{-1})^2 r_0 \in [\![r_2, r_1, r_0]\!],$$

且显然如何通过归纳法证明

$$(AK^{-1})^i r_0 \in [\![r_i, \dots, r_0]\!]. \tag{5.20}$$

将其代入 (5.18) 我们最终得到

$$x_{i+1} = x_0 + \sum_{j \le i} K^{-1} r_j \alpha_{ji}. \tag{5.21}$$

容易看出方案 (5.17) 属于形式 (5.21)，且反向蕴含同样成立。

S总结而言，迭代方法的基础是一种方案，其中迭代通过所有残差进行更新

$$x_{i+1} = x_i + \sum_{j \le i} K^{-1} r_j \alpha_{ji}. \tag{5.22}$$

与静态迭代法（章节 5.5.1 ）相比，后者仅根据上一次残差更新迭代值，且系数保持恒定。

关于 α_{ij} 系数，我们可以进一步阐述。若将方程（ 5.22 ）乘以 A，并从等式两边减去 b，可得

$$r_{i+1} = r_i + \sum_{j \le i} AK^{-1} r_j \alpha_{ji}. \tag{5.23}$$

让我们稍加思考这个方程。若给定初始残差 r_0，下一个残差的计算方式为

$$r_1 = r_0 + AK^{-1} r_0 \alpha_{00}.$$

由此可得 $AK^{-1} r_0 = \alpha_{00}^{-1}(r_1 - r_0)$，因此对于下一个残差，

$$\begin{aligned} r_2 &= r_1 + AK^{-1} r_1 \alpha_{11} + AK^{-1} r_0 \alpha_{01} \\ &= r_1 + AK^{-1} r_1 \alpha_{11} + \alpha_{00}^{-1} \alpha_{01} (r_1 - r_0) \\ \Rightarrow AK^{-1} r_1 &= \alpha_{11}^{-1} (r_2 - (1 + \alpha_{00}^{-1} \alpha_{01}) r_1 + \alpha_{00}^{-1} \alpha_{01} r_0) \end{aligned}$$

可见 $AK^{-1} r_1$ 可表示为求和式 $r_2 \beta_2 + r_1 \beta_1 + r_0 \beta_0$，且 $\sum_i \beta_i = 0$。

将其推广后，我们发现（使用不同于上述的 α_{ij} ）

$$\begin{array}{rll} r_{i+1} & = r_i + AK^{-1} r_i \delta_i + \sum_{j \le i+1} r_j \alpha_{ji} & \\ r_{i+1}(1 - \alpha_{i+1,i}) & = AK^{-1} r_i \delta_i + r_i (1 + \alpha_{ii}) + \sum_{j<i} r_j \alpha_{ji} & \\ r_{i+1} \alpha_{i+1,i} & = AK^{-1} r_i \delta_i + \sum_{j \le i} r_j \alpha_{ji} & \text{substituting } \begin{array}{ll} \alpha_{ii} & := 1 + \alpha_{ii} \\ \alpha_{i+1,i} & := 1 - \alpha_{i+1,i} \end{array} \\ & & \text{note that } \alpha_{i+1,i} = \sum_{j \le i} \alpha_{ji} \\ r_{i+1} \alpha_{i+1,i} \delta_i^{-1} & = AK^{-1} r_i + \sum_{j \le i} r_j \alpha_{ji} \delta_i^{-1} & \\ r_{i+1} \alpha_{i+1,i} \delta_i^{-1} & = AK^{-1} r_i + \sum_{j \le i} r_j \alpha_{ji} \delta_i^{-1} & \\ r_{i+1} \gamma_{i+1,i} & AK^{-1} r_i + \sum_{j \le i} r_j \gamma_{ji} & \text{substituting } \gamma_{ij} = \alpha_{ij} \delta_j^{-1} \end{array}$$

且有 $\gamma_{i+1,i} = \sum_{j \le i} \gamma_{ji}$。

我们可以将最后一个方程改写为 $AK^{-1}R = RH$ 其中

$$H = \begin{pmatrix} -\gamma_{11} & -\gamma_{12} & \cdots & & \\ \gamma_{21} & -\gamma_{22} & -\gamma_{23} & \cdots & \\ 0 & \gamma_{32} & -\gamma_{33} & -\gamma_{34} & \\ \emptyset & \ddots & \ddots & \ddots & \ddots \end{pmatrix}$$

其中，H 是一个所谓的海森堡矩阵：它是一个上三角矩阵加上一条下子对角线。同时我们注意到 H 的每一列元素之和为零。

由于恒等式 $\gamma_{i+1,i} = \sum_{j \le i} \gamma_{ji}$，我们可以从 r_{i+1} 的等式两边减去 b 并 ‘约去 A’，得到

$$x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j \le i} x_j \gamma_{ji}.$$

这给出了迭代方法的一般形式：

$$\begin{cases} r_i = Ax_i - b \\ x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j \le i} x_j \gamma_{ji} \\ r_{i+1}\gamma_{i+1,i} = AK^{-1}r_i + \sum_{j \le i} r_j \gamma_{ji} \end{cases} \qquad \text{where } \gamma_{i+1,i} = \sum_{j \le i} \gamma_{ji}. \tag{5.24}$$

这种形式适用于许多迭代方法，包括您之前见过的静态迭代方法。在接下来的章节中，您将看到 γ_{ij} 系数如何从残差的正交性条件中推导出来。

5.5.9 通过正交化迭代

上述静态方法（第 5.5.1 节）以某种形式存在已久：高斯在给学生的一封信中描述了某些变体。这些方法在 1950 年杨的论文 [198] 中得到完善；最终参考文献可能是瓦尔加的著作 [186]。如今这些方法很少使用，除了在专门的多重网格平滑器 *multigrid smoothers* 领域，本课程不讨论这一主题。

几乎在同一时期，基于正交化的方法领域由两篇论文 [129, 102]，开创，尽管它们花费了几十年时间才获得广泛应用（更多历史参见 [79]）。

基本思路如下：

> 若能使所有残差向量彼此正交，且矩阵维度为 n，则经过 n 次迭代后必然收敛：不可能存在一个 $n+1$ 阶残差既与之前所有残差正交又非零。由于零残差意味着对应迭代值即为解，我们可得出结论：经过 n 次迭代即可获得真实解。

随着当代应用生成的矩阵规模增大，这种推理已不再适用：迭代 n 次在计算上并不现实。此外，舍入误差很可能会破坏解的精度。然而，后来人们意识到 [165] 这类方法在对称正定（SPD）矩阵的情况下是可行的选择。其推理如下：

> 残差序列跨越了一系列维度递增的子空间，通过正交化处理，新的残差被投影到这些空间上。这意味着它们的规模将逐渐减小。

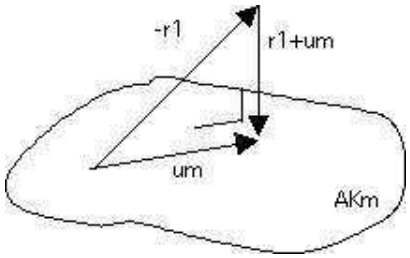


Figure 5.10: The optimal update u_m make the new residual orthogonal to the AK_m subspace.

如图 5.10 所示。

本节将介绍通过正交化进行迭代的基本思想。这里呈现的方法仅具有理论意义；接下来你将看到共轭梯度法（CG）和广义最小残差法（GMRES），这些方法是许多实际应用的基础。

现在让我们采用基本方案（5.24）并对残差进行正交化。我们使用 K^{-1}- 内积而非常规内积：

$$(x,y)_{K^{-1}} = x^t K^{-1} y$$

并且我们将强制残差为 K^{-1}- 正交：

$$\forall_{i\neq j}: r_i \perp_{K^{-1}} r_j \Leftrightarrow \forall_{i\neq j}: r_i K^{-1} r_j = 0$$

这被称为完全正交化方法（*FOM*）方案：

> 给定 r_0 对于 $i \geq 0$：令 $s \leftarrow K^{-1} r_i$ 令
> $t \leftarrow AK^{-1} r_i$ 对于 $j \leq i$：令 γ_j 为系数，使得
> $t - \gamma_j r_j \perp r_j$ 对于 $j \leq i$：形成 $s \leftarrow s - \gamma_j x_j$ 和
> $t \leftarrow t - \gamma_j r_j$ 令 $x_{i+1} = (\sum_j \gamma_j)^{-1} s$,
> $r_{i+1} = (\sum_j \gamma_j)^{-1} t$。

您可能认出了其中的 *Gram-Schmidt* 正交化过程（详见附录 14.2 说明）：在每次迭代中，r_{i+1} 初始设为 $AK^{-1}r_i$，并通过 $j \le i$ 与 r_j 进行正交化。

我们可以使用修正的 *Gram-Schmidt* 方法，将算法改写为：

给定 r_0，对于 $i \ge 0$：设 $s \leftarrow K^{-1}r_i$，设 $t \leftarrow AK^{-1}r_i$，对于 $j \le i$：设 γ_j 为系数，使得 $t - \gamma_j r_j \perp r_j$ 形成 $s \leftarrow s - \gamma_j x_j$ 与 $t \leftarrow t - \gamma_j r_j$，设 $x_{i+1} = (\sum_j \gamma_j)^{-1} s$、$r_{i+1} = (\sum_j \gamma_j)^{-1} t$。

FOM 算法的这两个版本在精确算术下等价，但在实际应用中存在两方面差异：

- 修正的 Gram-Schmidt 方法在数值上更稳定；
- 未经修改的方法允许您同时计算所有内积，从而减少网络延迟。我们将在以下章节讨论这一点：7.1.3 、7.6.1.2 、 7.6。

尽管 FOM 算法在实际中未被使用，但这些计算考量同样适用于下文中的 GMRES 方法。

5.5.10 迭代方法的耦合递推形式

前文已展示生成迭代和搜索方向的通用方程（ 5.24 ）。该方程通常被拆分为

- 基于单一搜索方向对 x_i 迭代的更新：

$$x_{i+1} = x_i - \delta_i p_i,$$

and

- 基于当前已知残差构建搜索方向：

$$p_i = K^{-1}r_i + \sum_{j<i} \beta_{ij} K^{-1} r_j.$$

通过归纳不难看出，我们同样可以定义

$$p_i = K^{-1}r_i + \sum_{j<i} \gamma_{ji} p_j,$$

而最后这种形式正是实际应用中采用的。

迭代依赖系数通常被选择为使残差满足各种正交性条件。例如，可以选择让方法通过使残差正交（$r_i^t r_j = 0$若 $i \ne j$）或 A- 正交（$r_i^t A r_j = 0$若 $i \ne j$）来定义。还存在更多方案。此类方法可比静态迭代收敛更快，或适用于更广泛的矩阵和预条件子类型。下文我们将看到两种此类方法；但其分析已超出本课程范围。

5.5.11 共轭梯度法

本节我们将推导共轭梯度法（CG），这是 FOM 算法的一种具体实现。特别地，当矩阵 A 为对称正定（SPD）时，该方法具有优良的计算特性。

共轭梯度法采用上述耦合递推公式作为基本形式，其系数通过要求残差序列 r_0、r_1、r_2、… 满足特定条件来定义。

$$r_i^t K^{-1} r_j = 0 \quad \text{if } i \neq j.$$

我们首先推导非对称系统的共轭梯度法，随后展示其在对称情况下的简化形式（此处方法源自 [58]）

基本方程为

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_{i+1} = K^{-1} r_{i+1} + \sum_{j \leq i} \gamma_{ji+1} p_j, \end{cases} \tag{5.25}$$

其中第一和第三个方程已在前面引入，第二个方程可通过将第一个方程乘以 A 得到（请验证！）。

现在我们将通过归纳法推导该方法中的系数。本质上，我们假设当前残差为 $r$cur，待计算的残差为 $r$new，以及已知残差集合 Rold。为避免使用下标 ‘old,cur,new’，我们采用以下约定：

- x_1、r_1、p_1 分别表示当前迭代步、残差和搜索方向。请注意此处下标 1 并不代表迭代次数。
- x_2、r_2、p_2 分别表示即将计算的迭代步、残差和搜索方向。同样地，该下标不等于迭代次数。
- X_0、R_0、P_0 是所有先前的迭代步、残差和搜索方向，它们被打包成一个向量块。

I根据这些量，更新方程可表示为

$$\begin{cases} x_2 = x_1 - \delta_1 p_1 \\ r_2 = r_1 - \delta_i A p_1 \\ p_2 = K^{-1} r_2 + v_{12} p_1 + P_0 u_{02} \end{cases} \tag{5.26}$$

其中 δ_1、v_{12} 为标量，u_{02} 是一个向量，其长度为当前迭代之前的迭代次数。现在我们从残差的正交性推导出 δ_1、v_{12}、u_{02}。具体而言，残差必须在 K^{-1} 内积下正交：我们希望满足

$$r_2^t K^{-1} r_1 = 0, \qquad r_2^t K^{-1} R_0 = 0.$$

结合这些关系式，我们可以得到例如

$$\left.\begin{array}{l} r_1^t K^{-1} r_2 = 0 \\ r_2 = r_1 - \delta_i A K^{-1} p_1 \end{array}\right\} \Rightarrow \delta_1 = \frac{r_1^t K^{-1} r_1}{r_1^t K^{-1} A p_1}.$$

寻找 v_{12}, u_{02} 稍难一些。为此，我们首先将方程 (5.25) 中残差和搜索方向的关系以块形式总结为

$$(R_0, r_1, r_2)\left(\begin{array}{ccc|c|c} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{array}\right) = A(P_0, p_1, p_2)\,\mathrm{diag}(D_0, d_1, d_2)$$

$$(P_0, p_1, p_2)\begin{pmatrix} I - U_{00} & -u_{01} & -u_{02} \\ & 1 & -v_{12} \\ & & 1 \end{pmatrix} = K^{-1}(R_0, r_1, r_2)$$

or abbreviated $RJ = APD$, $P(I-U) = R$ where J is the matrix with identity diagonal and minus identity subdiagonal. We then observe that

- $R^tK^{-1}R$ 是对角矩阵，表达了残差的正交性。
- 结合 $R^tK^{-1}R$是对角矩阵且 $P(I-U) = R$ 可得 $R^tP = R^tK^{-1}R(I-U)^{-1}$。现在我们推断 $(I-U)^{-1}$ 是上对角矩阵，因此 R^tP是上三角矩阵。这表明诸如 $r_2^tp_1$ 等量值为零。

- 结合 R 和 P的关系，我们首先得到

  $$R^tK^{-t}AP = R^tK^{-t}RJD^{-1}$$

  这表明 $R^tK^{-t}AP$是下双对角矩阵。在此方程中展开 R 可得

  $$P^tAP = (I-U)^{-t}R^tRJD^{-1}.$$

  此处 D 与 $R^tK^{-1}R$为对角矩阵，$(I-U)^{-t}$与 J 为下三角矩阵，因此 P^tAP是下三角矩阵。

- 这表明 $P_0^tAp_2 = 0$与 $p_1^tAp_2 = 0$。- 取 P_0^tA与 p_1^tA的乘积，结合方程 (5.26) 中的定义可得

  $$u_{02} = -(P_0^tAP_0)^{-1}P_0^tAK^{-1}r_2, \qquad v_{12} = -(p_1^tAp_1)^{-1}p_1^tAK^{-1}r_2.$$

- 若 A 对称，则 P^tAP为下三角矩阵（见上文）且对称，故实际上是对角矩阵。此外，$R^tK^{-t}AP$ 为下双对角矩阵，因此利用 $A = A^t$可知 $P^tAK^{-1}R$为上双对角矩阵。由于 $P^tAK^{-1}R = P^tAP(I-U)$，我们得出 $I-U$ 为上双对角矩阵，故仅在对称情况下 $u_{02} = 0$。

关于此推导的一些观察。

- 严格来说，此处我们仅证明必要条件关系。可以证明这些条件也是充分的。

- 在精确算术中，存在不同公式最终计算出相同向量的情况。例如，很容易推导出 $p_1^tr_1 = r_1^tr_1$，因此可以将其代入刚才推导的公式中。CG 方法（共轭梯度法）的典型实现如图 5.11 所示。

Compute $r^{(0)} = Ax^{(0)} - b$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Kz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\delta_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} - \delta_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \delta_i q^{(i)}$
  check convergence; continue if necessary
end

图 5.11：预处理共轭梯度方法。

- 在第 k 次迭代中，计算 $P_0^t A r_2$（这是 u_{02} 所需的）需要进行 k 次内积运算。首先，内积运算在并行环境下存在劣势。其次，这要求我们无限期存储所有搜索方向。第二点意味着随着迭代次数的增加，计算量和存储需求都会上升。相比之下，稳态迭代方案中存储仅限矩阵和少量向量，且每次迭代的计算量相同。
- 上述问题在对称情况下不复存在。由于 u_{02} 为零，对 P_0 的依赖消失，仅保留对 p_1 的依赖。因此，存储需求恒定，每次迭代的计算量也恒定。可以证明每次迭代仅需两次内积运算。

练习 5.57. 对共轭梯度方法单次迭代中的各类运算进行浮点操作计数。假设 A 是五点模板的矩阵且预条件子 M 为 A 的不完全分解（章节 5.5.6.1 ）。设 N 为矩阵规模。

5.5.12 从最小化推导

上述 CG 方法的推导在文献中并不常见。典型的推导从一个具有对称正定（*SPD*）矩阵 A 的最小化问题开始:

$$\text{For which vector } x \text{ with } \|x\| = 1 \text{ is } f(x) = 1/2\, x^t Ax - b^t x \text{ minimal?} \tag{5.27}$$

如果我们接受函数 f 存在最小值这一事实（这源于正定性），则通过计算导数来找到最小值

$$f'(x) = Ax - b.$$

并询问 $f'(x) = 0$的零点位置。于是，原线性系统便奇迹般地出现了。

练习 5.58. 推导上述导数公式。（提示：将导数定义写为 $\lim_{h\downarrow 0}\ldots$。）注意这要求 A 对称。关于迭代方法的推导，我们声明迭代 x

i 以特定步长 δ 更新 i

a长 a 搜索方向 p_i:

$$x_{i+1} = x_i + p_i\delta_i$$

最优步长

$$\delta_i = \frac{r_i^t p_i}{p_1^t A p_i}$$

随后被推导为沿直线 $x_i + \delta\delta p_i$ 最小化函数 f 的值:

$$\delta_i = \operatorname*{argmin}_{\delta} \|f(x_i + p_i\delta)\|$$

通过归纳证明，从残差构建搜索方向的要求是残差必须正交。典型证明可参考 [5]。

5.5.13 GMRES 方法

在上述 CG 方法的讨论中曾指出，残差的正交性需要存储所有残差，并在第 k 次迭代中进行 k 次内积运算。遗憾的是，可以证明 CG 方法的工作量节省在实际应用中几乎无法在 *SPD* 矩阵 [60] 之外实现。

GMRES 方法是这类完全正交化方案的一种流行实现。为了将计算成本控制在合理范围内，通常采用**重启策略** —— 即仅保留一定数量（如 $k = 5$ 或 20 个）的残差，每 k 次迭代后重新启动算法。

还存在其他不像 GMRES 那样存储需求持续增长的方法，例如 QMR [67] 和 BiCGstab [185]。尽管根据前述说明这些方法无法使残差正交化，但在实践中仍具吸引力。

5.5.14 收敛性与复杂度

高斯消元法的效率相对容易评估：确定性地分解和求解一个系统需要 $\frac{1}{3}n^3$ 次运算。对于迭代方法而言，运算次数是每次迭代的运算量乘以迭代次数的乘积。尽管每次单独的迭代易于分析，但尚无完善的理论能预测迭代次数。（事实上，迭代方法甚至可能一开始就不收敛。）

一个基本定理是关于共轭梯度法（ CG ）的

$$\#it \sim \sqrt{\kappa(A)}$$

而对于二阶偏微分方程

$$\kappa(A) \sim h^{-2} = N,$$

with N the matrix size.

预条件子的影响较难量化。对于许多预条件子（如 ILU ），

$$\kappa(M^{-1}A) \sim N,$$

b但比例常数较低。大量研究致力于开发能实现
o阶数改进的预条件子：

$$\kappa(M^{-1}A) \sim \sqrt{N}.$$

然而，这通常仅在受限情况下可证明，例如 M 矩阵 [91, 11]。限制条件更严格的是多重网格 [22], ，理想情况下它能实现

$$\kappa(M^{-1}A) = O(1).$$

评估迭代方法效率的最终考量与处理器利用率相关。高斯消元法可通过编码实现显著的缓存复用，使算法以计算机峰值速度的较高百分比运行。而迭代方法在每秒浮点运算基础上则慢得多。

所有这些考虑因素使得将迭代方法应用于线性系统求解介于一门手艺与一门黑色艺术之间。实践中，人们会进行大量实验来决定迭代方法是否值得采用，以及如果值得，哪种方法更可取。

5.6 特征值方法

本章至此我们仅限于讨论线性系统求解。特征值问题是线性代数应用中另一重要类别，但其意义更多在于数学层面而非计算本身。我们将简要概述相关计算类型。

5.6.1 幂方法

该幂方法是一种简单迭代过程（详见附录 14.3 ）：给定矩阵 A 和任意初始向量 v，重复计算

$$v \leftarrow Av, \quad v \leftarrow v/\|v\|.$$

向量 v 会迅速成为对应于绝对值最大特征值的特征向量，因此 $\|Av\|/\|v\|$ 会成为该最大特征值的近似值。

将幂方法应用于 A^{-1} 被称为逆迭代，它能得到绝对值最小的特征值的倒数。

幂方法的另一种变体是位移逆迭代，可用于寻找内部特征值。如果 σ 接近某个内部特征值，那么对 $A - \sigma I$ 进行逆迭代将找到该内部特征值。

5.6.2 正交迭代方案

利用不同特征值对应的特征向量正交这一特性，可衍生多种方法。例如，在找到一个特征向量后，可在其正交补空间中进行迭代。另一种方案是对向量块进行迭代，并在每次幂方法迭代后对该块进行正交化处理。由此可获得与块大小相同数量的主特征值。重启 *Arnoldi* 方法 [134] 就是此类方案的典型代表。

5.6.3 全谱方法

前述迭代方案仅能获得局部特征值。其他方法则可计算矩阵的完整谱，其中最著名的是 *QR* 方法。

5.7 延伸阅读

迭代方法是一个非常深奥的领域。作为相关问题的实用入门，你可以阅读 ‘模板书’ [9], 在线版本位于 http://netlib.org/templates/。若需更深入的理论探讨，可参阅 Saad 的著作 [166], 其第一版可在 http://www-users.cs.umn.edu/~saad/books.html 下载。

第 6 章

性能导向编程

本节我们将探讨不同编程方式如何影响代码性能。这仅是该主题的入门介绍。

完整代码清单及图中数据的详细说明可参阅第 25 章。

6.1 峰值性能

出于营销目的，可能需要为处理器定义一个 ‘ 最高速度 ’ 。由于流水线浮点运算单元在理论上每个周期可渐近输出一个结果，其理论峰值性能可计算为时钟频率（每秒周期数）、向量指令宽度、浮点运算单元数量及核心数量的乘积。这一极限速度在实际中无法达到，且极少有代码能接近该值。

少数能接近峰值性能的运算之一是矩阵 - 矩阵乘积，若编码得当；参见 7.4.1 节。这是 *Linpack* 基准测试的基础；该基准的并行版本记录于 ‘ 全球超级计算机 500 强 ’ 榜单；参见 2.11.4 节。

从某种意义上说，本章后续内容将探讨阻碍获得峰值性能的所有因素。

6.2 带宽

过去至少三十年的处理器基本事实是：处理单元速度远超内存 —— 处理器每秒能运算的数字量超过内存能供给的量。

我们将算术强度定义为每字操作数；若该数值过低，则称算法受带宽限制，因其性能由内存带宽而非处理器速度决定。

或称之为复用因子。数据复用是通过利用内存缓存实现高性能的关键；参见 1.3.5 节。

你可能会认为通过执行一个简单的 *streaming kernel* 就能测量带宽：

```
// hardware/allocation.cpp
template< typename R >
R sum_stream( span<R> stream ) {
  R sum{static_cast<R>(0)};
  for ( auto e : stream )
    sum += e;
  return sum;
};
```

这会一次性加载一段连续的内存，并仅返回一个数值。我们可以简单地通过加载的字节数除以执行时间来计算带宽。

然而，有几个因素会使情况复杂化。

1. 数据流的长度很关键：如果数据流较短，可能会完全放入缓存中，此时测得的带宽会比必须从主内存加载时更高。2. 即使数据流短到可以放入缓存，最初仍需从内存加载到缓存；我们将在第 **??** 节讨论前两点。3. 前两点假设所有加载的数据都被实际使用。若非如此，有效带宽会降低。4. 最后，现代处理器具有多核架构，主内存带宽由多个核心共享。单独运行流式内核时测得的带宽，很可能高于所有核心同时运行该内核时的带宽。

6.2.1 聚合带宽

我们通过在多组流长度和多核数量上运行流式内核来探究上述因素。

```
// hardware/bandwidth.cpp
  vector<real> results(nthreads,0.);
  float bw{0};
#   pragma omp parallel proc_bind(spread) reduction(+:bw)
  {
    int my_thread_number = omp_get_thread_num();
    auto my_data_stream = memory.get_stream(my_thread_number);
    for (int irepeat=0; irepeat<how_many_repeats; irepeat++)
      results.at(my_thread_number) += sum_stream(my_data_stream);
    bw += how_many_repeats * memory.stream_bytes();
  }
```

我们在 TACC 的 *Frontera* 集群上对此进行测试，该集群配备双路 *Intel Cascade Lake* 处理器，每个节点共计 56 个核心。

图 6.1 显示，对于小规模数据集，聚合带宽随核心数量线性增长。这是因为数据集能完全放入私有缓存中。反之，对于较大规模的数据集，聚合带宽在核心数约一半时趋于平缓。

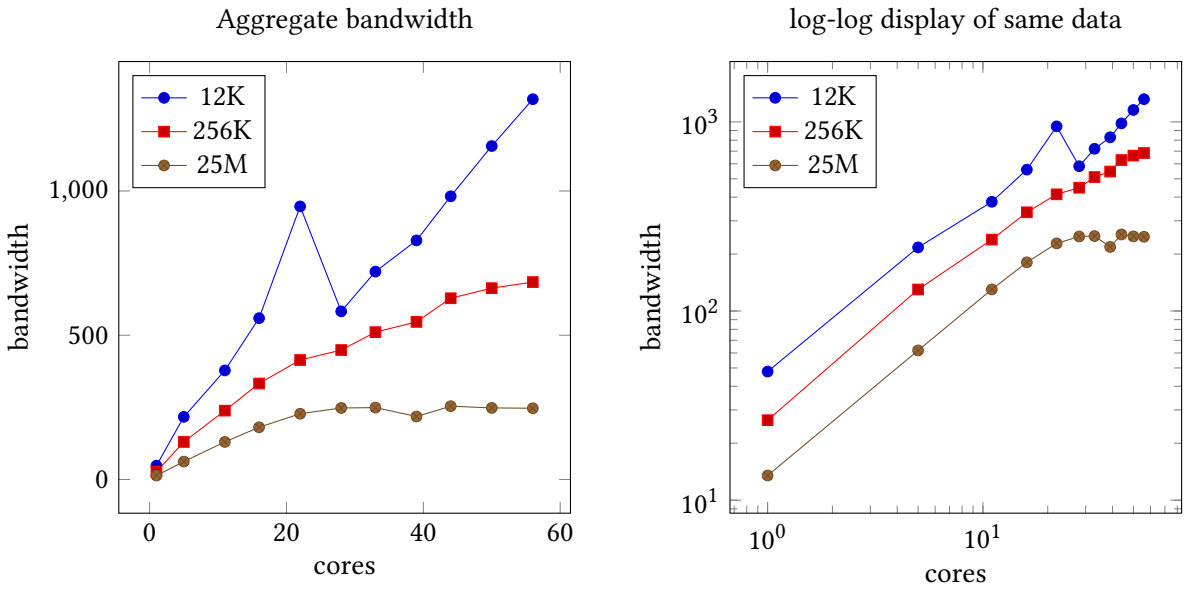


图 6.1：聚合带宽测量与核心数量及数据集大小的函数关系

6.2.2 跨步访问

用更传统的编程术语来说：

```
for ( size_t i=0; i<cachesize_in_words; i+=stride )
   f( thecache[i] );
```

由于数据是以名为缓存行的连续块从内存移动到缓存中的（参见章节 1.3.5.7 ），未充分利用缓存行中所有数据的代码会遭受带宽惩罚。为探究此现象，我们采用跨步操作

```
// hardware/allocation.cpptemplate <typename R>
void Cache<R>::transform_in_place( std::function< void(R&) > f,int stride,int nrepeats ) {
for ( int irepeat=0; irepeat<nrepeats; irepeat++ ) {// for short loops,
 the range version is 3x slower
//    std::ranges::for_each( *this | std::ranges::views::stride(stride),f );
for ( size_t i=0; i<cachesize_in_words; i+=stride )f( thecache[i] );}};
```

针对多个跨步值。

如上所述，我们尝试多种流大小和核心数量：

```
// hardware/stride.cpp
    #pragma omp parallel proc_bind(spread)
    {
```

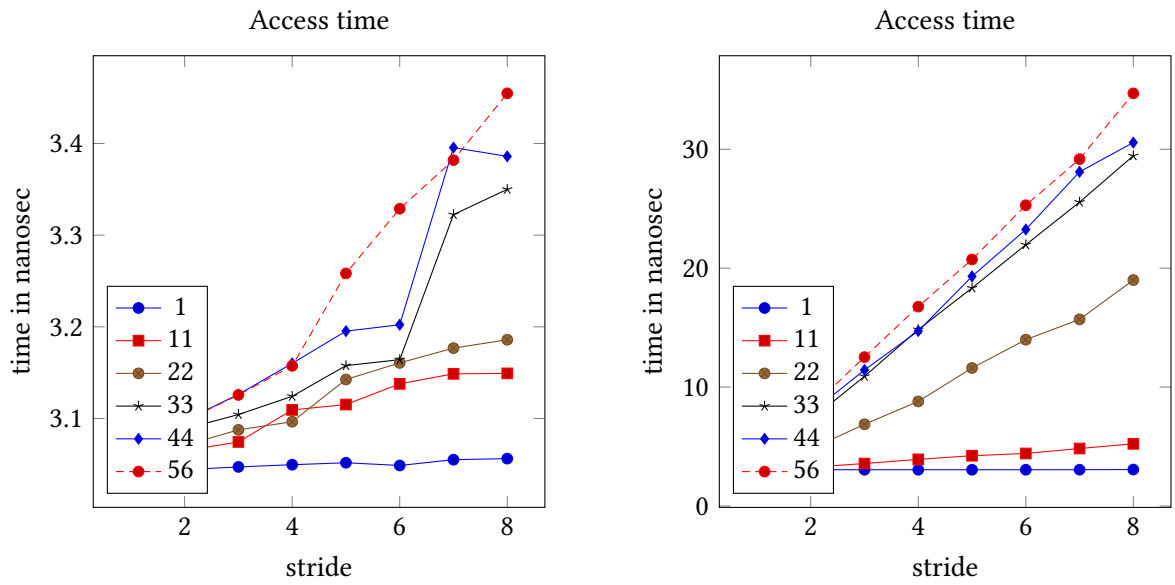


图 6.2：不同步长和核心数下的每字访问时间，Frontera；左侧数据集大小：10 万，右侧 100 万；注意不同的比例尺！

```
  int my_thread_number = omp_get_thread_num();
  auto my_data_stream = write_data.get_stream(my_thread_number);
  auto my_flush_stream = flush_data.get_stream(my_thread_number);
  for (int irepeat=0; irepeat<how_many_repeats; irepeat++) {
    my_data_stream.transform_in_place( [] (floattype& x ) { x+=1.; }, stride );
    if (flush)
      my_flush_stream.transform_in_place( [] (floattype& x ) { x+=1.; }, stride );
  }
}
```

这里我们使用恒定大小的数据集 s，并根据处理的元素数量计算有效带宽 $d = s/$ 步长。图 6.2 显示，对于小数据集，每个元素的访问时间大致恒定，因为数据以高速从缓存中流式传输。然而，对于较大的数据集，数据从主内存流式传输，主内存没有足够的带宽供所有核心使用，因此我们看到每字的访问时间随着步长的增加而上升。

更多内容参见章节 1.6.1.3。

6.3 算术运算性能

6.3.1 下溢计算

许多处理器可以正常处理非规格化数，但必须通过微代码模拟这些运算。这会显著降低性能。

	Intel Cascade Lake	AMD Milan
Flush-to-zero	1.5	1.4
	1.5	1.4
No Ftz	1.5	1.7
	45	1.7

图 6.3：每个字的访问时间

例如，我们计算一个几何序列 $n \mapsto r^n$，其中包含 $r < 1$。对于足够小的值 r，该序列会出现下溢，计算速度会变慢。为了获得宏观计时，在此代码中我们实际上操作的是一个由相同数字组成的数组。

```
// hardware/denormal.cpp
memory.set(startvalue);
/* repeated scale the array */
for (int r=0; r<repeats; r++) {
  memory.transform_in_place( [ratio] (floattype &x) { x *= ratio; } );
  memory.front() = memory.back();
}
```

其中转换例程为：

```
// hardware/allocation.cpptemplate <typename R>
void Cache<R>::transform_in_place( std::function< void(R&) > f,int stride,int nrepeats ) {
for ( int irepeat=0; irepeat<nrepeats; irepeat++ ) {// for short loops,
 the range version is 3x slower
//    std::ranges::for_each( *this | std::ranges::views::stride(stride),f );
for ( size_t i=0; i<cachesize_in_words; i+=stride )f( thecache[i] );}};
```

首先我们采用 *flush-to-zero* 的默认行为：任何次正规数会被置零；随后展示正确处理非正规数时伴随的性能下降。

6.3.2 流水线技术

在章节 1.2.1.3 中已了解到现代 CPU 的浮点单元采用流水线设计，且流水线需要大量独立操作才能高效运行。典型的可流水线化操作是向量加法；而无法流水线化的操作实例是内积累积运算

```
for (i=0; i<N; i++)
  s += a[i]*b[i];
```

由于 `s` 同时涉及读取和写入操作，这会中断加法流水线。填充浮点流水线的一种方法是采用循环展开：

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += a[2*i] * b[2*i];
  sum2 += a[2*i+1] * b[2*i+1];
}
```

现在两次独立的乘法运算被插入在累加操作之间。通过少量索引优化后，代码变为：

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += *(a + 0) * *(b + 0);
  sum2 += *(a + 1) * *(b + 1);

  a += 2; b += 2;
}
```

在进一步优化中，我们将每条指令的加法部分与乘法部分解耦。这样当累加操作等待乘法结果时，插入的指令能保持处理器忙碌，实际上提升了每秒操作数。

```
for (i = 0; i < N/2-1; i ++) {
  temp1 = *(a + 0) * *(b + 0);
  temp2 = *(a + 1) * *(b + 1);

  sum1 += temp1; sum2 += temp2;

  a += 2; b += 2;
}
```

最终，我们意识到将加法运算尽可能远离乘法运算的最远位置，就是将其直接置于下一次迭代的乘法运算之前：

```
for (i = 0; i < N/2-1; i ++) {
  sum1 += temp1;
  temp1 = *(a + 0) * *(b + 0);

  sum2 += temp2;
  temp2 = *(a + 1) * *(b + 1);

  a += 2; b += 2;
}
s = temp1 + temp2;
```

当然，我们可以以超过两倍的因子展开操作。虽然我们预期由于流水线操作序列的延长会带来性能提升，但大的展开因子需要大量寄存器。当请求的寄存器数量超过 CPU 实际拥有时，就会发生寄存器溢出，这将导致性能下降。

另一需要注意的事项是，操作总数很可能无法被展开因子整除。这需要在循环后添加清理代码来处理剩余的迭代。因此，展开后的代码比直接编写的代码更难编写，为此人们开发了工具来自动执行这类源码到源码的转换。

6.3.3 循环展开的语义

关于循环展开变换的一个观察是，我们隐式利用了加法的结合律与交换律：虽然相加的量相同，但实际上它们现在是以不同顺序相加的。正如你将在第 3 章看到的，在计算机算术中这并不能保证得到完全相同的结果。

因此，编译器仅在明确允许时才会应用此变换。例如，对于 *Intel* 编译器，选项 `fp-model precise` 表示代码变换应保留浮点运算的语义，此时不允许循环展开。而 `fp-model fast` 则表示可以为了速度牺牲浮点运算精度。

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                Processing 16384 words
                using memory model: fast
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Unroll  1.. ran for      518 usec (efficiency: 100% )
Unroll  2.. ran for      451 usec (efficiency: 114% )
Unroll  4.. ran for      258 usec (efficiency: 200% )
unroll factor 8 not implemented
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                Processing 16384 words
                using memory model: precise
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Unroll  1.. ran for     1782 usec (efficiency: 100% )
Unroll  2.. ran for      444 usec (efficiency: 401% )
Unroll  4.. ran for      256 usec (efficiency: 696% )
unroll factor 8 not implemented
```

6.4 硬件探索

6.4.1 缓存大小

前文提到，从 L1 缓存移动数据比从 L2 缓存具有更低的延迟和更高的带宽，而 L2 又比 L3 或主内存更快。因此，若代码存在数据复用的可能性，就应当以能让复用数据适配缓存容量的方式编写。

以一个典型示例来说明：考虑需要重复访问相同数据的代码：

```
for (int i=0; i<NRUNS; i++)
  for (int j=0; j<size; j++)
    array[j] = 2.3*array[j]+1.2;
```

I若 size 参数能使数组适配缓存容量，该操作将相对较快。随着
t随着数据集增长，其部分数据会将其他部分从 L1 缓存中驱逐出去，因此操作速度将
b由 L2 缓存的延迟和带宽决定。

Exercise6.1. 论证当问题规模足够大且采用 LRU 替换策略时（参见章节 1.3.5.6 ），外层循环每次迭代中 L1 缓存的数据几乎都会被替换。能否编写示例代码让部分 L1 数据保持驻留?

通过合理安排运算顺序，有可能将数据保留在 L1 缓存中。例如，在我们的示例中，可以这样编写

```
for (int b=0; b<size/l1size; b++) {
  blockstart = 0;
  for (int i=0; i<NRUNS; i++) {
    for (int j=0; j<l1size; j++)
      array[blockstart+j] = 2.3*array[blockstart+j]+1.2;
  }
  blockstart += l1size;
}
```

假设 L1 缓存大小能整除数据集大小。这种策略被称为缓存分块或缓存重用分块。

R备注 21 与循环展开类似，分块代码可能改变表达式求值顺序。由于浮点
*p*运算不具备结合律，分块并非编译器允许执行的转换操作。您
*c*提供编译器选项以指示是否严格遵守语言规则。

如上例过于简单，我们接下来将论证这一点。然而，分块技术是诸如矩阵 - 矩阵乘积内核等操作中的关键手段。

6.4.2 演示缓存对性能的影响

您可以尝试编写一个如上方所示的小型数组循环，并多次执行它，期望在数组大小超过缓存容量时观察到性能下降。迭代次数的选择应确保测量时间远超过时钟分辨率。

这会遇到几个未预料到的问题。简单循环嵌套的计时

```
for (int irepeat=0; irepeat<how_many_repeats; irepeat++) {
  for (int iword=0; iword<cachesize_in_words; iword++)
    memory[iword] += 1.1;
}
```

看似与数组大小无关。

要了解编译器如何处理这段代码，可让您的编译器生成一份优化报告。对于 *Intel* 编译器，请使用 `-qopt-report`。报告中可见编译器已决定交换循环顺序：此时数组的每个元素仅被加载一次。

```
remark #25444: Loopnest Interchanged: ( 1 2 ) --> ( 2 1 )
remark #15542: loop was not vectorized: inner loop was already vectorized
```

现在它仅遍历数组一次，对每个元素执行累加循环。此处缓存大小的确无关紧要。

为防止这种循环交换，您可以尝试使内层循环复杂化以阻碍编译器分析。例如可将数组转换为需要遍历的链表结构：

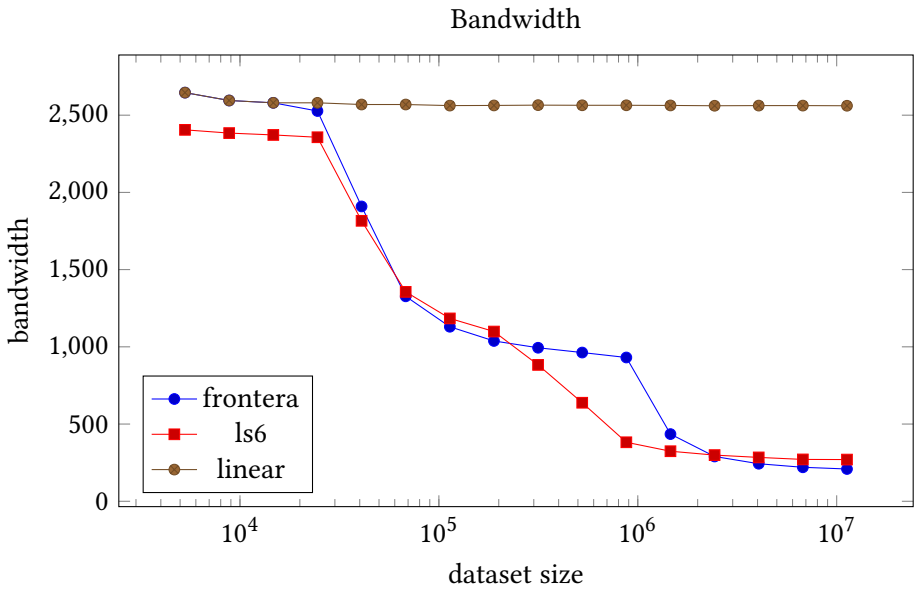


图 6.4：单核带宽随数据集大小的变化关系

```
// setup
for (int iword=0; iword<cachesize_in_words; iword++)
    memory[iword] = (iword+1) % cachesize_in_words

// use:
ptr = 0
for (int iword=0; iword<cachesize_in_words; iword++)
    ptr = memory[ptr];
```

此时编译器不会交换循环顺序，但你仍无法观察到缓存大小阈值。其原因是：在规律访问模式下，内存预取器 (章节 1.3.6) 会介入：CPU 的某个组件会预测你接下来要请求的地址，并提前获取这些数据。

To stymie this bit of cleverness you need to make the linked list more random:

```
for (int iword=0; iword<cachesize_in_words; iword++)
    memory[iword] = random() % cachesize_in_words
```

若给定足够大的缓存容量，这将形成一个遍历所有数组位置的循环，或者你也可以通过生成索引位置的全排列来显式确保这一点。

练习 6.2. 虽然刚才概述的策略能够证明缓存容量的存在，但它无法报告缓存支持的最大带宽。问题出在哪里？该如何解决？

6.4.3 详细时序分析

若我们能访问周期精确计时器或硬件计数器，便可实际绘制每次访问所需的周期数。

Such a plot is given in figure 6.5. The full code is given in section 25.2.

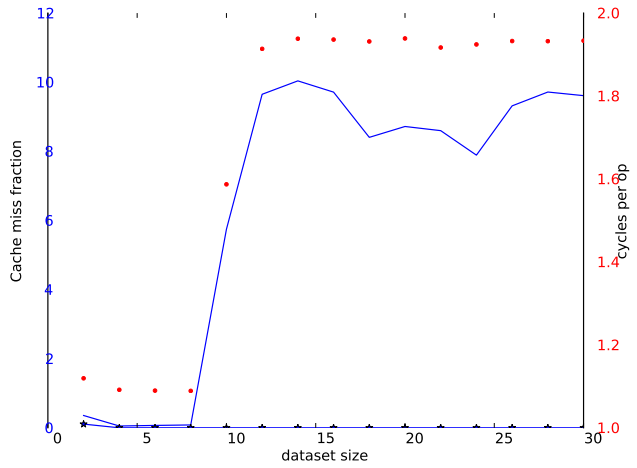


图 6.5：单次操作平均周期数随数据集大小的变化趋势。

6.4.4 TLB

如章节 1.3.9.2 所述，转译后备缓冲器（ TLB ）维护了一个高频使用内存页及其位置的小型列表；访问位于这些页上的数据远比访问非列表内数据更快。因此，编程时应尽量控制访问的页数量。

考虑以两种不同方式遍历二维数组元素的代码。

```
#define INDEX(i,j,m,n) i+j*m
array = (double*) malloc(m*n*sizeof(double));

/* traversal #1 */
for (j=0; j<n; j++)
  for (i=0; i<m; i++)
    array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;

/* traversal #2 */
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;
```

实验结果（源代码见附录 25.5 ）绘制于图 6.7 与 6.6 中。

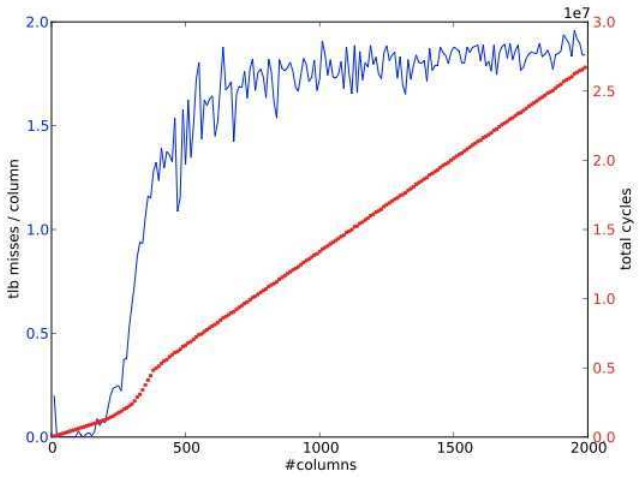


图 6.6：按列遍历数组时，每列的 TLB 失效次数随列数变化的函数关系。

U使用 $m = 1000$ 意味着，在 *AMD Opteron* 处理器（其页面大小为 512 个双精度浮点数）上，我们大约需要两个p页面来存储每列数据。我们运行此示例时，绘制了 ‘TLB 失效次数’，即t页面被引用但未记录在 TLB 中的次数。

1. 在第一次遍历中确实如此。当我们访问一个元素后，TLB 会记录其所在页面，随后该页面的其他元素都会被使用，因此不会发生进一步的 TLB 失效。图 6.6 显示，随着 n 的增加，每列的 TLB 失效次数大约为两次。2. 在第二次遍历中，我们为第一行的每个元素访问一个新页面。第二行的元素位于这些页面上，因此只要列数少于 TLB 条目数，这些页面仍会被记录在 TLB 中。随着列数增长，TLB 失效次数增加，最终每次元素访问都会产生一次 TLB 失效。图 6.7 表明，当列数足够大时，每列的 TLB 失效次数等于每列的元素数量。

6.5 缓存关联性

许多算法通过递归分解问题来工作，例如快速傅里叶变换（*FFT*）算法。因此，这类算法的代码通常会对长度为 2 的幂次的向量进行操作。遗憾的是，这可能与 CPU 的某些架构特性产生冲突，其中许多特性本身就涉及 2 的幂次。

在章节 1.3.5.9 中，你已了解如何对少量向量进行加法运算

$$\forall_j : y_j = y_j + \sum_{i=1}^{m} x_{i,j}$$

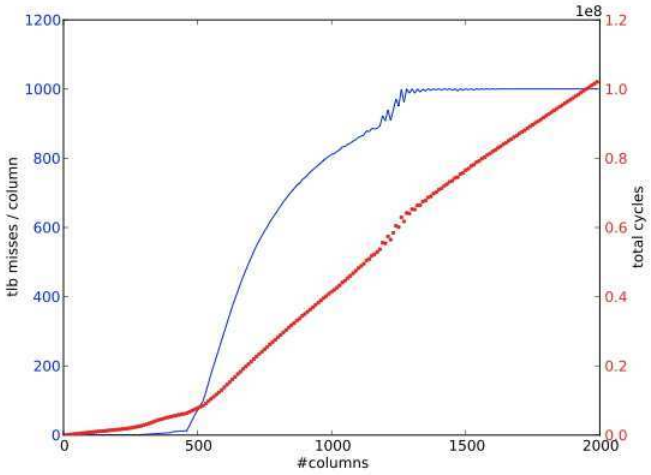


Figure 6.7: Number of TLB misses per column as function of the number of columns; rowwise traversal of the array.

对于直接映射缓存或具有关联度的组相联缓存来说是一个问题。

我们将其简化为一个代码，该代码循环遍历同一关联等价类中的多个位置：

```
// hardware/associativity.cpp
auto data = write_data.get_stream().data();
for ( int irepeat=0; irepeat<how_many_repeats; irepeat++ )
  for ( int loc=0; loc<data.size(); loc+=displacement_in_words )
    data[loc] += loc;
```

其中数据大小已计算为可容纳长度为 `assoc` 的循环：

```
// hardware/associativity.cpp
auto datasize_in_bytes = displacement_in_bytes * assoc;
auto datasize_in_words = datasize_in_bytes / sizeof(floattype);
```

为评估关联性的影响，我们测量了不同代际英特尔处理器上每个元素的访问时间。图 6.8 展示了 英特尔 *Sky Lake* 与 英特尔 *Cascade Lake* 的性能，两者均配备 32KiB 、 8 路组关联的 L1 缓存，以及英特尔 *Ice Lake* 的 48KiB 、 12 路组关联 L1 缓存。

6.6 循环嵌套

若代码中存在嵌套循环，且外层循环的迭代相互独立，则可自主选择将哪个循环设为外层、哪个设为内层。

习题 6.3. 举例说明可交换顺序的双重嵌套循环；再举一个不可交换的例子。尽可能使用本书中的实际案例。

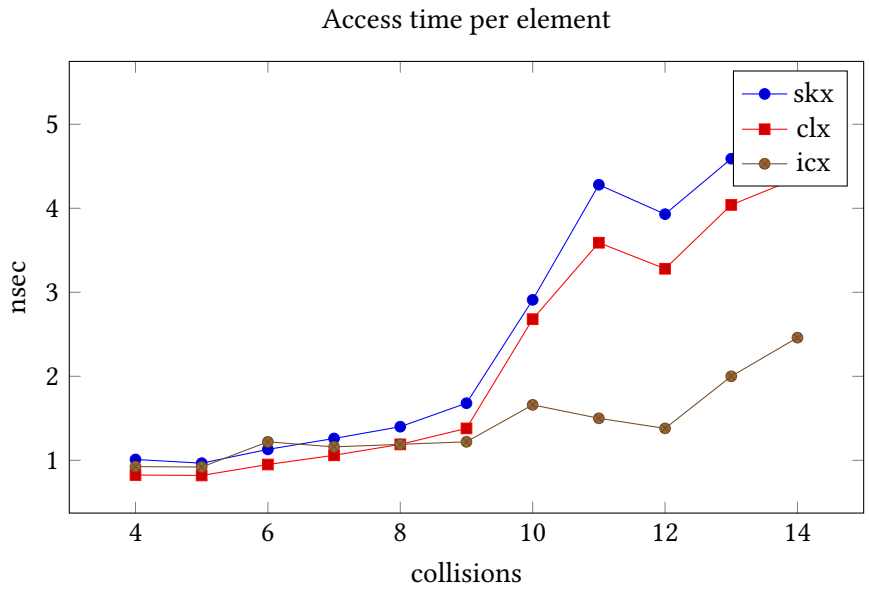


图 6.8：缓存关联性的影响

如果你有选择权，许多因素都可能影响你的决定。

编程语言：***C*** 与 ***Fortran*** 如果你的循环描述的是二维数组的 (i,j) 索引，通常最好让 i 索引在 Fortran 的内层循环中，而 j 索引在 C 的内层循环中。

练习 6.4. 你能提出至少两个为什么这可能对性能更好的原因吗?

然而，这并不是一成不变的规则。它可能取决于循环的大小以及其他因素。例如，在矩阵 - 向量乘法中，改变循环顺序会影响输入和输出向量的使用方式。

并行模型若想用 *OpenMP* 并行化循环，通常需要外层循环大于内层循环。外层循环过短绝对不利，而短内层循环常可被编译器向量化。但若目标设备是 *GPU*，则需将大循环置于内层。并行工作单元应避免分支或嵌套循环。

循环顺序在 OpenMP 中的其他影响详见《并行程序设计》第 *19.6.2* 节

6.7 循环分块

某些情况下，将循环拆分为两个嵌套循环（外层遍历迭代空间中的块，内层遍历块内元素）可提升性能，此技术称为循环分块：（短）内层循环即分块单元，多个连续分块构成完整迭代空间。

例如

```
for (i=0; i<n; i++)...
```

变为

```
bs = ...       /* the blocksize */
nblocks = n/bs /* assume that n is a multiple of bs */
for (b=0; b<nblocks; b++)
for (i=b*bs,j=0; j<bs; i++,j++)...
```

对于单个循环而言，这可能不会产生任何差异，但在合适的上下文中可能会有所不同。例如，如果一个数组被重复使用，但其大小超过了缓存容量：

```
for (n=0; n<10; n++)
  for (i=0; i<100000; i++)
    ... = ...x[i] ...
```

那么循环分块可能导致数组被分割成适合放入缓存的块：

```
bs = ... /* the blocksize */
for (b=0; b<100000/bs; b++)
  for (n=0; n<10; n++)
    for (i=b*bs; i<(b+1)*bs; i++)
      ... = ...x[i] ...
```

因此，循环分块也被称为 *cacheblocking*。块大小取决于循环体中访问的数据量；理想情况下，您会尝试让数据在 L1 缓存中重用，但也可以针对 L2 重用进行分块。当然，L2 重用不会像 L1 重用那样带来高性能。

练习 6.5. 分析此示例。 x 何时被载入缓存，何时被重用，何时被清除？此示例中所需的缓存大小是多少？使用常量重写此示例，

```
#define L1SIZE 65536
```

举个稍复杂的例子，我们来看 *matrix transposition* $A \leftarrow B^t$。通常你会遍历输入和输出矩阵：

```
// regular.c
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    A[i][j] = B[j][i];
```

采用分块技术后变为：// blocked.c

```
for (int ii=0; ii<N; ii+=blocksize)
for (int jj=0; jj<N; jj+=blocksize)
for (int i=ii*blocksize; i<MIN(N,(ii+1)*blocksize); i++)
```

```
for (int j=jj*blocksize; j<MIN(N,(jj+1)*blocksize); j++)
  A[i][j] = B[j][i];
```

与上述示例不同，输入和输出的每个元素仅被访问一次，因此不存在直接的重用。然而，缓存行存在重用情况。

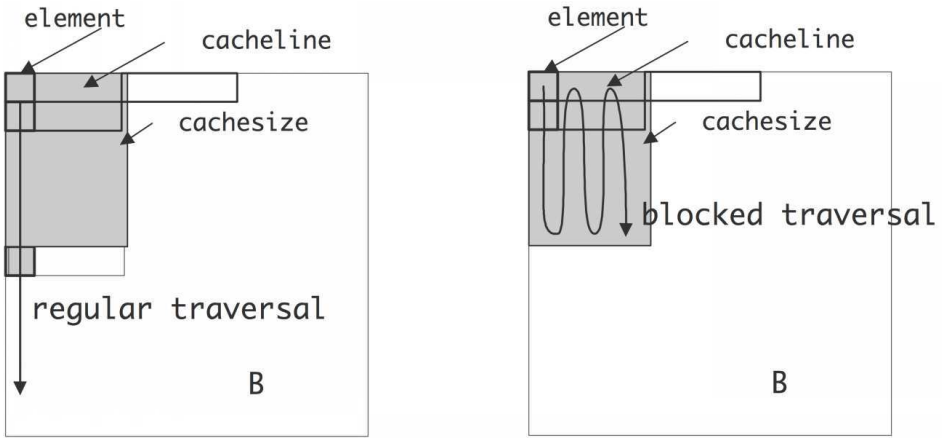


图 6.9：矩阵的常规遍历与分块遍历对比。

图 6.9 展示了矩阵如何以不同于存储顺序的方式遍历，例如按列访问而实际按行存储。这导致每次元素加载都会传输一整条缓存行，但其中只有一个元素被立即使用。在常规遍历中，这种连续的缓存行流会迅速溢出缓存，无法实现重用。而在分块遍历中，只需遍历少量缓存行后就会用到这些行的下一个元素，因此实现了缓存行的重用，即空间局部性。

通过分块提升性能的最重要案例是矩阵！矩阵乘积！分块。在 1.6.2 节中我们讨论了矩阵乘法，并指出缓存中几乎无法保留数据。通过循环分块技术，我们可以改善这一状况。例如，标准写法中该乘积

```
for i=1..n
  for j=1..n
    for k=1..n
      c[i,j] += a[i,k]*b[k,j]
```

只能优化以保持 `c[i,j]` 在寄存器中：

```
for i=1..n
  for j=1..n
    s = 0
    for k=1..n
      s += a[i,k]*b[k,j]
    c[i,j] += s
```

通过循环分块技术，我们可以将 `a[i,:]` 的部分保留在缓存中，前提是 `a` 按行存储：

```
for kk=1..n/bs
  for i=1..n
    for j=1..n
      s = 0
      for k=(kk-1)*bs+1..kk*bs
        s += a[i,k]*b[k,j]
      c[i,j] += s
```

6.8 优化策略

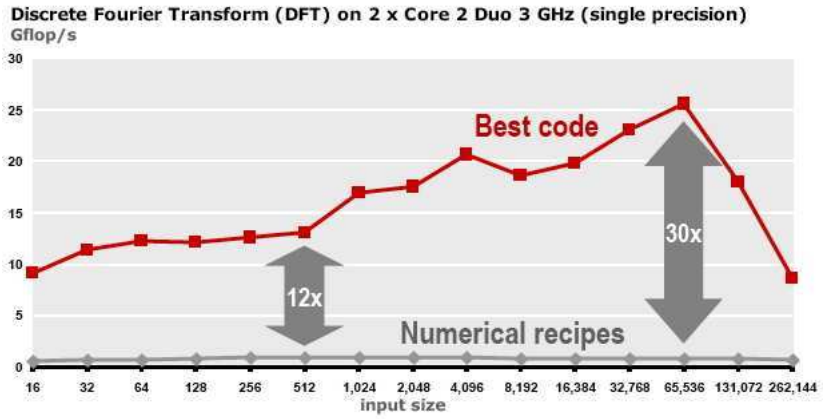


图 6.10：离散傅里叶变换的原始实现与优化实现的性能对比 .

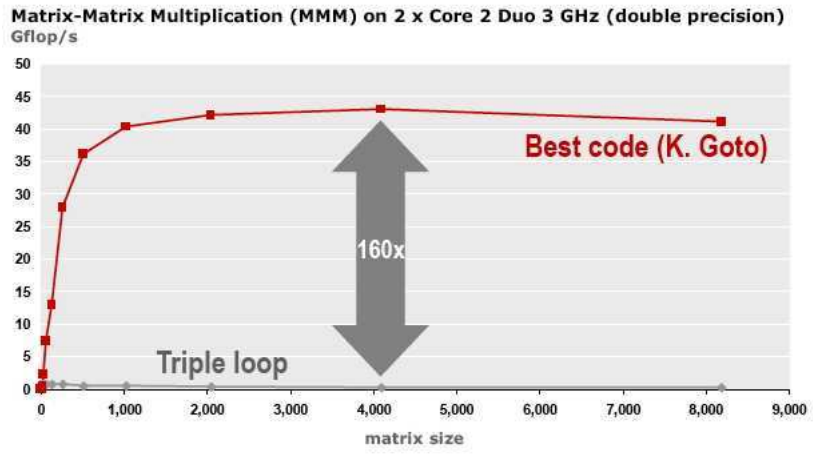


图 6.11：矩阵乘法运算的原始实现与优化实现的性能对比

Figures 6.10 和 6.11 表明，某项操作的原始实现（有时称为 “ 参考实现 ” ）与优化实现的性能可能存在巨大差异。遗憾的是，优化实现并不容易获得。一方面，由于它们依赖于

在分块处理时，其循环嵌套深度会翻倍：矩阵 - 矩阵乘法会变成一个六层嵌套循环。此时，最优分块大小取决于目标架构等因素。

我们得出以下观察结论：

- 编译器无法提取接近最优性能的代码 1 。
- 存在自动调优项目，用于自动生成针对特定架构优化的实现。这种方法可能取得中等乃至显著的成功。其中最著名的项目包括针对 Blas 内核的 Atlas [190]，以及针对变换的 Spiral [163] 。

6.9 缓存感知与缓存无关编程

与寄存器和主内存不同（二者均可通过汇编代码寻址），缓存的使用是隐式的。程序员无法显式地将数据加载到特定缓存中，即便使用汇编语言也无法实现。

然而，以“缓存感知”的方式编写代码是可行的。假设某段代码反复操作的数据量小于缓存容量。我们可以认为首次访问数据时，数据会被载入缓存；后续访问时数据已存在于缓存中。反之，若数据量超过缓存容量 2 ，在访问过程中数据将部分或全部被挤出缓存。

我们可以通过实验验证这一现象。使用高精度计时器时，代码片段

```
for (x=0; x<NX; x++)
  for (i=0; i<N; i++)
    a[i] = sqrt(a[i]);
```

所耗时间将与 `N` 呈线性关系，直至 `a` 填满缓存。更直观的方法是计算标准化时间，即内循环单次执行耗时：

```
t = time();
for (x=0; x<NX; x++)
  for (i=0; i<N; i++)
    a[i] = sqrt(a[i]);
t = time()-t;
t_normalized = t/(N*NX);
```

归一化时间将保持恒定，直到数组 `a` 填满缓存，随后时间增加并最终再次趋于平稳。（详见章节 6.4.1 的详细讨论。）

原因在于，只要 `a[0]...a[N-1]` 能放入 L1 缓存，内层循环将使用 L1 缓存中的数据。此时访问速度由 L1 缓存的延迟和带宽决定。当数据量超过 L1 缓存容量时，部分或全部数据将从 L1 缓存中清除，性能将由 L2 缓存的特性决定。若数据量进一步增大，性能将再次下降至由主存带宽决定的线性行为。

1. 向编译器提供参考实现仍可能实现高性能，因为某些编译器经过训练能识别此操作。它们会跳过翻译步骤，直接替换为优化后的变体。2. 此处我们暂且忽略组相联性问题，基本假设采用全相联缓存。

若已知缓存大小，在上述情况下可对算法进行优化以充分利用缓存。然而，不同处理器的缓存大小各异，这将导致代码丧失可移植性，或至少其高性能特性无法跨平台。此外，针对多级缓存的区块划分也极为复杂。基于这些原因，部分研究者提倡缓存无关编程 [68]。

缓存无关编程可被描述为一种能自动利用缓存层级结构中所有级别的编程方法。其典型实现方式是采用分治策略，即通过问题的递归细分来完成。

缓存无关编程的一个简单示例是矩阵转置操作 $B \leftarrow A^t$。首先我们观察到两个矩阵的每个元素仅被访问一次，因此唯一的复用机会在于缓存行的利用。若两个矩阵均按行存储且我们按行遍历 B，则 A 需按列遍历，此时每个被访问元素都会加载一个缓存行。若行数乘以每缓存行元素数超过缓存容量，缓存行将在被复用前就被置换出。

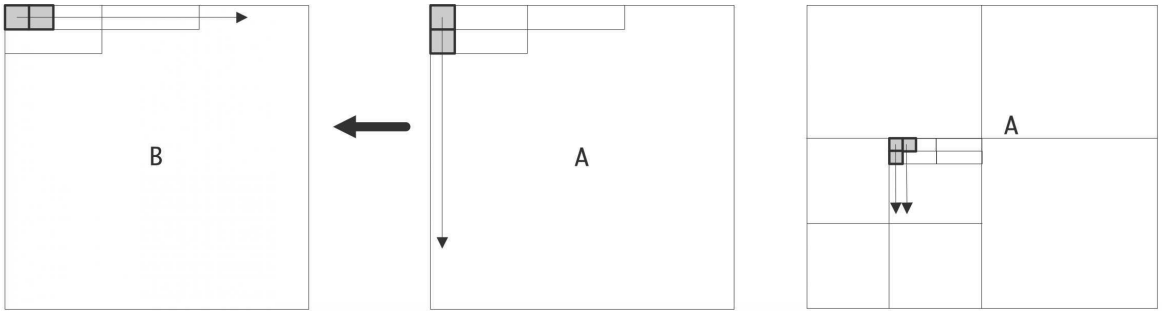


图 6.12：矩阵转置操作，包括对源矩阵的简单遍历与递归遍历 .

在缓存无关的实现中，我们将 A 和 B 划分为 2×2 块矩阵，并递归计算 $B_{11} \leftarrow A_{11}^t$、$B_{12} \leftarrow A_{21}^t$ 等；参见图 6.12。在递归的某个阶段，块 A_{ij} 会变得足够小以适应缓存，而 A 的缓存行将被充分利用。因此，该算法相比简单算法的提升因子等于缓存行大小。

缓存无关策略通常能带来性能提升，但它不一定是最优的。在矩阵 - 矩阵乘积中，它优于朴素算法，但不及专门为优化缓存使用而设计的算法 [82]。

关于此类技术在模板计算中的应用讨论，参见章节 7.8.4 。

6.10 案例研究：矩阵 - 向量乘积

让我们详细探讨矩阵 - 向量乘积

$$\forall_{i,j} : y_i \leftarrow a_{ij} \cdot x_j$$

这涉及对 $n^2 + 2n$ 个数据项进行 $2n^2$ 次操作，因此重用率为 $O(1)$：内存访问与操作次数处于同一量级。但需注意此处存在双重循环，且 x、 y 向量仅含单一下标，故其中每个元素会被多次使用。

利用这种理论上的重用并非易事。在

```
/* variant 1 */
for (i)
  for (j)
    y[i] = y[i] + a[i][j] * x[j];
```

元素 y[i] 似乎被重用了。然而，此处给出的语句会在每次内部迭代中将 y[i] 写入内存，因此我们必须将循环改写为

```
/* variant 2 */
for (i) {
  s = 0;
  for (j)
    s = s + a[i][j] * x[j];
  y[i] = s;
}
```

以确保重用。此变体使用了 $2n^2$ 次加载和 n 次存储。这一优化很可能会由编译器自动完成。

此代码片段仅显式利用了 y 的重用。若缓存过小无法容纳整个向量 x 加上 a 的一列，则 x 的每个元素在每次外层迭代中仍会被重复加载。若将循环顺序反转

```
/* variant 3 */
for (j)
  for (i)
    y[i] = y[i] + a[i][j] * x[j];
```

可显露出 x 的重用特性，特别是若我们将其改写为

```
/* variant 3 */
for (j) {
  t = x[j];
  for (i)
    y[i] = y[i] + a[i][j] * t;
}
```

但此时 y 不再被重用。此外，现在需要进行 $2n^2+n$ 次加载（与变体 2 相当），但 n^2 次存储操作，其数量级更高。

I要实现 x 和 y的双重重用是可能的，但这需要更复杂的编程技巧。关键在于
i将循环分割成块的方法。例如：

```
for (i=0; i<M; i+=2) {
  s1 = s2 = 0;
  for (j) {
    s1 = s1 + a[i][j] * x[j];
    s2 = s2 + a[i+1][j] * x[j];
  }
  y[i] = s1; y[i+1] = s2;
}
```

这也被称为循环展开，或条带挖掘。循环展开的程度由可用寄存器的数量决定。

第 7 章

高性能线性代数

本节将探讨并行计算机上线性代数的若干关键问题。我们将以现实视角切入，假设处理器数量有限，且问题数据规模始终远大于处理器数量。同时会重点关注处理器间通信网络的物理特性。

我们将分析各类线性代数运算，包括线性方程组求解的迭代方法（若简称 ' 迭代方法 ' 时，默认特指线性方程组情形），以及这些方法在有限带宽和有限连通性网络环境中的表现。

7.1 集体操作

集体操作是分布式内存并行架构的特有产物（相关讨论见 2.4 节）。数学上它们描述的是极其简单的操作，例如对数字数组求和。然而由于这些数字可能分布在具有独立内存空间的不同处理器上，该操作在计算层面变得非平凡，值得深入研究。

集体操作在线性代数运算中扮演着重要角色。事实上，即使是矩阵 - 向量乘积这类简单操作的扩展性，也可能取决于这些集体操作的成本，下文将详细阐述。我们首先简要讨论集体操作背后的核心理念；具体细节可参考 [29] 。随后我们将在第 7.2 节及其他多处看到其重要应用。

计算集体操作成本时，三个架构常数足以给出下界：α（发送单条消息的延迟）、β（数据传输带宽的倒数，参见第 1.3.2 节），以及 γ（计算速率的倒数，即单次算术运算耗时）。发送 n 个数据项耗时 $\alpha+\beta n$。我们进一步假设处理器每次只能发送一条消息，且不限定处理器间的连接拓扑（相关讨论见第 2.7 节），因此此处推导的下界适用于多种架构。

上述架构模型的主要含义在于，算法的每一步中活跃处理器数量只能翻倍。例如，进行广播时，首先处理器 0 向 1 发送数据，接着 0 和 1 可向 2 与 3 发送，随后 0–3 向 4–7 发送，依此类推。这种消息级联被称为 *minimum*

spanning tree 处理器网络的，因此任何集合算法至少需要承担与累积延迟相关的 $\alpha \log_2 p$ 成本。

7.1.1 广播

在 *broadcast* 操作中，单个处理器拥有 n 个数据元素需要发送给所有其他处理器：其他处理器需要完整复制所有 n 个元素。根据上述倍增原理，我们得出结论：向 p 个处理器进行广播至少需要 $\lceil \log_2 p \rceil$ 步时间，总延迟为 $\lceil \log_2 p \rceil \alpha$。由于发送了 n 个元素，这至少还需要 $n\beta$ 时间让所有元素离开发送处理器，因此总成本下限为

$$\lceil \log_2 p \rceil \alpha + n\beta.$$

我们可以通过以下方式说明生成树方法：

	$t=1$	$t=2$	$t=3$
p_0	$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	x_0, x_1, x_2, x_3
p_1		$x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$	x_0, x_1, x_2, x_3
p_2			x_0, x_1, x_2, x_3
p_3			x_0, x_1, x_2, x_3

（在 $t=1$ 上，p_0 发送给 p_1；在 $t=2$ p_0 上，p_1 发送给 p_2, p_3。）该算法具有正确的对数 $_2\, p \cdot \alpha$ 项，但处理器 0 会重复发送整个向量，因此带宽开销为对数 $_2\, p \cdot n\beta$。若 n 较小，延迟开销将占主导地位，故可将其归类为短向量集合操作

T以下算法通过组合散播与桶式传递来实现广播
*a*算法。首先是散播阶段：

	$t=0$	$t=1$	$t=2$	$t=3$
p_0	$x_0\downarrow, x_1, x_2, x_3$	$x_0, x_1\downarrow, x_2, x_3$	$x_0, x_1, x_2\downarrow, x_3$	$x_0, x_1, x_2, x_3\downarrow$
p_1		x_1		
p_2			x_2	
p_3				x_3

这涉及 $p-1$ 条大小为 N/p 的消息，总耗时为

$$T_{\text{scatter}}(N,P) = (p-1)\alpha + (p-1)\cdot \frac{N}{p}\cdot \beta.$$

于是，桶链式传递让每个处理器在每一步都保持活跃状态，接收部分消息（第一步除外），并将其传递给下一个处理器。

	$t=0$	$t=1$	*etcetera*
p_0	$x_0\downarrow$	$x_0 \quad x_3\downarrow$	$x_0, \quad x_2, x_3$
p_1	$x_1\downarrow$	$x_0\downarrow, x_1$	$x_0, x_1, \quad x_3$
p_2	$x_2\downarrow$	$x_1\downarrow, x_2$	x_0, x_1, x_2
p_3	$x_3\downarrow$	$x_2\downarrow, x_3$	x_1, x_2, x_3

每条部分消息会被发送 $p-1$ 次，因此此阶段的复杂度同样为

$$T_{\text{bucket}}(N,P) = (p-1)\alpha + (p-1)\cdot \frac{N}{p}\cdot beta.$$

此时复杂度变为

$$2(p-1)\alpha + 2\beta n(p-1)/p$$

这在延迟上并非最优，但当 n 较大时是更优算法，使其成为长向量集体操作。

7.1.2 归约

在归约操作中，每个处理器拥有 n 个数据元素，需由一个处理器按元素进行组合，例如计算 n 个求和或乘积。

通过逆向推演广播操作，我们发现归约运算在通信复杂度上同样存在 $\lceil \log_2 p\rceil\alpha + n\beta$ 的下界。归约运算还涉及计算环节，若串行执行总耗时为 $(p-1)\gamma n$：即 n 个数据项需在 p 个处理器上完成归约。由于这些操作可能并行化，计算环节的下界为 $\frac{p-1}{p}\gamma n$，因此总耗时为

$$\lceil \log_2 p\rceil\alpha + n\beta + \frac{p-1}{p}\gamma n.$$

W我们以 $x_i^{(j)}$ 表示最初数据项 i 的标记，以此阐述生成树算法
o在处理器 j 上，$x_i^{(j:k)}$ 表示处理器 $j\ldots k$ 中各数据项 i 的求和结果

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$	$x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)}\uparrow, x_1^{(1)}\uparrow, x_2^{(1)}\uparrow, x_3^{(1)}\uparrow$		
p_2	$x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$	$x_0^{(2:3)}\uparrow, x_1^{(2:3)}\uparrow, x_2^{(2:3)}\uparrow, x_3^{(2:3)}\uparrow$	
p_3	$x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}\uparrow, x_3^{(3)}\uparrow$		

在时间点 $t=1$，处理器 p_0 和 p_2 分别从 p_1 和 p_3 接收数据；而在 $t=2$ p_0 时刻，处理器 p_2 接收数据

As with the broadcast above, this algorithm does not achieve the lower bound; instead it has a complexity

$$\lceil \log_2 p\rceil(\alpha + n\beta + \frac{p-1}{p}\gamma n).$$

对于短向量而言，α 项占主导地位，因此该算法已足够高效。针对长向量，则可如前所述采用其他算法 [29]。

一个长向量归约可通过桶式接力后接聚集操作完成。其复杂度与前述相同，只是桶式接力会执行部分归约，耗时 $\gamma(p-1)N/p$。聚集操作不再进行额外计算。

7.1.3 全归约

全归约操作会计算每个处理器上 n 元素的相同逐项归约结果，但将结果保留在每个处理器上，而不仅仅是生成树的根节点。这可以通过先归约后广播的方式实现，但存在更精妙的算法。

	$t=1$	$t=2$	$t=3$
p_0	$x_0^{(0)}\downarrow, x_1^{(0)}\downarrow, x_2^{(0)}\downarrow, x_3^{(0)}\downarrow$	$x_0^{(0:1)}\downarrow\downarrow, x_1^{(0:1)}\downarrow\downarrow, x_2^{(0:1)}\downarrow\downarrow, x_3^{(0:1)}\downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_1	$x_0^{(1)}\uparrow, x_1^{(1)}\uparrow, x_2^{(1)}\uparrow, x_3^{(1)}\uparrow$	$x_0^{(0:1)}\downarrow\downarrow, x_1^{(0:1)}\downarrow\downarrow, x_2^{(0:1)}\downarrow\downarrow, x_3^{(0:1)}\downarrow\downarrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_2	$x_0^{(2)}\downarrow, x_1^{(2)}\downarrow, x_2^{(2)}\downarrow, x_3^{(2)}\downarrow$	$x_0^{(2:3)}\uparrow\uparrow, x_1^{(2:3)}\uparrow\uparrow, x_2^{(2:3)}\uparrow\uparrow, x_3^{(2:3)}\uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$
p_3	$x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}\uparrow, x_3^{(3)}\uparrow$	$x_0^{(2:3)}\uparrow\uparrow, x_1^{(2:3)}\uparrow\uparrow, x_2^{(2:3)}\uparrow\uparrow, x_3^{(2:3)}\uparrow\uparrow$	$x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$

allreduce 操作的成本下限，令人惊讶的是，几乎与简单规约操作相同：因为在规约过程中并非所有处理器同时处于活跃状态，我们假设额外的工作可以被完美地分摊。这意味着延迟和计算的下限保持不变。

由于每个处理器都生成一个最小生成树，我们得到 $\log_2 p\alpha$ 延迟。对于带宽，我们的推理如下：为了使通信完全并行化，$\frac{p-1}{p}n$ 个数据项必须到达并离开每个处理器。因此，我们得到的总时间为

$$\lceil \log_2 p \rceil \alpha + 2\frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma.$$

allreduce 是线性系统迭代方法中的重要操作；例如参见第 5.5.9 节。在那里，规约通常应用于单个标量，这意味着延迟相对重要。

7.1.4 全收集 (Allgather)

在针对收集 *(gather)* 操作中，每个处理器拥有 n 个元素，一个处理器收集所有元素而不像规约那样进行合并。全收集 *(allgather)* 执行相同的收集操作，但将结果保留在所有处理器上。您将在稠密矩阵 - 向量乘积（第 7.2.3.3 节）和稀疏矩阵 - 向量乘积的初始化（第 7.5.6 节）中看到其两个重要应用。

我们再次假设多目标聚集操作是同时激活的。由于每个处理器都生成一棵最小生成树，因此存在 $\log_2 p\alpha$ 的延迟；由于每个处理器从 $p-1$ 个处理器接收 n/p 个元素，故存在 $(p-1)\times(n/p)\beta$ 的带宽开销。最终，通过 allgather 操作构建长度为 n 的向量总成本为

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n\beta.$$

我们通过以下示例说明:

$$\left|\begin{array}{c|ccc} & t=1 & t=2 & t=3 \\ p_0 & x_0\downarrow & x_0x_1\downarrow & x_0x_1x_2x_3 \\ p_1 & x_1\uparrow & x_0x_1\downarrow & x_0x_1x_2x_3 \\ p_2 & x_2\downarrow & x_2x_3\uparrow & x_0x_1x_2x_3 \\ p_3 & x_3\uparrow & x_2x_3\uparrow & x_0x_1x_2x_3 \end{array}\right|$$

在时间 $t=1$，邻居 p_0、 p_1 之间发生交换，同样地 p_2、 p_3 之间也发生交换；在 $t=2$ 时刻，p_0、 p_2 之间以及 p_1、 p_3 之间会进行跨度为二的远程交换。

7.1.5 归约 - 散射

在归约 - 散射操作中，每个处理器拥有 n 个元素，并对它们进行 n 路归约。与归约或全归约不同，结果会被拆分，并像散射操作那样分发。

形式上，处理器 i 持有数据项 $x_i^{(i)}$，且需要 $\sum_j x_i^{(j)}$ 。可通过执行规模为 p 的归约操作，将向量 $(\sum_i x_0^{(i)}, \sum_i x_1^{(i)}, \ldots)$ 收集到一个处理器上，再散射结果来实现。但也可以采用所谓的双向交换算法合并这些操作:

$$\left|\begin{array}{c|ccc} & t=1 & t=2 & t=3 \\ p_0 & x_0^{(0)}, x_1^{(0)}, x_2^{(0)}\downarrow, x_3^{(0)}\downarrow & x_0^{(0:2:2)}, x_1^{(0:2:2)}\downarrow & x_0^{(0:3)} \\ p_1 & x_0^{(1)}, x_1^{(1)}, x_2^{(1)}\downarrow, x_3^{(1)}\downarrow & x_0^{(1:3:2)}\uparrow, x_1^{(1:3:2)} & x_1^{(0:3)} \\ p_2 & x_0^{(2)}\uparrow, x_1^{(2)}\uparrow, x_2^{(2)}, x_3^{(2)} & x_2^{(0:2:2)}, x_3^{(0:2:2)}\downarrow & x_2^{(0:3)} \\ p_3 & x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}, x_3^{(3)} & x_0^{(1:3:2)}\uparrow, x_1^{(1:3:2)} & x_3^{(0:3)} \end{array}\right|$$

归约 - 散射可视为反向运行的全收集操作并附加算术运算，因此其成本为

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} n(\beta+\gamma).$$

7.2 并行稠密矩阵 - 向量乘积

本节我们将深入探讨并行稠密矩阵 - 向量乘积的性能表现，特别是其可扩展性。首先我们将分析一个简单案例，并从并行化角度进行一定程度的详细讨论。

7.2.1 实现块行分解方案

在设计算法的并行版本时，通常需要对涉及的对象进行数据分解。对于矩阵 - 向量操作（如乘积 $y = Ax$），我们可以选择从向量分解入手，探究其对矩阵分解方式的影响；或者从矩阵本身出发，推导出相应的向量分解方式。本例中，从矩阵而非向量开始分解似乎更为自然，因为矩阵往往具有更大的计算意义。此时我们面临两种选择：

1. 对矩阵进行一维分解，将其划分为块行或块列，并将这些块（或其组合）分配给不同处理器。2. 或者进行二维分解，为每个处理器分配一个或多个通用子矩阵。

我们首先考虑块行分解。假设处理器 p 及其拥有的行索引集合 I_p[1]，令 $i \in I_p$ 表示分配给该处理器的某一行。在运算过程中会用到行 i 中的元素：

$$y_i = \sum_j a_{ij} x_j$$

我们现在进行推理：

- 如果处理器 p 拥有所有 x_j 值，矩阵 - 向量乘积可以简单地执行，完成后，处理器将获得正确的 y_j 值 $j \in I_p$。
- 这意味着每个处理器都需要保留 x 的副本，这是浪费的。同时也引发了数据完整性问题：你需要确保每个处理器拥有正确的 x 值。
- 在某些实际应用中（例如之前提到的迭代方法），
  矩阵 - 向量乘积的输出直接或间接地成为下一次矩阵 - 向量运算的输入。对于计算 x、Ax、A^2x、... 的幂方法而言，情况确实如此。由于我们的操作开始时每个处理器拥有完整的 x，但结束时仅拥有 Ax 的本地部分，这就产生了不匹配。
- 或许更好的假设是：在操作开始时，每个处理器仅拥有 x 的本地部分，即那些满足 $i \in I_p$ 条件的 x_i，这样算法的初始状态和结束状态就能保持一致。这意味着我们需要修改算法，加入一些通信机制，使每个处理器能获取满足 $i \notin I_p$ 条件的 x_i 值。

练习 se7.1. 针对矩阵按块行分解的情况进行类似推理
块列分解的情况。详细描述并行算法，如同上文所述，但无需给出伪代码。

现在让我们详细考察通信过程：假设固定处理器 p，分析其执行的操作及引发的通信。根据上述分析，在执行语句 $y_i = \sum_j a_{ij} x_j$ 时需明确 j 值 ' 所属 ' 的处理器。为此我们表示为

$$y_i = \sum_{j \in I_p} a_{ij} x_j + \sum_{j \notin I_p} a_{ij} x_j \tag{7.1}$$

1. For ease of exposition we will let I_p be a contiguous range of indices, but any general subset is allowed.

输入： 处理器编号 p；元素 x_i 与 $i \in I_p$；矩阵元素 A_{ij} 与 $i \in I_p$。
输出： 元素 y_i 与 $i \in I_p$ **对于** $i \in I_p$ **执行** $s \leftarrow 0$；**对于** $j \in I_p$ **执行** $s \leftarrow s + a_{ij}x_j$ **对于** $j \notin I_p$ **执行**发送 x_j 从拥有它的处理器到当前处理器，然后；$s \leftarrow s + a_{ij}x_jy_i \leftarrow s$**过程**朴素并行 MVP$(A, x_{local}, y_{local}, p)$

图 7.1：一种简单编码的并行矩阵 - 向量乘积。

如果 $j \in I_p$，指令 $y_i \leftarrow y_i + a_{aij}x_j$ 仅涉及处理器本地已有的量。因此，让我们专注于 $j \notin I_p$ 的情况。如果能直接写出这样的语句就好了

```
y(i) = y(i) + a(i,j)*x(j)
```

而某些底层会自动传输 `x(j)` 从存储它的任意处理器到本地寄存器。（第 2.6.5 节的 PGAS 语言旨在实现这一点，但其效率远未得到保证。）基于这种乐观并行观点的实现如图 7.1 所示。

这种 ‘局部’ 方法的直接问题在于会产生过多的通信。

- 若矩阵 A 是稠密的，元素 x_j 对每一行 $i \in I_p$ 仅需一次，因此它将被每行 $i \in I_p$ 获取一次。
- 对于每个处理器 $q \neq p$，将有（大量）元素 x_j 与 $j \in I_q$ 需要从处理器 q 传输至 p。以单独消息而非批量传输方式进行此操作极其低效。

在共享内存环境下这些问题不大，但在分布式内存场景中，采用缓冲策略更为适宜。

不同于逐个通信 x 的元素，我们为每个处理器 $q \neq p$ 使用本地缓冲区 B_{pq} 来收集来自 q 的元素，这些元素用于在 p 上执行乘积运算。（图示参见图 7.2。）并行算法如图 7.3 所示。

除了防止元素被重复获取外，这种做法还能将许多小消息合并为一个大消息，通常效率更高；回顾我们在章节 2.7.8 中关于带宽与延迟的讨论。

练习 7.2. 使用非阻塞操作（章节 2.6.3.6 ）给出矩阵 - 向量乘法的伪代码

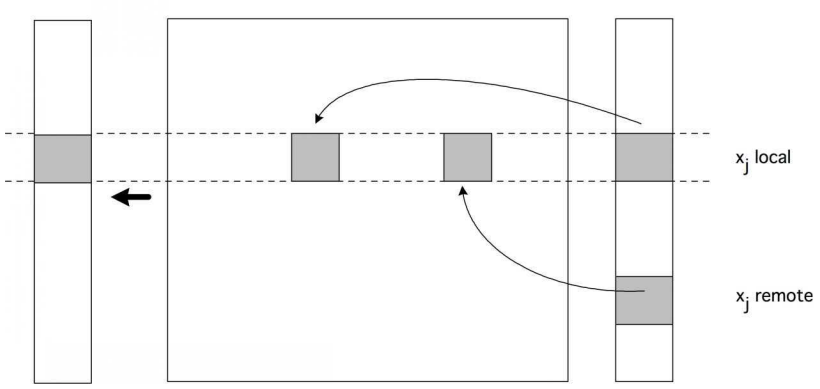


图 7.2：采用块行分布的并行矩阵 - 向量乘积示意图。

输入： 处理器编号 p；元素 x_i 与 $i \in I_p$；矩阵元素 A_{ij} 与 $i \in I_p$。

输出： 元素 y_i 与 $i \in I_p$

for $q \neq p$ **do**
    Send elements of x from processor q to p, receive in buffer B_{pq}.

$y_{local} \leftarrow Ax_{local}$
for $q \neq p$ **do**
    $y_{local} \leftarrow y_{local} + A_{pq}B_q$

Procedure Parallel MVP($A, x_{local}, y_{local}, p$)

图 7.3：并行矩阵 - 向量乘积的缓冲实现方案。

前文提到，在每个处理器上保留完整的 x 副本会造成存储空间的浪费。这里隐含的论点是，通常情况下我们不希望本地存储量随处理器数量增长：理想情况下它应仅与本地数据量相关。（这与弱可扩展性相关；参见章节 2.2.5。）

可见，由于通信因素的考量，我们实际上已认定每个处理器存储完整输入向量是不可避免的，或至少是更优的选择。这种空间效率与时间效率的权衡在并行编程中相当常见。对于稠密矩阵 - 向量积，我们确实可以为此开销辩护，因为向量存储量级低于矩阵存储，因此按比例计算我们的过量分配较小。下文（章节 7.5 ）将说明，稀疏矩阵 - 向量积的开销可能小得多。

显然，若忽略通信时间，前述并行稠密矩阵 - 向量积具有完美的加速比。接下来几节将表明，若考虑通信因素，上述块行实现并非最优方案。为实现可扩展性，我们需要二维分解。首先讨论集合通信。

7.2.2 块列情形

我们同样可以按块列分布矩阵。若已研究过块行情形，这并无太大差异：

- 我们再次从每个进程存储一组不同列开始；
- 且每个进程初始存储输入的对应部分；
- 乘积运算完成后，每个进程存储其对应的输出部分。

练习 7.3. 用伪代码勾勒算法。通信现在相对于计算发生在何处？你使用了哪种集合操作？（提示：每个进程仅需部分输出，而非整个向量。）

7.2.3 稠密矩阵 - 向量乘积的可扩展性

本节将全面分析并行计算 $y \leftarrow Ax$，其中 x、 $y \in \mathbb{R}^n$ 与 $A \in \mathbb{R}^{n\times n}$。我们假设将使用 p 个进程，但不对其连接性做任何假设。我们将看到矩阵的分布方式对算法扩展性有重大影响；原始研究可参阅 [99, 170, 178], ，各种扩展形式的定义见章节 2.2.5 。

7.2.3.1 矩阵 - 向量乘积，按行划分

划分

$$A \to \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix} \quad x \to \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}, \quad \text{and} \quad y \to \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix},$$

其中 $A_i \in \mathbb{R}^{m_i\times n}$ 与 x_i、 $y_i \in \mathbb{R}^{m_i}$，且 $\sum_{i=0}^{p-1} m_i = n$ 和 $m_i \approx n/p$。我们首先假设 A_i、 x_i 和 y_i 最初分配给 P_i。

该计算的特点是每个处理器都需要完整的向量 x，但自身仅持有其中的 n/p 部分。因此，我们执行一次 *allgather* 操作以收集 x。此后，处理器可执行本地乘积 $y_i \leftarrow A_i x$，之后无需进一步通信。

于是，针对 $y = Ax$ 的并行计算成本算法如下给出：

Step	成本（下限）
通过 Allgather 操作使 x 在所有节点上可用	$\lceil \log_2(p) \rceil \alpha + \frac{p-1}{p} n\beta \approx \log_2(p)\alpha + n\beta$
本地计算 $y_i = A_i x$	$\approx 2\frac{n^2}{p}\gamma$

成本分析 该算法的总成本近似为，

$$T_p(n) = T_p^{\text{1D-row}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + n\beta.}_{\text{Overhead}}$$

由于串行成本为 $T_1(n) = 2n^2\gamma$，加速比由下式给出

$$S_p^{\text{1D-row}}(n) = \frac{T_1(n)}{T_p^{\text{1D-row}}(n)} = \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \log_2(p)\alpha + n\beta} = \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}$$

并行效率则为

$$E_p^{\text{1D-row}}(n) = \frac{S_p^{\text{1D-row}}(n)}{p} = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}.$$

7.2.3.1.1 **乐观视角** 现在，若固定 p 并让 n 趋近无穷大，

$$\lim_{n\to\infty} E_p(n) = \lim_{n\to\infty} \left[\frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} \right] = 1.$$

因此，若能将问题规模扩大到足够大，最终并行效率将接近完美。然而，这假设了内存无限，故该分析不具备实际意义。

7.2.3.1.2 **悲观视角** 在强可扩展性 分析中，固定 n 并让 p 趋近无穷大，可得

$$\lim_{p\to\infty} E_p(n) = \lim_{p\to\infty} \left[\frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} \right] \sim \frac{1}{p} \downarrow 0.$$

因此，最终并行效率几乎消失殆尽。

7.2.3.1.3 **现实主义的视角** 在更现实的视角下，我们随着数据量的增加而增加处理器数量。这被称为弱可扩展性，它使得可用于存储问题的内存量随 p 线性扩展。

设 M 等于单个节点内存中可存储的浮点数数量，那么聚合内存由 Mp 给出。设 $n\max(p)$ 等于 p 个节点的聚合内存中可存储的最大问题规模。那么，如果所有内存都可用于矩阵存储，

$$(n_{\max}(p))^2 = Mp \quad \text{or} \quad n_{\max}(p) = \sqrt{Mp}.$$

现在的问题变为：对于可存储在 p 个节点上的最大问题规模，其并行效率如何：

$$E_p^{\text{1D-row}}(n_{\max}(p)) = \frac{1}{1+\frac{p\log_2(p)}{2(n_{\max}(p))^2}\frac{\alpha}{\gamma}+\frac{p}{2n_{\max}(p)}\frac{\beta}{\gamma}} = \frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}}.$$

现在，如果分析节点数量变大时的情况，会发现

$$\lim_{p\to\infty} E_p(n_{\max}(p)) = \lim_{p\to\infty} \left[\frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}} \right] \sim \frac{1}{\sqrt{p}} \downarrow 0.$$

因此，这种矩阵 - 向量乘法的并行算法同样不具备可扩展性。

如果仔细审视这个效率表达式，你会发现主要问题在于其中的 $1/\sqrt{p}$ 部分。这一项涉及系数 β，若逆向推导其来源，会发现它源自处理器间数据传输的时间。非正式地说，这表明消息规模过大导致问题难以扩展。实际上，消息大小恒定为 n，与处理器数量无关，而要实现可扩展性，该值可能需要降低。

另一种现实主义的认知是：完成计算存在一个时间上限 $T\max$。在最理想情况下（即通信开销为零时），我们能在 $T\max$ 时间内解决的最大问题规模由下式决定：

$$T_p(n_{\max}(p)) = 2\frac{(n_{\max}(p))^2}{p}\gamma = T_{\max}.$$

Thus

$$(n_{\max}(p))^2 = \frac{T_{\max}p}{2\gamma} \quad \text{or} \quad n_{\max}(p) = \frac{\sqrt{T_{\max}\sqrt{p}}}{\sqrt{2\gamma}}.$$

那么该算法针对可在 $T\max$ 时间内解决的最大问题所达到的并行效率为：

$$E_{p,n_{\max}} = \frac{1}{1+\frac{\log_2 p}{T}\alpha+\sqrt{\frac{p}{T}}\frac{\beta}{\gamma}}$$

随着节点数量增加，并行效率将趋近于

$$\lim_{p\to\infty} E_p = \sqrt{\frac{T\gamma}{p\beta}}.$$

同样地，随着处理器数量增加，效率无法维持，且执行时间存在上限。

我们还可以计算该操作的等效率曲线，即保持效率恒定时 n, p 之间的关系（参见章节 2.2.5.1 ）。若将上述效率简化为 $E(n,p) = \frac{2\gamma}{\beta}\frac{n}{p}$，则 $E \equiv c$ 等价于 $n = O(p)$，因此

$$M = O\left(\frac{n^2}{p}\right) = O(p).$$

因此，为维持效率，我们需要快速提升每个处理器的内存容量。这合乎逻辑，因为此举能降低通信的重要性。

7.2.3.2 矩阵 - 向量乘积的列划分

Partition

$$A \to \begin{pmatrix} A_0, A_1, \ldots, A_{p-1} \end{pmatrix} \quad x \to \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}, \quad \text{and} \quad y \to \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix},$$

其中 $A_j \in \mathbb{R}^{n\times n_j}$ 和 x_j， $y_j \in \mathbb{R}^{n_j}$ 与 $\sum_{j=0}^{p-1} n_j = n$ 及 $n_j \approx n/p$。

我们首先假设 A_j 、 x_j 和 y_j 最初被分配给 P_j （但现在 A_i 是一个列块）。在此按列算法中，处理器 i 无需预先通信即可计算长度为 n 的向量 $A_i x_i$。这些部分结果随后需要被汇总相加。

$$y \leftarrow \sum_i A_i x_i$$

在 *reduce-scatter* 操作中：每个处理器 i 将其结果的一部分 $(A_i x_i)_j$ 散射给处理器 j。接收处理器随后执行归约操作，将所有片段相加：

$$y_j = \sum_i (A_i x_i)_j.$$

带成本的算法如下所示：

Step	成本（下限）
Locally compute $y^{(j)} = A_j x_j$	$\approx 2\frac{n^2}{p}\gamma$
Reduce-scatter the $y^{(j)}$s so that $y_i = \sum_{j=0}^{p-1} y_i^{(j)}$ is on P_i	$\lceil \log_2(p) \rceil \alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$ $\approx \log_2(p)\alpha + n(\beta+\gamma)$

7.2.3.2.1 **成本分析** 该算法的总成本近似为，

$$T_p^{\text{1D-col}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + n(\beta+\gamma).}_{\text{Overhead}}$$

注意这与 T^1D 行 $_p$ (n) 的成本相同，只是将 β 替换为 $(\beta+\gamma)$。不难看出关于可扩展性的结论也是相同的。

7.2.3.3 二维分区

显然，一维分区（无论是按行还是按列）是一种扩展性极差的解决方案。留给我们的选择是使用二维分区。这里我们将进程排列在二维网格中（无论这在网络层面是否合理），并为每个进程分配一个不跨越整行或整列的子矩阵。我们仅考虑方阵，但允许进程网格不完全方正。

$$A \to \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,c-1} \\ A_{10} & A_{11} & \cdots & A_{1,c-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{r-1,0} & A_{r-1,0} & \cdots & A_{r-1,c-1} \end{pmatrix} \quad x \to \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{c-1} \end{pmatrix}, \quad \text{and} \quad y \to \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{r-1} \end{pmatrix},$$

其中 $A_{ij} \in \mathbb{R}^{n_i \times n_j}$和 x_i， $y_i \in \mathbb{R}^{n_i}$且 $\sum_{i=0}^{p-1} n_i = N$和 $n_i \approx N/\sqrt{P}$。

我们将这些进程视为一个 $r \times c$ 网格，其中 $P = rc$，并将其索引为 p_{ij}，其中 $i = 0, \ldots, r-1$ 和 $j = 0, \ldots, c-1$。图 7.4 展示了一个 12×12 矩阵在 3×4 处理器网格上的数据分配情况，其中 i, j ‘单元格’ 显示了由 p_{ij} 拥有的矩阵和向量元素。

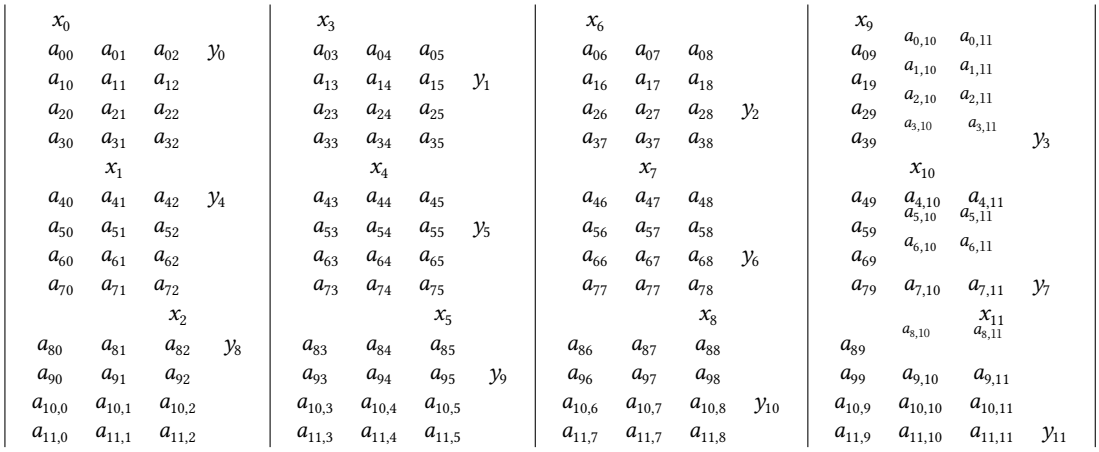


Figure 7.4: Distribution of matrix and vector elements for a problem of size 12 on a 4×3 processor grid.

换言之， p_{ij} 拥有矩阵块 A_{ij} 以及 x 和 y 的部分。这使得以下算法 2 成为可能：

- 由于 x_j 分布在 j 列上，算法首先通过处理器列内部的全收集操作在每个处理器 p_{ij} 上收集 x_j。
- 每个处理器 p_{ij} 随后计算 $y_{ij} = A_{ij}x_j$。此过程无需进一步通信。
- 结果 y_i 通过收集各处理器行中的片段 y_{ij} 以形成 y_i，随后将其分布到处理器行中。这两个操作实际上被合并为一个规约 - 分散操作。
- 若 $r = c$，我们可以转置处理器间的 y 数据，使其可作为后续矩阵 - 向量乘积的输入。反之，若我们正在计算 A^tAx，则 y 当前已为 A^t 乘积正确分布。

7.2.3.3.1 **算法** 附带成本分析的算法是

Step	成本（下限）
在列内全局收集 x_i	$\lceil\log_2(r)\rceil\alpha + \frac{r}{p}^{-1}n\beta$
执行局部矩阵 - 向量乘法	$\log_2(r)\alpha + \frac{n}{c}\beta \approx 2\frac{n^2}{p}\gamma$
在行内对 y_i 进行规约分散	$\lceil\log_2(c)\rceil\alpha + \frac{c}{p}^{-1}n\beta + \frac{c-1}{p}n\gamma$
	$\log_2(c)\alpha + \frac{n}{r}\beta + \frac{n}{r}\gamma$

7.2.3.3.2 **成本分析** 该算法的总成本近似为

$$T_p^{r\times c}(n) = T_p^{r\times c}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + \left(\frac{n}{c} + \frac{n}{r}\right)\beta + \frac{n}{r}\gamma.}_{\text{Overhead}}$$

我们现在简化假设 $r = c = \sqrt{p}$，从而

$$T_p^{\sqrt{p}\times\sqrt{p}}(n) = T_p^{\sqrt{p}\times\sqrt{p}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + \frac{n}{\sqrt{p}}(2\beta+\gamma)}_{\text{Overhead}}$$

由于串行成本为 $T_1(n) = 2n^2\gamma$，加速比由下式给出

$$S_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{T_1(n)}{T_p^{\sqrt{p}\times\sqrt{p}}(n)} = \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \frac{n}{\sqrt{p}}(2\beta+\gamma)} = \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

2. 该图展示了矩阵被划分为连续块的分区方式，向量分布似乎是与这种矩阵分布协同工作所必需的。您也可以从另一个角度理解：从输入输出向量的分布出发，推导出矩阵应采用的分布方式。例如，若将 x 和 y 以相同方式分布，会得到不同的矩阵分布，但除此之外乘积算法大致相同；参见 [55]。

以及并行效率通过

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p}{2n^2}\log_2(p)\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

我们再次提出这个问题：在 p 个节点上能存储的最大规模问题，其并行效率是多少。

$$\begin{aligned} E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) &= \frac{1}{1 + \frac{p}{2n^2}\log_2(p)\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}} \\ &= \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}} \end{aligned}$$

因此 still

$$\lim_{p\to\infty} E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) = \lim_{p\to\infty} \frac{1}{1 + \frac{\log_2(p)}{2M}\frac{\alpha}{\gamma} + \frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}} = 0.$$

然而 2 表现得几乎像一个常数，在这种情况下，$E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p))$ 下降得非常缓慢，该算法在实
2. log grows very slowly with p and is therefore considered
际应用中被认为是可扩展的。

注意当 $r = p$ 时，二维算法变为 “ 按行划分 ” 算法；当 $c = p$ 时，则变为 “ 按列划分 ” 算法。不难证明，只要 r/c 保持近似恒定，二维算法在上述分析意义上是可扩展的。

Exercise7.4. 计算该操作的等效率曲线。

7.3 并行 LU 分解

从某种意义上说，矩阵 - 向量和矩阵 - 积运算很容易并行化。输出的所有元素都可以独立且以任意顺序计算，因此我们在并行化算法时拥有很大的自由度。但对于计算 LU 分解或使用分解后的矩阵求解线性系统而言，这一特性并不成立。

7.3.1 求解三角系统

T三角系统 $y = L^{-1}x$ (当 L 为下三角矩阵时) 的求解本质上是一种矩阵 - 向量运算，因此它与矩阵 - 向量积具有相同的 $O(N^2)$ 复杂度特性。然而与乘积运算不同的是，该求解过程包含了输出元素间的递推关系：

$$y_i = \ell_{ii}^{-1}(x_i - \sum_{j<i} \ell_{ij}x_j).$$

这意味着并行化并非易事。对于稀疏矩阵的情况，可能有特殊策略；参见章节 7.10。此处我们将针对一般的密集矩阵情况作若干说明。

对于 $k = 1, n-1$: 对于
$i = k+1$ 到 n:
$a_{ik} \leftarrow a_{ik}/a_{kk}$ 对于 $i = k+1$
到 n: 对于 $j = k+1$ 到 n:
$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

图 7.5: LU 分解算法

为简化起见，我们假设通信不耗时，且所有算术运算均采用相同的单位时间。首先考虑按行分布矩阵，即处理器 p 存储元素 ℓ_{p*}。基于此，三角解法可实现为：

- 处理器 1 求解 $y_1 = \ell_{11}^{-1} x_1$ 并将其值发送至下一处理器。• 通常，处理器 p 从处理器 $p-1$ 获取值 $y_1, \ldots, y_{p-1}$，并计算 y_p；
- 随后每个处理器 p 将 $y_1, \ldots, y_p$ 发送至 $p+1$。

练习 7.5. 证明该算法与串行算法一样，耗时 $2N^2$。

该算法让每个处理器以流水线方式将所有计算出的 y_i 值传递给后继处理器。但这意味着处理器 p 直到最后一刻才收到 y_1，而该值实际上已在第一步完成计算。我们可以这样构造求解算法：使计算出的元素尽可能早地被利用：

- 处理器 1 求解 y_1，并将其发送给所有后续处理器。
- 通常，处理器 p 会等待包含 $q < p$ 的 y_q 值的独立消息。
- 处理器 p 随后计算 y_p 并将其发送给满足 $q > p$ 条件的 q 号处理器。

在通信时间可忽略的假设下，该算法速度可以显著提升。例如所有处理器 $p > 1$ 会同时接收 y_1，并同步计算 $\ell_{p1} y_1$。

练习 7.6. 证明若忽略通信，该算法变体耗时 $O(N)$。若将通信成本纳入考量，总成本是多少？

练习 7.7. 现考虑按列分布的矩阵：处理器 p 存储 ℓ_{*p}。概述采用此分布的三角解法算法，并证明并行求解时间为 $O(N)$。

7.3.2 因式分解，稠密情形

高斯消元法或 LU 分解的分析略显复杂。首先，高斯消元法可分为右视法、左视法（杜利特尔）和克劳特变体；参见章节 5.3.1.1 及 [183]。虽然这些算法计算结果相同，但选择不同算法可能带来计算性能上的差异。

我们限定采用常规右视高斯消元法，如章节 5.3.1 所述，其流程再现于图 7.5。

其中运算次数为

$$\sum_{k=1}^{n} \Big(\sum_{i=k+1}^{n} 1 + \sum_{i=k+1}^{n} \sum_{j=k+1}^{n} 1 \Big) = n^3/3 + O(n^2).$$

此处我们并不关心精确计算；通过类比积分，您会理解主导项中 1/3 系数的合理性。此外，我们将始终忽略低阶项。

若手动执行该算法的几个步骤，您会发现

$$a_{22} \leftarrow a_{22} - a_{21} * a_{11}^{-1} a_{12}$$
$$\cdots$$
$$a_{33} \leftarrow a_{33} - a_{32} * a_{22}^{-1} a_{23}$$

这意味着存在一条依赖操作链。

练习 7.8. 请完整推演此论证。试说明存在一个非平凡的关键路径，其概念如第 2.2.4 节所述。该路径长度是多少？在关键路径分析中，此路径对最小并行执行时间和加速比上限有何启示？

Now let's take a closer look at how we would actually parallelize this. The major work step is the subblock update:

对于 $i = k+1$ 到 n：对于
$j = k+1$ 到 n：
$a_{ij} \leftarrow a_{ij} - a_{ik} * a_{kj}$

在 $k = 1$ 步骤中，这涉及 $(n-1)^2$ 次操作，且这些操作都是并行的，因此理论上可以使用 $(n-1)^2$ 个处理单元（处理器、核心、 SIMD 线路等）。

练习 7.9. 延续这一推理。使用 $p = n^2$ 个处理单元时，子块中的每个 (i,j) 更新都可以同时完成。具体而言，任意 k 次迭代需要多长时间？对所有 k 求和后，得到的 T_p、 S_p、 E_p 是多少？这与之前推导的界限有何关联？此外，使用 $p = n$ 个处理单元时，可以让子块更新的每一行或每一列并行执行。此时第 k 次外层迭代的时间是多少？最终的 T_p、 S_p、 E_p 又是多少？

迄今为止我们探讨了并行执行的各个方面，但尚未考虑内存因素。针对内存分析，我们建立了以下关于问题规模 N 和处理器频率 f 的时间与内存关系式：

$$T = \frac{1}{3} N^3 / f, \qquad M = N^2.$$

请注意，此公式适用于单处理器计算机。

练习 7.10. 假设你购买了一台速度翻倍的处理器，并希望进行基准测试，使其再次耗时 T。你需要多少内存？

With p parallel processing elements, the relations become

$$T = \frac{1}{3} N^3 / (pf), \qquad M = N^2.$$

现在让我们探讨处理器数量增加的影响，这里假设每个处理器都有自己的内存，其中包含总问题的 $M_p = N^2/p$。（由于保持进程本身不变，为简化起见可设 $f = 1$。）

习题 7.11. 假设你有一个包含 p 个处理器的集群，每个处理器拥有 M_p 内存，能在 T 时间内完成 $N \times N$ 矩阵的高斯消元：

$$T = \frac{1}{3}N^3/p, \qquad M_p = N^2/p.$$

现在你将集群扩展至 $2P$ 个相同主频的处理器，并希望进行基准测试运行，同样耗时 T。每个计算节点需要多少内存？提示：对于扩展后的集群：

$$T' = \frac{1}{3}N'^3/p', \qquad M'_p = N'^2/p'.$$

问题转化为在给定条件下计算 M'_p。

7.3.3 稠密情形的因式分解实现

A 对并行稠密 LU 分解可扩展性的完整分析相当复杂，因此我们将不作推导直接给出结论 f进一步证明了在矩阵 - 向量情形下需要二维分布。

习题 7.12. 若采用一维分布且处理器数量足以每个处理器存储一列，分析运行时间、加速比和效率作为 N 的函数关系。证明加速比存在上限。并对二维分解情形（每个处理器存储一个元素）重复上述分析。

因此，采用过度分解策略：将矩阵划分为超过处理器数量的块，每个处理器存储多个不连续的子矩阵。如图 7.6 所示

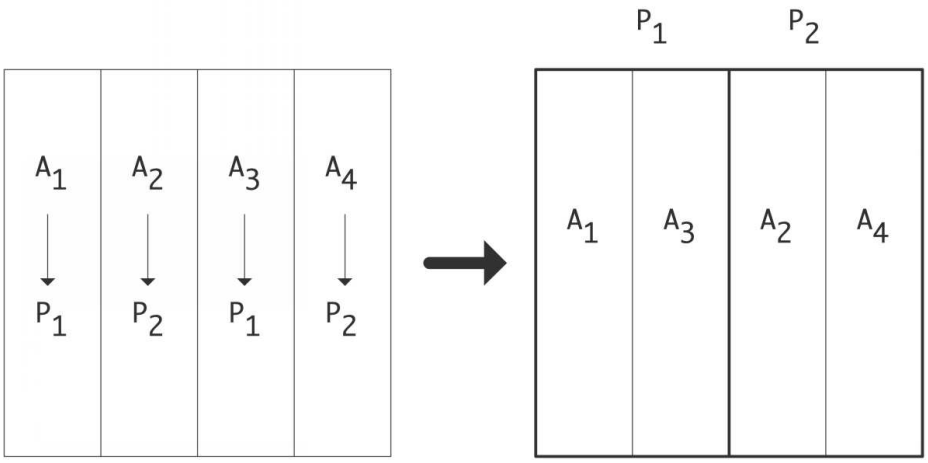


图 7.6：一维循环分布：将四个矩阵列分配给两个处理器，以及存储到矩阵列的映射结果。

此例中将矩阵的四个块列分配给两个处理器：每个处理器在连续内存块中存储两个不连续的矩阵列。

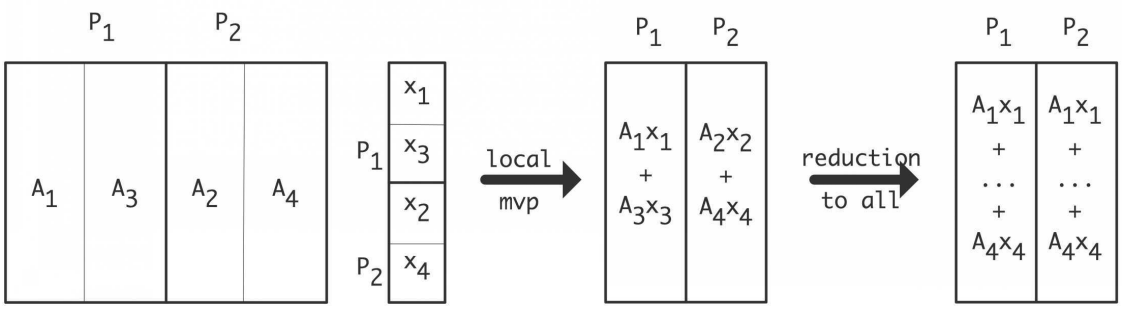


图 7.7：循环分布矩阵的矩阵 - 向量乘法。

N接下来，我们在图 7.7 中说明，即使处理器存储的是矩阵的非连续部分，也能执行此类矩阵的矩阵 - 向量乘积。只需确保输入向量也采用循环分布即可。

习题 7.13. 现在考虑一个 4×4 矩阵和一个 2×2 处理器网格。将矩阵按行和列双向循环分布。证明只要输入分布正确，仍可使用连续矩阵存储执行矩阵 - 向量乘积。输出如何分布？证明其通信开销高于一维案例的归约操作。

具体而言，使用 $P < N$ 个处理器时（为简化假设 $N = cP$），我们让处理器 0 存储行 $0, c, 2c, 3c, \ldots$；处理器 1 存储行 $1, c+1, 2c+1, \ldots$，以此类推。若满足 $N = c_1 P_1 = c_2 P_2$ 和 $P = P_1 P_2$ 条件，此方案可推广至二维分布，称为二维循环分布。进一步可通过划分块行 / 列（小分块尺寸）扩展该方案，例如分配块行 $0, c, 2c, \ldots$ 给处理器 0。

练习 7.14. 考虑一个正方形矩阵 $n \times n$ 和一个正方形处理器网格 $p \times p$，其中 p 能整除 n 无余数。考虑上述的超分解方案，针对特定情况 $n = 6, p = 2$ 绘制矩阵元素分配的示意图。即绘制一个 $n \times n$ 表格，其中位置 (i, j) 包含存储相应矩阵元素的处理器编号。同时为每个处理器制作一个描述本地到全局映射关系的表格，即给出本地矩阵中元素的全局 (i, j) 坐标。（采用从零开始的编号方式会简化此任务。）现编写函数 P, Q, I, J 来描述全局到本地的映射关系，即矩阵元素 a_{ij} 存储在处理器 $(P(i,j), Q(i,j))$ 的 $(I(i,j), J(i,j))$ 位置上。

7.3.4 因式分解 · 稀疏情形

并行稀疏矩阵 LU 分解是一个众所周知的难题。任何类型的分解都涉及顺序组件，这本身就使其非平凡。要特别理解稀疏情况下的问题，假设您按顺序处理矩阵行。密集情况下每行有足够多的元素可以从中派生出并行性，但在稀疏情况下这个数量可能非常少。

解决这个问题的方法是意识到我们并不关心分解本身，而是我们可以用它来解线性系统。由于对矩阵进行置换会得到相同的解，也许

其本身经过置换后，我们可以探索具有更高并行度的置换方式。

矩阵排序的主题已在第 5.4.3.5 节中提及，动机是减少填充量。下文将探讨具有良好并行特性的排序方法：嵌套剖分法见第 7.8.1 节，多色排序法见第 7.8.2 节。

7.4 矩阵 - 矩阵乘积

本节我们将探讨顺序和并行矩阵 - 矩阵乘积，两者均存在一定的计算挑战。

矩阵 - 矩阵乘积 $C \leftarrow A \cdot B$ （或 BLAS 中使用的 $C \leftarrow A \cdot B + \gamma C$）具有简单的并行结构。假设所有大小为 $N \times N$ 的方阵

- 这些 N^3 乘积 $a_{ik}b_{kj}$ 可以独立计算，之后
- 这些 N^2 元素 c_{ij} 通过独立的求和归约形成，每次归约耗时 $\log_2 N$。

然而，数据移动的问题则显得更为有趣 t:

- 通过 N^3 对 $O(N^2)$ 元素的操作，存在大量数据复用的机会（章节 1.6.1）。我们将在 7.4.1 节探讨 ‘Goto 算法’。
- 根据矩阵遍历方式的不同，还需考虑 TLB 复用问题（章节 1.3.9.2）。
- 分布式内存版本的矩阵 - 矩阵乘积尤为复杂。假设 A、B、C 均分布在处理器网格上，A 的元素需通过处理器行传输，而 B 的元素则需通过处理器列传输。下文我们将讨论 ‘Cannon 算法’ 及外积方法。

7.4.1 Goto 矩阵 - 矩阵乘积

在章节 1.6.1 中我们论证了 矩阵矩阵乘积 (或 *dgemm* 根据 *BLAS* 术语) 具有大量潜在的数据复用：存在 $O(n^3)$ 次操作作用于 $O(n^2)$ 数据。现在我们将探讨由 *Kazushige Goto* [82], 提出的实现方案，该方案确实达到了接近峰值性能。

矩阵 - 矩阵算法包含三个循环，每个循环均可分块处理，从而形成六层嵌套循环。由于输出元素不存在递归依赖，所有循环交换操作均合法。结合循环分块引入了三个分块参数这一事实，你会发现潜在实现方式的数量极其庞大。此处我们展示支撑 Goto 实现的全局逻辑；详细讨论请参阅所引论文。

我们首先将乘积 $C \leftarrow A \cdot B$ (或按 Blas 标准记为 $C \leftarrow C + AB$) 表示为一系列低秩更新的序列：

$$C_{**} = \sum_k A_{*k} B_{k*}$$

参见图 7.8。接着我们通过将 A 的块与 B 的 ‘条状切片’ 相乘来推导 ‘块 - 面板’ 乘法；见图 7.9。最终，内层算法通过累加方式处理小型行 $C_{i,*}$，典型尺寸如 4：

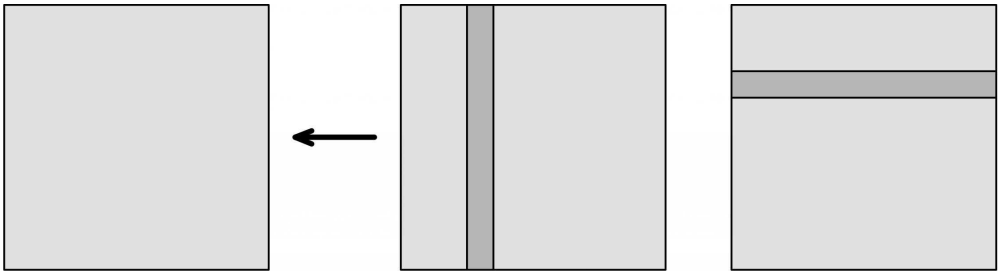


图 7.8：矩阵乘法作为一系列低秩更新的过程。

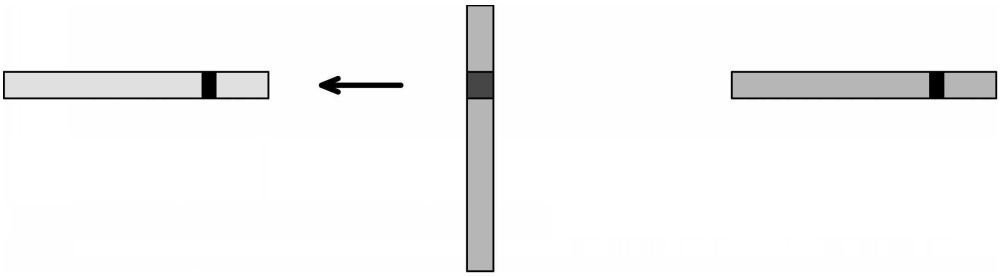


图 7.9：矩阵乘法算法中的块 - 面板相乘

```
// compute C[i,*] :
for k:
   C[i,*] = A[i,k] * B[k,*]
```

参见图 7.10。现在该算法已调优。

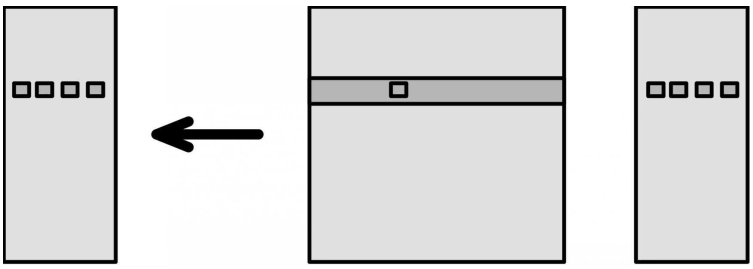


图 7.10：矩阵乘法寄存器驻留核心算法

- 我们需要足够的寄存器来存储 `C[i,*]`、 `A[i,k]` 和 `B[k,*]`。在当前处理器上，这意味着我们需累积 C 的四个元素。
- 这些 C 的元素会被累加，因此它们保留在寄存器中，唯一的数据传输是加载 A 和 B 的元素；无需存储操作！

- 元素 `A[i,k]` 和 `B[k,*]` 从 L1 缓存流出。
- 由于同一块 A 会用于 B 的多个连续切片，我们希望它保持驻留状态；

我们选择 A 的块大小使其能保留在 L2 缓存中。

- 为了防止 TLB 问题， A 按行存储。如果初始矩阵采用 (Fortran) 列优先存储，这意味着我们需要制作一个副本。由于复制的复杂度较低，这一成本是可分摊的。

该算法可在专用硬件中实现，如 *Google TPU* [81]，能效极高。
e

7.4.2 分布式内存矩阵 - 矩阵乘积的 Cannon 算法

在章节 7.4.1 中，我们探讨了单处理器 矩阵 - 矩阵乘积的高性能实现。现在我们将简要讨论该操作的分布式内存版本。（值得注意的是，这并非基于章节 7.2 中矩阵 - 向量乘积算法的泛化。）

该操作的一种算法被称为 *Cannon* 算法。它假设一个方形处理器网格，其中处理器 (i,j) 逐步累积 (i,j) 块 $C_{i,j} = \sum_k A_{i,k}B_{k,j}$；参见图 7.11。（本质上这是 分块矩阵 操作，如章节 5.3.6 所述。由于实现的分布式特性，此处我们实际存储的是块，而非仅概念性考虑。）

若从左上角开始，可见处理器 $(0,0)$ 同时拥有 A_{00} 和 B_{00}，因此可立即执行本地乘法运算。处理器 $(0,1)$ 持有 A_{01} 和 B_{01}，二者虽无需同时使用，但若将 B 的第二列向上旋转一位，处理器 $(0,1)$ 将获得 A_{01}、 B_{11}，此时便需进行乘法运算。同理，将 B 的第三列向上旋转两位，使 $(0,2)$ 包含 A_{02}、B_{22}。

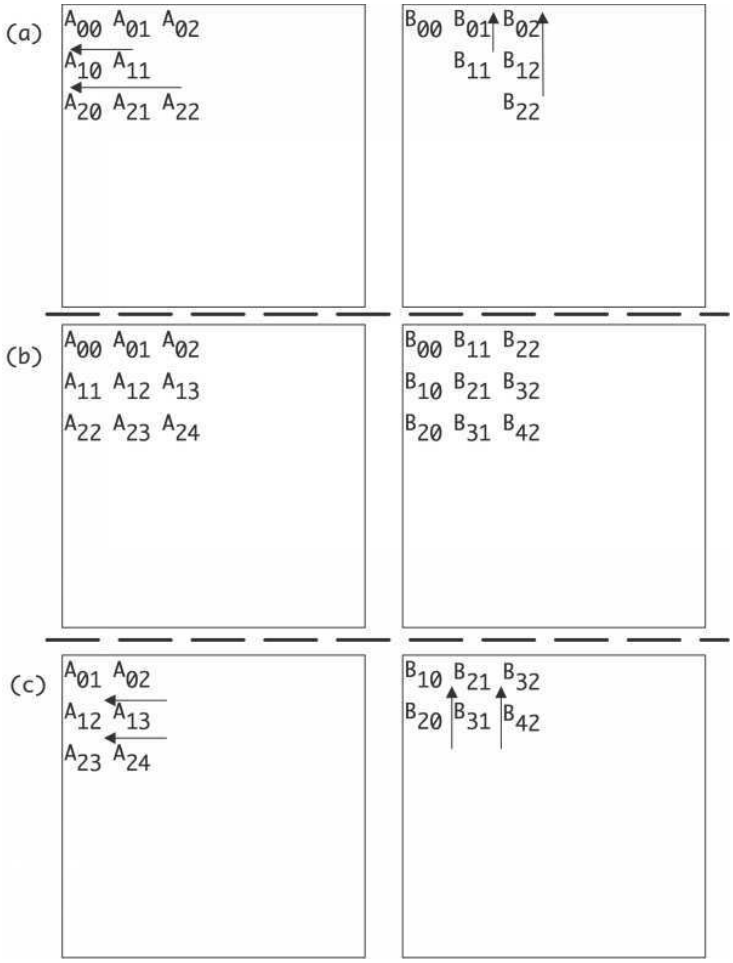


图 7.11：矩阵 - 矩阵乘法的 Cannon 算法：(a) 矩阵行与列的初始旋转，(b) 处理器 (i,j) 可开始累加 $\sum_k A_{ik}B_{kj}$的结果位置，(c) 为下一项进行的 A、 B 后续旋转。

这个故事在第二行如何展开
? 处理器 $(1,0)$ 拥有 A_{10} 和 B_{10}，这两者并不需要同时使用。若我们将 A 的第二行向左旋转一个位置，此时它包含 A_{11} 和 B_{10}，这两者正是一个部分积所需的元素。现在，处理器 $(1,1)$ 则持有 A_{11} 和 B_{11}。
若延续这一逻辑，我们从一个经过行左旋的矩阵 A 和一个经过列上旋的矩阵 B 开始。在此配置下，处理器 (i,j) 包含 $A_{i,i+j}$ 及 $B_{i+j,j}$，其中加法运算对矩阵尺寸取模。这意味着每个处理器可以先进行本地乘积运算。

现在注意到 $C_{ij} = \sum_k A_{ik}B_{kj}$ 表明下一个部分积项来自将 k 增加 1。通过将 A 和 B 的对应元素再移动一个位置（即再次旋转行和列），这些元素可被移入处理器中。

7.4.3 分布式矩阵 - 矩阵乘积的外积方法

Cannon 算法受限于需要方形处理器网格的要求。而外积方法更为通用。它基于以下矩阵 - 矩阵乘积计算的排列方式:

```
for ( k )
  for ( i )
    for ( j )
      c[i,j] += a[i,k] * b[k,j]
```

即，对于每个 k，我们取 A 的一列和 B 的一行，并计算秩 -1 矩阵（或称 ‘外积’） $A_{*,k} \cdot B_{k,*}$。然后对 k 求和。
观察该算法的结构，我们注意到在步骤 k 中，每列 j 接收 $A_{*,k}$，每行 i 接收 $B_{k,*}$。换言之，$A_{*,k}$ 的元素通过其行广播，而 $B_{k,*}$ 的元素通过其列广播。

使用 MPI 库，这些同步广播通过为每行和每列创建子通信器来实现。

7.5 稀疏矩阵 - 向量乘积

在通过迭代方法求解线性系统时（参见章节 5.5），矩阵 - 向量乘积在计算上是一个重要的核心操作，因为它可能在数百次迭代中每次都要执行。本节我们首先关注单处理器上矩阵 - 向量乘积的性能方面；多处理器的情况将在章节 7.5.3 中讨论。

7.5.1 单处理器稀疏矩阵 - 向量乘积

在迭代方法的背景下，我们并不太担心稠密矩阵 - 向量乘积，因为通常不会对稠密矩阵进行迭代。在处理块矩阵的情况下，可以参考章节 6.10 对稠密乘积的分析。稀疏乘积则要复杂得多，因为大部分分析都不适用。

稀疏矩阵 - 向量乘积中的数据复用按行执行的稠密矩阵 - 向量乘积与 CRS 稀疏乘积（章节 5.4.1.4 ）存在一些相似之处。两者中所有矩阵元素都是按顺序使用的，因此加载的任何缓存行都能被充分利用。然而，CRS 乘积至少在以下方面表现更差：

- 间接寻址需要加载整型向量的元素。这意味着在相同操作数量下，稀疏乘积会产生更多的内存流量。
- 源向量的元素并非按顺序加载，实际上它们可能以近乎随机的顺序加载。这意味着包含源元素的缓存行很可能无法被充分利用。此外，内存子系统的预取逻辑（章节 1.3.6 ）在此场景下也无法发挥作用。

由于这些原因，以稀疏矩阵 - 向量乘积为主要计算任务的应用很可能仅以 $\approx 5\%$ 的处理器峰值性能运行。

如果矩阵结构在某种意义上是规则的，或许有可能提升这一性能。其中一种情况是处理完全由小型稠密块组成的块矩阵。这至少能减少索引信息量：若矩阵由 2×2 个块组成，整数数据传输量将减少 $4\times$ 。

练习 7.15. 列举该策略可能提升性能的另外两个原因。提示：
缓存线及复用。

此类矩阵镶嵌可能带来 2 倍的性能提升。假设存在此等改进，即便矩阵并非完美块矩阵，我们仍可采用该策略：若每个 2×2 块含有一个零元素，仍可能获得 1.5 倍的性能提升 [188, 27]。

稀疏乘积中的向量化在其他情况下，带宽和复用并非主要考量：

- 在旧式向量计算机（如老式 *Cray* 机型）上，内存速度足以匹配处理器需求，但向量化至关重要。这对稀疏矩阵构成难题，因为矩阵行中的零元素数量（即向量长度）通常较低。
- 在 GPU 上，内存带宽相当高，但需要找到大量相同的操作。矩阵可以被独立处理，但由于各行长度可能不等，这并非合适的并行化来源。

基于这些原因，针对稀疏矩阵的对角线存储方案最近重新受到关注；参见 5.4.1.5 节。这里的观察结果是，若按行数对矩阵行进行排序，会得到少量行块；每个块将相当大，且块内各行具有相同数量的元素。

具有此类结构的矩阵非常适合向量架构 [39]。此时乘积通过对角线计算。

习题 7.16. 为此情况编写伪代码。行的排序如何改善了该情况?

T这种排序存储方案也解决了我们在 GPU 上注意到的问题 [20]。此时我们采用传统的 CRS 乘积算法，并行度等于一个块中的行数。

当然，这里存在一个复杂性：我们对矩阵进行了置换，输入和输出向量也需要相应地进行置换。如果乘积操作是迭代方法的一部分，在每次迭代中来回进行这种置换很可能会抵消任何性能增益。相反，我们可以置换整个线性系统，并在置换后的系统上进行迭代。

练习 7.17. 你能想到这种方法可行的理由吗？以及它不可行的理由？

7.5.2 并行稀疏矩阵 - 向量乘积

在章节 5.4 中，你首次了解了稀疏矩阵的讨论，但仅限于单处理器上的使用。现在我们将探讨并行化的方面。

稠密矩阵 - 向量乘积，正如你上面所看到的，要求每个处理器与其他所有处理器通信，并且需要一个基本上与全局向量大小相同的本地缓冲区。在稀疏情况下，需要的缓冲区空间以及通信量都显著减少。

让我们详细分析并行稀疏矩阵 - 向量乘积。我们假设矩阵按块行分布，其中处理器 p 拥有索引在某个集合 I_p 中的矩阵行。

在稠密情况下，行 $y_i = y_i + a_{ij}x_j$ 会对所有 j 执行，因此会引发全收集操作；而现在必须考虑到 a_{ij} 可能为零，因此对于某些 $i \in I_p, j \notin I_p$ 对无需通信。为每个 $i \in I_p$ 声明一个稀疏模式集

$$S_{p;i} = \{j : j \notin I_p, a_{ij} \neq 0\}$$

我们的乘法指令变为

$$y_i += a_{ij}x_j \qquad \text{if } j \in S_{p;i}.$$

若要避免大量小消息的洪泛，我们将所有通信合并为每个处理器的单一消息。定义

$$S_p = \cup_{i \in I_p} S_{p;i},$$

算法现在变为：

- 将所有必要的非本处理器元素 x_j 与 $j \in S_p$ 收集到一个缓冲区中；
- 执行矩阵 - 向量乘积操作，从本地存储中读取 x 的所有元素。

当然，这一整套分析同样适用于稠密矩阵。但如果我们考虑稀疏矩阵的来源，情况就有所不同。让我们从一个简单案例开始。

回顾图 4.1，该图展示了最简单区域（正方形）上的离散化边界值问题，现在让我们将其并行化。我们通过划分区域来实现这一点：每个处理器获取与其子区域对应的矩阵行。图 7.12 展示了这如何引发处理器间的连接：元素 a_{ij} 与 $i \in I_p$、$j \notin I_p$ 现在成为跨越处理器边界的模板‘腿’。所有这些 j 的集合，正式定义为

$$G = \{j \notin I_p : \exists_{i \in I_p} : a_{ij} \neq 0\}$$

被称为处理器的幽灵区域，参见图 7.13。

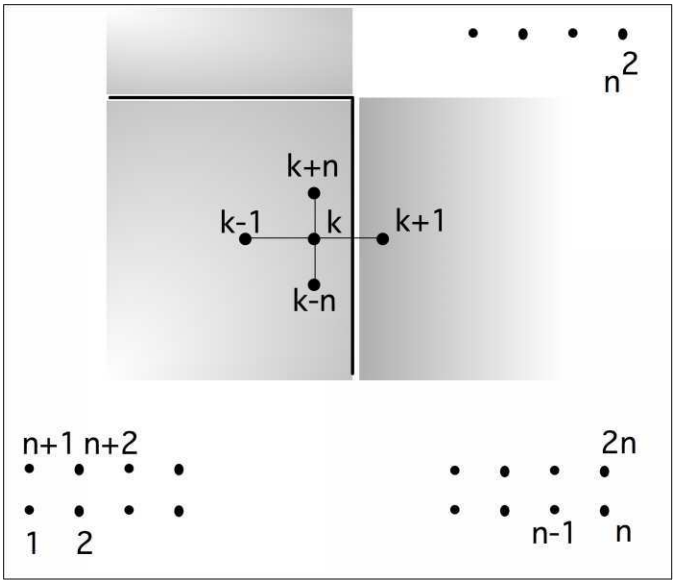


图 7.12：应用于二维方形区域的差分模板，分布式处理中跨处理器连接以 A 表示。 ors.

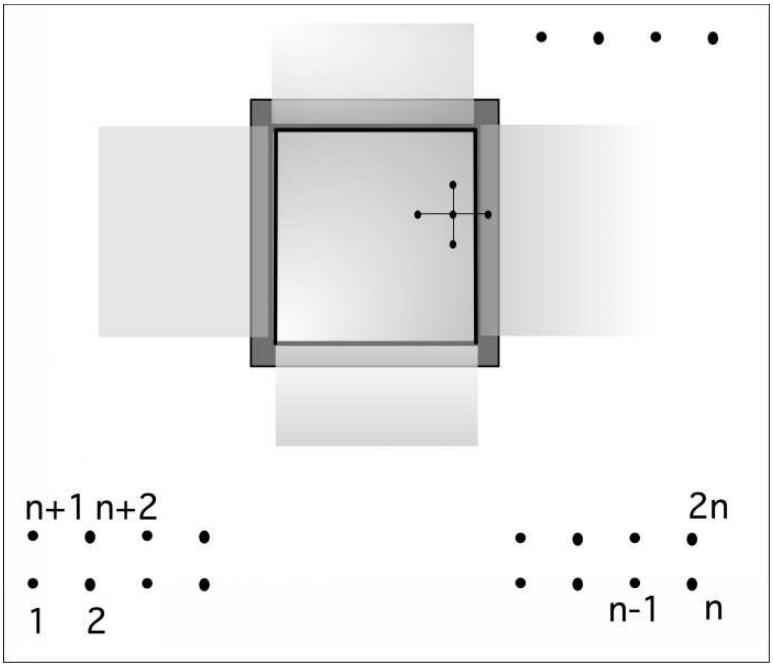


图 7.13：由模板引发的处理器幽灵区域。

练习 7.18. 证明域的一维划分会导致矩阵被划分为块行，而域的二维划分则不会。你可以从抽象角度进行论证，也可以通过示例说明：取一个 4×4 域（生成一个大小为 16 的矩阵），并将其划分到 4 个处理器上。一维域划分对应每个处理器获得域中的一行，而二维划分则让每个处理器获得一个 2×2 子域。绘制这两种情况下的矩阵。

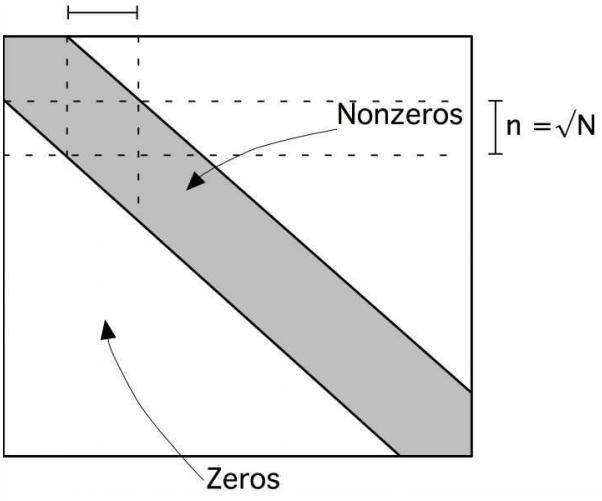


图 7.14: 半带宽为 $\sqrt{N}$ 的带状矩阵。

练习 7.19. 图 7.14 展示了一个大小为 N 、半带宽为 $n = \sqrt{N}$ 的稀疏矩阵。即，

$$|i - j| > n \Rightarrow a_{ij} = 0.$$

我们将该矩阵在 p 个处理器上进行一维分布，其中 $p = n = \sqrt{N}$。通过计算效率随处理器数量的函数关系，证明使用此方案的矩阵 - 向量乘积具有弱可扩展性。

$$E_p = \frac{T_1}{pT_p}.$$

为何这种方案不符合通常定义的弱扩展性标准?

关于并行稀疏矩阵 - 向量乘积的一个关键观察是：对于每个处理器而言，其需要交互的其他处理器数量存在严格限制。这一特性直接影响着该运算的效率。

在稠密矩阵 - 向量乘积的情况下（章节 7.2.3 ），通过（块）行划分将矩阵分配到各处理器上并不能产生可扩展的算法。从全局角度可以这样解释：随着处理器数量的增加，消息大小的下降速度无法与每个处理器的工作量下降速度同步。

习题 7.20. 取一个方形域并按图 7.12 的方式对处理器的变量进行分区。对于图 4.3 中的盒式模板，处理器需要通信的最大邻居数是多少？在三维空间中，若使用 7 点中心差分模板，邻居数量又是多少？

各处理器仅需与少数邻居通信这一观察结论，在从方形域扩展到更复杂的物理对象时依然成立。如果处理器接收到的是一个大致连续的子域，其邻居数量将受到限制。这意味着即使在复杂问题中，每个处理器也只需与少量其他处理器通信。相比之下，在密集矩阵情况下，每个处理器必须从所有其他处理器接收数据。显然，稀疏情形对互连网络更为友好。（对于大型系统而言，稀疏情形更常见，这可能影响您在采购新并行计算机时对网络架构的选择。）

7.5.3 稀疏矩阵 - 向量乘积的并行效率

对于方形域，我们可以将上述论证形式化。设单位域 $[0,1]^2$ 被划分为 P 个处理器，排列成 $\sqrt{P}\times\sqrt{P}$ 网格。从图 7.12 可以看出，每个处理器最多与四个相邻处理器通信。设每个处理器的工作量为 w，与每个相邻处理器的通信时间为 c。则在单个处理器上完成总工作的时间为 $T_1 = Pw$，并行时间为 $T_P = w + 4c$，由此得到的加速比为

$$S_P = Pw/(w+4c) = P/(1+4c/w) \approx P(1-4c/w).$$

习题 7.21. 将 c 和 w 表示为 N 和 P 的函数，并证明在问题的弱扩展条件下，加速比是渐进最优的。

习题 7.22. 本练习中，你将针对一台假设但现实的并行机器分析稀疏矩阵 - 向量乘积的并行性能。设该机器的参数特征如下（参见章节 1.3.2 ）：

- 网络 延迟： $\alpha = 1\mu s = 10^{-6}s$。
- 网络带宽： $1Gb/s$ 对应于 $\beta = 10^{-9}$。
- 计算速率：单核浮点运算速率为 $1G$flops 意味着 $\gamma = 10^{-9}$。这个数值可能看起来较低，但请注意矩阵 - 向量乘积的重用性低于矩阵 - 矩阵乘积（后者可接近峰值性能），且稀疏矩阵 - 向量乘积更受带宽限制。

我们结合渐近分析与具体数值推导。假设一个由 10_4 个单核处理器 [3] 组成的集群，应用于大小为 $N = 25\cdot 10^{10}$的五点模板矩阵。这意味着每个处理器存储 $5\cdot 8\cdot N/p = 10^9$ 字节。若矩阵来自方形域上的问题，则域尺寸为 $n\times n$，其中 $n = \sqrt{N} = 5\cdot 10^5$。

案例 1. 我们并非划分矩阵，而是先通过尺寸为 $n\times(n/p)$ 的水平板块对域进行划分。论证其通信复杂度为 $2(\alpha + n\beta)$，计算复杂度为 $10\cdot n\cdot(n/p)$。证明最终计算结果超出通信量 250 倍。

3. 引入多核处理器会使情况复杂化，但由于核心数为 $O(1)$，且扩展集群的唯一方式是增加联网节点，这并不改变渐近分析的结果。

> 案例 2. 我们将域划分为大小为 $(n/\sqrt{p}) \times (n/\sqrt{p})$ 的区块。内存和计算时间与之前相同。推导通信时间并证明其改善了 50 倍。
>
> 论证第一种情况不具备弱扩展性：在假设 N/p 恒定的前提下，效率会下降。（证明加速比仍会随着 $\sqrt{p}$ 渐近上升。）论证第二种情况确实具有弱扩展性。

关于处理器仅与少数邻居通信的论点基于科学计算的性质。这对于有限差分法（ FDM ）和有限元法（ FEM ）方法成立。而在边界元法（ *BEM* ）中，每个子域需要与其周围半径 r 内的所有区域通信。随着处理器数量增加，每个处理器的邻居数量也会上升。

习题 7.23. 对 BEM 算法的加速比和效率进行形式化分析。假设每个处理器的工作量 w 为单位值，与每个邻居的通信时间为 c。由于邻居的定义基于物理距离而非图论属性，邻居数量将会增加。给出该情况下的 T_1、 T_p、 S_p、 E_p。

也存在稀疏矩阵需要像稠密矩阵一样处理的情况。例如， *Google* 的 *PageRank* 算法（参见章节 10.4 ）核心是重复操作 $x \leftarrow Ax$ 其中 A 是一个稀疏矩阵，当网页 j 链接到页面 i 时， $A_{ij} \neq 0$ ；参见章节 10.4。这使得 A 成为一个非常稀疏的矩阵，没有明显的结构，因此每个处理器很可能几乎与其他所有处理器通信。

7.5.4 稀疏矩阵 - 向量乘积的内存行为

在章节 6.10 中，你看到了单处理器上稠密情况下稀疏矩阵 - 向量乘积的分析。部分分析可直接应用于稀疏情况，例如每个矩阵元素仅使用一次，性能受限于处理器与内存之间的带宽。

关于输入和输出向量的重用，如果矩阵按行存储，例如 *CRS* 格式（章节 5.4.1.3 ），对输出向量的访问将限制为每矩阵行一次写入。另一方面，获取输入向量重用的循环展开技巧在此无法应用。合并两次迭代的代码如下：

```
for (i=0; i<M; i+=2) {
  s1 = s2 = 0;
  for (j) {
    s1 = s1 + a[i][j] * x[j];
    s2 = s2 + a[i+1][j] * x[j];
  }
  y[i] = s1; y[i+1] = s2;
}
```

问题在于，如果 a_{ij} 非零，并不能保证 $a_{i+1,j}$ 也非零。稀疏模式的非规则性使得矩阵 - 向量积的优化变得困难。通过识别矩阵中较小密集块的部分 [27, 44, 187]，可以实现适度的改进。

在 *GPU* 上，稀疏矩阵 - 向量乘积同样受限于内存带宽。由于 GPU 需以数据并行模式运行，需激活大量线程，编程难度因此增加。

若考虑稀疏矩阵 - 向量乘积的典型应用场景，则可能实现一种有趣的优化。该运算最常见于线性系统的迭代解法（第 5.5 节），其中同一矩阵可能被用于数百次迭代运算。因此可考虑将矩阵长期存储于 GPU，仅在各次乘积运算时传输输入输出向量。

7.5.5 转置乘积

第 5.4.1.3 节指出，常规与转置矩阵 - 向量乘积的代码均受限于仅按矩阵行遍历的循环顺序。（第 1.6.2 节讨论了循环顺序改变的计算效应；本例中因存储格式限制必须采用行遍历。）

本节我们将简要探讨并行转置乘积的概念。与按行划分矩阵并执行转置乘积类似，我们也可以观察按列存储和划分的矩阵，并执行常规乘积运算。

按列执行乘积的算法可表述为：

$y \leftarrow 0$ 对于 j：对于 i：$y_i \leftarrow y_i + a_{ij}x_j$

无论是在共享内存还是分布式内存中，我们都需要将外层迭代分配给各个处理器执行。此时的问题在于，每个外层迭代都会更新整个输出向量。这会引发两个问题：在共享内存中会导致对输出位置的多次写入；在分布式内存中则需要目前尚未明确的通信机制。

解决此问题的一种方法是为每个进程分配私有输出向量 $y^{(p)}$：(p)

$y \leftarrow 0$
for $j \in$ the rows of processor p
  for all i:
    $y_i^{(p)} \leftarrow y_i^{(p)} + a_{ji}x_j$

after which we sum $y \leftarrow \sum_p y^{(p)}$.

7.5.6 稀疏矩阵 - 向量乘积的配置

上文我们定义了包含待发送至进程 p 数据的进程集合 S_p。对于进程 p 而言，从其稀疏矩阵部分构建该集合相当容易。

练习 7.24. 假设矩阵通过块行划分至各处理器；示意图参见 7.2 。并假设每个处理器知晓其他处理器存储了哪些行。（你将如何实现这种信息共享？）

概述处理器如何确定其接收来源的算法；
该算法本身不应涉及任何通信。

然而，发送与接收之间存在不对称性。处理器较易确定其接收数据的来源处理器，但发现需要发送至哪些处理器则更为困难。

练习 7.25. 论证在结构对称矩阵情况下此问题的简易性：$a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$。

在一般情况下，处理器原则上可能被要求向任何其他处理器发送数据，因此简单算法如下：

- 每个处理器会统计其所需的非本地索引。基于已知其他处理器各自拥有的索引范围这一前提，随后决定从哪些相邻处理器获取哪些索引。
- 每个处理器向所有相邻处理器发送索引列表；对多数相邻处理器该列表为空，但仍需发送不可省略。
- 每个处理器接收来自其他所有处理器的这些列表后，会制定出待发送索引的清单。

你会注意到，尽管矩阵 - 向量乘积运算期间的通信仅涉及每个处理器的少量相邻节点（其成本 $O(1)$ 与处理器数量相关），但建立通信关系时需要全对全通信，其时间复杂度为 $O(\alpha P)$。

若某处理器的相邻节点较少，上述算法存在资源浪费。理想情况下，空间和时间复杂度应与相邻节点数成正比。若能在建立阶段预知待接收消息数量，则可优化接收时间。该数量可通过以下方式确定：

- 每个处理器生成一个长度为 P 的数组 `need`，其中 `need[i]` 为 1 表示该处理器需要从处理器 i 获取数据，否则为 0。
- 对此数组执行 *reduce-scatter* 集合操作（求和运算符），最终每个处理器上会留下一个数字，表示有多少处理器需要从其接收数据。
- 处理器可执行相应次数的接收调用。

reduce-scatter 调用的时间复杂度为 $\alpha \log P + \beta P$，与前一算法同阶，但比例常数可能更低。

T通过一些精妙技巧 [61,，可将配置所需的时间和空间降至 $O(\log P)$。
107].

7.6 迭代方法的计算特性

所有迭代方法都具有以下操作特征：

- 矩阵 - 向量乘积；这在第 5.4 节中讨论了串行情况，并在第 7.5 节中讨论了并行情况。在并行情况下，有限元矩阵的构建存在一个复杂性，我们将在第 7.6.2 节中讨论。
- 预条件矩阵的构建 $K \approx A$，以及方程组的求解 $Kx = y$。这在第 5.5.6 节中讨论了串行情况。我们将在第 7.7 节中讨论并行方面的内容。
- 一些向量操作（通常包括内积）。这些将在接下来讨论。

7.6.1 向量运算

T在典型的迭代方法中，存在两种类型的向量运算：向量加法和内积。

练习 7.26. 考虑第 5.5.11 节的 CG 方法，图 5.11，应用于来自二维边值问题的矩阵；方程 (4.56)，首先考虑无预处理情况 $M = I$。证明在矩阵 - 向量乘积和向量运算中执行的浮点运算数量大致相等。将所有内容用矩阵大小 N 表示，并忽略低阶项。如果矩阵每行有 20 个非零元素，这种平衡会如何变化？

> 接下来，研究第 5.5.9 节中 FOM 方案的向量与矩阵运算之间的平衡。由于向量运算的数量取决于迭代次数，考虑前 50 次迭代，并计算在向量更新和内积与矩阵 - 向量乘积中执行的浮点运算数量。矩阵需要有多少个非零元素才能使这些数量相等？

Exercise7.27. 浮点运算次数并非全部真相。对于在单处理器上执行的迭代方法中的向量和矩阵运算效率，你能得出什么结论？

7.6.1.1 向量加法

向量加法通常表现为 $x \leftarrow x + \alpha y$ 或 $x \leftarrow \alpha x + y$ 的形式。若假设所有向量分布方式相同，则该操作可完全并行化执行。

7.6.1.2 内积运算

内积虽属向量运算，但其计算过程比更新操作更具研究价值，因其涉及通信过程。

计算内积时，极有可能每个处理器都需要接收计算结果。我们采用如下算法：

> **for***processor* p **do** 计算 $a_p \leftarrow x_p^t y_p$ 其中 x_p, y_p 是 x, y 在处理器 p 上的部分 执行全局全归约以计算 $a = \sum_p a_p$ 广播结果**算法 1:** 计算 $a \leftarrow x^t y$ 其中 x, y 是分布式向量。

`Allreduce`) 在所有处理器间组合数据，因此其通信时间随处理器数量增加而增长。这使得内积可能成为一项昂贵操作（同时它们形成类似屏障的同步机制），人们已提出多种方法来降低其对迭代方法性能的影响。

习题 7.28. 迭代方法通常用于稀疏矩阵。在此背景下，可以论证内积涉及的通信对整体性能的影响可能大于矩阵 - 向量积中的通信。矩阵 - 向量积与内积的复杂度如何随处理器数量变化？

以下是已采取的部分方法。

- CG 方法每次迭代有两个相互依赖的内积运算。可以通过
  重写该方法，使其计算相同的迭代值（至少在精确算术中），并将每次迭代的两个内积合并。参见 [33, 40, 148, 195]。
- 有可能将内积计算与其他并行计算 [42] 重叠进行。
- 在 GMRES 方法中，使用经典 Gram-Schmidt(GS) 方法所需的内积运算量远少于改进的 GS 方法，但其稳定性较低。研究者已探索策略来决定何时可采用经典 GS 方法 [131]。

由于计算机算术不具备结合性，内积运算成为导致相同计算在不同处理器配置下结果差异的主要根源。在 3.6.5 节我们概述了解决方案。

7.6.2 有限元矩阵构建

FEM 在并行计算中引发了一个有趣的问题。为此，我们需要简要概述该方法的基本原理。FEM 得名于其将建模的物理对象划分为小的二维或三维形状（即单元），例如二维中的三角形和正方形，或三维中的金字塔和立方体。在每个单元上，我们建模的函数被假定为多项式，通常为低次多项式，如线性或双线性。

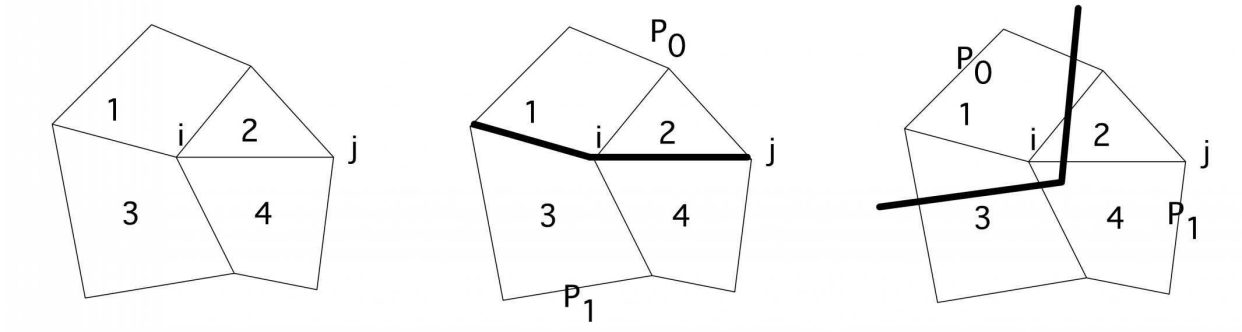


图 7.15：有限元域、矩阵构建的并行化及矩阵元素存储的并行化。

关键在于矩阵元素 a_{ij} 是所有同时包含变量 i 和 j 的单元上计算（特别是某些积分）的总和：

$$a_{ij} = \sum_{e\,:\,i,j\in e} a_{ij}^{(e)}.$$

每个单元中的计算存在大量共同部分，因此很自然地将每个单元 eu

唯一分配给处理器 P_e，由该处理器计算所有贡献 $a_{ij}^{(e)}$。如图 7.15 所示，单元 2 被分配
t将元素 0 分配给处理器 0，元素 4 分配给处理器 1。

现在考虑变量 i 和 j 以及矩阵元素 a_{ij}。它由分配给不同处理器的域元素 2 和 4 上的计算之和构成。因此，无论如何

无论处理器行 i 被分配给哪个处理器，至少有一个处理器需要将其对矩阵元素 a_{ij} 的贡献进行通信。

显然，无法通过分配 P_e 元素和 P_i 变量的方式使得 P_e 能完整计算出所有 a_{ij} 的系数 $i \in e$。换言之，若在本地计算贡献值，则需通过一定通信来组装特定矩阵元素。因此，现代线性代数库（如 PETSc ）《并行编程》书籍第 *32.4.3* 节允许任何处理器设置任意矩阵元素。

7.6.3 迭代方法性能的简单模型

前文已提及，迭代方法几乎不存在数据复用的机会，因此其本质属于带宽受限型算法。这使得我们可以对迭代方法的浮点性能进行简单预测。由于迭代次数难以预估，此处性能特指单次迭代的性能。与线性系统的直接解法不同（参见第 7.8.1 节），其求解所需浮点运算次数极难预先确定。

首先我们论证可以将关注点限制在稀疏矩阵向量乘法的性能上：该操作耗时远超向量运算。此外，大多数预条件子的计算结构与矩阵 - 向量乘积极为相似。

接下来我们考察 *CRS* 矩阵 - 向量乘积的性能。

- 首先观察到矩阵元素数组和列索引数组不存在数据复用，其加载性能完全取决于可用带宽。缓存和预取流仅能隐藏延迟，但无法提升带宽。每个矩阵元素与输入向量元素的乘法运算需加载一个浮点数和一个整数。根据索引是 32 位还是 64 位，这意味着每次乘法需加载 12 或 16 字节数据。
- 结果向量的存储需求相对次要：每个矩阵行仅需写入一次输出向量元素。
- 输入向量在带宽计算中亦可忽略。初看可能认为对输入向量的间接索引近乎随机因而代价高昂。但若从矩阵 - 向量乘积的算子视角出发，并考察矩阵来源的 PDE 空间域（参见 4.2.3 节），可发现看似随机的索引实际访问的是空间邻近的向量元素组。这意味着这些向量元素很可能驻留在 L3 缓存中，其访问带宽（例如至少提高 5 倍）将显著高于主存数据。

针对稀疏矩阵 - 向量乘法的并行性能，我们考虑在偏微分方程（ PDE ）背景下，每个处理器仅与少量邻近处理器通信。此外，通过表面积与体积比论证可知，通信数据量比节点内计算量低一个数量级。

综上所述，我们得出一个非常简单的稀疏矩阵向量乘法模型（进而适用于整个迭代求解器），该模型通过测量有效带宽并将性能计算为

每加载 12 或 16 字节数据对应一次加法与一次乘法运算。
o每加载 12 或 16 字节数据，执行一次加法和一次乘法运算。

7.7 并行预处理器

上文（第 5.5.6 节，特别是 5.5.6.1 小节）我们看到了几种不同的 K 选择。本节我们将开始讨论并行化策略，具体内容将在后续章节展开。

7.7.1 雅可比预处理

雅可比方法（第 5.5.3 节）利用 A 的对角线作为预处理器。其应用实现了最大程度的并行化：语句 $y \leftarrow K^{-1}x$ 可独立缩放输入向量的每个元素。但雅可比预处理对迭代次数的改善较为有限，因此我们需要考虑更复杂的方法如 ILU。与雅可比预处理不同，这类方法的并行化并非易事。

7.7.2 高斯 - 赛德尔与 SOR 方法

关于 GS （高斯 - 赛德尔）和 SOR （逐次超松弛）方法，我们可以指出它们不常被用作预条件子，因为其本质是非对称的。对于 SOR 方法，还存在确定 ω 合适值的问题。对称化的逐次超松弛（ SSOR ）方法与 ILU （不完全 LU 分解）足够相似，因此我们将不单独讨论它。

这些方法确实可作为多重网格平滑器使用，这里有一个值得注意的有趣现象。考虑通过波前法（章节 7.10.1 ）求解 GS 方程组，并设想多个迭代同时激活。不难看出这等效于多色方案，尽管起始向量不同。完整论证参见 [2] 。

7.7.3 并行计算中 ILU 的问题

前文提到，从浮点运算次数的角度来看，应用 ILU 预条件子（章节 5.5.6.1 ）的成本与矩阵 - 向量乘积相当。但在并行计算机上运行迭代方法时，这一结论不再成立。

乍看之下，这些操作颇为相似。矩阵 - 向量积 $y = Ax$ 看起来就像

```
for i=1..n
  y[i] = sum over j=1..n a[i,j]*x[j]
```

在并行计算中，情况会类似如下

```
for i=myfirstrow..mylastrow
  y[i] = sum over j=1..n a[i,j]*x[j]
```

假设处理器已本地缓存了其所需的所有 A 和 x 元素，那么该操作可完全并行化：每个处理器都能立即开始工作，若负载大致均衡，它们将同时完成运算。此时，矩阵 - 向量乘法的总耗时会被处理器数量均分，从而实现近乎完美的加速比。

现在考虑前向求解 $Lx = y$，例如在 ILU 预条件子的背景下：

```
for i=1..n
  x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j]) / a[i,i]
```

我们可以简单地编写并行代码：

```
for i=myfirstrow..mylastrow
  x[i] = (y[i] - sum over j=1..i-1 ell[i,j]*x[j]) / a[i,i]
```

但现在存在一个问题。我们不能再声称 ‘假设处理器拥有右侧所有数据的本地副本’，因为向量 x 同时出现在等式左右两侧。虽然矩阵 - 向量乘积原则上可以完全按矩阵行并行计算，但这类三角求解代码是递归的，因此本质上是串行的。

在并行计算环境中，这意味着第二个处理器要开始工作，必须等待第一个处理器计算出 x 的某些组件。显然，第二个处理器必须等待第一个处理器完成才能启动，第三个处理器又需要等待第二个，以此类推。令人失望的结论是：并行环境下任何时刻都只有一个处理器处于活跃状态，总耗时与串行算法相同。这对稠密矩阵来说并非大问题，因为单行操作中仍可发掘并行性（参见 7.12 节），但在稀疏矩阵场景下，这意味着我们必须重新设计才能使用不完全分解法。

在接下来的几个小节中，我们将探讨不同策略以寻找能在并行环境下高效运行的预条件子。

7.7.4 块雅可比方法

针对三角求解的序列性问题，已有多种改进方案被提出。例如，我们可以直接让处理器忽略来自其他处理器的 x 组件：

```
for i=myfirstrow..mylastrow
  x[i] = (y[i] - sum over j=myfirstrow..i-1 ell[i,j]*x[j])
         / a[i,i]
```

这与串行算法在数学上并不等价（技术上称为以 ILU 为局部求解的块雅可比方法），但由于我们仅寻求近似解 $K \approx A$，这不过是稍显粗略的近似。

习题 7.29. 利用你之前编写的高斯 - 赛德尔代码模拟并行运行。增加（模拟的）处理器数量会产生什么影响?

块方法背后的思想可以通过图示来理解；参见图 7.16。实际上，我们通过对忽略处理器间所有连接的矩阵进行不完全 LU 分解（ ILU ）。由于在边界值问题（ BVP ）中所有点相互影响（见章节 4.2.1 ），若在串行计算机上执行，使用连接较少的预条件子会增加迭代次数。然而，块方法是并行的，正如我们之前所观察到的，串行预条件子在并行环境中效率极低，因此我们接受了这种迭代次数的增加。

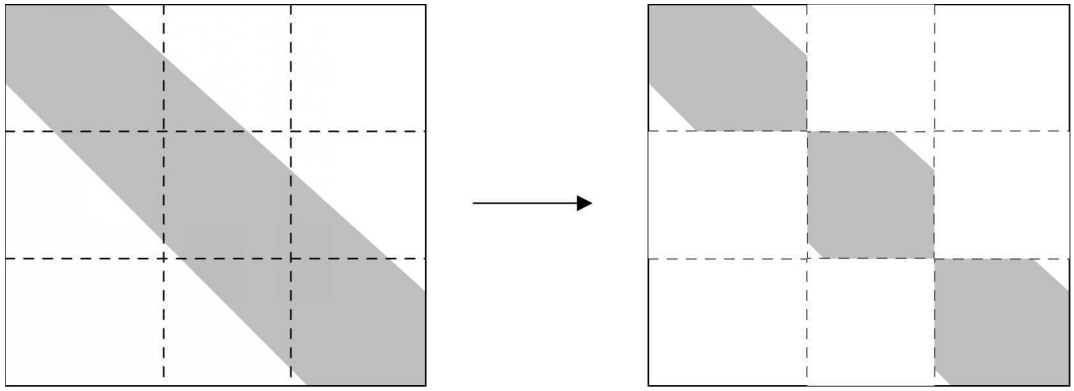


图 7.16：对应块雅可比预条件子的稀疏模式。

7.7.5 并行 ILU

块雅可比预条件子通过解耦区域部分来运作。虽然这种方法可能具有高度并行性，但其迭代次数可能多于真正的 ILU 预条件子。幸运的是，我们可以实现并行 ILU 方法。由于需要后续关于变量重排序的内容，我们将此讨论推迟至 7.8.2.3 节。

7.8 排序策略与并行性

前文已指出，求解线性方程组本质上是递归活动。对于稠密系统，由于操作数量相对于递归深度足够大，寻找并行性相对直接。而稀疏系统则需要更精巧的方法。本节将探讨多种重排方程（或等价地置换矩阵）的策略，以增加可用并行性。

这些策略均可视为高斯消元法的变体。通过构造不完全变体，

其中（参见章节 5.5.6.1），所有这些策略同样适用于构建迭代预条件子 solution methods.

7.8.1 嵌套分割法

在先前讨论中，我们通常考察方形域及变量的字典序排列。本节将介绍变量的嵌套分割排序法，该方法最初旨在减少填充量（见 [74]）。更重要的是，如后文所述，该排序法在并行计算场景中同样具有优势。

嵌套分割法是一种递归过程，用于确定计算域中未知量的非平凡排序。第一步将计算域划分为两部分，中间以分隔带隔开（见图 7.17）。具体而言，分隔带的宽度需确保左右两部分之间不存在任何连接关系。

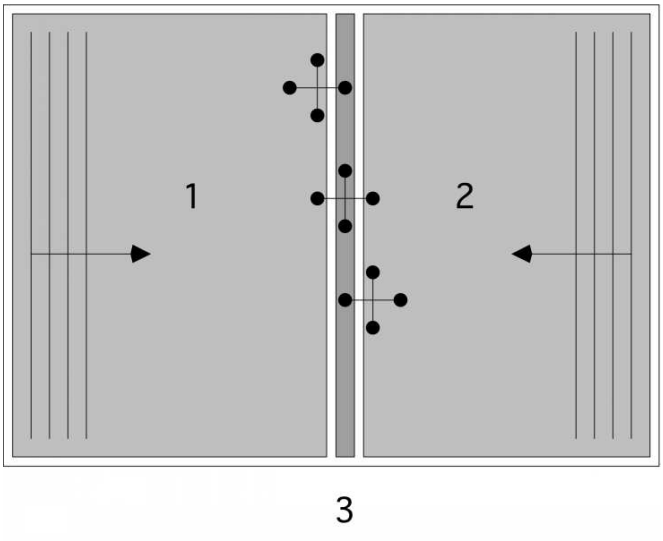


Figure 7.17: Domain dissection into two unconnected subdomains and a separator.

和右侧子域。生成的矩阵 ADD 具有 3×3 结构，对应域的三个部分。由于子域 Ω_1 和 Ω_2 互不连通，子矩阵 ADD$_{12}$ 与 ADD$_{21}$ 为零。

$$A^{\mathrm{DD}} = \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ \emptyset & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \left(\begin{array}{ccccc|ccccc|c} \star & \star & & & & & & & & & 0 \\ \star & \star & \star & & & & & & & & \vdots \\ & \ddots & \ddots & \ddots & & & & \emptyset & & & \vdots \\ & & \star & \star & \star & & & & & & 0 \\ & & & \star & \star & & & & & & \star \\ \hline & & & & & \star & \star & & & & 0 \\ & & & & & \star & \star & \star & & & \vdots \\ & & \emptyset & & & & \ddots & \ddots & \ddots & & \vdots \\ & & & & & & & \star & \star & \star & 0 \\ & & & & & & & & \star & \star & \star \\ \hline 0 & \cdots & \cdots & 0 & \star & 0 & \cdots & \cdots & 0 & \star & \star \end{array}\right) \begin{array}{l} \left.\vphantom{\begin{array}{c}a\\a\\a\\a\\a\end{array}}\right\} (n^2-n)/2 \\ \left.\vphantom{\begin{array}{c}a\\a\\a\\a\\a\end{array}}\right\} (n^2-n)/2 \\ \left.\vphantom{\begin{array}{c}a\end{array}}\right\} n \end{array}$$

这种通过分隔符划分域的过程也被称为域分解或子结构法，不过该名称也与结果矩阵的数学分析相关 [14]。在这个矩形域的例子中，寻找分隔符自然是简单的。但对于从边值问题得到的方程类型，通常可以高效地为任何域找到分隔符 [139]；另见章节 20.6.2。

现在让我们考虑该矩阵的 LU 分解。若按其 3×3 块结构进行分解，可得

$$A^{\mathrm{DD}} = LU = \begin{pmatrix} I & & \\ \emptyset & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ & A_{22} & A_{23} \\ & & S_{33} \end{pmatrix}$$

其中

$$S_{33} = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}.$$

此处关键事实在于

- 贡献项 $A_{31}A_{11}^{-1}A_{13}$ 与 $A_{32}A_{22}^{-1}A_{23}$ 可被并行计算，因此该分解过程具有高度并行性；且
- 在向前和向后求解过程中，解的组件 1 与组件 2 可被同时计算，故求解过程同样具备高度并行性。

第三区块无法简单地并行处理，因此这在算法中引入了顺序组件。让我们更仔细地观察 S_{33} 的结构。

练习 7.30。在 5.4.3.1 节中，您已经了解了 LU 分解与图论之间的联系：消除一个节点会导致该节点从图中移除，但同时会添加某些新的连接。证明在消除前两组 Ω_1, Ω_2 变量后，分隔符上剩余矩阵对应的图将完全连通。

最终结果是，在消去区块 1 和 2 中的所有变量后，我们得到一个大小为 $n \times n$ 的完全稠密矩阵 S_{33}。

引入分隔符使我们获得了一种双向并行的分解方式。现在，我们迭代这一过程：在区块 1 和 2 内部放置分隔符（见图 7.18），这将生成如下的矩阵结构：

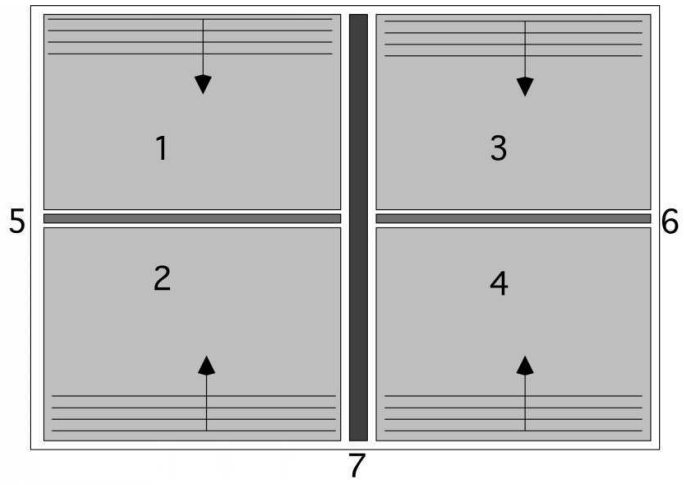


Figure 7.18: A four-way domain decomposition.

$$A^{\mathrm{DD}} = \left(\begin{array}{cccc|cc|c} A_{11} & & & & A_{15} & & A_{17} \\ & A_{22} & & & A_{25} & & A_{27} \\ & & A_{33} & & & A_{36} & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ \hline A_{51} & A_{52} & & & A_{55} & & A_{57} \\ & & A_{63} & A_{64} & & A_{66} & A_{67} \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right)$$

(注意与第 5.4.3.4 节中 ‘箭头’ 矩阵的相似性，并回顾该结构导致填充量减少的论点。) 其 LU 分解如下：

$$\left(\begin{array}{cccc|cc|c} I & & & & & & \\ & I & & & & & \\ & & I & & & & \\ & & & I & & & \\ \hline A_{51}A_{11}^{-1} & A_{52}A_{22}^{-1} & & & I & & \\ & & A_{63}A_{33}^{-1} & A_{64}A_{44}^{-1} & & I & \\ \hline A_{71}A_{11}^{-1} & A_{72}A_{22}^{-1} & A_{73}A_{33}^{-1} & A_{74}A_{44}^{-1} & A_{75}S_5^{-1} & A_{76}S_6^{-1} & I \end{array}\right).$$

$$\left(\begin{array}{cccc|cc|c} A_{11} & & & & A_{15} & & A_{17} \\ & A_{22} & & & A_{25} & & A_{27} \\ & & A_{33} & & & A_{36} & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ \hline & & & & S_5 & & A_{57} \\ & & & & & S_6 & A_{67} \\ \hline & & & & & & S_7 \end{array}\right)$$

其中

$$S_5 = A_{55} - A_{51}A_{11}^{-1}A_{15} - A_{52}A_{22}^{-1}A_{25}, \quad S_6 = A_{66} - A_{63}A_{33}^{-1}A_{36} - A_{64}A_{44}^{-1}A_{46}$$
$$S_7 = A_{77} - \textstyle\sum_{i=1,2,3,4} A_{7i}A_{ii}^{-1}A_{i7} - \sum_{i=5,6} A_{7i}S_i^{-1}A_{17}.$$

构建分解的过程如下：

- 区块 A_{ii} 可并行分解，对应 $i = 1,2,3,4$；类似地，贡献项 $A_{5i}A_{ii}^{-1}A_{i5}$（对应 $i = 1, 2$）、$A_{6i}A_{ii}^{-1}A_{i6}$（对应 $i = 3, 4$）及 $A_{7i}A_{ii}^{-1}A_{i7}$（对应 $i = 1, 2, 3, 4$）也可并行构建。
- 舒尔补矩阵 S_5、S_6 被并行生成并随后分解，而贡献项 $A_{7i}S_i^{-1}A_{17}$（对应 $i = 5, 6$）也并行构建。
- 最终形成并分解舒尔补 S_7。

类比上述推理过程，我们得出结论：在消去块 1、2、3、4 后，更新后的矩阵 S_5、S_6 是大小为 $n/2$ 的稠密矩阵；而在消去块 5、6 后，舒尔补 S_7 是大小为 n 的稠密矩阵。

练习 7.31. 证明用 ADD 求解系统具有与上文所述构建因子分解过程相似的并行性。

为便于后续引用，我们将集合 1 和 2 互为兄弟集，同理 3 和 4 也互为兄弟集。集合 5 是 1 和 2 的父集，6 是 3 和 4 的父集；5 和 6 互为兄弟集，7 则是 5 和 6 的父集。

7.8.1.1 区域分解

在图 7.18 中，我们通过递归过程将区域划分为四部分。这引出了我们对嵌套剖分法的讨论。也可以立即将区域分割为任意数量的带状子域或网格状子域。只要分隔带足够宽，就能形成具有多个独立子域的矩阵结构。如上述讨论所述，LU 分解的特征将表现为

- 子域的并行处理，包括分解过程及 L, U 求解过程，以及
- 需要在分隔带结构上求解的方程组。

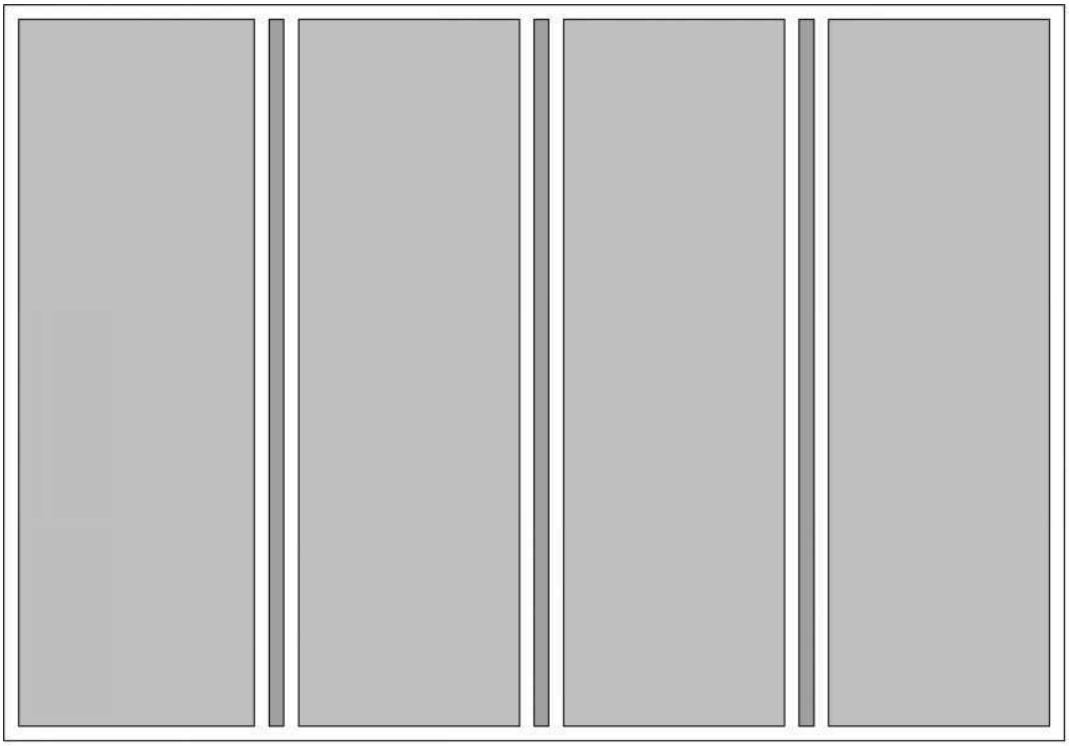


图 7.19：单向区域分解示意图。

习题 7.32 二维边界值问题的矩阵具有块三对角结构。将域划分为四条带，即使用三个分隔线（参见图 7.19 ）。注意原矩阵中的分隔线是解耦的。现绘出子域消去后分隔线上所得系统的稀疏结构，证明该系统是块三对角的。

在我们迄今讨论的所有域分解方案中，使用的域在二维以上都是矩形或 “ 砖块 ” 形状。这些讨论同样适用于二维或三维更一般的域，但诸如寻找分隔线等操作会变得困难得多 [138], —— 这在并行计算情形中更为突出。相关入门介绍可参阅章节 20.6.2 。

7.8.1.2 复杂度

嵌套分割方法会重复上述过程，直到子域变得非常小。在理论分析中，我们会持续分割直至子域大小为 1×1，但在实际应用中，可以在尺寸达到 32 等大小时停止，并使用高效的密集求解器对块进行分解和求逆。

为了推导算法的复杂度，我们再次观察图 7.18，并发现从复杂度角度考虑，完整的递归嵌套分割分解所需的总空间是以下各项之和

- 一个大小为 n 的分隔器上的密集矩阵，加上
- 两个大小为 $n/2$ 的分隔器上的密集矩阵，
- 综合考量 $3/2\,n^2$ 空间与 $5/12\,n^3$ 时间；
- 上述两项将在尺寸为 $(n/2)\times(n/2)$ 的四个子域上重复出现。

With the observation that $n=\sqrt{N}$, this sums to

$$\begin{aligned}\text{space} &= 3/2n^2 + 4\cdot 3/2(n/2)^2 + \cdots \\ &= N(3/2 + 3/2 + \cdots) \quad \log n \text{ terms} \\ &= O(N\log N)\end{aligned}$$

$$\begin{aligned}\text{time} &= 5/12n^3/3 + 4\cdot 5/12(n/2)^3/3 + \cdots \\ &= 5/12N^{3/2}(1 + 1/4 + 1/16 + \cdots) \\ &= O(N^{3/2})\end{aligned}$$

显然，我们现在得到的分解在很大程度上是并行的，且该过程完成于 $O(N$ 对数 $N)$ 空间，而非 $O(N^{3/2})$ （参见章节 5.4.3.3 ）。分解时间也从 $O(N^2)$降至 $O(N^{3/2})$。

遗憾的是，这种空间节省仅适用于二维情况：在三维空间中我们需要

- 一个尺寸为 $n\times n$ 的分隔器，占用 $(n\times n)^2 = N^{4/3}$ 空间和 $1/3\cdot(n\times n)^3 = 1/3\cdot N^2$ 时间，
- 两个尺寸为 $n\times n/2$ 的分隔器，共占用 $N^{3/2}/2$ 空间和 $1/3\cdot N^2/4$ 时间，
- 四个大小为 $n/2\times n/2$ 的分隔符，占用 $N^{3/2}/4$ 空间和 $1/3\cdot N^2/16$ 时间，
- 总计占用 $7/4N^{3/2}$ 空间和 $21/16N^2/3$ 时间；
- 下一层级有 8 个子域，这些子域以 $n\to n/2$ 和因此 $N\to N/8$ 的方式贡献这些项。

这使得总空间

$$\frac{7}{4}N^{3/2}(1 + (1/8)^{4/3} + \cdots) = O(N^{3/2})$$

以及总时间

$$\frac{21}{16}N^2(1 + 1/16 + \cdots)/3 = O(N^2).$$

我们不再拥有二维情况下的巨大节省优势。

7.8.1.3 不规则区域

S到目前为止，我们已分析了笛卡尔区域上的嵌套剖分。更复杂的分析表明
二维一般问题的阶数改进依然成立，而三维问题通常具有更高的复杂度 [138]。

对于不规则区域，诸如如何找到分隔符等问题变得更加复杂。为此，需将 “ 二维区域 ” 的概念推广至平面图。此类图中可找到足够 “ 优质 ” 的分隔符 [139]，但需依赖串行算法。并行算法则基于谱图论；参见 20.6 节。

7.8.1.4 并行性

嵌套剖分方法明显引入了大量并行性，我们可以将其归类为任务并行（章节 2.5.3 ）：每个分隔符关联的任务包括分解其矩阵，随后在其变量上求解线性系统。然而，这些任务并非独立：在图 7.18 中，域 7 的分解需等待 5 和 6 完成，而它们又需等待 1 、 2 、 3 、 4 完成。因此，我们得到具有树状依赖关系的任务：每个分隔符矩阵仅当其子矩阵完成分解后才能被分解。

将这些任务映射到处理器并非易事。首先，若处理共享内存，可采用简单的任务队列：

> 队列 $\leftarrow \{\}$ **用于** 所有底层子域 d **执行**添加 d 至队列**当**队列非空时**若** 处理器空闲**则**分配队列任务给它**若** 任务完成且其兄弟任务完成**则**将其父任务加入队列

这里的主要问题在于，当处理器数量超过任务数量时会导致负载不均衡。由于各级分隔符的规模呈指数级增长（稠密矩阵分解的工作量与矩阵尺寸的三次方成正比！），最后阶段的任务往往最为繁重，这使得问题进一步加剧。因此，对于较大的分隔符，我们必须从任务并行转向中等粒度的并行模式，即多个处理器协作完成一个区块的分解工作。

在分布式内存架构下，由于涉及大规模数据迁移（但请注意此时计算量随矩阵尺寸的高次方增长，反而使通信开销相对降低），我们可以通过简单的任务队列解决并行问题。解决方案是采用某种形式的域分解。如图 7.18 所示，可配置四个处理器分别对应区块 1 、 2 、 3 、 4。随后处理器 1 和 2 需协商由谁负责分解区块 5 （同理处理器 3 和 4 处理区块 6 ），或者两者冗余执行该任务。

7.8.1.5 预处理

与所有分解方法一样，可以通过使分解不完全，将嵌套剖分方法转化为预条件子。（关于不完全分解的基本思想，参见章节 5.5.6.1 ）。

图 7.20：直线上点的红黑排序。

然而，此处因子分解完全基于分块矩阵表述，除以主元元素的操作变为对主元分块矩阵求逆或求解系统。我们不再深入探讨；详情参阅文献 [6, 56, 149]。

7.8.2 变量重排序与着色：独立集

通过图着色技术（第 20.3 节）可在稀疏矩阵中实现并行化。由于 ‘颜色’ 被定义为仅与其他颜色相连的节点集合，它们本质上相互独立，因而可并行处理。这引出了以下策略：

1. 将问题的邻接图分解为少量称为 ‘颜色’ 的独立集；2. 用等同于颜色数量的顺序步骤求解问题；每一步骤中均有大量可并行处理的独立节点。

7.8.2.1 红黑着色

我们从一个简单的例子开始，考虑一个三对角矩阵 A。方程 $Ax = b$ 的形式如下

$$\begin{pmatrix} a_{11} & a_{12} & & & & \emptyset \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ & & \ddots & \ddots & \ddots & \\ \emptyset & & & & & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix}$$

我们观察到 x_i 直接依赖于 x_{i-1} 和 x_{i+1}，但不依赖于 x_{i-2} 或 x_{i+1}。因此，让我们看看如果对索引进行排列，将每隔一个组件分组会发生什么。

图示上，我们取点 $1, \ldots, n$ 并将其着色为红与黑（图 7.20 ），然后对它们进行排列，先取所有红点，再取所有黑点。经相应排列后的矩阵如下所示：

$$\begin{pmatrix} a_{11} & & & & a_{12} & & \\ & a_{33} & & & a_{32} & a_{34} & \\ & & a_{55} & & & \ddots & \ddots \\ & & & \ddots & & & \\ a_{21} & a_{23} & & & a_{22} & & \\ & a_{43} & a_{45} & & & a_{44} & \\ & & \ddots & \ddots & & & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_5 \\ \vdots \\ y_2 \\ y_4 \\ \vdots \end{pmatrix}$$

经过置换后的 A，高斯 - 赛德尔矩阵 $D_A + L_A$ 呈现如下形态

$$\begin{pmatrix} a_{11} & & & & \emptyset & & \\ & a_{33} & & & & & \\ & & a_{55} & & & & \\ & & & \ddots & & & \\ a_{21} & a_{23} & & & a_{22} & & \\ & a_{43} & a_{45} & & & a_{44} & \\ & & \ddots & \ddots & & & \ddots \end{pmatrix}$$

这能带来什么优势？让我们详细展开求解方程组 $Lx = y$ 的过程。

对于 $i = 1, 3, 5, \ldots$ **执行**求解
$x_i \leftarrow y_i / a_{ii}$ **对于** $i = 2, 4, 6, \ldots$ **执行**计算 $t = a_{ii-1}x_{i-1} + a_{ii+1}x_{i+1}$ 求解 $x_i \leftarrow (y_i - t)/a_{ii}$

显然该算法包含三个阶段，每个阶段在半数域点上并行执行。如图 7.21 所示。理论上我们可以配置相当于域点数量一半的处理器，但实际应用中每个处理器将负责一个子域。

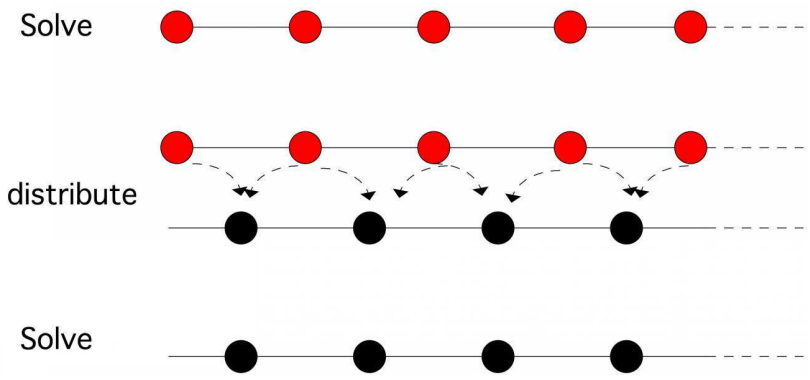


Figure 7.21: Red-black solution on a 1d domain.

域点数量的处理器，但实际上每个处理器将管理一个子域。通过图 7.22 可见，这仅需极少量通信：每个处理器最多向相邻节点发送两个红色数据点的信息。

练习 7.33. 论证此处的邻接图是一个二分图。我们可以看到这类图（以及广义上的着色图）与并行性相关联。你能否也指出非并行处理器的性能优势？

红黑排序同样适用于二维问题。让我们对点集 (i, j) 其中 $1 \leq i, j \leq n$ 应用红黑排序。这里我们首先对第一行的奇数点进行连续编号 $(1, 1), (3, 1), (5, 1), \ldots$，然后是第二行的偶数点 $(2, 2), (4, 2), (6, 2), \ldots$，第三行的奇数点，以此类推。完成对域内半数点的编号后，我们继续处理第一行的偶数点、第二行的奇数点等。如图 7.23 所示，此时

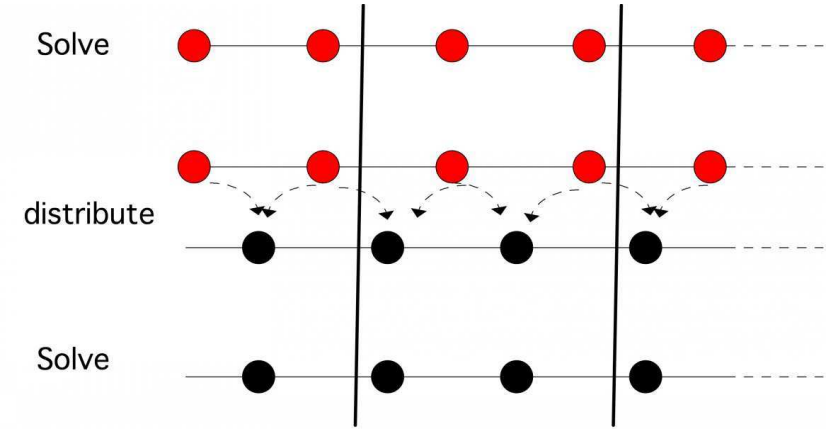


图 7.22：一维域上的红黑并行解法。

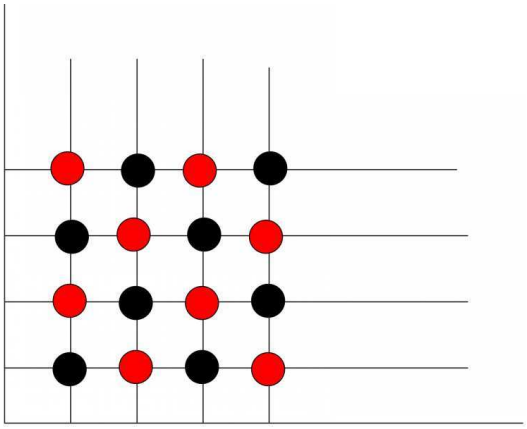


图 7.23：二维域变量的红黑排序。

红点仅与黑点相连，反之亦然。用图论术语来说，你已找到矩阵图的双色着色（定义见附录 20 ）。

习题 7.34. 将红黑排序应用于二维边值问题 (4.51)，绘制所得矩阵结构。

T红黑排序是图着色（有时称为多色着色）的简单示例。简而言之
c例如我们在章节 4.2.3 中考虑的单位正方形域或其向 3D 的扩展
*n*邻接图的数量可以轻易确定；在不太规则的情况下则较为困难。

习题 7.35. 你已看到，未知数的红黑排序与规则的五点星形模板结合会产生两组互不连接的变量子集，即它们构成了矩阵图的二着色。若节点通过图 4.3 中的第二个模板连接，你能找到一种着色方案吗?

7.8.2.2 通用着色

对于稀疏矩阵图所需的颜色数量存在一个简单的界限：颜色数量最多为 $d+1$，其中 d 是图的度数。为了证明我们可以用 $d+1$ 种颜色为度数为 d 的图着色，考虑一个度数为 d 的节点。无论其邻居如何着色，在可用的 $d+1$ 种颜色中总有一种未被使用。

练习 7.36. 考虑一个稀疏矩阵，其图可以用 d 种颜色着色。首先枚举第一种颜色的未知数，然后是第二种颜色，依此类推，对矩阵进行置换。你能说明置换后矩阵的稀疏模式有何特点吗？

如果你寻求线性系统的直接解法，可以在消除一种颜色后对剩余矩阵重复着色和置换的过程。对于三对角矩阵，你会发现剩余矩阵仍是三对角矩阵，因此如何继续该过程是显而易见的。这被称为**递归倍增**。如果矩阵不是三对角而是**块三对角**，则可以按块执行此操作。

7.8.2.3 多色并行 ILU

在章节 7.8.2 中，您看到了 图着色 与排列的结合。设 P 为将同色变量分组的排列矩阵，则 $\tilde{A} = P^t AP$ 是具有以下结构的矩阵：

- $\tilde{A}$ 具有块状结构，其块数等于 A 的邻接图中的颜色数量；且
- 每个对角块均为对角矩阵。

现在，如果您正在进行迭代系统求解，并寻找并行预处理器，可以使用此排列后的矩阵。考虑求解 $Ly = x$ 的排列系统。我们将常规算法（章节 5.3.4）改写为：

对于颜色集合中的 c：对于颜色 c 的变量 i：$y_i \leftarrow x_i - \sum_{j<i} \ell_{ij} y_j$

习题 7.37. 证明在从自然排序的 ILU 分解转为颜色置换排序时，求解系统 $LUx = y$ 的浮点运算次数（在最高阶项上）保持不变。

这些着色操作究竟有何意义？求解过程仍然是串行的 … 确实，外层循环按颜色顺序执行是串行的，但同一颜色的所有节点相互独立，因此可以并行求解。如果我们采用常规区域划分并结合多色着色（见图 7.24），所有处理器在所有颜色阶段都保持活跃；参见图 7.25。当然，细看该图会发现最后一个颜色阶段有一个处理器处于闲置状态。当每个处理器负责大量节点时，这种情况不太可能发生，但仍可能存在负载不均衡问题。

剩下的问题是如何并行生成多色着色。寻找最优颜色数量属于 NP 难问题。所幸我们并不严格要求最优解，因为无论如何都在进行不完全分解。（甚至有观点认为，使用稍多的颜色数可能减少迭代次数。）

一种优雅的并行多色着色算法由 [113, 142] 发现：

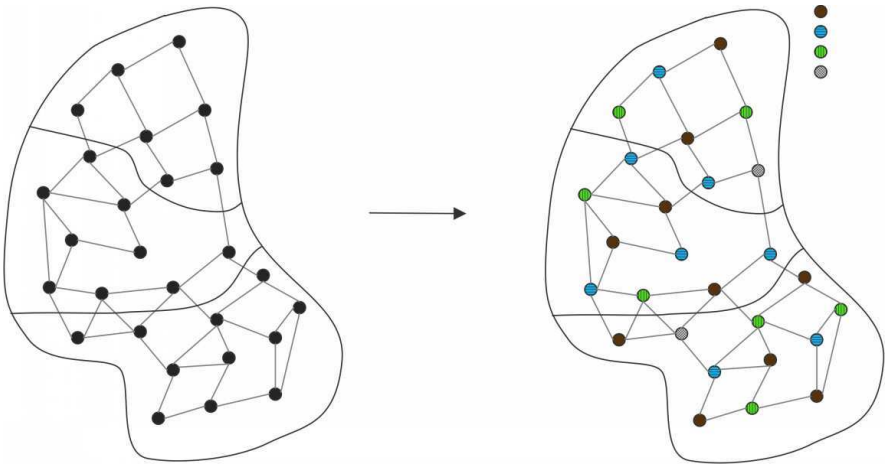


图 7.24：带有颜色标记节点的分区域。

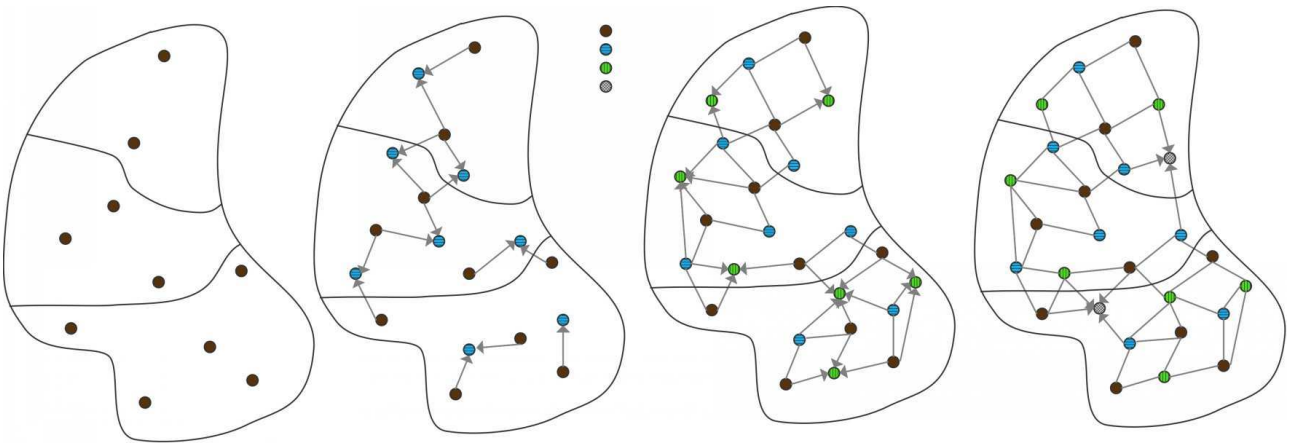


图 7.25：分四步求解并行多色 ILU。

1. 为每个变量分配一个随机值。2. 找出随机值高于所有相邻变量的变量，即标记为颜色 1。3. 接着找出随机值高于所有未被标记为颜色 1 的相邻变量的变量，即标记为颜色 2。4. 迭代此过程，直至所有点都被着色。

7.8.3 不规则迭代空间

A在显式时间步进或稀疏矩阵向量乘法的背景下，应用计算模板。
p乘积运算是可以并行化的。然而，在实际操作中，分割迭代空间可能并非易事。如果迭代
s若空间为笛卡尔砖块结构，即便存在嵌套并行性，处理起来也相当简便。

然而，实际上迭代空间可能复杂得多。在这种情况下，可以采取以下措施：

1. 遍历迭代循环以计算内部迭代的总次数；然后将其划分为与进程数量相等的若干部分。2. 再次遍历循环以确定每个进程的起始和结束索引值。3. 随后重写循环代码，使其能够在这样的子范围内运行。

更一般的讨论，请参阅章节 2.10 关于负载均衡 .

7.8.4 为缓存效率排序

模板操作的性能通常较低。这些操作没有明显的缓存重用；它们往往类似于流操作，即从内存中获取大量数据且仅使用一次。如果仅执行单次模板计算，故事到此结束。然而，通常我们会进行多次此类更新，此时可应用类似于循环分块的技术（见章节 6.7 ）。

若考虑模板的形状，我们能比简单分块做得更好。图 7.26 展示了

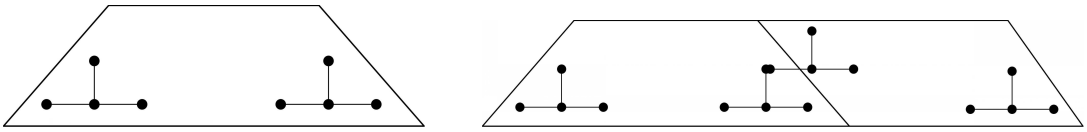


图 7.26：缓存优化迭代空间遍历的第一步与第二步。

(左) 我们首先计算一个被缓存容纳的时空梯形区域，随后 (右) 计算另一个基于首个梯形的缓存容纳梯形区域 [69]。

7.9 偏微分方程线性系统求解中的并行性

偏微分方程的数值求解是一项重要任务，其通常所需的高精度使其成为并行处理的理想候选对象。若我们探究其可并行化的程度，特别是能达到何种加速效果，需区分该问题的不同层面。

首先我们可以探讨问题本身是否存在内在的并行性。在全局层面上通常不会存在这种情况（如果问题的各个部分完全解耦，那它们就应该是独立的问题了，对吧？），但在更细粒度上可能存在并行性。

例如，观察时间相关问题，并参考第 4.2.1 节，我们可以说每个后续时间步当然依赖于前一个时间步，但并非下一步的每个单独点都依赖于上一步的每个点：存在一个影响区域。因此有可能通过划分问题域来实现并行化。

在偏微分方程中，我们还可以基于其以 PDE 理论中所谓特征线为表征的特性，对其并行求解做出高层论断。波动方程直观上是严格向前传播的现象，因此下一步时间步是前一步的显式、可并行计算的函数。而椭圆型问题（参见 refsec:elliptic 节）则存在所有点相互依赖的特性。因此并行求解很可能需要涉及一系列更显式的近似方法。我们将在下个小节通过简单案例进行讨论。

7.9.1 单位正方形上的 Laplace/Poisson 方程

The 单位正方形上的 *Laplace* 方程

$$-\Delta u = 0, \qquad \text{on } \Omega = [0,1]^2 \tag{7.2}$$

是最简单且兼具严肃性与实用性的偏微分方程问题。若采用 FEM 或 FDM 进行二阶离散化处理，即可得到矩阵 (4.56)。

现有以下方法可用于求解该隐式系统。

- The *Jacobi* 迭代法（方程 5.16 ）虽能实现全节点并行计算，但需对求解域进行多次全局扫描。
- 与之密切相关的 *Gauss-Seidel* 迭代法 收敛更快，但属于隐式方法。通常采用
  并行化的一种方式是通过多色着色；参见章节 7.8.2。另一种可行的方法是采用波前；参见章节 7.10.1。
- 更优的迭代方案是使用共轭梯度（ *CG* ）方法；参见章节 5.5.11。该方法通常通过预条件子加速，而预条件子常选用上述雅可比或高斯 - 赛德尔迭代的某种变体；参见章节 5.5.6 和 7.7。
- 最后，诸如拉普拉斯方程之类的简单方程可通过多重网格方法求解，无论是单独使用还是作为迭代方法的预条件子。本书将不对此展开讨论。

7.9.2 算子分裂

在某些情况下，需要对二维或三维数组的所有方向进行隐式计算。例如，在章节 4.3 中，您已了解热方程的隐式求解如何引发重复系统

$$(\alpha I + \frac{d^2}{dx^2} + \frac{d^2}{dy^2})u^{(t+1)} = u^{(t)} \tag{7.3}$$

无需证明，我们可以断言时变问题同样可通过

$$(\beta I + \frac{d^2}{dx^2})(\beta I + \frac{d^2}{dy^2})u^{(t+1)} = u^{(t)} \tag{7.4}$$

对于合适的 β 求解。该方案不会在每个独立时间步上计算相同的值，但会收敛至相同的稳态。此方案亦可用作边值问题（ BVP ）中的预处理器。

该方法具有显著优势，主要体现在运算量方面：原始系统的求解需对矩阵进行分解（这会导致填充），或通过迭代法求解。

习题 7.38. 分析这些方法的相对优劣，给出粗略运算量估算。需同时考虑 α 与 t 存在依赖关系及不存在依赖关系的情况，并讨论不同运算的预期速度。

当考虑并行求解 (7.4) 时，另一优势显现。注意我们拥有二维变量集合 u_{ij}，但算子 $I + d^2u/dx^2$ 仅连接 u_{ij}、 u_{ij-1}、 u_{ij+1}。即每个对应 i 值的行可独立处理。因此，两个算子均可通过一维区域划分实现完全并行求解。而 (7.3) 中的系统求解则存在并行性限制。

遗憾的是，存在一个严重的复杂性问题： x 方向的算子需要在单一方向上对域进行分区，而 y 方向的算子则需要在另一方向上进行分区。通常采用的解决方案是在两次求解之间转置 u_{ij} 值矩阵，以便相同的处理器分解能同时处理两者。这一转置操作可能消耗每个时间步长中相当大一部分的处理时间。

练习 7.39. 讨论采用 $P = p \times p$ 处理器网格对域进行二维分解的优势与问题。你能提出一种缓解这些问题的方法吗?

加速这些计算的一种方法是用显式操作替代隐式求解；参见章节 7.10.3。

7.10 并行性与隐式操作

在讨论 IBVP （章节 4.1.2.2 ）时，你已了解到隐式操作在数值稳定性方面可能具有显著优势。然而，你也看到它们使得基于简单操作（如矩阵 - 向量乘积）的方法与基于更复杂线性系统求解的方法之间产生差异。当你开始并行计算时，隐式方法还会带来更多问题。

以下练习旨在回顾我们在线性代数总体讨论中已指出的部分问题；参见第 5 章。

习题 7.40. 设 A 为矩阵

$$A = \begin{pmatrix} a_{11} & & \emptyset & \\ a_{21} & a_{22} & & \\ & \ddots & \ddots & \\ \emptyset & & a_{n,n-1} & a_{nn} \end{pmatrix}. \tag{7.5}$$

证明矩阵向量积 $y \leftarrow Ax$ 与通过求解三角系统 $Ax = y$（而非求逆 A）得到的系统解 $x \leftarrow A^{-1}y$具有相同的运算量。

现在考虑并行化乘积 $y \leftarrow Ax$。假设我们有 n 个处理器，每个处理器 i 存储 x_i 和 A 的第 i 行。证明乘积 Ax 可以在除第一个处理器外的所有处理器上无空闲时间地完成计算。

对于三角系统 $Ax = y$的求解能否实现同样的并行化？证明直接实现会导致每个处理器在计算的 $(n-1)/n$ 比例时间内处于空闲状态。

接下来我们将探讨处理这一固有顺序组件的多种方法。

7.10.1 波前法

前文提到，求解规模为 N 的下三角线性系统的串行时间复杂度可能高达 N 步。但实际上，情况往往没有这么糟糕。诸如求解三角系统这类隐式算法本质上是串行的，但其实际步骤数可能比初看时要少。

习题 7.41. 重新观察单位正方形上二维边值问题通过中心差分离散化得到的矩阵 (4.56)。该矩阵源于对变量进行字典序排列。若改为按对角线顺序排列未知数，试推导矩阵结构。关于块的大小及块本身的结构，你能得出什么结论？这对并行计算有何启示？

我们无需构造矩阵即可进行并行性分析。如前所述，采用模板方法往往更有效。让我们重新观察描述二维边值问题的有限差分模板的图 4.1 。下三角因子对应的模板图示见 7.27，该图描述了求解下三角系统时的串行性 $x \leftarrow L^{-1}y$:

$$x_k = y_k - \ell_{k,k-1}x_{k-1} - \ell_{k,k-n}x_{k-n}$$

换言之，若已知点 k 左侧（即变量 $k-1$）和下方（变量 $k-n$）的邻点值，便可求出该点的值。

反过来看，如果我们知道 x_1，就不仅能求出 x_2，还能得到 x_{n+1}。下一步我们可以确定 x_3、 x_{n+2} 和 x_{2n+1}。如此延续下去，我们就能通过波前法求解 x：每个波前上的 x 值相互独立，因此可在同一顺序步骤中并行求解。

习题 7.42. 完成这个论证。我们能使用的最大处理器数量是多少？需要多少顺序步骤？最终效率如何？

当然，你不必实际使用并行处理来利用这种并行性。你可以改用向量处理器、向量指令或 *GPU* [141]。

在 5.4.3.5 节中，你了解了用于减少矩阵填充的 ***Cuthill-McKee*** 排序法。我们可以按如下方式修改该算法以生成波前：

1. 选取任意节点，将其称为 ‘零级’。2. 对于 $n+1$ 级，找出与 n 级相连且自身互不连接的点。3. 对于所谓的 ‘逆 Cuthill-McKee 排序’，将各级编号进行倒序处理。

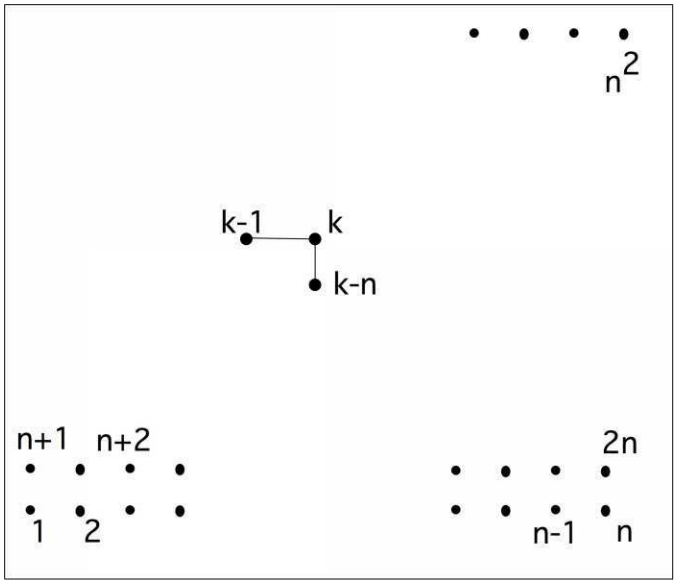


图 7.27：二维边值问题矩阵 L 因子的差分模板。

练习 7.43. 该算法并非完全正确。问题出在哪里？如何修正？证明置换后的矩阵不再是对角占优的三对角矩阵，但仍可能保持带状结构。

7.10.2 递归倍增法

递归关系 $y_{i+1} = a_i y_i + b_i$，例如出现在求解双线性方程组时（参见练习 4.4），看似本质上是串行的。但您已在练习 1.4 中看到，通过一些预处理操作，计算可以实现并行化。其核心思想是将序列进行变换：

$$x_{i+1} = a_i x_i + y_i \Rightarrow x_{i+2} = a_i' x_i + y_i' \Rightarrow x_{i+4} = a_i'' x_i + y_i'' \Rightarrow \cdots$$

现在我们将形式化这一策略，通常称为递归倍增法。首先，取一般形式的双对角矩阵（见公式 7.5），将其缩放为归一化形式，得到待求解问题 x：

$$\begin{pmatrix} 1 & & \emptyset \\ b_{21} & 1 & \\ & \ddots & \ddots \\ \emptyset & & b_{n,n-1} \quad 1 \end{pmatrix} x = y$$

我们将其写作 $A = I + B$。

练习 7.44. 证明通过乘以对角矩阵可实现归一化形式的缩放。求解系统 $(I + B)x = y$ 如何帮助求解 $Ax = y$？用两种不同方法求解该系统所需的运算量分别是多少？

现在我们进行类似高斯消元的操作，但并非从第一行开始，而是从第二行着手。（如果对矩阵 $I+B$ 执行高斯消元或 LU 分解会发生什么？）我们利用第二行来消去 b_{32}：

$$\begin{pmatrix} 1 & & & \varnothing \\ & 1 & & \\ & -b_{32} & 1 & \\ & & & \ddots \\ \varnothing & & & 1 \end{pmatrix} \times \begin{pmatrix} 1 & & & \varnothing \\ b_{21} & 1 & & \\ & b_{32} & 1 & \\ & & \ddots & \ddots \\ \varnothing & & b_{n,n-1} & 1 \end{pmatrix} = \begin{pmatrix} 1 & & & \varnothing \\ b_{21} & 1 & & \\ -b_{32}b_{21} & 0 & 1 & \\ \varnothing & & b_{n,n-1} & 1 \end{pmatrix}$$

将其记为 $L^{(2)}A = A^{(2)}$。同时我们计算 $L^{(2)}y = y^{(2)}$，使得 $A^{(2)}x = y^{(2)}$ 与 $Ax = y$ 具有相同的解。求解变换后的系统带来些许优势：在计算 x_1 后，x_2 和 x_3 可以并行计算。

现在我们重复此消元过程：用第四行消去 b_{54}，第六行消去 b_{76}，依此类推。最终结果是汇总所有 $L^{(i)}$ 个矩阵后得到：

$$\begin{pmatrix} 1 & & & & & & & \varnothing \\ 0 & 1 & & & & & & \\ & -b_{32} & 1 & & & & & \\ & & 0 & 1 & & & & \\ & & & -b_{54} & 1 & & & \\ & & & & 0 & 1 & & \\ & & & & & -b_{76} & 1 & \\ & & & & & & \ddots & \ddots \end{pmatrix} \times (I+B) = \begin{pmatrix} 1 & & & & & & & \varnothing \\ b_{21} & 1 & & & & & & \\ -b_{32}b_{21} & 0 & 1 & & & & & \\ & & b_{43} & 1 & & & & \\ & & -b_{54}b_{43} & 0 & 1 & & & \\ & & & & b_{65} & 1 & & \\ & & & & -b_{76}b_{65} & 0 & 1 & \\ & & & & & \ddots & \ddots & \ddots \end{pmatrix}$$

将其记为 $L(I+B) = C$，此时求解 $(I+B)x = y$ 就转化为 $Cx = L^{-1}y$。

这一最终结果需要仔细研究。

- 首先，计算 $y' = L^{-1}y$ 很简单。（请详述细节。可用的并行度有多少？）
- 求解 $Cx = y'$ 仍是顺序性的，但不再需要 n 步：从 x_1 可得 x_3，由此再得 x_5，依此类推。换言之，x 的奇数编号组件之间仅存在顺序关系。
- x 的偶数编号组件互不依赖，仅依赖于奇数组件：x_2 由 x_1 推导，x_4 由 x_3 推导，以此类推。一旦奇数组件被顺序计算完成，此步骤即可完全并行化。

我们可以单独描述奇数组件的顺序求解过程：

$$\begin{pmatrix} 1 & & \varnothing \\ c_{31} & 1 & \\ & \ddots & \ddots \\ \varnothing & & c_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y'_1 \\ y'_3 \\ \vdots \\ y'_n \end{pmatrix}$$

其中 $c_{i+1i} = -b_{2n+1,2n}b_{2n,2n-1}$。换言之，我们已将规模为 n 的顺序问题简化为同类顺序问题与规模同为 $n/2$ 的并行问题。现在可重复此过程

递归地，将原始问题转化为一系列并行操作，每个操作的规模是前一个的一半。

通过递归加倍计算所有部分和的过程也被称为并行前缀操作。这里我们使用前缀和，但在抽象层面上它可以应用于任何结合运算符；参见第 22 节。

7.10.3 通过显式操作和级数展开逼近隐式操作

有多种原因允许有时用实践中更具优势的不同操作来替代隐式操作，正如你上面所见，隐式操作在实践中可能存在问题。

- 对热方程使用显式方法（第 4.3 节）而非隐式方法同样合理，只要我们遵守步长对显式方法的限制。
- 调整预处理器（第 5.5.8 节）在迭代方法中是允许的，因为它只会影响收敛速度，而不会影响方法最终收敛到的解。您已经在块雅可比方法中看到了这一通用思想的示例；参见第 7.7.4 节。本节剩余部分将展示如何用显式操作替代预处理器中的递推（即隐式操作），从而获得多种计算优势。

求解线性系统是隐式操作的典型示例，由于这归结为求解两个三角系统，让我们探讨寻找替代计算方案来求解下三角系统的方法。若 U 是上三角非奇异矩阵，设 D 为 U 的对角线，并记 $U = D(I - B)$，其中 B 是对角线为零的上三角矩阵，亦称严格上三角矩阵；我们称 $I - B$ 为单位上三角矩阵。

习题 7.45. 设 $A = LU$ 是一个 LU 分解，其中 L 的对角线元素为 1，而 U 的对角线上存放着主元，如第 5.3 节所述。说明如何通过仅求解单位上三角和下三角方程组来解系统 $Ax = b$。证明在求解过程中无需进行除法运算。

我们当前关注的操作是求解系统 $(I - B)x = y$。注意到

$$(I - B)^{-1} = I + B + B^2 + \cdots \tag{7.6}$$

且 $B^n = 0$ 其中 n 为矩阵规模（验证此点！），因此我们可以通过以下方式精确求解 $(I - B)x = y$

$$x = \sum_{k=0}^{n-1} B^k y.$$

显然，我们希望避免显式计算幂 B^k，因此观察到

$$\begin{aligned} x_1 &= \textstyle\sum_{k=0}^{1} B^k y &&= (I + B)y, \\ x_2 &= \textstyle\sum_{k=0}^{2} B^k y &&= (I + B(I + B))y = y + Bx_1, \\ x_3 &= \textstyle\sum_{k=0}^{3} B^k y &&= (I + B(I + B((I + B))))y = y + Bx_2 \end{aligned} \tag{7.7}$$

等等。用于计算 $\sum_{k=0}^{n-1} B^k y$ 的最终算法称为 霍纳法则 （参见第 22.3 节），可见该算法避免了计算矩阵幂 B^k。

练习 7.46. 假设 $I-B$ 是双对角矩阵。证明上述计算需要 $n(n+1)$ 次运算。（如果显式计算 B 的幂次，运算量会是多少？）通过三角解法计算 $(I-B)x=y$ 的运算次数是多少?

我们已将隐式操作转化为显式操作，但不幸的是其运算量很高。然而在实际情况下，如前所述，我们通常可以合理截断矩阵幂级数的求和。

练习 7.47. 设 A 为三对角矩阵

$$A=\begin{pmatrix} 2 & -1 & & & \varnothing \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ \varnothing & & & -1 & 2 \end{pmatrix}$$

来自章节 4.2.2 的一维边值问题。

1. 回顾章节 5.3.3 中对角占优的定义。该矩阵是否对角占优? 2. 证明该矩阵（无选主元）LU 分解中的主元满足递推关系。提示：证明经过 n 次消元步骤（$n\geq 0$）后，剩余矩阵形如

$$A^{(n)}=\begin{pmatrix} d_n & -1 & & & \varnothing \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ \varnothing & & & -1 & 2 \end{pmatrix}$$

并展示 d_{n+1} 与 d_n 之间的关系。

3. 证明序列 $n\mapsto d_n$ 是递减的，并推导极限值。4. 用 d_n 主元表示 L 和 U 因子。5. L 和 U 因子是否对角占优?

上述练习暗示（注意我们并未实际证明！）对于来自边值问题的矩阵，我们发现 $B^k\downarrow 0$在元素大小和范数上均成立。这意味着我们可以通过例如 $x=(I+B)y$ 或 $x=(I+B+B^2)y$来近似求解 $(I-B)x=y$。这样做虽然运算量仍高于直接三角解法，但至少在两方面具有计算优势：

- 显式算法具有更好的流水线特性。
- 如你所见，隐式算法在并行方面存在问题；而显式算法更易于并行化。

另见 [184]。当然，这种近似可能会对整个数值算法的稳定性产生进一步影响。

习题 7.48. 描述霍纳法则的并行化特性；参见方程 (7.7)。

7.11 网格更新

第 4 章的结论之一是，对于时间相关问题，显式方法在计算上比隐式方法更容易。例如，它们通常涉及矩阵 - 向量乘积而非系统求解，且并行化显式操作相当简单：矩阵 - 向量乘积的每个结果值可以独立计算。这并不意味着没有其他值得注意的计算方面。

由于我们处理的是源自某种计算模板的稀疏矩阵，我们采用算子视角。在图 7.12 和 7.13 中，你已看到在域内每个点应用模板如何引发处理器间的特定关系：为了在处理器上评估矩阵 - 向量乘积 $y \leftarrow Ax$，该处理器需要获取其 x- 值的幽灵区域。在关于处理器间域划分的合理假设下，涉及的消息数量将相当少。

习题 7.49. 推断在 FEM 或 FDM 背景下，消息数量为 $O(1)$ 如 $h \downarrow 0$。

在章节 1.6.1 中你已了解到矩阵 - 向量乘积的数据复用率很低，尽管计算存在一定局部性；而在章节 5.4.1.4 中指出，由于稀疏性所需的索引方案，稀疏矩阵 - 向量乘积的局部性更差。这意味着稀疏乘积主要是一种带宽受限算法。

仅观察单次乘积运算时，我们对此能做的改变有限。然而，实践中我们常会连续执行多次此类乘积运算，例如在时间相关过程中的各个步骤。此时，通过调整运算顺序可能降低带宽需求。以下列简单示例说明

$$\forall_i : x_i^{(n+1)} = f\left(x_i^{(n)}, x_{i-1}^{(n)}, x_{i+1}^{(n)}\right) \tag{7.8}$$

假设集合 $\{x_i^{(n)}\}_i$ 过大无法存入缓存。这可以类比于一维空间热方程的显式格式求解场景（参见章节 4.3.1.1 ）。示意图如下：

$$\begin{array}{ccc} x_0^{(n)} & x_1^{(n)} & x_2^{(n)} \\ \downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\ x_0^{(n+1)} & x_1^{(n+1)} & x_2^{(n+1)} \\ \downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\ x_0^{(n+2)} & x_1^{(n+2)} & x_2^{(n+2)} \end{array}$$

在常规计算流程中（即先计算所有 $x_i^{(n+1)}$，再计算所有 $x_i^{(n+2)}$），层级 $n+1$ 的中间值生成后会立即从缓存中清除，随后又作为层级 $n+2$ 量的输入数据重新载入缓存。

然而，如果我们计算的不是一次迭代，而是两次，中间值可能会保留在缓存中。考虑 $x_0^{(n+2)}$：它需要 $x_0^{(n+1)}$、 $x_1^{(n+1)}$，而这些又需要 $x_0^{(n)}$、 … 、 $x_2^{(n)}$。

现在假设我们对中间结果不感兴趣，只关注最终迭代。这种情况适用于迭代至稳态、进行多重网格平滑等场景。图 7.28 展示了一个简单示例。第一个处理器计算层级 $n+2$ 上的 4 个点，为此需要来自层级的 5 个点

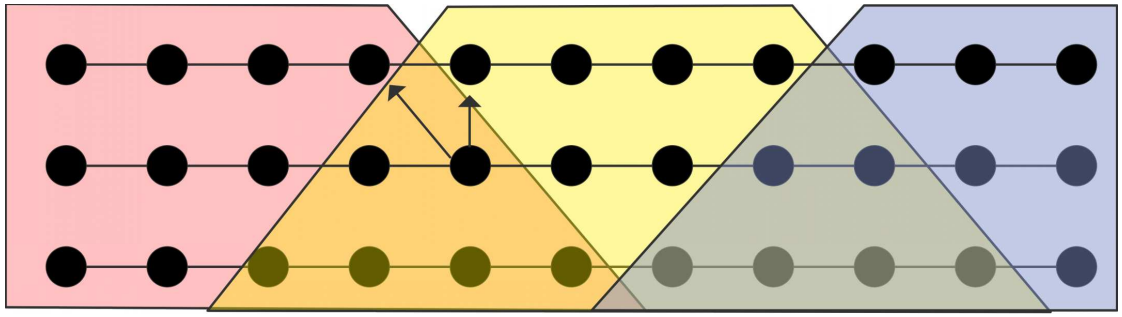


图 7.28：多次迭代中网格点块的计算过程。

$n+1$，这些也需要从层级 n 上的 6 个点计算得出。可见处理器显然需要收集宽度为二的幽灵区域，而非常规单步更新时仅需的宽度为一。第一个处理器计算的其中一个点是 $x_3^{(n+2)}$，它需要 $x_4^{(n+1)}$。该点同样被第二个处理器用于计算属于其的 $x_4^{(n+2)}$。

最简单的解决方案是让这类中间层上的点在冗余计算过程中，由两个不同处理器分别在所需块的计算中重复计算。

习题 7.50. 你能设想出某个点被超过两个处理器冗余计算的情况吗?

对于这种通过分块计算多步更新的方案，我们可以给出多种解读。

- 首先，正如我们之前所强调的，在单处理器上执行此操作可提升局部性：若彩色区块中的所有点（见图）都能放入缓存，我们就能复用中间点。
- 其次，若将此视为分布式内存计算的方案，它能减少消息流量。通常，每个更新步骤中处理器需交换边界数据。若允许一定程度的工作冗余，我们可消除中间层级的数据交换。通信量的减少通常能抵消工作量的增加。

练习 7.51. 讨论在多核计算中采用此策略的情况。节省了什么？潜在的缺陷有哪些?

7.11.1 分析

L让我们分析刚概述的算法。如方程（ 7.8 ）所示，我们限定在一维点集和三点函数上。
p描述问题的参数如下：

- N 表示待更新的点数， M 表示更新步骤数。因此，我们执行了 MN 次函数求值。
- α、 β、 γ 是描述延迟、单点传输时间和操作时间（此处指 f 求值）的常规参数。
- b 表示我们合并的步骤数。

每次光环通信包含 b 个点，共进行 $\sqrt{N/b}$ 次。执行的工作包括 MN/p 次本地更新，以及因光环而产生的冗余计算。后者包含 $b^2/2$次操作，在处理器域左右两侧均需执行。

将所有项相加后，我们得出总成本为

$$\frac{M}{b}\alpha + M\beta + \left(\frac{MN}{p} + Mb\right)\gamma.$$

我们观察到 $\alpha M/b + \gamma Mb$ 的开销与 p 无关 ，

练习 7.52. 计算 b 的最优值，并注意其仅取决于架构参数 α、 β、 γ，而与问题参数无关。

7.11.2 通信与工作量最小化策略

我们可以通过重叠计算与通信来提高该算法的效率。如图 7.29 所示，每个处理器首先通信其 halo 区域，并将此通信与可本地完成的计算部分重叠执行。最后再计算依赖于 halo 区域的数值。

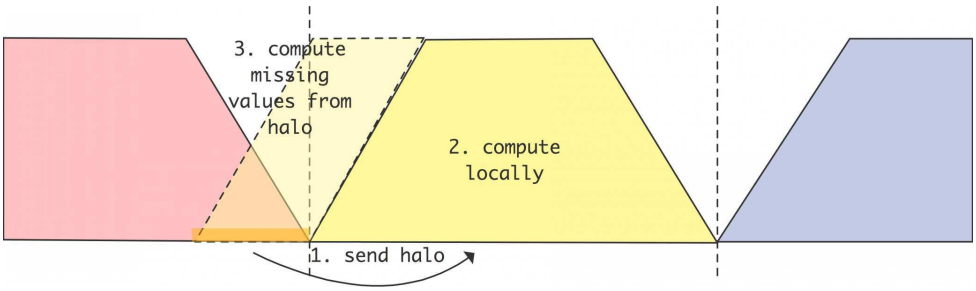


图 7.29：多次迭代中网格点块的计算过程。

习题 7.53. 以这种方式组织代码（重点在于 ‘代码’！）会带来什么实际难题?

若每个处理器分配的点数足够大，通信量相对于计算量较低，此时可选取 b 较大的值。但这类网格更新通常用于迭代方法，例如 *CG* 方法（第 5.5.11 节），而舍入误差的考量会限制 b 取值过大 [33]。

习题 7.54. 针对非重叠算法进行复杂度分析，假设点以二维网格形式组织。每个点更新需涉及四个相邻点（每个坐标方向两个相邻点）。

上述算法可进一步优化。如图 7.30 所示，可利用不同时间步的不同点构建光晕区域。该算法（参见 [43]）减少了冗余计算量，但需先计算待通信的光晕值，因此需将本地通信拆分为两个阶段。

7.12 多核架构上的分块算法

在章节 5.3.6 中你已了解到某些线性代数算法可通过子矩阵形式表述。这种视角对于在共享内存架构上高效执行线性代数运算具有显著优势

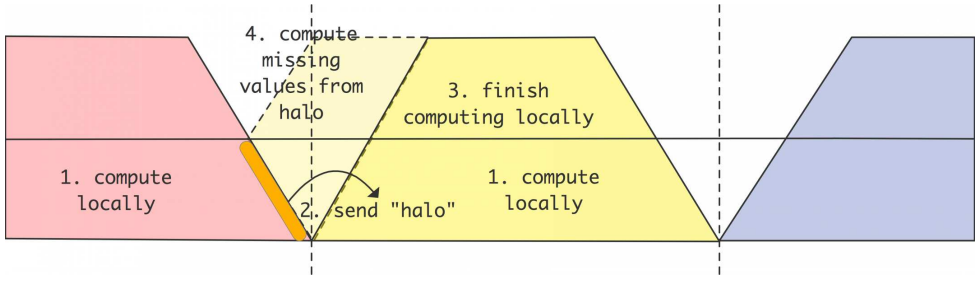


图 7.30：网格点块在多轮迭代中的计算过程。

内存架构，例如当前的多核处理器。

举例来说，让我们考虑 *Cholesky* 分解，它计算 $A = LL^t$ 对称正定矩阵 A；另见章节 5.3.1.2。递归地，我们可以如下描述该算法：

$$\text{Chol}\begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t \qquad \text{where} \quad L = \begin{pmatrix} L_{11} & 0 \\ \tilde{A}_{21} & \text{Chol}(A_{22} - \tilde{A}_{21}\tilde{A}_{21}^t) \end{pmatrix}$$

其中 $A_{21} = \tilde{A}_{21}L_{11}^{-t}$、 $A_{11} = L_{11}L_{11}^t$。

（该算法既可作为标量算法，也可视为分块矩阵算法。从性能角度出发，通过使用 *BLAS 3* 例程，我们将隐式仅考虑分块方法。）

实践中，分块实现被应用于分区处理

$$\left(\begin{array}{c|cc} & \text{finished} & \\ \hline A_{kk} & A_{k,>k} \\ A_{>k,k} & A_{>k,>k} \end{array}\right)$$

其中 k 表示当前块行的索引，且所有 $< k$ 索引的分解已完成。该分解过程使用 BLAS 操作命名表述如下：

for $k = 1$,nblocks: 分块数 : Chol $L_k L_k^t \leftarrow A_{kk}$: factor : 分解 : 求解 , ,

Trsm $A_{>k\,k} \leftarrow A_{>k}$

$\tilde{}$ $_kL_k^{-t}$: 形成乘积 $,_kA_{>k}^t$,

Gemm $A_{>k}$ $\tilde{}$ $\tilde{}$

k : 对称秩更新 $A_{>k,>k} \leftarrow A_{>k,>k} - \underline{A}_{>k,k}A_{>k,k}^t$

Syrk k

T 并行性能的关键在于对索引 $> k$ 进行分区，并以这些锁为基础编写算法：

b

$$\left(\begin{array}{c|cc} & \text{已完成} & \\ \hline A_{kk} & A_{k,k+1} & A_{k,k+2}\cdots \\ A_{k+1,k} & A_{k+1,k+1} & A_{k+1,k+2}\cdots \\ A_{k+2,k} & A_{k+2,k+2} & \\ \vdots & \vdots & \end{array}\right)$$

T该算法现在增加了一层内循环：

for $k = 1, \text{nblocks}$:
  `Chol`: factor $L_k L_k^t \leftarrow A_{kk}$
  for $\ell > k$:
    `Trsm`: solve $\tilde{A}_{\ell,k} \leftarrow A_{\ell,k} L_k^{-t}$
  for $\ell_1, \ell_2 > k$:
    `Gemm`: form the product $\tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$
  for $\ell_1, \ell_2 > k, \ell_1 \leq \ell_2$:
    `Syrk`: symmmetric rank-k update $A_{\ell_1,\ell_2} \leftarrow A_{\ell_1,\ell_2} - \tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$

现在，每个 ℓ 循环中的离子可以独立处理。然而，这些循环在外层 k 循环的每次迭代中会变短，因此我们无法立即确定可以容纳多少处理器。另一方面，无需严格保持上述算法的操作顺序。例如，在

$$L_1 L_1^t = A_{11}, \quad A_{21} \leftarrow A_{21} L_1^{-t}, \quad A_{22} \leftarrow A_{22} - A_{21} A_{21}^t$$

即使 $k = 1$ 迭代的其余部分尚未完成，分解 $L_2 L_2^t = A_{22}$ 也可以开始。因此，可能存在的并行性远超过仅通过并行化内层循环所能获得的。

在这种情况下，处理并行性的最佳方式是摆脱算法的控制流视角（即操作序列被严格规定），转向数据流视角。后者仅关注数据依赖关系，任何遵守这些依赖关系的操作顺序都是允许的。（从技术上讲，我们放弃了任务的程序顺序，代之以偏序关系 [4] 。）表示算法数据流的最佳方法是构建一个有向无环图（*DAG*）（关于图的简要教程见第 20 节），其中节点代表任务。如果任务 j 使用了任务 i 的输出，我们就在图中添加一条边 (i, j) 。

练习 7.55. 在章节 2.6.1.6 中，你学习了 顺序一致性的概念：一个线程并行代码程序在并行执行时应与顺序执行时产生相同的结果。我们刚刚指出，基于 DAG 的算法可以自由地以任何遵守图节点偏序关系的顺序执行任务。请讨论在此上下文中顺序一致性是否构成问题。

在我们的示例中，我们通过为每个内部迭代创建一个顶点任务来构建 DAG。图 7.31 展示了由 4×4 个块组成的矩阵所有任务的 DAG。该图是通过模拟上述 Cholesky 算法构建的，

练习 7.56. 该图的直径是多少？确定位于决定直径的路径上的任务。这些任务在算法上下文中意味着什么？

这条路径被称为 关键路径。其长度决定了并行计算时的执行时间，即使有无限数量的处理器可用。（这些主题已在章节 2.2.4 中讨论过。）

练习 7.57. 假设有 T 个任务，每个任务执行耗时均为单位时间，且有 p 个处理器。执行该算法的理论最短时间是多少？现根据关键路径修正此公式，设其长度为 C。

4. 假设 $a \leq b$ 若 a 在 b 之前执行，则当 $a \leq b \wedge b \leq a \Rightarrow a = b$ 和 $a \leq b \wedge b \leq c \Rightarrow a \leq c$ 时，关系 $\cdot \leq \cdot$ 构成偏序。与全序（如程序顺序）的区别在于 $a \leq b \vee b \leq a$ 不成立：可能存在未被排序的配对，意味着它们的时间顺序未被规定。

在执行任务的有向无环图（ DAG ）过程中，可以得出若干观察结果。

- 若对某一块进行多次更新，最好由同一进程完成这些更新计算。这有助于简化缓存一致性的维护。

- 若数据被使用后又被修改，必须在使用完成后才能开始修改。即使这两个操作位于不同处理器上，此规则依然适用，因为内存子系统通常维护缓存一致性，修改操作可能影响正在读取数据的进程。可通过在主内存中保留数据副本解决此问题，为读取进程提供预留数据（参见章节 1.4.1 ）。

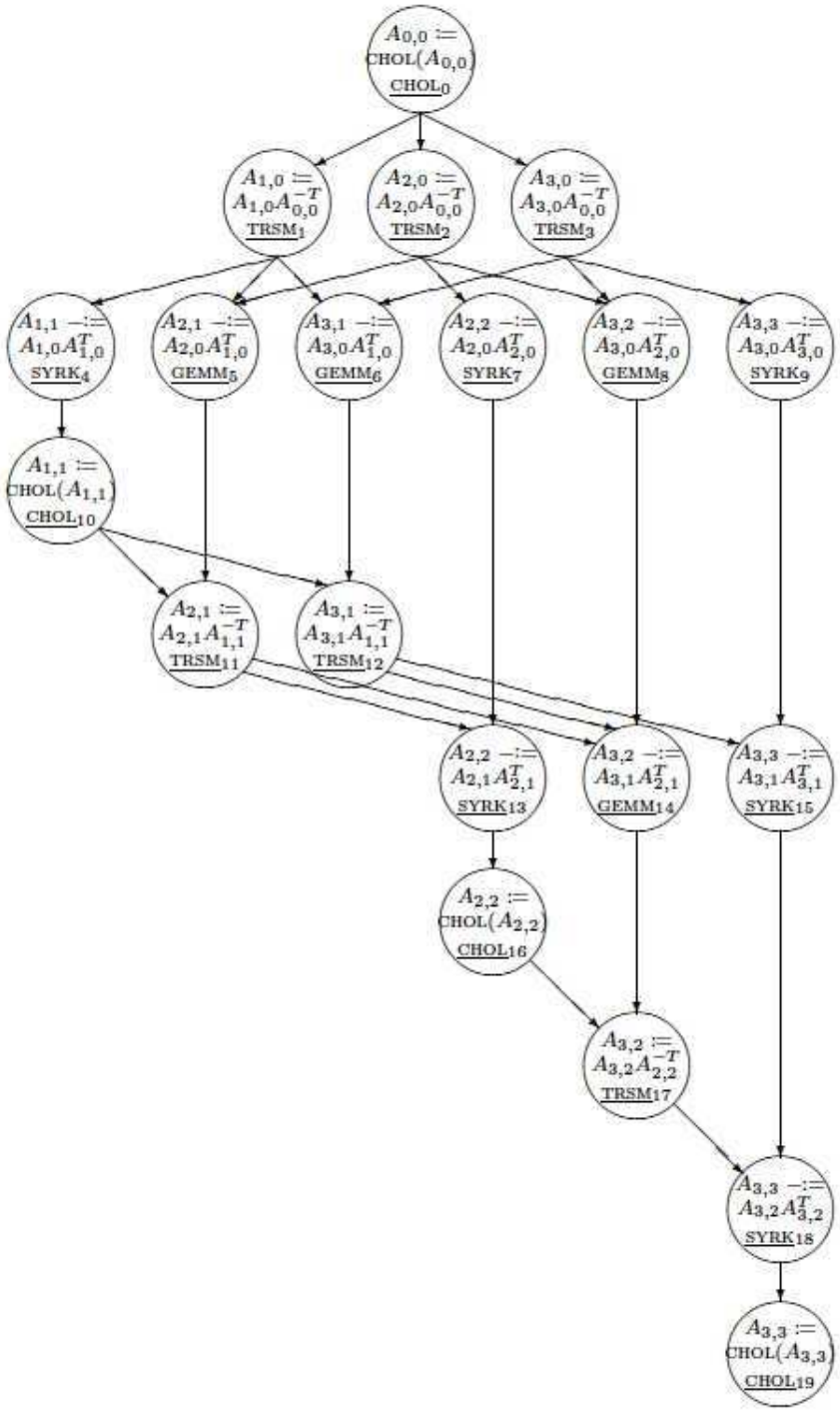


图 7.31: 4×4 Cholesky 分解中的任务依赖关系图。

第二部分

应用领域

第 8 章

分子动力学

分子动力学是一种通过模拟分子原子层面的行为并从中推导宏观性质的技术。它可应用于生物分子（如蛋白质和核酸）以及材料科学与纳米技术中的天然与合成分子。分子动力学属于粒子方法范畴，该范畴还包括天体力学和天体物理学中的 N 体问题，此处介绍的许多概念也适用于这些其他领域。此外，分子动力学还存在一些特殊类型，比如从头算分子动力学 —— 其中电子被量子力学处理，从而可以模拟化学反应。我们不会讨论这些特殊类型，而是专注于经典分子动力学。

分子动力学背后的原理非常简单：一组粒子根据牛顿运动定律 $F = ma$ 相互作用。给定粒子的初始位置与速度、粒子质量及其他参数，以及粒子间作用力的模型，便可通过数值积分牛顿运动定律，得到每个粒子在未来（及过去）所有时刻的运动轨迹。通常，这些粒子位于具有周期性边界条件的计算箱体中。

因此，分子动力学的时间步长由两部分组成：

1: 计算所有粒子上的作用力
2: 更新位置（积分）。

力的计算是耗时部分。最先进的分子动力学模拟在并行计算机上执行，因为力计算成本高昂且需要大量时间步长才能达到合理的模拟时长。在许多情况下，分子动力学也应用于具有极多原子的分子模拟，例如生物分子可达百万原子级并覆盖长时间尺度，而其他分子可达十亿级但时间尺度较短。

数值积分技术在分子动力学中同样重要。对于需要大量时间步长且保持能量等量守恒比精度阶数更关键的模拟，必须使用的求解器与第 4 章介绍的传统 ODE 求解器不同。

下文将介绍用于生物分子模拟的力场，并讨论快速计算这些力的方法。随后我们将分节阐述短程力分子动力学的并行化，以及用于长程力快速计算的 3-D FFT 并行化。

最后我们将介绍适用于分子动力学模拟的一类积分技术。本章对分子动力学主题的处理旨在入门和实践；如需更多信息，推荐参考 [66] 。

8.1 力计算

8.1.1 力场

在经典分子动力学中，描述原子间势能和作用力的模型称为力场。力场是对量子力学效应的可处理但近似的模型，因为对于大分子而言，精确计算量子力学效应在计算上过于昂贵。不同类型的分子以及不同研究者对同一分子会使用不同的力场，且没有一种是理想的。

在生化系统中，常用的力场将势能函数建模为键合能、范德华力和静电（库仑）能的总和：

$$E = E_{\text{bonded}} + E_{\text{Coul}} + E_{\text{vdW}}.$$

势能是模拟中所有原子位置的函数。原子所受的力是该原子位置处势能的负梯度。

键能源于分子内的共价键，

$$E_{\text{bonded}} = \sum_{\text{bonds}} k_i(r_i - r_{i,0})^2 + \sum_{\text{angles}} k_i(\theta_i - \theta_{i,0})^2 + \sum_{\text{torsions}} V_n(1 + \cos(n\omega - \gamma))$$

其中这三项分别对应：所有共价键的总和、由两个键形成的所有角度的总和，以及由三个键形成的所有二面角的总和。固定参数 k_i 、 $r_{i,0}$ 等取决于所涉及原子的类型，且可能因力场不同而异。额外项或具有不同函数形式的项也常被使用。

剩余两项势能 E 统称为非键合项。在计算上，它们构成了力计算的主体。静电能量源于原子电荷，并通过常见的

$$E_{\text{Coul}} = \sum_i \sum_{j>i} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

其中求和覆盖所有原子对， q_i 和 q_j 分别是原子 i 和 j 上的电荷， r_{ij} 是原子 i 和 j 之间的距离。最后，范德华力近似描述了剩余的吸引和排斥效应，通常用 Lennard-Jones 函数建模

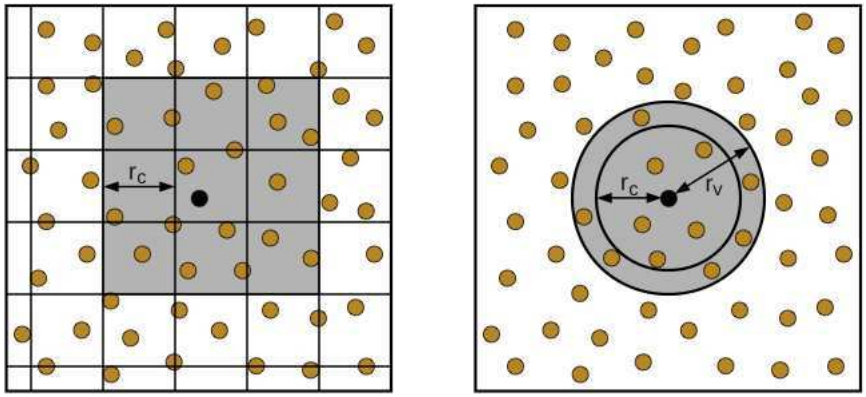
$$E_{\text{vdW}} = \sum_i \sum_{j>i} 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right]$$

其中 ϵ_{ij} 和 σ_{ij} 是取决于原子类型的力场参数。在短距离时，排斥项 (r^{12}) 起主导作用；而在长距离时，色散（吸引，$-r^6$）项起主导作用。

分子动力学力计算的并行化取决于各类力计算的并行化。键合力属于局部计算，因为对于给定原子只需附近原子的位置和数据。范德华力也是局部的，并因其在原子间距较大时可忽略而被称为短程力。静电力属于长程力，目前已开发多种技术来加速这类计算。接下来两小节将分别讨论短程和长程非键合力的计算。

8.1.2 计算短程非键合力

粒子短程非键合力的计算可在超出该粒子截断半径 r_c 时截断。对粒子 i 执行此计算的原始方法是检查所有其他粒子并计算它们与粒子 i 的距离。对于 n 个粒子，此方法的复杂度为 $O(n^2)$，相当于计算所有粒子对之间的作用力。有两种数据结构 —— 单元列表和 *Verlet* 邻居列表 —— 可单独用于加速此计算，亦可将二者结合使用。



(a) Cell list method. (b) Verlet neighbor list method.

图 8.1：计算截止半径 r_c 内的非键合力。要计算高亮粒子所涉及的力 d
时，仅需考虑阴影区域内的粒子。

8.1.2.1 单元格列表

单元格列表的思想常见于需要寻找给定点附近点集的问题中。参考图 8.1(a) 的二维示例：在粒子集合上覆盖一个网格。若网格间距不小于 r_c，则计算粒子 i 受力时，只需考虑该粒子所在单元格及相邻 8 个单元格内的粒子。通过一次全局粒子扫描即可构建每个单元格的粒子列表，这些列表用于计算所有粒子的受力。由于粒子位置随时间步长移动，单元格列表需在下一步重新生成或更新。该方法构建数据结构的复杂度为 $O(n)$，计算受力的复杂度为 $O(n \times n_c)$。

力计算过程中，n_c 代表 9 个单元格（三维情况下为 27 个单元格）内粒子的平均数量。单元格列表数据结构所需的存储空间为 $O(n)$。

8.1.2.2 Verlet 邻居列表

单元格列表结构存在一定低效性，因为对于每个粒子 i，n_c 个粒子会被纳入考量，但这远超过截断半径内的实际粒子数量 r_c。Verlet 邻居列表是记录粒子截断半径内其他粒子的列表 i。每个粒子拥有独立列表，因此所需存储空间为 $O(n \times n_v)$ 其中 n_v 表示截断半径内粒子的平均数量。这些列表构建完成后，力计算将极为高效，仅需最低复杂度 $O(n \times n_v)$。但列表构建过程代价较高，需要对每个粒子检查所有其他粒子，其复杂度不低于原始的 $O(n^2)$。然而优势在于，若采用扩展截断半径，邻居列表可跨多个时间步重复使用 r_v。参考图 8.1(b) 的二维示例，只要没有粒子从两个圆圈外部移动到内圈，邻居列表就可重复使用。若能估算或限定粒子最大速度，则可确定安全复用邻居列表的时间步数。（或者，可在粒子进入截断半径时触发信号。）严格来说，Verlet 邻居列表记录的是扩展截断半径内的粒子集合 r_v。

8.1.2.3 联合使用单元列表与邻居列表

混合方法的核心在于利用 Verlet 邻居列表，同时通过单元列表来构建邻居列表。这显著降低了需要重新生成邻居列表时的高昂计算成本。该混合策略极为高效，常被用于尖端分子动力学软件中。

单元列表和 Verlet 邻居列表均可进行优化，以利用粒子 i 受到粒子 j 的作用力 f_{ij} 等于 $-f_{ji}$ 这一特性（牛顿第三定律），该力只需计算一次。例如对于单元列表，在二维情况下只需考虑 8 个单元中的 4 个。

8.1.3 长程力计算

静电力计算具有挑战性，因其属于长程力：模拟中每个粒子都会受到其他所有粒子不可忽略的静电作用。有时会采用截断半径近似法（类似短程范德华力的处理方式），即在特定截断半径后停止粒子间作用力计算。但这种方法通常会导致模拟结果出现不可接受的伪影。

有几种更精确的方法可以加速静电力的计算，避免对所有 n 粒子对进行 $O(n^2)$ 求和。我们在此简要概述其中一些方法。

8.1.3.1 分层 *N* 体方法

分层 N 体方法（包括 Barnes-Hut 方法和快速多极方法）在天体物理粒子模拟中很流行，但对于生物分子模拟所需的精度通常成本过高。在 Barnes-Hut 方法中，空间被递归地划分为 8 个相等的单元（在三维情况下），直到每个单元包含零个或一个粒子。邻近粒子之间的力按常规单独计算，但对于远处粒子，力是计算一个粒子与单元内一组远处粒子之间的力。通过精度测量来确定是否可以使用远处单元来计算力。

必须通过单独考虑其子单元来计算。Barnes-Hut 方法的复杂度为 $O(n \log n)$。快速多极方法的复杂度为 $O(n)$；该方法计算电势而非直接计算力。

8.1.3.2 粒子 - 网格方法

在粒子 - 网格方法中，我们利用泊松方程

$$\nabla^2 \phi = -\frac{1}{\epsilon} \rho$$

该方程将电势 ϕ 与电荷密度 ρ 联系起来，其中 $1/\epsilon$ 为比例常数。为应用此方程，我们使用网格对空间进行离散化，将电荷分配到网格点，在网格上求解泊松方程以获得网格上的电势。力是电势的负梯度（对于静电力等保守力而言）。目前已开发出多种技术用于将空间中的点电荷分布到一组网格点，并通过数值插值计算网格点电势对点电荷产生的力。求解泊松方程的快速方法包括多重网格法和快速傅里叶变换等。术语上，粒子 - 网格方法与朴素的粒子 - 粒子方法形成对比，后者需计算所有粒子对之间的作用力。

事实证明，粒子 - 网格方法精度不高，更精确的替代方案是将每种力分解为短程快速变化部分和长程缓慢变化部分：

$$f_{ij} = f_{ij}^{sr} + f_{ij}^{lr}.$$

实现方式之一是通过函数 $h(r)$ 对 f 进行加权 —— 该函数突出短程部分（小 r），而 $1 - h(r)$ 则强调长程部分（大 r）。短程部分通过计算截断半径内所有粒子对的相互作用（粒子 - 粒子方法）获得，长程部分则采用粒子 - 网格方法计算。这种被称为粒子 - 粒子 - 粒子 - 网格（ PPPM 或 P3M ）的方法是 Hockney 和 Eastwood 于 1973 年在一系列论文中提出的。

8.1.3.3 埃瓦尔德方法

埃瓦尔德方法是目前生物分子模拟中静电力计算最流行的方法，专为周期性边界条件开发。其结构与 PPPM 类似，将力分解为短程和长程两部分：短程部分采用粒子 - 粒子方法计算，长程部分则使用傅里叶变换。埃瓦尔德方法的变体与 PPPM 高度相似 —— 长程部分采用网格计算，并利用快速傅里叶变换求解网格上的泊松方程。更多细节可参阅 [66]。在 8.3 节中，我们将描述用于求解三维泊松方程的三维快速傅里叶变换并行化实现。

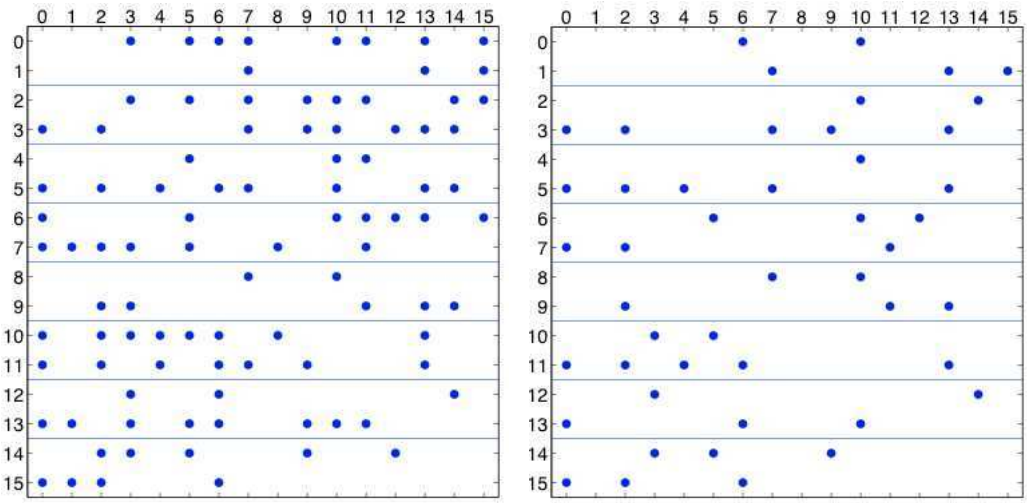
8.2 并行分解

我们现在讨论力的并行计算。Plimpton [160] 创建了一个非常有用的分子动力学并行化方法分类，识别了原子、力和空间分解方法。在此，我们严格遵循他对这些方法的描述。我们还增加了第四种类别，该类别具有

逐渐被认识到与早期类别不同，称为中立领域方法，这一名称由 Shaw 创造 [172]。中立领域方法目前被许多先进的分子动力学代码所采用。空间分解和中立领域方法在并行化基于截断的计算时尤其具有优势。

8.2.1 原子分解

在原子分解中，每个粒子被分配给一个处理器，该处理器负责计算粒子的力并更新其在模拟过程中的位置。为了使计算大致平衡，每个处理器被分配大致相同数量的粒子（随机分布效果良好）。原子分解的一个重要特点是，每个处理器通常需要与所有其他处理器通信以共享更新后的粒子位置。



(a) Force matrix.　　(b) Force matrix, redundancies removed.

图 8.2：原子分解，展示了 16 个粒子分布在 8 个处理器上的力矩阵。点代表力矩阵中的非零元素。左侧的矩阵是对称的；右侧，仅计算一对反对称元素中的一个，以利用牛顿第三定律。

原子分解通过力矩阵在图 8.2(a) 中展示。对于 n 个粒子，力矩阵是一个 $n \times n$ 的矩阵；行和列由粒子索引编号。矩阵中的非零项 f_{ij} 表示粒子 i 由于粒子 j 所受到的非零力，必须被计算。此力可能是非键合力和 / 或键合力。当使用截断时，矩阵是稀疏的，如本例所示。如果计算所有粒子对之间的力，则矩阵是稠密的。由于牛顿第三定律，矩阵是斜对称的，$f_{ij} = -f_{ji}$。图 8.2(a) 中的线条展示了粒子如何被划分。图中，16 个粒子被划分到 8 个处理器中。

算法 1 展示了一个处理器视角下的一个时间步。在时间步开始时，每个处理器持有分配给它的粒子的位置。

一个优化方案是将计算量减半，这是可行的，因为力矩阵是

Algorithm 1 原子分解时间步长。
1: 发送 / 接收粒子位置至 / 从所有其他处理器
2: (若使用非键截断) 确定需要计算哪些非键作用力
3: 计算分配给该处理器的粒子所受作用力
4: 更新分配给该处理器的粒子位置 (积分)

为实现斜对称性，我们精确选择 f_{ij} 或 f_{ji} 来处理所有斜对称对，确保每个处理器负责计算大致相同数量的力。选择力矩阵的上三角或下三角部分会导致计算负载不均衡，这是不佳的选择。更优方案是：当 $i+j$ 在矩阵上三角中为偶数时计算 f_{ij}，或当 $i+j$ 在矩阵下三角中为奇数时计算，如图 8.2(b) 所示。此外还存在多种其他选择方案。

当利用力矩阵的斜对称性时，处理器不再独立计算其所属粒子的全部作用力。例如图 8.2(b) 中，粒子 1 的作用力不再仅由第一个处理器计算。为完成力计算，处理器需通过通信交换数据：发送其他处理器所需的作用力，并接收其他处理器计算的结果。此时需在上述算法中增加通信步骤（步骤 4 ），如算法 2 所示。

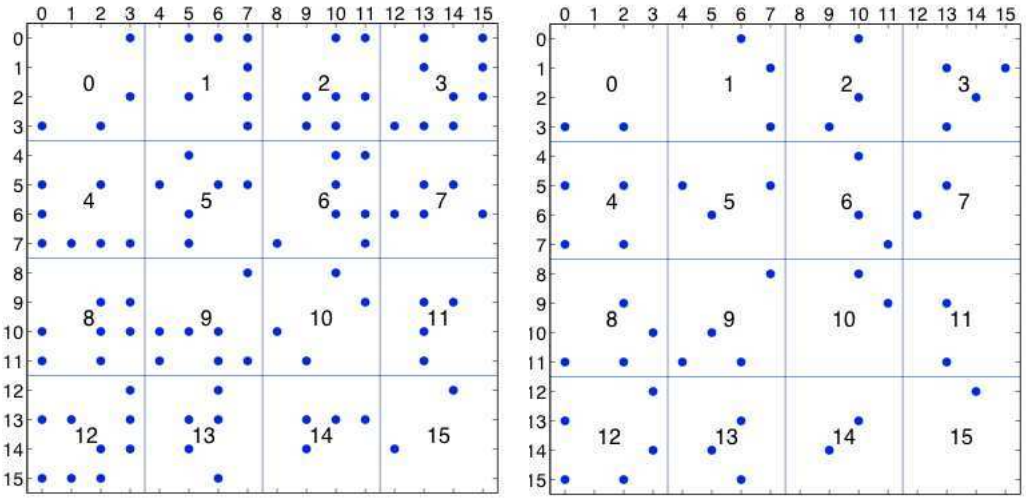
算法 2 原子分解时间步，无冗余计算。
1: 向 / 从所有其他处理器发送 / 接收粒子位置
2: (若使用非键截断) 确定需要计算哪些非键力
3: 计算部分 分配给该处理器的粒子受力
4: 发送其他处理器所需的粒子力，并接收本处理器所需的粒子力 r
5: 更新分配给该处理器的粒子位置 (积分)

若额外通信的开销被计算量的节省所抵消，则该算法具有优势。需注意的是，通信量通常会翻倍。

8.2.2 力分解

在力分解中，力被分配给各处理器进行计算。最直接的方法是将力矩阵划分为块，并将每个块分配给一个处理器。图 8.3(a) 展示了 16 个粒子和 16 个处理器的情况。为了更新粒子位置，粒子也需要分配给处理器（如同原子分解）。在该图示例中，处理器 i 负责更新粒子 i 的位置；在实际问题中，一个处理器通常会负责更新多个粒子的位置。再次注意，我们首先考虑的是斜对称力矩阵的情况。

我们现在分析力分解在单个时间步内所需的通信。以处理器 3 为例，它需要计算粒子 0 、 1 、 2 、 3 的部分力，并获取粒子 0 、 1 、 2 、 3 及 12 、 13 、 14 、 15 的位置信息。因此处理器 3 需与处理器 0 、 1 、 2 、 3 及 12 、 13 、 14 、 15 进行通信。当所有处理器完成力计算后，处理器 3 还需收集作用于



(a) Force matrix.　　(b) Force matrix, redundancies removed.

图 8.3：力分解示意图，展示了一个包含 16 个粒子的力矩阵以及这些力在 16 个处理器间的分配情况。

粒子 3 由其他处理器计算。因此处理器 2 需要再次与处理器 0 、 1 、 2 、 3 进行通信。

算法 3 展示了从一个处理器的视角出发，在一个时间步内执行的操作。时间步开始时，每个处理器持有分配给它的所有粒子的位置信息。

算法 3 力分解时间步。

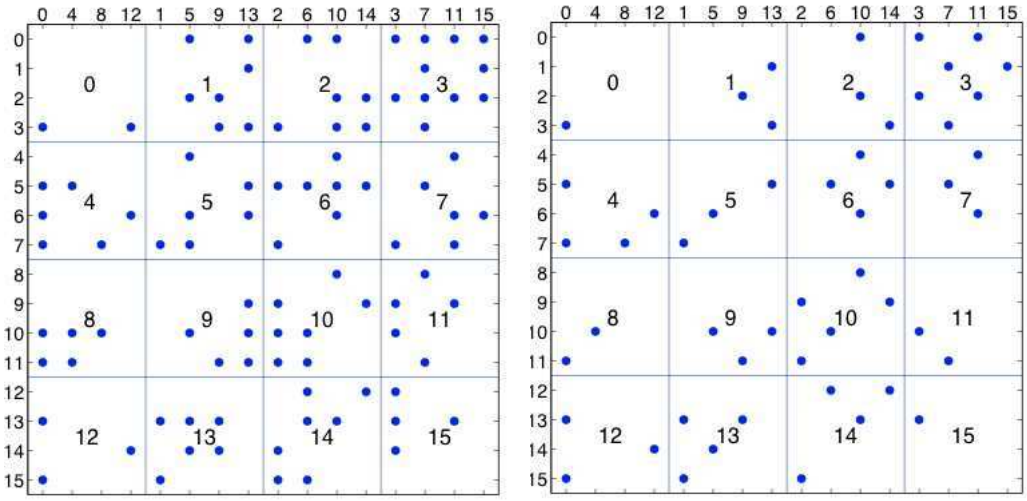
1: 发送其他处理器所需的、分配给本处理器的粒子位置；接收行 处理器所需的粒子位置（此通信发生在同一处理器行的处理器之间，例如处理器 3 与处理器 0 、 1 、 2 、 3 通信） 2: 接收列处理器所需的粒子位置（此通信通常发生在不同处理器行的处理器之间，例如处理器 3 与处理器 12 、 13 、 14 、 15 通信） 3: （若使用非键合截断）确定需要计算哪些非键合力 4: 计算分配给本处理器的粒子受力 5: 发送其他处理器所需的力；接收本处理器分配粒子所需的力（此通信发生在同一处理器行的处理器之间，例如处理器 3 与处理器 0 、 1 、 2 、 3 通信） 6: 更新分配给本处理器的粒子位置（积分）

一般情况下，若有 p 个处理器（且为简化起见假设 p 为方阵），则力矩阵会被划分为 $\sqrt{p} \times \sqrt{p}$ 个区块。前述力分解方法要求处理器分三步进行通信，每步涉及 $\sqrt{p}$ 个处理器。这比原子分解法高效得多，后者需要所有 p 个处理器之间进行通信。

我们同样可以在力分解中利用牛顿第三定律。与原子分解类似，我们首先

选择一个修正后的力矩阵，其中仅计算 f_{ij} 和 f_{ji} 中的一个。粒子 i 上的力由一行处理器计算，现在也由一列处理器计算。因此，每个处理器需要额外的通信步骤，从一列处理器中收集分配给它的粒子的力。原本有三个通信步骤，现在利用牛顿第三定律时有四个通信步骤（这种情况下通信量并未像原子分解那样翻倍）。

对力分解的修改节省了部分通信。如图 8.4 所示，列通过块循环排序重新排列。再次考虑处理器 3，它计算粒子 0 、 1 、 2 、 3 的部分力。与之前一样，它需要粒子 0 、 1 、 2 、 3 的位置，但现在还需要处理器 3 、 7 、 11 、 15 的数据。后者是与处理器 3 位于同一处理器列中的处理器。因此，所有通信均在同一处理器行或列内进行，这在基于网格的网络架构中可能具有优势。修改后的方法如算法 4 所示。



(a) Force matrix.　　(b) Force matrix, redundancies removed.

图 8.4：力分解示意图，其中力矩阵的列经过置换。注意第 3 、 7 、 11 、 15 列现在位于对应处理器 3 、 7 、 11 、 15 的块列中（索引相同），以此类推。

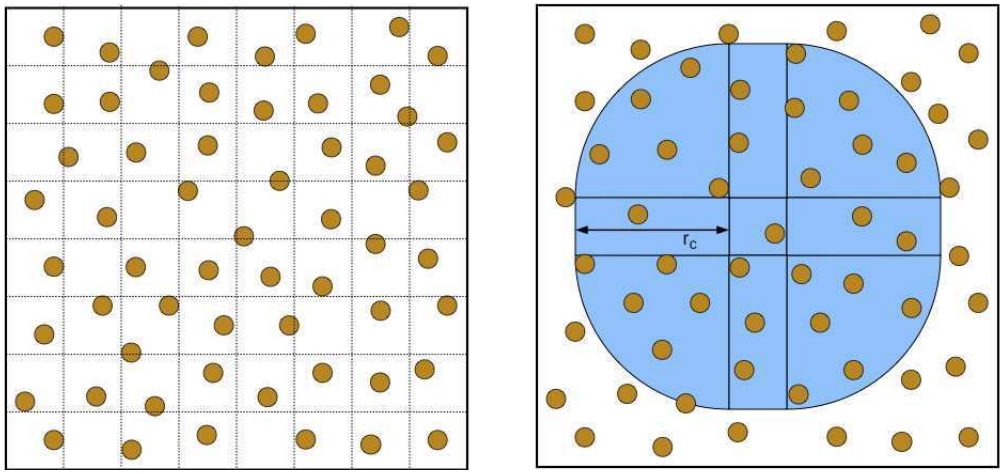
8.2.3 空间分解

在空间分解中，空间被划分为若干单元。每个单元分配给一个处理器，负责计算位于该单元内粒子所受的力。图 8.5(a) 展示了一个二维模拟案例中将空间分解为 64 个单元的情况。（这是对空间的分解，切勿与力矩阵混淆。）通常，单元数量选择与处理器数量相等。由于粒子在模拟过程中会移动，粒子与单元的对应关系也会随之变化。这与原子分解和力分解形成鲜明对比。

图 8.5(b) 展示了一个单元（中央方块）及可能位于截止半径内的粒子所在空间区域（阴影部分），该区域包含与给定单元中粒子相互作用的粒子。阴影区域常被称为导入区域，因为给定单元必须导入位于该区域内粒子的位置信息才能完成其

Algorithm4 强制分解时间步长，对力矩阵的列进行置换。发送其他处理器所需的已分配粒子位置；重新 ceive *row* particle positions needed by my processor (this communication is between processors in the same processor row, e.g., processor 3 communicates with processors 0, 1, 2, 3)
2: receive *column* particle positions needed by my processor (this communication is generally with processors the same processor column, e.g., processor 3 communicates with processors 3, 7, 11, 15)
3: (if nonbonded cutoffs are used) determine which nonbonded forces need to be computed
4: compute forces for my assigned particles
5: send forces needed by other processors; receive forces needed for my assigned particles (this communication is between processors in the same processor row, e.g., processor 3 communicates with processors 0, 1, 2, 3)
6: update positions (integration) for my assigned particles

力计算。需注意，给定单元中的粒子并非必须与导入区域内的所有粒子发生相互作用，特别是当导入区域远大于截断半径时。



(a) Decomposition into 64 cells. (b) Import region for one cell.

图 8.5：空间分解示意图，展示二维计算框中的粒子，(a) 被划分为 6 4 个单元，(b) 单个单元的导入区域。

Algorithm 5 展示了每个处理器在单个时间步长内的操作流程。我们假设在时间步长开始时，每个处理器已持有其单元内粒子的位置信息。

为利用牛顿第三定律，导入区域的形状可被减半。此时每个处理器仅计算其单元内粒子上的部分力，并需接收来自其他处理器的力以计算这些粒子上的总力。因此需额外增加一个通信步骤。我们将修改后的导入区域细节及此情况下的伪代码实现留给读者作为练习。

在空间分解方法的实现中，每个单元关联着一个包含粒子的列表

算法 5 空间分解时间步长。

1: 发送其他处理器所需的位置数据 for particles in their import regions; receive positions for 我的导入区域内的粒子
2: 计算我分配到的粒子的作用力 3: 更新我分配到的粒子的位置（积分） ned particles

其导入区域，类似于 Verlet 邻居列表。与 Verlet 邻居列表一样，如果导入区域略微扩大，则无需在每个时间步都更新此列表。这使得导入区域列表可重复使用多个时间步，相当于粒子穿过扩展区域宽度所需的时间。这与 Verlet 邻居列表完全类似。

总之，空间分解方法的主要优势在于仅需在对应邻近粒子的处理器之间进行通信。其劣势在于，当处理器数量极大时，每个单元内所含粒子数量相比导入区域显得过小。

8.2.4 中立区域方法

我们对中立区域方法的描述基本遵循 Shaw [172] 的框架。中立区域方法可视为融合了空间分解与力分解的特性。在并行化积分步骤时，粒子根据空间划分方案分配给处理器；而在并行化力计算时，每个处理器计算两组粒子间的相互作用力，这些粒子可能与积分步骤分配给该处理器的粒子无关。这种额外灵活性使得中立区域方法所需的通信量可能远少于空间分解方法。

图 8.6 展示了一个二维模拟场景下的中立区域方法示例。图中所示方法将水平条带粒子与垂直条带粒子间的力计算分配给指定处理器，这两个区域由此构成该方法的导入区域。对比图 8.6(b) 可见，此中立区域方法的导入区域远小于对应空间分解方法的区域。当每个处理器对应的单元尺寸远小于截断半径时，该优势更为显著。

在计算完作用力后，给定处理器将已计算的作用力发送给需要这些力进行积分的处理器。因此我们得到算法 6。

算法 6 中立领域方法时间步。

1: 发送与接收粒子位置分配给该处理器的作用力 3：发送和接收积分所需的作用力 4：更新分配给该处理器的粒子位置（积分）

与其他方法类似，中立领域方法的导入区域可进行调整以利用牛顿第三定律。更多细节及中立领域图示请参阅 Shaw [172]。

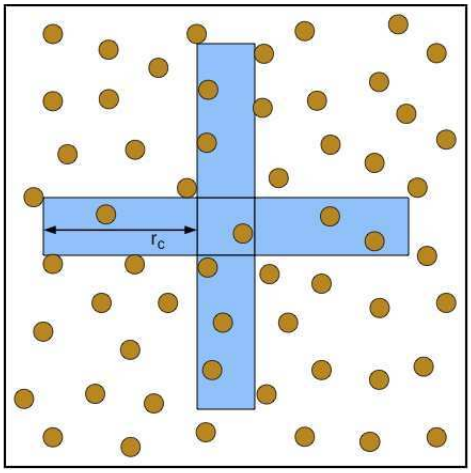


图 8.6：中性区域方法，展示二维计算盒中的粒子及一个单元（中心方块）的导入区域（阴影部分）。该图可直接与图 8.5(b) 的空间分解情况进行对比。详见 Shaw [172] 的更多细节。

在三维模拟中的方法。

8.3 并行快速傅里叶变换

许多计算长程力方法的共同组件是用于在三维网格上求解泊松方程的 3-D FFT。傅里叶变换将对泊松算子（称为拉普拉斯算子）进行对角化，求解过程中需要一个正向和一个逆向 FFT 变换。考虑离散拉普拉斯算子 L （具有周期性边界条件）及 $-L\phi = \rho$ 中 ϕ 的求解。设 F 表示傅里叶变换，原问题等价于

$$\begin{aligned} -(FLF^{-1})F\phi &= F\rho \\ \phi &= -F^{-1}(FLF^{-1})^{-1}F\rho. \end{aligned}$$

首先对 ρ 应用傅里叶变换，随后傅里叶空间组件通过对角矩阵的逆进行缩放，最后应用逆傅里叶变换 F^{-1} 以获得解 ϕ。

对于实际蛋白质尺寸，通常使用约 1 埃的网格间距，由此产生的三维网格按许多标准衡量都相当小： $64 \times 64 \times 64$ 或 $128 \times 128 \times 128$。并行计算通常不会应用于这种规模的问题，但必须采用并行计算，因为数据 ρ 已经分布在并行处理器之间（假设使用了空间分解）。

三维快速傅里叶变换（ FFT ）通过依次沿三个维度计算一维 FFT 来实现。对于 $64 \times 64 \times 64$ 的网格尺寸，这相当于 4096 个维度为 64 的一维 FFT。并行 FFT 计算通常受限于通信。最佳并行化方案取决于变换规模及计算机网络架构。下文将先阐述并行一维 FFT 的相关概念，再讨论并行三维 FFT 的部分原理。当前专注于该领域的软件与研究

关于大规模一维变换的并行化与高效计算（使用 SIMD 操作），我们推荐 SPIRAL 和 FFTW 包。这些包采用自动调优技术生成针对用户计算机架构优化的 FFT 代码。

8.3.1 并行一维 FFT

8.3.1.1 无转置一维 *FFT*

图 8.7 展示了 16 点基 2 频率抽取 FFT 算法的输入（左侧）与输出（右侧）之间的数据依赖关系（数据流图）。（图中未显示计算过程中可能需要的位反转置换操作。）该图还展示了四个处理器之间的计算任务划分。在此并行化方案中，初始数据不在处理器间移动，但计算过程中会发生通信。图中示例显示，通信发生在 FFT 的前两个阶段；最后两个阶段不涉及通信。当通信发生时，每个处理器仅与另一个特定处理器进行数据交换。

8.3.1.2 使用转置的 *1-DFFT*

利用转置并行化 FFT 计算是常见做法。图 8.8(a) 展示了与图 8.7 相同的数据流图，但移除了水平线并添加了额外索引标签以提高清晰度。如前所述，前两个 FFT 阶段无需通信即可完成。随后数据在处理器间进行转置。通过这种转置后的数据布局，最后两个 FFT 阶段也可无需通信即可执行。最终数据并非原始顺序；可能需要额外转置，或可直接使用转置后的顺序。图 8.8(b) 展示了转置前后四个处理器间的索引分区情况。从这两幅图可见，前两个阶段的数据依赖仅涉及同一分区内的索引。转置后的分区中，后两个阶段同样如此。还需注意转置前后的计算结构完全一致。

8.3.2 并行三维快速傅里叶变换

8.3.2.1 基于块分解的三维 *FFT*

图 8.9(a) 展示了当空间分解应用于大小为 $8 \times 8 \times 8$ 的网格时，FFT 输入数据的块分解情况，该网格分布在 64 个处理器上，排列成 $4 \times 4 \times 4$ 拓扑结构。并行一维 FFT 算法可应用于每个维度。图中示例中，每个一维 FFT 计算涉及 4 个处理器。每个处理器同时执行多个一维 FFT （本例中为四个）。在处理器内部，若沿某一维度遍历，数据是连续排列的，因此在对另外两个维度进行计算时，数据访问是跨步的。跨步数据访问可能较慢，因此在计算每个维度的 FFT 时，对处理器内部数据进行重排序可能是有价值的。

8.3.2.2 基于平板分解的三维 *FFT*

板分解如图 8.9(b) 所示，展示了 4 个处理器的情况。每个处理器持有一个或多个输入数据平面。当输入数据已以板形式分布时，采用此种分解方式。

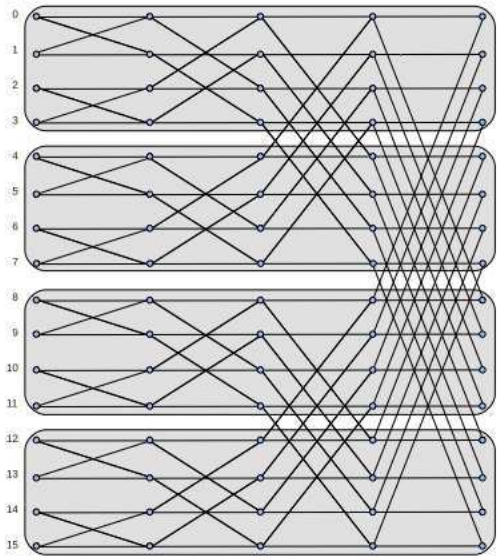


图 8.7：16 点一维 FFT 的数据流图。阴影区域展示了 4 个处理器（每个处理器负责一个区域）的分解方式。在此并行化方案中，前两个 FFT 阶段无需通信；后两个 FFT 阶段则需通信。

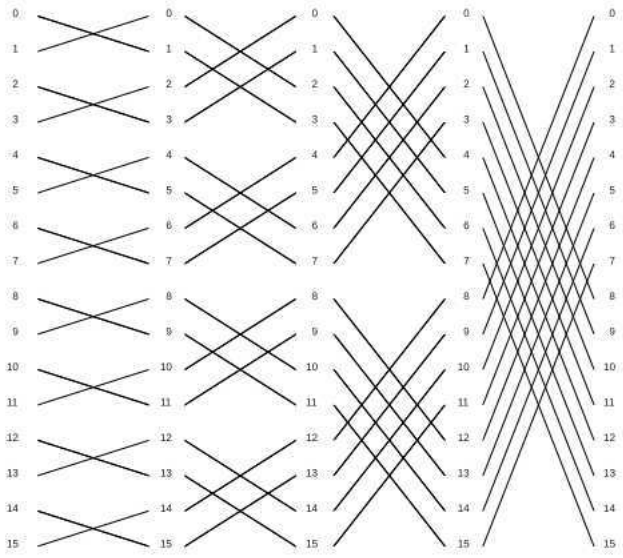
或者是否可以通过此方式重新分配数据。板片平面内的两次一维 FFT 无需通信。其余的一维 FFT 需要通信，可采用上述两种并行一维 FFT 方法之一。板片分解的缺点是当处理器数量较大时，处理器数量可能超过三维 FFT 任一维度上的点数。另一种选择是下文所述的铅笔分解法。

8.3.2.3 基于铅笔分解的三维 *FFT*

铅笔分解如图 8.9(c) 所示（以 16 个处理器为例）。每个处理器持有输入数据的一根或多根铅笔。若原始输入数据如图 8.9(a) 所示以块状分布，则通过三维处理器网格中一行处理器间的通信可将数据分配为铅笔分解。随后无需通信即可执行一维 FFT。若需在另一维度执行一维 FFT，则需将数据重新分配为该维度的铅笔状。整个三维 FFT 计算共需四个通信阶段。

8.4 分子动力学积分

为了对分子动力学中的常微分方程组进行数值积分，需要采用特殊的方法，这与第 4 章研究的传统 ODE 求解器不同。这些特殊方法被称为辛方法，在生成具有恒定能量的解方面优于其他方法，例如对于所谓的哈密顿系统（包括来自



(a) Data flow diagram (shown without horizontal lines for clarity) for 1-D FFT for 16 points.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

(b) 转置前（左）与转置后（右）的索引分区。

图 8.8: 带转置的一维快速傅里叶变换。前两个阶段不涉及通信。数据随后在处理器间转置，因此后两个阶段同样无需通信。

分子动力学）。当哈密顿系统在长时间区间内通过多时间步长进行积分时，保持结构（如总能量）通常比方法的精度阶数更为重要。本节将阐述斯托默 - 韦莱方法的部分原理与细节，该方法足以应对简单的分子动力学模拟。

哈密顿系统是一类能保持能量守恒且可表示为哈密顿方程形式的动力系统。为简化起见，考虑简谐振荡器

$$u'' = -u$$

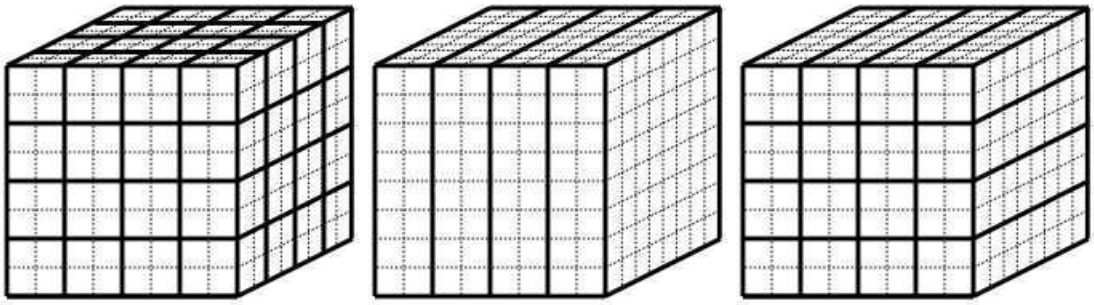
w其中 u 表示单个粒子偏离平衡点的位移。该方程可模拟
p质量为单位的物体挂在弹性系数为单位的弹簧上。位于位置 u 的粒子所受的力为 $-u$。
T该系统虽看似与分子动力学系统无关，但对阐释若干概念颇具助益。

上述二阶方程可改写为一阶方程组

$$\begin{aligned} q' &= p \\ p' &= -q \end{aligned}$$

其中 $q = u$ 和 $p = u'$ 是经典力学中常用的符号表示。其通解为

$$\begin{pmatrix} q \\ p \end{pmatrix} = \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix}.$$



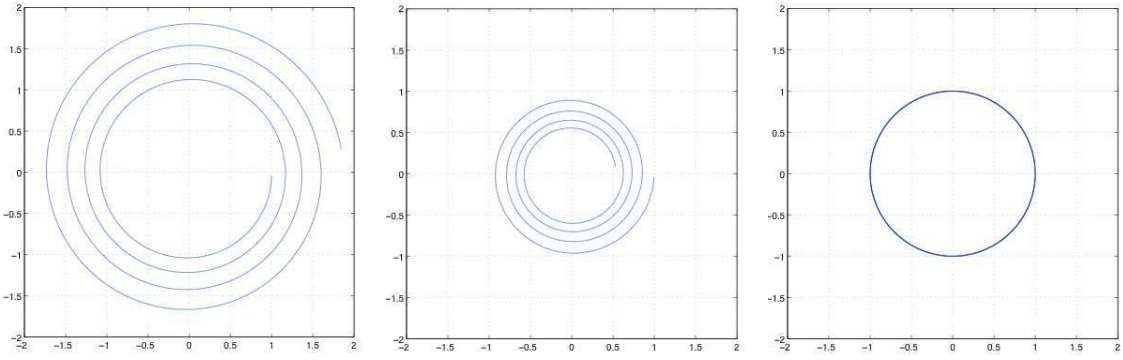
(a) Block Decomposition (b) Slab Decomposition (c) Pencil Decomposition

图 8.9：三维快速傅里叶变换的三种数据分解方式。

简谐振子的动能为 $p^2/2$，势能为 $q^2/2$（势能的负梯度即作用力，$-q$）。因此总能量与 q^2+p^2 成正比。

现在考虑通过三种方法求解一阶方程组：显式欧拉法、隐式欧拉法以及称为 Störmer-Verlet 方法的方法。初始条件为 $(q,p)=(1,0)$。我们采用时间步长 $h=0.05$ 并运行 500 步。分别在横纵轴上绘制 q 和 p（称为相图）。如前所述，精确解是原点为中心的单位圆。

图 8.10 展示了数值解。显式欧拉法的解呈外扩螺旋状，意味着解的位移和动量随时间递增；隐式欧拉法则呈现相反特性。两种情形下总能量图示将分别显示能量增减。采用更小步长或高阶方法可改善结果，但这些方法完全不适用于长时间积分辛系统。图 8.10(c) 显示了使用辛方法 Störmer-Verlet 所得解，可见 q^2+p^2 的保持效果明显优于前两种方法。



(a) Explicit Euler (b) Implicit Euler (c) Störmer-Verlet Method

图 8.10：初始值为 (1,0) 、时间步长 0.05 、 500 步长条件下，三种方法对简谐振荡器解的相位图。显式欧拉法中解呈外扩螺旋；隐式欧拉法中解呈内收螺旋；Störmer-Verlet 方法最能保持总能量守恒。

我们针对二阶方程推导出 Störmer-Verlet 方法

$$u'' = f(t, u)$$

只需用有限差分近似替换左侧项

$$\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} = f(t_k, u_k)$$

经重新整理后可得到该方法

$$u_{k+1} = 2u_k - u_{k-1} + h^2 f(t_k, u_k).$$

该公式同样可通过泰勒级数推导得出。此方法与线性多步法类似，需要其他技术来提供方法的初始步骤。由于交换 $k+1$ 和 $k-1$ 后公式保持不变，该方法还具有时间可逆性。遗憾的是，解释该方法为何是辛方法已超出本导论的范畴。

上述方法存在若干缺陷，最严重的是微小项 h^2 的加法运算会出现灾难性抵消。因此不应直接使用该公式形式，目前已发展出若干数学等价公式（均可由原公式推导得出）。

另一种替代公式是 *leap-frog* 方法：

$$\begin{aligned} u_{k+1} &= u_k + h v_{k+1/2} \\ v_{k+1/2} &= v_{k-1/2} + h f(t_k, u_k) \end{aligned}$$

其中 v is the first derivative (velocity) at 与位移相差半个步长。此公式不受相同舍入误差问题影响，且能提供速度值，不过需与位移重新对中以计算特定步骤的总能量。该方程组中的第二个方程本质上是有限差分公式。

斯托默 - 韦莱特方法的第三种形式是 *velocity Verlet* 变体：

$$\begin{aligned} u_{k+1} &= u_k + h v_k + \frac{h^2}{2} f(t_k, u_k) \\ v_{k+1} &= v_k + \frac{h^2}{2} (f(t_k, u_k) + f(t_{k+1}, u_{k+1})) \end{aligned}$$

其中点与位移相同。这些算法均可实现为仅需存储两组量（两个先前位置，或一个位置和一个速度）。斯托默 - 韦莱特方法的这些变体因其简洁性而广受欢迎，每步仅需一次高成本力评估。高阶方法通常不具备实用性。

速度韦莱特方案也是分子动力学多时间步算法的基础。在这些算法中，慢变（通常是长程）力的评估频率低于快变（通常是短程）力，位置更新频率也较低。最终，许多最先进的分子动力学模拟通过修正哈密顿系统来集成，以控制模拟温度和压力。针对这些系统已开发出更为复杂的辛算法。

第 9 章

组合算法

本章我们将简要探讨几种组合算法：排序，以及使用埃拉托斯特尼筛法寻找素数。

排序在科学计算中并不常见：人们更期待它在数据库（无论是金融还是生物领域，例如序列比对）中发挥更大作用。然而，它偶尔也会出现，比如在自适应网格细化（*AMR*）及其他涉及数据结构重大操作的场景中。

本节我们将简要了解一些基础算法及其并行实现方式。更多细节请参阅 [125] 及其参考文献。

9.1 排序算法简介

9.1.1 复杂度

排序算法种类繁多。传统上，人们通过计算复杂度来区分它们，即给定一个包含 n 个元素的数组，对其进行排序所需的操作次数如何随 n 变化。

理论上可以证明，排序算法的复杂度至少为 $O(n \log n)$。将排序算法视为决策树：首先进行一次比较，根据结果再进行两次比较，依此类推。因此，实际排序过程就是遍历该决策树的某条路径。若每条路径的运行时间为 h，则该树有 2^h 个节点。由于 n 个元素的序列有 $n!$ 种排列方式，决策树需要有足够路径容纳所有这些可能性；换言之，$2^h \geq n!$。根据斯特林公式，这意味着 $n \geq O(n \log n)$。

确实存在多种算法能保证达到此复杂度，但一种非常流行的算法 —— 快速排序仅具有「期望」复杂度 $O(n \log n)$，最坏情况复杂度则高达 $O(n^2)$。这种行为源于快速排序需要选择「枢轴元素」（下文第 9.3 节将详细说明），若持续选择最不理想的枢轴，则无法达到最优复杂度。

while *the input array has length* > 1 **do** 选取一个中等大小的枢轴元素 根据枢轴将数组分割为两部分 分别对两个子数组排序。**Algorithm2:** 快速排序算法。

另一方面，极为简单的冒泡排序 算法由于具有静态结构，其复杂度始终不变：

for 趟数从 1 到 $n-1$ **dofor** e 从 *1* 到 $n-$ 趟数 **doif** 元素 e 与 $e+1$ 顺序错误 **then** 交换它们 **Algorithm3:** 冒泡排序算法。

显然该算法复杂度为 $O(n^2)$：内层循环进行 t 次比较和最多 t 次交换。将 1 到 $n-1$ 的运算累加，可得约 $n^2/2$ 次比较及最多相同次数的交换。

9.1.2 排序网络

上文提到，某些排序算法的运行与实际输入数据无关，而另一些则基于数据做出决策。前一类有时被称为排序网络。它可以被视为专门实现单一算法的定制硬件。其基本硬件单元是比较交换器，该元件具有两个输入和两个输出。对于输入 x 和 y，其输出分别为 $\max(x,y)$ 和 $\min(x,y)$。

图 9.1 展示了由比较交换器构建的冒泡排序。

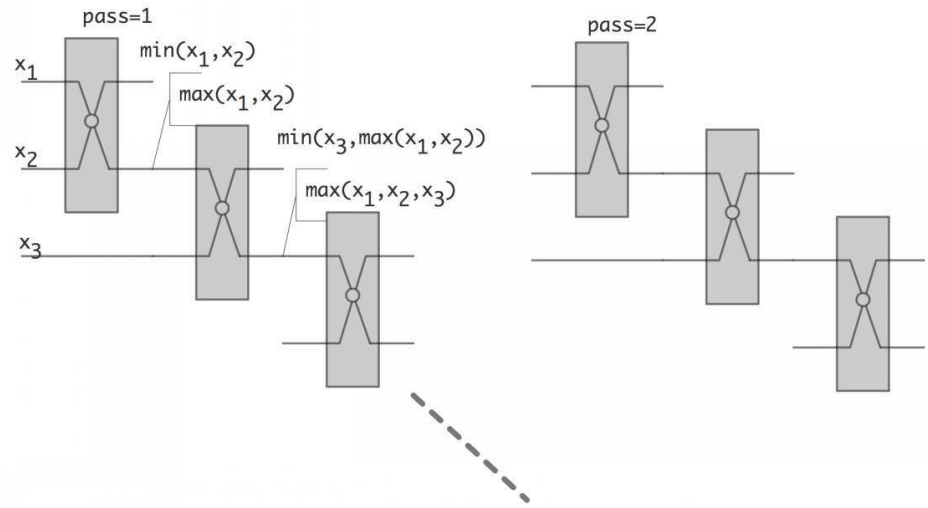


图 9.1：作为排序网络的冒泡排序。

下文我们将以双调排序算法为例，探讨排序网络的典型实现。

9.1.3 并行与顺序复杂度

前文曾提及，顺序排序至少需要 $O(N\log N)$ 时间。若假设能实现完美加速比，为简化起见使用 $P = N$ 个处理器，则并行时间应为 $O(\log N)$。若并行时间超过该值，我们将顺序复杂度定义为所有处理器上的操作总数。

该数值等同于用一个进程模拟所有其他进程执行并行算法时的操作量。理想情况下，这应与单进程算法的操作量相同，但实际情况未必如此。若数值更大，则表明存在另一种开销来源：使用并行算法带来的固有代价。例如后文将看到，排序算法的顺序复杂度通常为 $O(N\log^2 N)$。

9.2 奇偶转置排序

重新观察图 9.1 可见，第二遍操作实际上可以远早于第一遍操作就开始

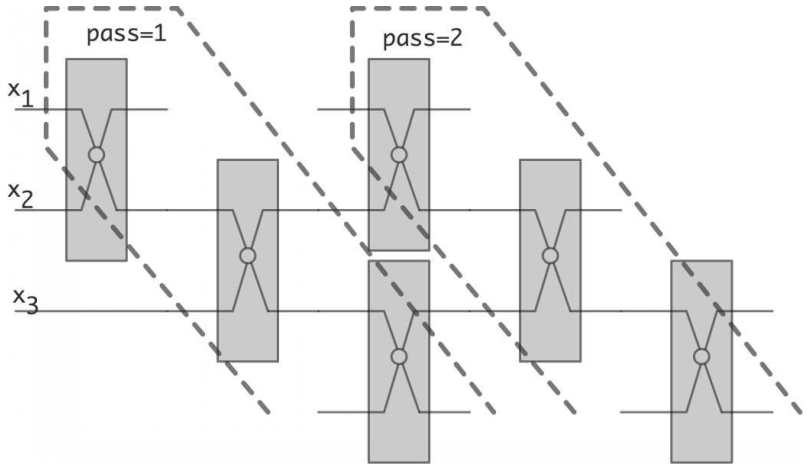


图 9.2：冒泡排序网络中的重叠遍历过程。

当一次遍历完全结束时。如图 9.2 所示。若观察任意时刻的运行状态，即可得到奇偶转置排序算法。

奇偶转置排序是一种简单的并行排序算法，其主要优点在于相对容易在处理器线性阵列上实现。但另一方面，其效率并不突出。

该算法的单步操作包含两个子步骤：

- 所有偶数编号处理器与其右侧邻居执行比较交换操作；随后
- 所有奇数编号处理器与其右侧邻居执行比较交换操作。

Theorem 3 *After $N/2$ steps, each consisting of the two substeps just given, a sequency is sorted.*

证明：在每个三元组 $2i, 2i+1, 2i+2$ 中，经过一次偶步骤和奇步骤后，最大元素将位于最右侧位置。后续可通过归纳法证明。

在并行时间为 N 的情况下，其串行复杂度为 N^2 次比较交换操作。

练习 9.1. 讨论奇偶换位排序的加速比和效率，其中使用 P 个处理器对 N 个数字进行排序；为简化假设 $N = P$，使每个处理器仅含一个数字。以比较交换操作表示执行时间。

1. 并行代码总共需要多少次比较交换操作？2. 该算法需要多少顺序步骤？当排序 N 个数字时，T_1、T_p、T_∞、S_p、E_p 分别为何值？平均并行量是多少？3. 奇偶换位排序可视为冒泡排序的并行实现。现设 T_1 表示（串行）冒泡排序的执行时间，这会如何改变 S_p 和 E_p？

9.3 快速排序

快速排序是一种递归算法，与冒泡排序不同，它不具有确定性。这是一个基于序列重排的两步过程 [1]：

算法：荷兰国旗问题数组排序
输入：一个元素数组及一个“基准”值**输出：**将输入数组按红 - 白 - 蓝顺序排列，其中红色元素大于基准值，白色元素等于基准值，蓝色元素小于基准值

我们不加证明地声明这可在 $O(n)$ 次操作内完成。基于此，快速排序算法如下：

算法：快速排序**输入：**一个元素数组**输出：**排序后的输入数组 **while** 数组长度大于 *1* 个元素 **do** 选取任意值作为基准 对该数组应用荷兰国旗重排 快速排序 (蓝色元素部分) 快速排序 (红色元素部分)

该算法的不确定性及其复杂度的变化源于枢轴（pivot）的选择。在最坏情况下，枢轴始终是数组中（唯一）最小的元素。此时将不存在蓝色元素，唯一的白色元素是枢轴本身，递归调用将作用于由 $n-1$ 红色元素组成的数组上。显而易见，此时的运行时间将达到 $O(n^2)$。反之，若枢轴始终（接近）中位数，即大小居中的元素，则递归调用的运行时间将大致相等，从而得到关于运行时间的递归公式：

$$T_n = 2T_{n/2} + O(n)$$

1. 该名称源自荷兰计算机科学家 Edsger Dijkstra，具体解释见 http://en.wikipedia.org/wiki/Dutch_national_flag_problem。

即 $O(n$ 对数 $n)$；证明详见章节 15.2。

接下来我们将探讨快速排序的并行实现方案。

9.3.1 并行快速排序

快速排序算法的简单并行化可通过并行执行两个递归调用来实现。采用共享内存模型最为便捷，并通过线程（章节 2.6.1 ）处理递归调用。

```
function par_qs( data,nprocs ) {
data_lo, data_hi = split(data);
parallel do:
par_qs( data_lo,nprocs/2 );
par_qx( data_hi,nprocs/2 );}
```

然而，若不加以优化，此实现方式效率不高。首先，递归会执行 $\log_2 n$ 步，且每一步的并行量都会翻倍，因此我们可能会认为在 n 个处理单元下整个算法耗时 $\log_2 n$。但这忽略了每一步并非消耗固定单位时间的问题。关键在于分割步骤无法简单地并行化。

练习 9.2. 请精确阐述这一论点。以此方式并行化快速排序算法的总运行时间、加速比和效率分别是多少?

是否存在更高效的数组分割方法? 事实上确实存在，关键在于使用并行前缀操作（参见附录 22 ）。若数值数组为 x_1 、 … 、 x_n，可通过并行前缀计算有多少元素小于基准值 π:

$$X_i = \#\{x_j : j < i \wedge x_j < \pi\}.$$

基于此，若处理器检查 x_i 时发现 x_i 小于基准值，则该元素需被移至数组中按基准值分割后的 $X_i + 1$ 位置。同理，可统计大于基准值的元素数量，并相应移动这些元素。

这表明每个枢轴选择步骤可在 $O(\log n)$ 时间内完成，由于排序算法共有 $\log n$ 个步骤，整个算法运行时间为 $O((\log n)^2)$。快速排序的串行复杂度为 $(\log_2 N)^2$。

9.3.2 超立方体上的快速排序

从前文可知，要实现快速排序算法的高效并行化，我们还需使荷兰国旗重排过程并行化。假设数组已被划分到维度 d 的超立方体（即 $p = 2^d$）的 p 个处理器上。

并行算法的第一步是选择枢轴并将其广播至所有处理器。随后所有处理器将独立对其本地数据执行重排操作。

为了在第一层级聚集红色与蓝色元素，每个处理器现与另一个二进制地址仅最高位不同的处理器配对。在每个

对于每一对元素，蓝色元素被发送到该比特位值为 1 的处理器；红色元素则送往该比特位值为 0 的处理器。

完成此次交换（本地操作，因此完全并行）后，地址为 $1xxxxx$ 的处理器将拥有所有红色元素，而地址为 $0xxxxx$ 的处理器则持有全部蓝色元素。前述步骤现在可以在子立方体上重复执行。

该算法确保所有处理器在每一步都处于工作状态；然而，若所选枢轴远离中位数，则易出现负载不均问题。此外，这种负载失衡在排序过程中并不会减轻。

9.3.3 通用并行处理器上的快速排序

快速排序同样可在任何具有处理器线性排序的并行机器上实现。我们首先假设每个处理器恰好持有一个数组元素，且由于标志位重排序，排序操作始终涉及一组连续的处理器。

数组（或递归调用中的子数组）的并行快速排序始于在存储该数组的处理器上构建一棵二叉树。选定一个基准值并通过该树进行广播。随后利用树结构统计每个处理器上左右子树中元素小于、等于或大于基准值的数量。

基于此信息，根处理器可计算出红 / 白 / 蓝区域将被存储的位置。该信息沿树向下传递，每个子树计算其子树中元素的目标存储位置。

若忽略网络争用，重排序操作现可在单位时间内完成，因为每个处理器最多发送一个元素。这意味着每个阶段仅需累加子树中蓝红元素的数量，顶层耗时 $O(\log n)$，下一层 $O(\log n/2)$，依此类推。这几乎实现了完美的加速比。

9.4 基数排序

大多数排序算法基于对完整项目值的比较。而基数排序则通过对数字的各位进行多轮部分排序来实现。为每个数字值分配一个 “ 桶 ”，并将数字移入这些桶中。连接这些桶可得到部分排序的数组，通过逐位处理，数组逐渐变得有序。

考虑一个最多两位数的例子，因此需要两个阶段：

数组 25 52 71 12				
last digit	5	2	1	2
(only bins 1,2,5 receive data)				
sorted	71	52	12	25
next digit	7	5	1	2
sorted	12	25	52	71

关键在于确保每一阶段的偏序关系在下一阶段得以保持。通过归纳法，我们最终能得到一个完全有序的数组。

9.4.1 并行基数排序

分布式内存排序算法天然具有数据的 ‘ 分箱 ’ 特性，因此基数排序的自然并行实现方式是以进程数 P 作为基数。

我们以双处理器为例进行说明，这意味着需要观察数值的二进制表示形式。

	proc0			proc1
数组	2	5	7	1
二进制	010	101	111	001

阶段 1：按最低有效位排序

末位数字	0	1	1	1

（此处用作桶编号）

已排序	010	101	111	001

阶段 2：按中间位排序

下一位数字	1	0	1	0

（此处作为桶编号）

已排序	101	001	010	111

阶段 3：按最高有效位排序

下一位数字	1	0	0	1

（此处用作箱号）

已排序	001	010	101	111
十进制	1	2	5	7

（我们注意到算法执行过程中可能出现负载不均衡的情况。）

分析：

- 确定当前处理的数位以及统计本地数值应归入哪个桶属于本地操作。可将其视为连通矩阵 C，其中 $C[i,j]$ 表示进程 i 将要发送给进程 j 的数据量。每个进程拥有该矩阵中对应自身的那一行。
- 为了在数据混洗阶段接收数据，每个进程需要知道将从其他每个进程接收多少数据。这需要对连通矩阵进行 ' 转置 ' 操作。用 MPI 术语来说，这是一个 *all-to-all* 集体通信操作：`MPI_Alltoall`。
- 此后，实际数据可通过另一个 all-to-all 操作进行混洗。但由于不同 i 与 j 组合对应的数据量各异，此时需要调用 `MPI_Alltoallv` 例程。

9.4.2 按最高有效位进行基数排序

完全可以让基数排序的阶段从最高有效位到最低有效位进行，而非相反顺序。在串行处理中，这不会改变任何实质性的内容。

然而，

- 与完全打乱不同，我们是在越来越小的子集中进行打乱，因此即使是串行算法也会提升空间局部性。
- 共享内存的并行版本将显示出类似的局部性改进。
- MPI 版本不再需要全对全操作。如果我们认识到每个后续阶段都将在进程的子集内进行，就可以利用通信器分割机制。

练习 9.3. 上述局部性论证略显粗略。请论证该算法可采用广度优先 和深度优先 两种方式实现，并讨论这种区分与局部性论证的关系。

9.5 样本排序

你在快速排序（第 9.3 节）中已看到概率元素可用于排序算法。我们可以将快速排序中选取单个枢轴的思想扩展为为每个处理器选取多个枢轴。这种方式不再对元素进行二分，而是将元素划分为与处理器数量相等的 “ 桶 ”，各处理器随后并行完成其元素的完整排序。

输入： p：处理器数量，N：待排序元素数量；$\{x_i\}_{i<N}$ 待排序元素设 $x_0 = b_0 < b_1 < \cdots < b_{p-1} < b_p = x_N$（其中 $x_N > x_{N-1}$ 可任意）**对于** $i = 0, \ldots, p-1$ **执行**设 $s_i = [b_i, \ldots b_{i+1} - 1]$ **对于** $i = 0, \ldots, p-1$ **执行**将 s_i 中的元素分配给处理器 i **对于** $i = 0, \ldots, p-1$ 并行**执行**令处理器 i 排序其元素**算法 4：** 样本排序算法

显然，若未谨慎选择桶，该算法可能出现严重的负载不均问题。随机选取 p 元素很可能不够理想；相反，需要对元素进行某种形式的采样。因此，该算法被称为**样本排序** [17]。

W尽管桶的分配后排序过程是完全并行的，该算法仍存在一些问题
r关于并行性。首先，采样步骤是算法的顺序瓶颈。此外，将桶分配
w给处理器的步骤本质上是一种全对全操作，

对此进行分析时，假设存在 P 个进程，它们首先作为映射器，随后作为归约器。设 N 为数据点数量，并定义块大小 $b \equiv N/P$。算法各处理步骤的成本为：

- 本地确定每个元素的所属桶，耗时 $O(b)$；以及

- 局部排序的复杂度可假设为最优值 $b \log b$。

然而，数据混洗步骤并非易事。除非数据已部分预排序，否则混洗操作将是一个完整的全对全通信，时间复杂度为 $P\alpha + b\beta$，并可能成为网络瓶颈。注意在超立方体上的快速排序中，线路从未出现争用问题。

习题 9.4. 论证当进程数 $P \ll N$ 较小时，该算法具有完美加速比，且其顺序复杂度（见上文）为 $N \log N$。

与双调排序等排序网络相比，该排序算法看似显著简化：仅需单步网络。前文论证在 ' 乐观扩展 ' 场景下（工作量增加而处理器数不变），其顺序复杂度与串行算法相同。但在弱扩展分析中，当工作量与处理器数成比例增加时，顺序复杂度会显著恶化。

习题 9.5. 考虑同时缩放 N 和 P 而保持 b 不变的情况。论证在此情况下，洗牌步骤会给算法引入一个 N^2 项。

9.5.1 通过 MapReduce 排序

Terasort 基准测试涉及 对大型基于文件的数据集进行排序。因此，它某种程度上成为大数据 系统的标准。特别是，*MapReduce* 是主要候选方案；参见 http://perspectives.mvdirona.com/2008/07/hadoop-wins-terasort/。

使用 MapReduce 时，算法按以下步骤进行：

- 通过采样或先验信息确定一组键值：这些键值应确保每对键值之间预期落入的记录数大致相等。区间数量等于归约器进程的数量。
- 接着，映射器进程生成键 / 值对，其中键为区间号或归约器编号。
- 归约器进程随后执行局部排序。

可见，除了术语变更外，这实际上就是样本排序 (samplesort)。

9.6 双调排序

为引入双调排序，假设序列 $x = \langle x_0, \dots, x_n - 1\rangle$ 由升序部分后接降序部分组成。现将其拆分为两个等长子序列，定义如下：

$$\begin{aligned} s_1 &= \langle \min\{x_0, x_{n/2}\}, \dots \min\{x_{n/2-1}, x_{n-1}\} \\ s_2 &= \langle \max\{x_0, x_{n/2}\}, \dots \max\{x_{n/2-1}, x_{n-1}\} \end{aligned} \tag{9.1}$$

从图示可直观看出 s_1, s_2 仍是包含升序与降序部分的序列。此外，s_1 中所有元素均小于 s_2 中的所有元素。

我们将 (9.1) 称为升序双调排序器，因第二个子序列包含比第一个子序列更大的元素。同理可通过调换最大值与最小值角色构造降序排序器。

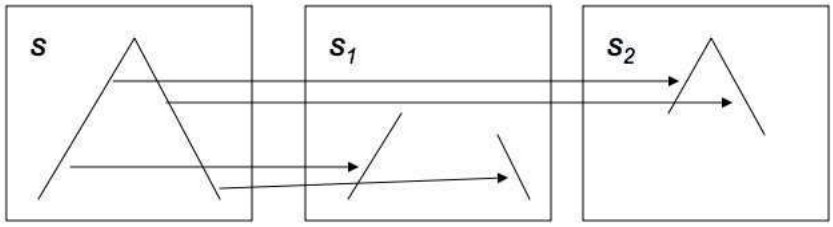


图 9.3：双调序列分割示意图。

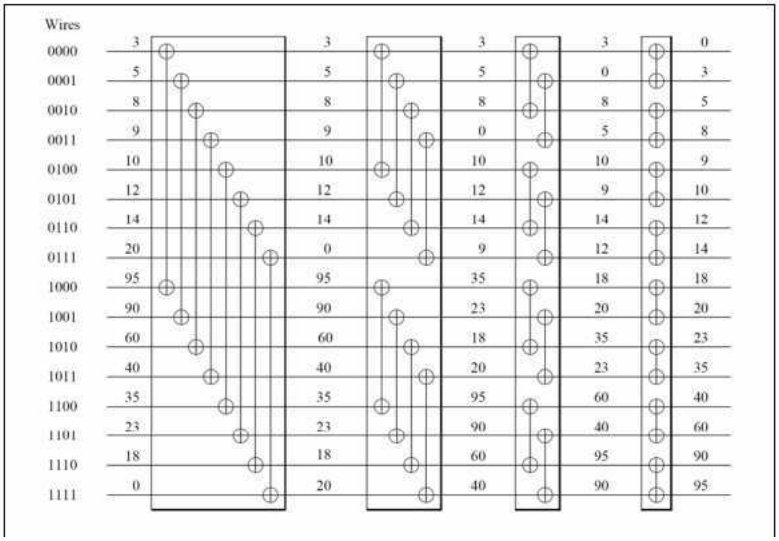


Figure 9.4: Illustration of a bitonic network that sorts a bitonic sequence of length 16.

不难想象这是排序算法中的一个步骤：从这种形式的序列开始，递归应用公式 (9.1) 将得到有序序列。图 9.4 展示了 4 个距离分别为 8 、 4 、 2 、 1 的双调排序器如何对长度为 16 的序列进行排序。

双调序列的实际定义稍复杂些：若一个序列由递增部分后接递减部分组成，或是此类序列的循环排列，则该序列为双调序列。

习题 9.6. 证明根据公式 (9.1) 分割双调序列可得到两个双调序列。

因此问题在于如何获得双调序列。答案是使用规模递增的双调网络。

- 对两个元素进行双调排序可得到一个有序序列。
- 若有两个长度为二的序列，一个升序排列，另一个降序排列，便构成双调序列。

- 因此这个长度为四的序列可通过两步双调排序完成排序。
- 而两个长度为四的有序序列可组成一个长度为八的双调序列；
- 该序列可通过三步双调排序完成排序；以此类推。

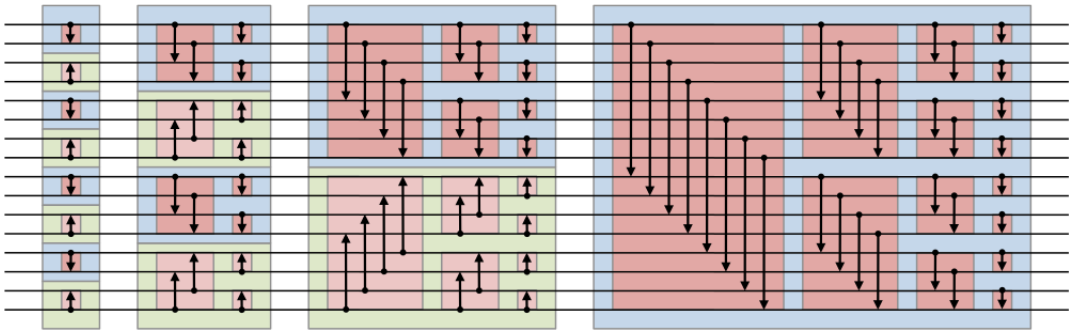


图 9.5：16 个元素的完整双调排序。

从这个描述中可以看出，你需要 $\log_2 N$ 个阶段来排序 N 个元素，其中第 i 个阶段的长度为 $\log_2 i$。这使得双调排序的串行复杂度总计为 $(\log_2 N)^2$。

图中的操作序列 9.5 被称为排序网络，由简单的比较交换元素构建而成。与快速排序不同，它对数据元素的值没有依赖性。

9.7 素数查找

埃拉托斯特尼筛法是一种非常古老的查找素数的方法。在某种程度上，它仍然是许多更现代方法的基础。

列出从 2 到某个上限 N 的所有自然数，我们将把这些数字标记为素数或明确非素数。所有数字初始状态均为未标记。

- 第一个未标记的数字是 2：将其标记为素数，并标记其所有倍数为非素数。
- 第一个未标记的数字是 3：将其标记为素数，并标记其所有倍数为非素数。
- 下一个数字是 4，但已被标记过，因此标记 5 及其倍数。
- 下一个数字是 6，但已被标记过，因此标记 7 及其倍数。
- 数字 8 、 9 、 10 已被标记，故继续处理 11。
- 等等。

第 10 章

图分析

科学计算中的许多问题可以表述为图问题；关于图论的介绍请参阅附录 20 ）。例如，您已经遇到过负载均衡问题（第 2.10.4 节）和寻找独立集问题（第 7.8.2 节）。

许多传统的图算法并不能直接或至少不能高效地应用，因为这些图通常是分布式的，而传统图论假设对整个图有全局了解。此外，图论通常关注寻找最优算法，而最优算法通常不是并行的。相反，在并行计算中，我们可能获得足够的加速比，从而可以承受在非顺序最优算法上浪费一些计算周期。因此，并行图算法本身就是一个研究领域。

最近，科学计算中出现了新型的图计算。在这里，图不再是工具，而是研究对象本身。例如万维网或社交图谱 *Facebook*，或者生物体内所有可能的蛋白质相互作用图。

正因如此，组合计算科学正逐渐成为一门独立的学科。本节我们将探讨图分析：针对大型图的计算。我们首先讨论一些经典算法，但会将其置于代数框架中呈现，这将使并行实现更为便捷。

10.1 传统图算法

我们首先考察几个 ‘经典’ 图算法，并探讨其并行实现方式。图与稀疏矩阵之间的关联（参见附录 20 ）在此至关重要：许多图算法具有稀疏矩阵 - 向量乘法的结构。

10.1.1 最短路径算法

最短路径算法有多种类型。例如，在单源最短路径算法中，需求解从给定节点到任意其他节点的最短路径；而全对最短路径算法则旨在计算任意两节点间的最短距离。实际路径的求解并非其组成部分

这些算法的实现中通常不会显式存储路径；不过，一般很容易通过附加信息来后续重建路径。

我们从简单算法开始：在无权图中寻找单源最短路径。这可以通过广度优先搜索（ BFS ）轻松实现：每一步我们都有一组已知最短距离的节点 U，然后考虑它们的邻居集合 V。

输入： 一个图及起始节点 s **输出：** 测量从 s 到 v 距离的函数
$d(v)$ 给定 s，并设置 $d(s) = 0$ 将已完成集合初始化为 $U = \{s\}$
设置 $c = 1$ **当**未完成时**执行**设 V 为 U 的邻居集合且这些邻居
不在 U 中**如果** $V = \varnothing$ **则**完成**否则对于** $v \in V$ **执行**设置
$d(v) = c + 1 U \leftarrow U \cup V$ 增加 $c \leftarrow c + 1$

这种算法表述方式对理论分析很有帮助：通常你可以为 while 循环的每次迭代定义一个谓词。这使你能够证明算法会终止，并且计算出了预期的结果。在传统处理器上，这确实是编写图算法的方式。然而，如今像 Facebook 这样的图数据可能非常庞大，你需要以并行方式编写图算法。

在这种情况下，传统表述方式存在不足：

- 它们通常基于节点的增减队列；这意味着存在某种形式的共享内存。
- 关于节点和邻居的陈述没有考虑它们是否满足任何空间局部性；如果一个节点被多次访问，也无法保证时间局部性。

10.1.2 Floyd-Warshall 全对最短路径

Floyd-Warshall 算法是全对最短路径算法的一个例子。它是一种基于逐步扩大路径中间节点集合的动态规划算法。具体来说，在第 k 步中，会考虑所有中间节点位于集合 $k = \{0, \dots, k-1\}$ 内的路径 $u \rightsquigarrow v$，并将 $\Delta_k(u, v)$ 定义为从 u 到 v 且所有中间节点都在 k 中的路径长度。初始时，对于 $k = 0$ 这意味着没有中间边，因此只考虑图的边；当 $k \equiv |V|$ 时，我们已经考虑了所有可能的路径并完成计算。

计算步骤是

$$\Delta_{k+1}(u, v) = \min\{\Delta_k(u, v), \Delta_k(u, k) + \Delta_k(k, v)\}. \tag{10.1}$$

即，距离 $\Delta(u,v)$ 的第 k 次估计值取旧值与新路径中的较小者，该新路径因当前考虑节点 k 而变得可行。后者通过连接路径 $u \rightsquigarrow k$ 和 $k \rightsquigarrow v$ 获得。

用算法形式表示：

for *k from zero to* $|V|$ **do**
  for *all nodes* u, v **do**
    $\Delta_{uv} \leftarrow f(\Delta_{uv}, \Delta_{uk}, \Delta_{kv})$

可见该算法与高斯消元法结构相似，只是内层循环应为 ‘对所有 $u, v > k$’。（亦可类比 *Gauss Jordan* 算法。）

代数形式：

for *k from zero to* $|V|$ **do**
  $D \leftarrow D._{\min}\left[D(:,k) \min \cdot_{+} D(k,:)\right]$

Floyd-Warshall 算法不提供实际路径。在上述距离计算过程中存储这些路径会耗费大量时间和内存。可采用更简单的解决方案：存储第二个矩阵 $n(i,j)$，其中包含 i 与 j 之间路径的最高节点编号。

练习 10.1. 在 Floyd-Warshall 算法中加入 $n(i,j)$ 的计算，并描述在已知 $d(\cdot,\cdot)$ 和 $n(\cdot,\cdot)$ 后如何利用该算法找到 i 与 j 之间的最短路径。

在 5.4.3 节中已了解到稀疏矩阵的分解会导致 填充（*fill-in*）现象，此处同样存在该问题。这需要灵活的数据结构支持，且在并行环境下问题会进一步加剧；详见 10.5 节。

10.1.3 生成树

在无向图 $G = \langle V, E \rangle$ 中，若 $T \subset E$ 连通且无环，则称其为 ‘树’。若其边集包含所有顶点，则称为生成树。若图具有边权 $w_i : i \in E$，则树的权重为 $\sum_{e \in T} w_e$，当该树具有最小权重时，称为最小生成树。最小生成树不唯一。

普里姆算法是迪杰斯特拉最短路径算法的一个轻微变体，它从根节点开始计算生成树。根节点的路径长度为零，所有其他节点的路径长度初始为无穷大。在每一步中，会考虑所有与已知树节点相连的节点，并更新它们当前已知的最佳路径长度。

输入： 图 $\langle V、\ E\rangle$ 及节点 $s \in V$ **输出：** 函数 $\ell(\cdot)$ 给出任意节点到 s 的最短距离 **对于** 所有顶点 v **执行** $\ell(v) \leftarrow \infty \ell(s) \leftarrow 0$ $Q \leftarrow V - \{s\}$ 和 $T \leftarrow \{s\}$ **当** $Q \neq \emptyset$ **执行** 令 u 为 Q 中具有最小 $\ell(u)$ 值的元素，将 u 从 Q 移除并加入 T **对于** $v \in Q$ 与 $(u, v) \in E$ **执行 若** $\ell(u) + w_{uv} < \ell(v)$ **则** 设置 $\ell(v) \leftarrow \ell(u) + w_{uv}$

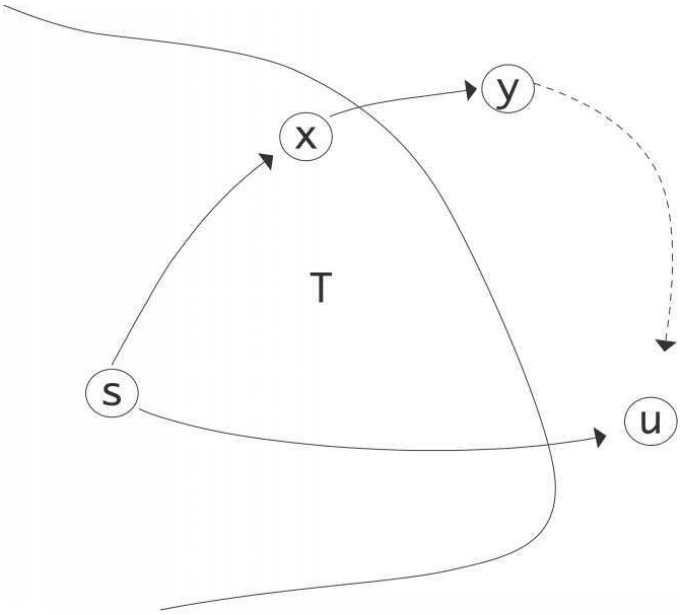


图 10.1：迪杰斯特拉算法正确性示意图

定理 4 上述算法计算了每个节点到根节点的最短距离

证明。该算法正确性的关键在于我们选择具有最小 $\ell(u)$ 值的 u。设顶点 $L(v)$ 的真实最短路径长度为 $L(v)$。由于我们初始时 ℓ 值为无穷大且仅会递减，因此始终满足 $L(v) \leq \ell(v)$。

我们的归纳假设是：在算法的任何阶段，对于当前 T 中的节点，路径长度已被正确确定：

$$u \in T \Rightarrow L(u) = \ell(u).$$

当树仅由根节点 s 组成时，这显然成立。现在我们需要证明归纳步骤：如果当前树中所有节点的路径长度都正确，那么我们也将得到 $L(u) = \ell(u)$。

假设此命题不成立，则存在另一条更短的路径。该路径需经过某个节点 y，而该节点当前不在 T 中；如图 *10.1* 所示。设 x 为 T 中这条所谓最短路径上紧邻 y 之前的节点。此时有 $\ell(u) > L(u)$（因尚未获得正确路径长度），且 $L(u) > L(x) + w_{xy}$（因 y 与 u 之间至少存在一条具有正权重的边）。但 $x \in T$，故 $L(x) = \ell(x)$ 且 $L(x) + w_{xy} = \ell(x) + w_{xy}$。注意到当 x 被加入 T 时，其邻接节点已被更新，因此 $\ell(y)$ 的值应 $\ell_x + w_{xy}$ 或更小。串联这些不等式可得

$$\ell(y) < \ell(x) + w_{xy} = L(x) + w_{xy} < L(u) < \ell(u)$$

这与我们选择 u 具有最小 ℓ 值的事实矛盾 .

要实现该算法的并行化，我们注意到内层循环的迭代相互独立，因此可并行执行。然而外层循环每次迭代需通过最小化函数值进行选择。该选择计算属于归约操作，随后需进行广播。此策略使得串行时间复杂度为 $d \log P$，其中 d 为生成树的深度。

在单处理器上，查找数组中的最小值直观上是一个 $O(N)$ 操作，但通过使用优先级队列可将其降低至 $O(\log N)$。对于并行版本的生成树算法，对应项为 $O(\log(N/P))$，不考虑归约的 $O(\log P)$ 成本。

Bellman-Ford 算法与 Dijkstra 算法类似，但能处理负权边。其时间复杂度为 $O(|V| \cdot |E|)$。

10.1.4 图切割

有时可能需要划分图结构，例如为了并行处理。若通过对顶点进行划分实现，则相当于切割边，因此这被称为顶点切割划分。

优质顶点切割有多种评判标准。例如，为使并行工作负载均衡，需确保切割后的各部分规模大致相当。由于顶点常对应通信过程，还需使切割线上的顶点数量（或加权图中的权重总和）尽可能小。图拉普拉斯算子（章节 20.6.1 ）是解决该问题的常用算法。

图拉普拉斯算法相较于传统图分割算法的优势在于 [139] 其更易于并行化。另一种流行的计算方法采用多层次策略：

1. 对图的粗化版本进行分割；2. 逐步细化图，并调整分割方案。

图切割的另一个例子是二分图的情况：具有两类节点且边仅存在于两类节点之间的图。此类图可建模为人群与属性集的关系：边表示某人具有特定兴趣。

10.1.5 图算法的并行化

许多图算法（如 10.1 节所述）的并行化并非易事。以下是几点考量：

首先，与其他算法不同，由于最外层循环通常是 ‘while’ 循环，因此难以直接并行化该层级，这使得并行停止测试成为必要。另一方面，通常存在

宏观步骤是顺序执行的，但其中多个变量被独立考虑。因此确实存在可挖掘的并行性。

T图算法中的独立工作具有一种有趣的结构。虽然我们可以识别出 ‘forall’
l这些是并行化的候选操作，但它们与我们之前所见的情况不同。

- 传统实现通常涉及逐步构建或消耗的变量集合。这可通过共享数据结构和任务队列来实现，但会将实现限制为某种形式的共享内存。
- 其次，虽然每次迭代中的操作是独立的，但它们所操作的动态集合意味着将数据元素分配给处理器是复杂的。固定分配可能导致大量空闲时间，而动态分配则带来较大开销。
- 采用动态任务分配时，该算法将几乎不具备空间或时间局部性。

10.1.5.1 全对算法的并行化

在单源最短路径算法（第 10.1.1 节）中，我们别无选择，只能通过分布向量而非矩阵来实现并行化。这种分布方式在此同样适用，它对应于 $D(\cdot,\cdot)$ 量的一维分布。

习题 10.2. 勾画该算法变体的并行实现。证明每次 k 次迭代都涉及以处理器 k 为根的广播操作。

然而，这种方法会遇到与采用矩阵一维分布的矩阵 - 向量积相同的扩展性问题；参见第 7.2.3 节。因此我们需要采用二维分布。

习题 10.3. 进行扩展性分析。在内存恒定的弱扩展场景下，其渐近效率是多少?

练习 10.4. 使用 $D(\cdot,\cdot)$ 量的二维分布来勾勒 Floyd-Warshall 算法。

练习 10.5. 并行 Floyd-Warshall 算法在早期阶段确实会对零值执行大量操作。你能设计一个避免这些操作的算法吗?

10.2 邻接矩阵上的线性代数

A 图通常可以方便地通过其邻接矩阵来描述；参见 20.5 节。在本节中，我们
讨论将此矩阵作为线性代数对象进行操作的算法。

在许多情况下，我们实际上需要左积，即从左侧用行向量乘以邻接矩阵。例如，设 e_i 为在位置 i 处为 1 而其他位置为零的向量。那么 $e_i^t G$ 在 G 的邻接图中，每个 j （即 i 的邻居）处都有一个 1。

更一般地，如果我们计算单位基向量 x 对应的 $x^t G = y^t$，则有

$$\begin{cases} x_i \neq 0 \\ x_j = 0 \quad j \neq i \end{cases} \wedge G_{ij} \neq 0 \Rightarrow y_j \neq 0.$$

非正式地说，我们可以认为 G 从状态 i 转换到了状态 j。

另一种理解方式是，如果 $x_i \neq 0$，那么对于满足 $(i,j) \in E$ 的那些 j 来说， $y_j \neq 0$ 成立。

10.2.1 状态转移与马尔可夫链

我们经常遇到仅含非负元素的向量，在这种情况下

$$\begin{cases} x_i \neq 0 \\ x_j \geq 0 \quad \text{all } j \end{cases} \wedge G_{ij} \neq 0 \Rightarrow y_j \neq 0.$$

T其左侧的矩阵 - 向量乘积有一个简单应用：马尔可夫链。假设我们有一个系统（参见例如 21 ）可以处于 n 种状态中的任意一种，且处于这些状态的概率之和为 1。随后我们可以利用邻接矩阵来建模状态转移。

设 G 为一个邻接矩阵，其属性是所有元素均为非负数。对于马尔可夫过程，从给定状态出发的所有状态转移概率之和为 1。现在我们将邻接矩阵 g_{ij} 的元素解释为从状态 i 转移到状态 j 的概率。此时，矩阵中每一行的元素之和均为 1。

设 x 为概率向量，即 x_i 表示处于状态 i 的非负概率，且 x 中各元素之和为 1，那么 $y^t = x^t G$描述了一次状态转移后的这些概率。

习题 10.6. 要使向量 x 成为规范的概率向量，其元素之和需为 1。证明：对于前述矩阵， y 的元素之和仍为 1。

10.2.2 图算法的计算形式

我们为图操作引入两个抽象运算符 $\bigoplus$和 $\otimes$ 。例如，计算最短距离可以表示为基本步骤的迭代应用：

到节点 y_i 的距离是已知距离与到节点 x_j 的距离加上弧 a_{ij} 值之间的最小值。

我们将这一步骤更抽象地表示为

$$y_i \bigoplus a_{ij} \otimes x_j$$

类比于矩阵 - 向量乘法

$$y_i \leftarrow y_i + a_{ij} x_j$$

但采用以下定义

$$y_i \bigoplus \xi_j \equiv y_i \leftarrow \min\{y_i, \xi_j\} \qquad \text{and} \qquad a_{ij} \otimes x_j \equiv \begin{cases} x_j + a_{ij} & a_{ij} \neq 0 \wedge x_j \neq \infty \\ \infty & a_{ij} = 0 \vee x_j = \infty \end{cases}$$

最短距离算法现在变为：

直到为节点 i 和节点 j
$y_i \bigoplus a_{ij} \otimes x_j$ 计算出所有距
离

您可能会注意到这看起来像一系列矩阵 - 向量乘积运算，实际上，使用常规的 $+, \times$ 运算符时，该算法确实会进行此类计算。我们将继续探讨图算法与线性代数之间的相似性，但需指出上述 $\bigoplus$, $\otimes$ 运算符并非线性，甚至不满足结合律。

该形式化方法将用于下文多个算法；其他应用场景参见 [127]。

10.2.2.1 算法变换

类似于矩阵 - 向量乘积的变换，我们现在探讨由循环变换产生的该图算法的不同变体。

我们首先通过交换 i 、 j 索引来转换上述算法，得到如下结果：

for nodes j
    for nodes i
        $y_i \bigoplus a_{ij} \otimes x_j$

接下来，我们通过移除某些迭代使外层循环更高效。通过观察发现 x_j 在内层循环中保持不变，且内层循环的更新对 $x_j = \infty$ 无效，因此我们将外层循环限制在 $x_j < \infty$ 的索引上。同样地，我们限制内层循环，并应用一种表示法

$$j \to i \equiv a_{ij} \neq 0.$$

Together this gives:

for nodes $\{j : x_j < \infty\}$
    for nodes $\{i : j \to i\}$
        $y_i \bigoplus a_{ij} \otimes x_j$

在此我们识别出图算法的经典表述：
令已知距离集合 $D = \{$ 根节点 $\}$ 直到 $D = E$,
即已知每条边对于每个 $j \in D$ 的距离：对于每
个与 j 相连的 i ： 将 i 添加到 D，距离为
$d_j + e_{j \to i}$

10.2.2.2 “推送模型”

从图示中可以看出，我们可以将刚刚推导出的算法称为 “推送模型”，因为它反复将一个节点的值推送到其相邻节点上。这与矩阵 - 向量乘积的一种形式类似，其中结果向量是矩阵各列的和。

虽然该算法直观易懂，但就性能而言并不出色 .

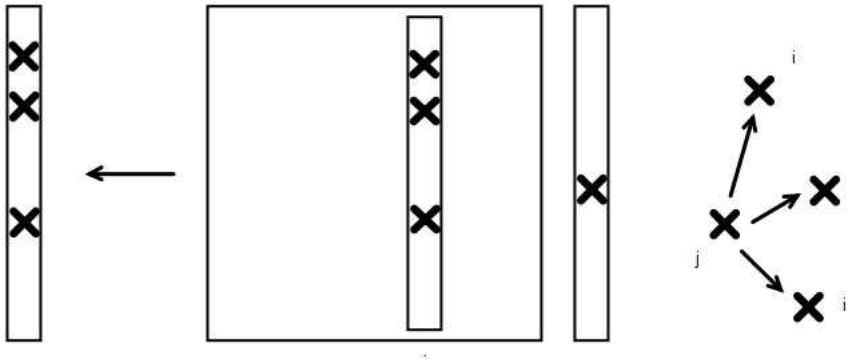


Figure 10.2: Push model of graph algorithms, depicted in matrix-vector form

- 由于结果向量被反复更新，很难对其进行缓存。
- 基于推送模型的本质特性，该变体难以并行化，因为它可能涉及冲突更新。因此，线程化代码需要原子操作、临界区、锁或其他类似机制。分布式内存代码的编写则更为困难。

10.2.2.3 “拉取”模型

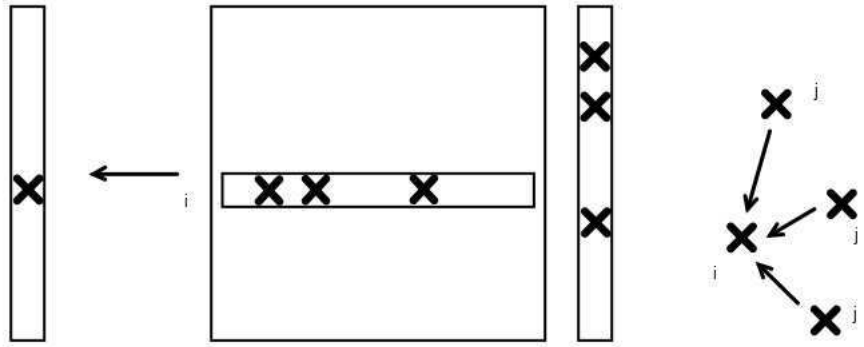


图 10.3：图算法的拉取模型（以矩阵 - 向量形式表示）

若保持原始 i, j 循环嵌套结构，我们无法通过移除某些迭代来优化外层循环，最终得到：

对于所有节点 i，对于
节点 $\{j : j \to i\}$
$y_i \bigoplus a_{ij} \otimes x_j$

这与传统的基于内积的矩阵向量积类似，但作为一种图算法，该变体在传统文献中并不常见。我们可以将其描述为 ‘拉取模型’，因为每个节点 i 都会从其所有邻居 j 拉取更新。

关于性能，我们现在可以声明：

- 作为顺序算法，这可以轻松缓存输出向量。
- 该算法可轻松在 i 索引上并行化；不存在写冲突。
- 分布式算法同样简单直接，计算前需对输入向量执行（邻居）全收集操作。

- 但作为传统表述的图算法，它存在大量冗余，每个 y_j 组件会被多次更新。

10.3 ‘ 真实世界 ’ 图

在诸如 4.2.3 节的讨论中，您已看到偏微分方程离散化如何导致具有图论特性的计算问题。此类图具有使其适用于特定问题的属性。例如，使用有限差分法或有限元法对二维或三维对象建模时，生成的图中每个节点仅与少数相邻节点相连。这使得寻找分隔符变得容易，从而可以采用嵌套剖分等求解方法；参见 7.8.1 节。

然而存在一些计算密集型图论问题，其图结构不同于有限元图。我们将简要考察万维网的例子，以及诸如 *Google* 的 *PageRank* 这类试图寻找权威节点的算法。

目前我们将此类图称为随机图，尽管该术语也有其技术含义 [59]。

10.3.1 并行化考量

在 7.5 节中我们讨论了稀疏矩阵 - 向量乘积的并行计算。由于稀疏性，仅按块行或块列划分才有意义。实际上，我们让划分由其中一个问题变量决定。这也是单源最短路径算法唯一可行的策略。

练习 10.7. 你能为基于向量分布的并行化提出一个先验论证吗？该操作涉及多少数据量、工作量以及需要多少顺序步骤？

10.3.2 随机图的属性

我们在本课程中见到的大多数图都具有某些属性，这些属性源于它们模拟的是我们三维世界中的对象。因此，两个节点之间的典型距离通常是 $O(N^{1/3})$，其中 N 代表节点数量。随机图则并非如此：它们常具备小世界属性，其典型距离为 $O(\log N)$。一个著名的例子是电影演员及其通过共同出演同一部电影建立的连接图：根据 “ 六度分隔理论 ”，在该图中任意两位演员之间的距离不超过六。用图论术语来说，这意味着该图的直径为六。

S小世界图还具有其他属性，例如存在派系（尽管这些特征也
i高阶有限元问题中的枢纽节点：高度数节点。这会导致诸如
f以下结果：在此类图中随机删除节点对最短路径影响甚微。

练习 10.8. 考虑由机场及其间航线构成的图。若只存在枢纽与非枢纽两类节点，请论证删除一个非枢纽节点不会影响其他机场间的最短路径。反之，假设节点按二维网格排列，删除任意一个节点后，会有多少最短路径受到影响?

10.4 超文本算法

存在多种基于线性代数的算法，用于衡量网站重要性 [132] 并排序网页搜索结果。我们将简要定义其中几种算法并讨论其计算意义。

10.4.1 HITS 算法

在 HITS （超文本诱导主题搜索）算法中，网站具有枢纽分数（衡量其指向其他网站的数量）和权威分数（衡量被其他网站指向的数量）。为计算这些分数，我们定义一个关联矩阵 L，其中

$$L_{ij} = \begin{cases} 1 & \text{document } i \text{ points to document } j \\ 0 & \text{otherwise} \end{cases}$$

权威分数 x_i 被定义为指向 i 的所有事物的枢纽分数 y_j 之和，反之亦然。因此

$$\begin{aligned} x &= L^t y \\ y &= Lx \end{aligned}$$

或 $x = LL^t x$和 $y = L^t Ly$，表明这是一个特征值问题。我们需要的特征向量仅包含非负项；这被称为 *Perron* 向量 对于非负矩阵，参见附录 14.4。Perron 向量通过幂方法计算；参见章节 14.3。

一个实用的搜索策略是：

- 找到所有包含搜索词的文档；
- 构建这些文档的子图，以及可能与它们相关的一到两级文档；
- 计算这些文档的权威分数和枢纽分数，并按有序列表呈现给用户。

10.4.2 PageRank 算法

PageRank 算法的 [156] 基本思想与 HITS 类似：它通过建模回答 ‘如果用户持续点击页面上最具吸引力的链接，最终哪些链接会成为整体上最具吸引力的集合’。其核心模型定义为：网页的排名等于所有链接到该页面的其他网页排名的总和。该算法通常以迭代方式表述：

while *Not converged* **do**
  for *all pages i* **do**
    $\text{rank}_i \leftarrow \epsilon + (1-\epsilon)\sum_{j\,:\,\text{connected}j\to i} \text{rank}_j$

  对排名向量进行归一化处理

其中排名是该算法的固定点。ϵ 项解决了若某页面无外链时，用户陷入后无法退出的问题。

练习 10.9. 论证该算法可被两种方式解读，大致对应雅可比与高斯 - 赛德尔迭代方法；参见章节 5.5.3。

练习 10.10. 在 PageRank 算法中，每个页面将其排名 ' 赋予 ' 所链接的页面。给出该变体的伪代码。证明其对应按列的矩阵 - 向量积，而非前述按行的表述。在共享内存并行实现中会存在什么问题?

关于此方法收敛性的问题，最好采用线性代数语言来描述。我们定义连通性矩阵

$$M_{ij} = \begin{cases} 1 & \text{if page } j \text{ links to } i \\ 0 & \text{otherwise} \end{cases}$$

通过 $e = (1,\ldots,1)$，向量 $d^t = e^t M$统计页面上的链接数量：d_i 表示页面 i 上的链接数。我们构建一个对角矩阵 $D = \text{diag}(d_1,\ldots)$，并将 M 归一化为 $T = MD^{-1}$。

现在 T 的列和（即任一列中元素的总和）均为 1，可表示为 $e^tT = e^t$，其中 $e^t = (1,\ldots,1)$。这样的矩阵称为随机矩阵。其含义如下：

> 若 p 是概率向量，即 p_i 表示用户正在浏览页面 i 的概率，则 Tp 是用户点击随机链接后的概率向量。

练习 10.11 从数学角度看，概率向量的特征是其元素之和为 1。证明随机矩阵与概率向量的乘积确实仍是概率向量。

如上所述的 PageRank 算法将对应于取任意随机向量 p，计算幂方法 Tp 、T^2p、T^3p、 …，并观察该序列是否收敛于某值。

这个基础算法存在若干问题，例如没有出链的页面。从数学角度而言，我们通常处理的是 ‘ 不变子空间 ’ 。例如，考虑一个仅含 2 个页面且具有如下邻接矩阵的网络：

$$A = \begin{pmatrix} 1/2 & 0 \\ 1/2 & 1 \end{pmatrix}.$$

请自行验证这对应于第二个页面没有出链的情况。现在令 p 作为初始向量 $p^t = (1,1)$，并计算幂方法的数次迭代。你是否注意到用户位于第二个页面的概率趋近于 1? 此处的问题在于我们面对的是一个可约矩阵。

为防止此问题，PageRank 引入了另一个要素：有时用户会因连续点击感到厌倦，转而跳转至任意页面（同时也考虑了无外链页面的情况）。若设 s 为用户点击链接的概率，则跳转至任意页面的概率为 $1-s$。综合起来，我们得到以下过程

$$p' \leftarrow sTp + (1-s)e,$$

即，若 p 是概率向量，则 p' 是描述用户通过点击链接或 ‘瞬移’ 方式进行一次页面跳转后所处位置的概率向量。

PageRank 向量是该过程的稳态点；可将其视为用户进行无限次跳转后的概率分布。该向量满足

$$p = sTp + (1-s)e \Leftrightarrow (I - sT)p = (1-s)e.$$

因此，我们现在需要探讨 $I - sT$ 是否存在逆矩阵。若逆矩阵存在，则满足

$$(I - sT)^{-1} = I + sT + s^2T^2 + \cdots$$

不难证明逆矩阵存在：根据 Gershgorin 定理（附录 14.5 ）可知 T 的特征值满足 $|\lambda| \leq 1$。现利用 $s < 1$，故该级数部分和收敛。

上述逆矩阵公式也指明了一种通过一系列矩阵 - 向量乘法计算 PageRank 向量 p 的方法。

练习 10.12. 编写计算 PageRank 向量的伪代码，给定矩阵 T。证明你无需显式计算 T 的幂次。（这是霍纳法则的一个实例）。

当 $s = 1$ 时（即排除随机跳转的情况），PageRank 向量满足 $p = Tp$，这 h
又回到了寻找 *Perron* 向量的问题；参见附录 14.4。

我们通过幂迭代法（第 14.3 节）寻找 Perron 向量

$$p^{(i+1)} = Tp^{(i)}.$$

这是一个稀疏矩阵向量乘积，但与边界值问题不同，其稀疏性不太可能具有带状结构等特定模式。在计算上，可能需要采用与稠密矩阵相同的并行化论证：矩阵需要二维分布 [152]。

10.5 大规模计算图论

在前面的章节中，你已经看到许多图算法的计算结构使得矩阵 - 向量乘积成为其最重要的核心运算。由于大多数图相对于节点数量而言度数较低，该乘积属于稀疏矩阵 - 向量乘积。

稠密矩阵 - 向量乘积依赖于集合通信（参见第 7.2 节），而稀疏情形则采用点对点通信，即每个处理器仅向少数相邻节点发送数据，并仅从少数节点接收数据

少数。这对于来自偏微分方程（ PDEs ）的稀疏矩阵类型是有意义的，这些矩阵具有清晰的结构，正如你在第 4.2.3 节中所见。然而，有些稀疏矩阵是如此随机，以至于你基本上不得不使用密集技术；参见第 10.5 节。

在许多情况下，我们可以像第 7.5 节那样，通过图节点的分布来诱导矩阵的一维分布：如果一个处理器拥有一个图节点 i，它就拥有所有的边 i, j。

然而，计算往往非常不平衡。例如，在单源最短路径算法中，只有前沿的顶点是活跃的。因此，有时边的分布比顶点的分布更有意义。为了均衡负载，甚至可以使用随机分布。

弗洛伊德 - 沃舍尔算法（ Floyd-Warshall algorithm ）的并行化（第 10.1.2 节）采用了不同的方法。这里我们不计算每个节点的量，而是计算节点对的函数，即类似矩阵的量。因此，我们不是分布节点，而是分布对距离。

10.5.1 Distribution of nodes vs edges

在处理稀疏图矩阵时，有时我们需要超越一维分解的范畴。假设我们处理的图结构大致是随机的，例如任意两个顶点之间存在边的概率相同。同时假设我们拥有大量顶点和边，那么每个处理器将存储一定数量的顶点。由此可得出结论：任何一对处理器需要交换消息的概率相同，因此消息数量为 $O(P)$。（另一种可视化方式是将非零元素视为随机分布在矩阵中。）这并未给出一个可扩展的算法。

解决之道是将该稀疏矩阵视为稠密矩阵，并援引第 7.2.3 节的论述来决定矩阵的二维分布。（参见 [197] 中关于 BFS 问题的应用；他们以图论术语表述算法，但二维矩阵 - 向量乘法的结构仍清晰可辨。）二维乘积算法仅需在处理器行和列中进行集合通信，因此涉及的处理器数量为 $O(\sqrt{P})$。

10.5.2 超稀疏数据结构

在稀疏矩阵的二维分布中，可以想象某些进程最终得到的矩阵可能包含空行。从形式上看，若矩阵的非零元素数量（渐近地）小于其维度，则称该矩阵为超稀疏矩阵。

In this case, CRS format can be inefficient, and something called *double-compressed storage* can be used [24].

第 11 章

N 体问题

在第 4 章中，我们探讨了连续现象，例如一根受热杆在整个区间 $[0,1]$ 内某段时间内的行为。此外，也存在一些应用场景，您可能对有限数量的点感兴趣。其中一个应用就是研究粒子集合（可能是非常大的粒子，如行星或恒星）在诸如重力或电力等作用力下的运动。（也可能存在外力，但我们将忽略这些；同时假设没有碰撞发生，否则我们需要考虑最近邻相互作用。）这类问题被称为 N 体问题；入门介绍可参阅

http://www.scholarpedia.org/article/N-body_simulations_(gravitational)。

针对该问题的基本算法相当简单：

- 选择一个较小的时间间隔，
- 计算每个粒子上的作用力，给定所有粒子的位置，
- 移动每个粒子的位置，假设其上的作用力在该时间间隔内保持恒定。

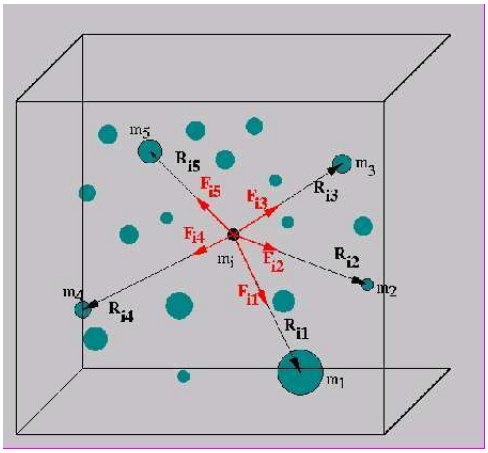


图 11.1：对单个粒子所受所有力求和。

For a small enough time interval this algorithm gives a reasonable approximation to the truth.

最后一步 —— 更新粒子位置 —— 简单且完全可并行：问题在于力的计算。以最朴素的方式，这一计算足够简单，甚至完全可并行：

对于每个粒子 i

对于每个粒子 j，令 r_{ij} 为 i 与 j 之间的向量；则 i 因 j 所受的力为 $f_{ij} = -r_{ij} \frac{m_i m_j}{|r_{ij}|}$（其中 m_i、m_j 为质量或电荷）且 $f_{ji} = -f_{ij}$。

该算法的主要缺陷在于其计算复杂度为二次方：对于 N 个粒子，操作次数为 $O(N^2)$。

习题 11.1. 如果我们有 N 个处理器，一次更新步骤的计算将耗时 $O(N)$。通信复杂度是多少？提示：是否存在可用的集体操作？

已有多种算法被发明出来将串行复杂度降低至 $O(N \text{ 对数 } N)$ 甚至 $O(N)$。正如预期，这些算法比朴素算法更难实现。我们将讨论一种流行的方法：巴尼斯 - 胡特算法 [8]，，其复杂度为 $O(N \text{ 对数 } N)$。

11.1 巴尼斯 - 胡特算法

导致复杂度降低的基本观察如下：若计算两个相邻粒子 i_1、 i_2 所受来自另两个相邻粒子 j_1、 j_2 的力时，可将 j_1、 j_2 聚合成单一粒子，并同时用于 i_1、 i_2 的计算。

接着，该算法采用空间递归划分策略，二维空间划分为象限，三维空间划分为八分体；参见图 11.2。

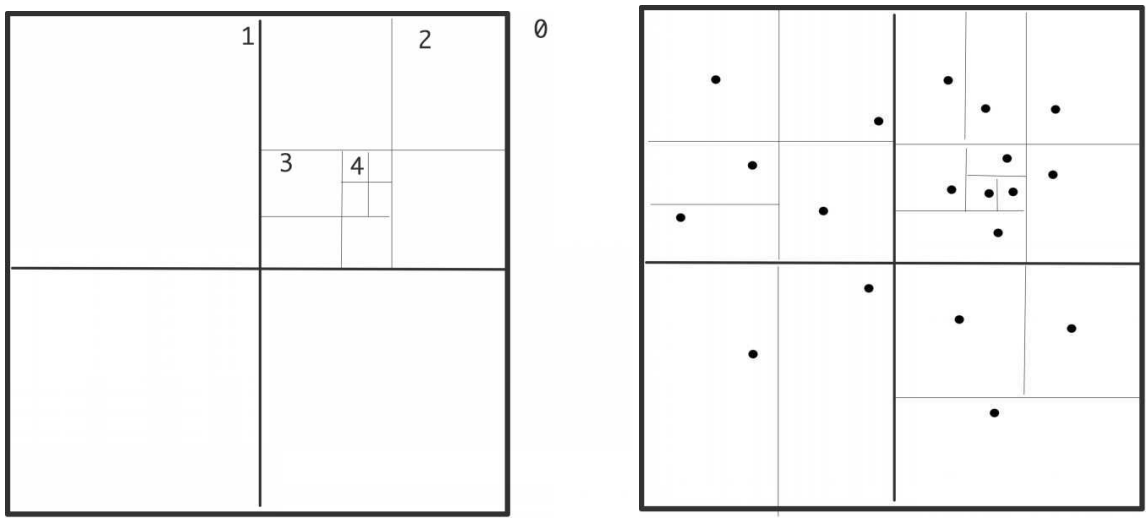


图 11.2：区域按象限递归细分并标注层级（左）；实际细分情况为每个方框含一个粒子（右）。

算法流程如下：首先计算所有层级上各单元的总质量及质心：

> 对于每个层级 ℓ，从细到粗：对于层级 ℓ 上的每个单元 c：通过考虑其子单元计算该单元 c 的总质量及质心，若该单元内无粒子则设其质量为零

T随后利用这些层级计算每个粒子的相互作用：

对于每个粒子 p：遍历顶层每个单元 c

若 c 与 p 距离足够远：使用其总质量与质心

否则考虑 c 的子单元

判断一个单元是否足够远的标准通常实现为其直径与距离之比是否足够小。这有时被称为 ‘单元开启准则’。通过这种方式，每个粒子与一系列同心环状排列的单元相互作用，每个下一环的宽度加倍；参见图 11.3。

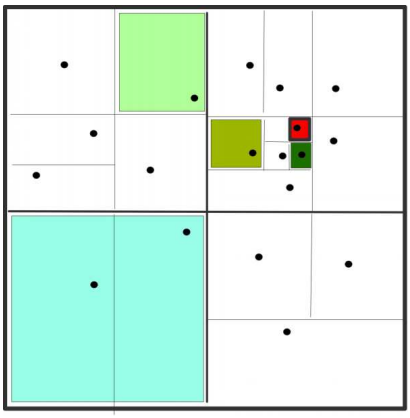


图 11.3：具有恒定距离 / 直径比例的方框。

若将单元组织为树形结构，该算法易于实现。在三维情况下，每个单元拥有八个子单元，因此被称为八叉树。

每次粒子移动后都需重新计算质心位置。更新计算可能比从头计算更高效。此外，粒子可能跨越单元边界，此时需更新数据结构。最坏情况下，粒子会移入原本为空的单元。

11.2 快速多极子方法

快速多极子方法（*FMM*）通过表达式计算
各点电势（而非 Barnes-Hut 算法计算的力）。FMM 利用了比箱体内粒子质量和质心更丰富的信息。这种更复杂的展开式精度更高，但计算成本也更大。作为补偿，FMM 使用固定箱体集合计算电势，而非随精度参数 θ 和质心位置变化的集合。

然而，从计算角度来看，FMM 与 Barnes-Hut 方法非常相似，因此我们将一并讨论它们的实现。

11.3 完全计算

尽管存在上述用于明智近似的方法，但也有努力完全计算 N^2相互作用；例如参见 *NBODY6* Sverre Aarseth 的代码；见 http://www.ast.cam.ac.uk/~sverre/web/pages/home.htm。此类代码使用高阶积分器和自适应时间步长。在 *Grape* 计算机 上存在快速实现；由于需要定期负载平衡，通常难以进行通用并行化。

11.4 实现

八叉树方法在高性能架构上带来了一些挑战。首先，问题本身具有不规则性；其次，这种不规则性会动态变化。第二个方面在分布式内存中尤为突出，需要负载重新平衡（参见 2.10 节）。本节我们专注于单一步骤中的力计算。

11.4.1 向量化

如图 11.2 所示问题的结构具有高度不规则性。这对于小规模的 *SSE/AVX* 指令集及大规模向量流水线处理器（相关解释见 2.3.1 节）的向量化都是难题。程序步骤如 ' 对某盒子的所有子节点执行操作 ' 将面临不规则长度问题，且数据存储方式也可能非连续。

该问题可通过细分网格来缓解 —— 即便这意味着会产生空盒子。若底层完全细分，总能保证有八个（三维空间）粒子可供操作。更高层级也可填充，但这会导致底层空盒子数量增加，因此需在增加计算量与提升效率之间权衡取舍。

11.4.2 共享内存实现

在顺序架构上执行时，该算法的复杂度为 $O(N\log\ N)$。显然，若将每个粒子转化为任务，该算法同样适用于共享内存环境。由于并非所有单元格都包含粒子，各任务的运行时间将有所不同。

11.4.3 分布式内存实现

上述 Barnes-Hut 算法的共享内存版本无法直接应用于分布式内存环境，因为原则上每个粒子都可能需访问总数据任意部分的信息。虽然可以通过哈希八叉树实现此类方案，但我们暂不深入探讨。

我们观察到数据访问模式比初看起来更具结构性。考虑粒子 p 及其交互的第 ℓ 层单元格。靠近 p 的粒子会与相同单元格交互，因此可通过分析第 ℓ 层单元格及其同层交互单元格来重构交互关系。

由此我们得到以下算法 [118]：计算质心转化为计算 $g_p^{(\ell)}$施加的由粒子 p 在层级 ℓ 上的作用力：

> 对于从最精细层之上到最粗糙层的每一层级 ℓ：对于层级 ℓ 上的每个单元 c，令 $g_c^{(\ell)}$为 c所有子单元 i 的 $g_i^{(\ell+1)}$组合。

由此我们计算作用在一个细胞上的力：

> 对于从次粗到最精细的层级 ℓ：对于层级 ℓ 上的每个单元 c：

令 $f_c^{(\ell)}$为以下两者之和：1. 作用于 c 的父节点 p 上的力 $f_p^{(\ell-1)}$，以及 2. 在层级 ℓ 上所有满足胞体开放准则的 i 对应的总和 $g_i^{(\ell)}$

可以看出，在每一层级上，每个胞体仅与该层级的少量相邻胞体交互。算法的前半部分仅利用胞体间的父子关系沿树结构向上遍历，这部分理应相对简单。

T算法的后半部分涉及更复杂的数据访问。第二项中的胞体 i 都
a与当前计算作用力的目标胞体 c 存在一定距离。在图论术语中，这些胞体可被
d表述为表亲关系：即 c 父节点的兄弟节点的子代。若收紧开放准则，我们则需要使用
s二代表亲： c 的祖父母兄弟姐妹的孙子女，以此类推。

练习 11.2. 论证这一力计算操作在结构上与稀疏矩阵 - 向量乘积有许多共同之处。

在共享内存情况下，我们已经注意到不同子树处理时间不同，但由于任务数量可能超过处理器核心数，这一差异会自然平衡。分布式内存环境下，我们无法随意分配任务至任意处理器，因此需谨慎分配负载。空间填充曲线（ SFCs ）在此可发挥良好效果（参见章节 2.10.5.2 ）。

11.4.4 完整方法的 1.5 维实现

可以通过分布粒子来实现完整 N^2 方法的直接并行化，让每个粒子计算来自其他所有粒子的作用力。

练习 11.3. 由于力的对称性，通过仅让粒子 p_i 与满足 $j > i$ 条件的粒子 p_j 相互作用，可节省一半计算量。此方案中，均匀的数据（如粒子）分布会导致非均匀的力相互作用计算负载。应如何分布粒子才能实现负载均衡?

假设我们不在意利用力的对称性带来的两倍增益因子，并采用均匀分布的粒子分配方案，该方案存在一个更严重的问题：其渐进复杂度无法扩展。

其通信成本为

- $O(P)$ 延迟（因为需要接收来自所有处理器的消息）
- $O(N)$ 带宽（因为需要接收所有粒子数据）

但若改为分配力计算任务而非粒子本身，则会得到不同的边界值，详见 [53] 及其所引文献。

当每个处理器计算边长为 $N/\sqrt{P}$ 的区块内的相互作用时，存在粒子数据冗余且需要汇总力。因此，现在的成本是

- $O(\log p)$ 广播与归约操作的延迟
- $O(N/\sqrt{P} \cdot \log\ P)$ 带宽，因为这是每个处理器对最终求和结果贡献的力数据量。

11.4.4.1 问题描述

我们将 N 体问题抽象描述如下：

- 假设存在一个向量 $c(\cdot)$，包含粒子电荷 / 质量及位置信息。
- 为计算位置更新，需要基于存储的元组 $C_{ij} = \langle c_i, c_j \rangle$ 计算两两相互作用 $F(\cdot,\cdot)$。
- 相互作用随后被汇总为一个力向量 $f(\cdot)$。

在并行集成模型（ IMP ）中，只要知晓相应的数据分布情况，算法便足以被描述；我们并不立即关注局部计算问题。

11.4.4.2 粒子分布

基于粒子分布的实现以分布为 c 的粒子向量作为起点，直接计算同分布下的力向量 $f(u)$。我们将计算过程描述为三个内核的序列，其中两个涉及数据移动，一个负责局部计算 [1]：

$$\begin{cases} \{\alpha : c(u)\} & \text{initial distribution for } c \text{ is } u \\ C(u,*) = c(u) \cdot_{\times} c(*) & \text{replicate } c \text{ on each processor} \\ \text{local computation of } F(u,*) \text{ from } C(u,*) & \\ f(u) = \sum_2 F(u,*) & \text{local reduction of partial forces} \\ \text{particle position update} & \end{cases} \tag{11.1}$$

最终内核是具有相同 α 和 β 分布的归约操作，因此不涉及数据移动。局部计算内核同样无需数据迁移。剩下的任务是从初始分布 $c(u)$ 中收集 $C(u,*)$。在缺乏 u 分布进一步信息的情况下，这相当于对 N 个元素进行全收集操作，成本为 $\alpha \log p + \beta N$。特别值得注意的是，通信成本并不随 p 的增加而降低。

W若采用基于分布的合适编程系统，方程组 (11.1) 可被转译 -
l编码化。

11.4.4.3 工作分配

粒子分布的实现采用了一维分布 $F(u,*)$ 以与粒子分布形式匹配的作用力。由于 F 是二维对象，也可以使用二维分布。我们现在将探讨这一选项，该方案先前已在 [53] 中描述；此处旨在展示如何在 IMP 框架中实现和分析该策略。我们不再将算法表述为分布式计算和复制数据，而是采用分布式临时方案，并通过子通信器上的集合操作推导复制方案。

对于 F 的二维分布，需要 $(N/b)\times(N/b)$ 个处理器，其中 b 是块大小参数。为便于说明，我们使用 $b = 1$，得到处理器数量 $P = N^2$。

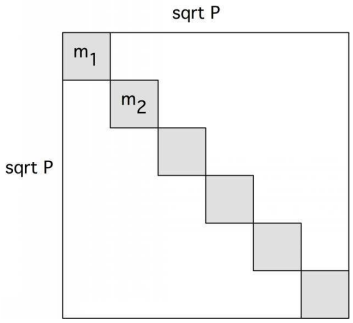
现在我们将进行必要的推理论证。图 11.4 给出了接近可实施形式的完整算法。为简化起见，我们采用标识分布 $I : p \mapsto \{p\}$。

1. 在完整的 IMP 理论中 [57] 前两个内核可以被合并。

$$
\begin{array}{ll}
\{\alpha : C(D : \ < I, I >)\} & \text{initial distribution on the processor diagonal} \\
C(I,I) = C(I,p) + C(q,I) & \text{row and column broadcast of the processor diagonal} \\
F(I,I) \leftarrow C(I,I) & \text{local interaction calculation} \\
f(D : \ < I, I >) = \sum_2 F(I, D : *) & \text{summation from row replication on the diagonal}
\end{array}
\tag{11.2}
$$

图 11.4：1.5 维 N 体问题的 IMP 实现。

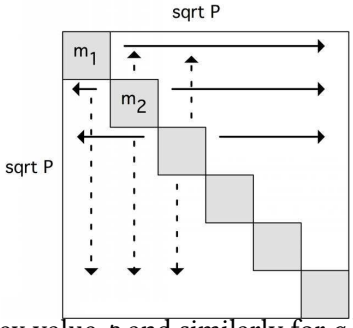
Initial store on the diagonal of the processor grid 我们首先考虑收集元组 $C_{ij} = \langle c_i \text{ 和 } c_j \rangle$。初始时，我们决定让 c 向量存储在处理器网格的对角线上；换言之，对于所有 p，处理器 $\langle p, p \rangle$ 包含 C_{pp}，而其他处理器的内容未定义。



为形式化表达，我们设 D 为对角线 $\{\langle p, p \rangle : \ p \in P\}$。利用部分定义分布的机制，初始的 α 分布即为

$$
C(D : \ < I, I >) \equiv D \ni \langle p, q \rangle \mapsto C(I(p), I(q)) = C_{pq}. \tag{11.3}
$$

Replication for force computation 局部力计算 $f_{pq} = f(C_{pq})$ 需要每个处理器 $\langle p, q \rangle$ 获取量 C_{pq}，因此我们需要 $C(I,I)$ 的 β 分布。本方案的关键在于认识到



$$
C_{pq} = C_{pp} + C_{qq}
$$

以便

$$
C(I,I) = C(I,p) + C(q,I)
$$

(其中 p 代表将每个处理器映射到输入的分布) dex value p and similarly for q.)

为了找到从 α 到 β 分布的转换，我们考虑表达式 $C(D : \langle I, I \rangle)$ 的转换。一般而言，任何集合 D 都可以写作从第一坐标到第二坐标值集合的映射：

$$
D \equiv p \mapsto D_p \qquad \text{where} \qquad D_p = \{q : \langle p, q \rangle \in D\}.
$$

在我们对角线的特定情况下，有

$$
D \equiv p \mapsto \{p\}.
$$

由此我们可写出

$$
C(D : \langle I, I \rangle) = C(I, D_p : \ I) = C(I, \{p\} : \ p). \tag{11.4}
$$

注意，方程（11.4）仍然是 α- 分布。 β- 分布是 $C(I,p)$，通过模式匹配可以看出，这是通过每行广播实现的，其成本为

$$\alpha \log \sqrt{p} + \beta N/\sqrt{P}.$$

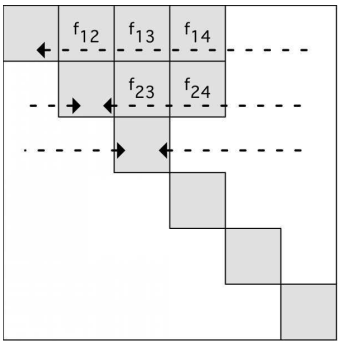
同理， $C(q,I)$ 是通过列广播从 α- 分布中得到的。我们得出结论，此变体的通信成本确实会随处理器数量的增加而按比例降低。

局部相互作用计算 $F(I,I) \leftarrow C(I,I)$ 的计算具有相同的 α 和 β 分布，因此是平凡并行的。

力的求和 我们需要将力向量 $f(\cdot)$ 扩展为二维处理器网格上的 $f(\cdot,\cdot)$。但为了与初始粒子在对角线上的分布保持一致，我们仅在对角线上求和。指令

$$f(D:\ <I,I>) = \sum_{2} F(I,D:*)$$

具有 β- 分布 $I,D:*$，这是通过行内聚集从 α- 分布 I,I 形成的。



第 12 章

蒙特卡洛方法

蒙特卡洛模拟是一个广义术语，指代利用随机数和统计抽样来解决问题的各类方法，而非精确建模。由于这种抽样的本质特性，结果会存在一定不确定性，但统计学的 ‘ 大数定律 ’ 将确保随着样本数量的增加，不确定性会逐渐降低。

统计抽样的一个重要工具是随机数生成器。关于随机数生成，参见附录 19 。

12.1 动机

让我们从一个简单例子开始：测量一个区域的面积，例如 π 是内接于边长为 2 的正方形中的圆的面积。若在正方形内随机选取一个点，其落入圆内的概率为 $\pi/4$，因此可以通过获取大量随机点 (x, y) 并观察其中长度 $\sqrt{x^2 + y^2}$ 小于 1 的比例来估算这个比值。

你甚至可以将此作为一个物理实验来进行：假设你家后院有一个不规则形状的池塘，而院子本身是已知尺寸的矩形。如果你现在向院子里随机投掷小石子，使它们落在任意位置的概率均等，那么落入池塘的石子与落在池塘外的石子数量之比，就等于两者面积之比。

更少幻想色彩且更数学化的表述是，我们需要将 “ 落在待测形状内部或外部 ” 这一概念形式化。因此，设 $\Omega \in [0,1]^2$ 为该形状，并用函数 $f(x)$ 描述 Ω 的边界，即

$$\begin{cases} f(\bar{x}) < 0 & x \notin \Omega \\ f(\bar{x}) > 0 & x \in \Omega \end{cases}$$

现在取随机点 x_0、x_1、$x_2 \in [0,1]^2$，则我们可以通过统计 $f(x_i)$ 为正或负的频率来估算 Ω 的面积。

W我们可以将此概念推广至积分运算。函数在区间 (a, b) 上的平均值定义为

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x) dx$$

另一方面，我们也可以将平均值估计为

$$\langle f \rangle \approx \frac{1}{N} \sum_{i=1}^{n} f(x_i)$$

如果点 x_i 分布合理且函数 f 不过于奇异。这引导我们得出

$$\int_a^b f(x)dx \approx (b-a) \frac{1}{N} \sum_{i=1}^{n} f(x_i)$$

我们不会深入探讨的统计理论表明，积分中的不确定性 σ_I 与标准差 σ_f 的关系为

$$\sigma_I \sim \frac{1}{\sqrt{N}} \sigma_f$$

对于正态分布而言。

12.1.1 吸引力何在?

到目前为止，蒙特卡洛积分看起来与通过黎曼和进行的经典积分并无太大差异。差异出现在我们进入更高维度时。在那种情况下，经典积分需要在每个维度上取 N 个点，导致在 d 维空间中需要 N^d 个点。而蒙特卡洛方法则是从 d 维空间中随机选取点，所需点数要少得多。

从计算角度看，蒙特卡洛方法之所以吸引人，是因为所有函数评估都可以并行执行。

其背后的统计规律如下：如果对标准差为 σ 的量进行 N 次独立观测，则均值的标准差为 $\sigma/\sqrt{N}$。这意味着更多观测会带来更高精度；蒙特卡洛方法的有趣之处在于，这种精度提升与原问题的维度无关。

蒙特卡洛技术天然适合模拟具有统计本质的现象，如放射性衰变或布朗运动。蒙特卡洛模拟的其他应用领域已超出科学计算范畴。例如，布莱克 - 斯科尔斯模型对股票期权定价 [15] 就采用了蒙特卡洛模拟。

一些你之前见过的问题，例如求解线性方程组，可以用蒙特卡洛技术来处理。然而，这并不是典型的应用场景。下面我们将讨论两种应用场景，在这些场景中，精确方法需要耗费大量时间计算，而统计抽样可以快速给出相当准确的答案。

12.2 示例

12.2.1 伊辛模型的蒙特卡洛模拟

伊辛模型（入门介绍可参阅 [34] ）最初是为模拟铁磁性而提出的。磁性源于原子自旋方向的排列：假设自旋只能 ‘向上’ 或 ‘向下’，当更多原子自旋向上而非向下时（或反之），材料就表现出磁性。这些原子被认为处于一种称为 ‘晶格’ 的结构中。

现在想象加热材料会使原子变得松散。若施加外部磁场，原子将开始沿磁场方向排列；撤去磁场后磁性会再次消失。但在特定临界温度以下，材料会保持其磁性。我们将通过蒙特卡洛模拟来寻找保持稳定的配置。

假设晶格 Λ 包含 N 个原子，我们将原子配置表示为 $\sigma = (\sigma_1, \dots, \sigma_N)$，其中每个 $\sigma_i = \pm 1$。晶格的能量建模为

$$H = H(\sigma) = -J \sum_i \sigma_i - E \sum_{ij} \sigma_i \sigma_j.$$

第一项模拟了单个自旋 σ_i 与强度为 J 的外场相互作用。第二项对最近邻原子对求和，模拟原子对的排列：若原子自旋方向相同，乘积 $\sigma_i \sigma_j$ 为正；若相反则为负。

在统计力学中，某个配置的概率为

$$P(\sigma) = \exp(-H(\sigma))/Z$$

其中 ‘配分函数’ Z 定义为

$$Z = \sum_\sigma \exp(H(\sigma))$$

此处求和遍历所有 2^N 配置。

若某配置的能量在微小扰动下不降低，则该配置稳定。为探究此现象，我们遍历晶格，测试改变原子自旋是否会降低能量。引入随机因素以避免人为解（此即 *Metropolis* 算法 [147] ）。

> **for** 固定迭代次数 **dofor** 每个原子 i **do** 计算 ΔE 改变 σ_i **符号引起的能量变化 if** $\Delta E <$ 或 $\exp(-\Delta E)$ 大于某随机数 **then** 接受该改变

该算法可并行化处理，若我们注意到其与稀疏矩阵 - 向量乘积结构的相似性。在该算法中，同样通过结合少数最近邻的输入来计算局部量。这意味着我们可以对晶格进行分区，并在每个处理器收集一个 *ghostregion* 后计算局部更新。

让每个处理器迭代晶格中的局部点对应于晶格的特定全局排序；为了使并行计算等效于顺序计算，我们还需要一个并行随机数生成器（章节 19.3 ）。

第 13 章

机器学习

机器学习（ ML ）是一系列解决诸如图像识别等 “ 智能 ” 问题的技术统称。抽象来看，这类问题是从一个特征向量空间（如图像中的像素值）到结果向量空间的映射。以字母图像识别为例，最终空间可能是 26 维的，第二组件中的最大值将指示识别出的是字母 ‘B’ 。

机器学习技术的核心特征在于：这种映射由一组（通常数量庞大的）内部参数描述，且这些参数会逐步优化。其学习性体现在：通过将输入数据与基于当前参数的预测输出及预期输出进行对比，从而完成参数优化。

13.1 神经网络

当前最流行的机器学习形式是深度学习（ DL ），或称神经网络。神经网络（或称深度学习网络）是一种给定（通常为多维）输入点计算数值输出的函数。假设输出归一化至区间 $[0,1]$，，我们便可通过设置输出阈值将神经网络用作分类工具。

为何称为 ‘ 神经 ’？

在生物体内，神经元是一种会 ‘ 激活 ’ 的细胞，即当接收到特定输入时会释放电压脉冲。在机器学习中，我们将其抽象为感知器：一种根据特定输入输出值的函数。具体而言，输出值通常是输入的线性函数，结果被限制在 $(0,1)$ 范围内。

13.1.1 单数据点

最简单的形式中，我们有一个由特征向量 x 描述的输入和一个标量输出 y。我们可以通过使用相同尺寸的权重向量和一个标量偏置 b，将 y 计算为 x 的线性函数：

$$y = \bar{w}\bar{x} + b.$$

13.1.2 激活函数

为了实现某种尺度不变性，我们引入一个称为激活函数的函数 σ，它将 $\mathbb{R} \to (0,1)$ 映射，并实际计算标量输出 y 为：

$$\mathbb{R} \ni y = \sigma(\bar{w}^t \bar{x} + b). \tag{13.1}$$

一个流行的 *sigmoid* 函数选择是

$$\sigma(z) = \frac{1}{1+e^{-z}}.$$

这具有一个有趣的属性，即

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

因此计算函数值和导数并不比仅计算函数值昂贵多少。

对于向量值输出，我们逐点应用 sigmoid 函数：

```
// funcs.cpptemplate <typename V>void sigmoid_io(const V &m, V &a) {
a.vals.assign(m.vals.begin(),m.vals.end());for (int i = 0; i < m.r * m.c; i++) {
// a.vals[i]*=(a.vals[i]>0); // values will be 0 if negative, and equal to themselves
if positivea.vals[i] = 1 / (1 + exp(-a.vals[i]));}}
```

其他激活函数包括 $y = \tanh(x)$ 或 'ReLU'（线性整流单元）

$$f(x) = \max(0, x).$$

在其他场景（如深度学习网络的最终层），*softmax* 函数可能更为适用。

13.1.3 多维输出

仅靠由 w, b 定义的单一层很难实现神经网络的全部需求。通常我们会将某一层的输出作为下一层的输入。这意味着我们不再计算标量 y，而是计算多维向量 y。

现在，我们为输出的每个组件都配备了权重和偏置项，因此

$$\mathbb{R}^n \ni \bar{y} = \sigma(W\bar{x} + \bar{b})$$

其中 W 现在是一个矩阵。

几点观察结果：

- 如前所述，输出向量通常比输入向量的组件更少，因此该矩阵不是方阵，尤其不可逆。

- Sigmoid 函数使得整体映射呈现非线性。
- 神经网络通常具有多个层，每一层都是形如 $x \to y$ 的映射。

13.1.4 卷积

上述关于权重应用的讨论将输入视为一组无额外结构的特征。然而，在诸如图像识别等应用中，输入向量是一幅图像，存在需要被识别的结构。将输入向量线性化后，水平方向上相邻的像素在输入向量中会彼此靠近，但垂直方向上则不然。

因此，我们致力于寻找一个能反映这种局部性的权重矩阵。为此，我们引入核函数：一种在图像不同位置应用的微小 ‘模板’。（关于偏微分方程背景下模板的讨论，请参阅章节 4.2.4。）此类核通常是一个小的方阵，其应用方式是通过计算模板值与图像值的内积。（这是对信号处理中“卷积”术语的不精确使用。）

示例：https://aishack.in/tutorials/image-convolution-examples/。

13.2 深度学习网络

We will now present a full neural network. This presentation follows [103].

使用一个包含 $L \geq 1$ 层的网络，其中第 $\ell = 1$ 层为输入层，第 $\ell = L$ 层为输出层。

对于 $\ell = 1$、 ...、 L，第 ℓ 层计算

$$\begin{aligned} z^{(\ell)} &= W^{(\ell)} a^{(\ell)} + b^{(\ell)} \\ y^{(\ell)} &= \sigma\left(y^{(\ell)}\right) \end{aligned} \tag{13.2}$$

其中 $a^{(1)}$ 为输入， $z^{(L+1)}$ 为最终输出。

我们将其简洁地表示为

$$y^{(L)} = N_{\{W^{(\ell)}\}_\ell, \{b^{(\ell)}\}_\ell}\left(a^{(1)}\right) \tag{13.3}$$

通常我们会省略网络对 $W^{(\ell)}$、 $b^{(\ell)}$ 集合的依赖关系。

```
// net.cpp
void Net::feedForward(const VectorBatch &input) {
    this->layers.front().forward(input); // Forwarding the input

    for (unsigned i = 1; i < layers.size(); i++) {
        this->layers.at(i).forward(this->layers.at(i - 1).activated_batch);
    }
}
```

```
// layer.cpp
void Layer::forward(const VectorBatch &prevVals) {

        VectorBatch output( prevVals.r, weights.c, 0 );
    prevVals.v2mp( weights, output );
        output.addh(biases); // Add the bias
    activated_batch = output;
    apply_activation<VectorBatch>.at(activation)(output, activated_batch);
}
```

13.2.1 分类

在上述描述中，输入 x 和输出 y 均为向量值。也存在需要不同类型输出的情况。例如，假设我们希望表征数字的位图图像；此时输出应为整数 $0 \cdots 9$。

我们通过让输出 y 属于 $\mathbb{R}^{10}$来适应这一点，并规定当 y_5 显著大于其他输出组件时，网络即识别出数字 5。通过这种方式，我们仍保持整个过程的实数值特性。

13.2.2 误差最小化

通常我们拥有已知输出数据点 $\{x_i\}_{i=1,N}$，希望网络能尽可能准确地预测并重现这种映射关系。从形式上看，我们力求最小化成本或误差：

$$C = \frac{1}{N} L(N(x_i), y_i)$$

o在所有选择 $\{W\}, \{b\}$ 中。（通常我们不会明确说明该成本是所有 $W^{[\ell]}$ 权重
m矩阵和 $b^{[\ell]}$ 偏置项的函数。）

```
float Net::calculateLoss(Dataset &testSplit) {
    testSplit.stack();
    feedForward(testSplit.dataBatch);
    const VectorBatch &result = output_mat();

    float loss = 0.0;
    for (int vec=0; vec<result.batch_size(); vec++) { // iterate over all items
      const auto& one_result = result.get_vector(vec);
      const auto& one_label  = testSplit.labelBatch.get_vector(vec);
      assert( one_result.size()==one_label.size() );
      for (int i=0; i<one_result.size(); i++) // Calculate loss of result
        loss += lossFunction( one_label[i], one_result[i] );
    }
    loss = -loss / (float) result.batch_size();

    return loss;
}
```

最小化成本意味着选择权重 $\{W^{(\ell)}\}_\ell$ 和偏置 $\{b^{(\ell)}\}_\ell$，使得对于每个 x:

$$\left[\{W^{(\ell)}\}_\ell, \{b^{(\ell)}\}_\ell\right] = \operatorname*{argmin}_{\{W\},\{b\}} L(N_{\{W\},\{b\}}(x), y) \tag{13.4}$$

其中 $L(N(x), y)$ 是一个损失函数，用于描述计算输出 $N(x)$ 与预期输出 y之间的距离。

We find this minimum using *gradient descent*:

$$w \leftarrow w + \Delta w, \qquad b \leftarrow b + \Delta b$$

其中

$$\Delta W = \frac{\partial L}{\partial W_{ij}^{(\ell)}}$$

这是一个复杂的表达式，我们现在将直接给出而不进行推导。

13.2.3 系数计算

我们关注的是成本相对于各种权重、偏置和计算量的偏导数。为此，引入一个简写符号会很方便:

$$\delta_i^{[\ell]} = \frac{\partial C}{\partial z_i^{[\ell]}} \qquad \text{for } 1 \le i \le n_\ell \text{ and } 1 \le \ell < L. \tag{13.5}$$

现在应用链式法则（完整推导参见上述引用的论文），我们得到，使用 $x \circ y$ 表示点积（或 *Hadamard*）向量 - 向量积 $\{x_i y_i\}$:

- 在最后一层:
$$\delta^{[L-1]} = \sigma'\left(z^{[L-1]}\right) \circ \left(a^{[L]} - y\right)$$
- recursively for the earlier levels:
$$\delta^{[\ell]} = \sigma'\left(z^{[\ell]}\right) \circ \left(W^{[\ell+1]^t} \delta^{[\ell+1]}\right)$$
- 对偏置的敏感度:
$$\frac{\partial C}{\partial b_i^{[\ell]}} = \delta_i^{[\ell]}$$
- sensitivity wrt the weights:
$$\frac{\partial C}{\partial w_{ik}^{[\ell]}} = \delta_i^{[\ell]} a_k^{[\ell-1]}$$

Using the special form

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

得出

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

输入层 $\ell = 1$ 起始于：$a^{(\ell)} = x$ 网络输入 $n_{ell} \times b$ 对于层 $\ell = 1$、 …、 L

$a^{(\ell)}$ $n_\ell \times b$ 层输入权重

$W^{(\ell)}$

$n_{\ell+1} \times n_\ell$ 偏置

$b^{(\ell)}$

$n_{\ell+1} \times b$ 偏置乘积

$z^{(\ell)} \leftarrow W^{(\ell)} a^{(\ell)} + b^{(\ell)}$

$n_{\ell+1} \times b$ 激活乘积 最终输

$a^{(\ell+1)} = y^{(\ell)} \leftarrow \sigma\left(z^{(\ell)}\right)$ $n_{\ell+1} \times b$

出：$y = a^{(L+1)} = z^{(L)}$ $n_{L+1} \times b$ 对于层 $\ell = L$、

$L-1$、 …， $1 D^{(\ell+1)} \leftarrow$ 对角矩阵 (13.6)

$\left(\sigma'\left(z^{(\ell)}\right)\right)$ $n_{\ell+1} \times n_{\ell+1}$

$\delta^{(\ell)} \leftarrow \begin{cases} D^{L+1}\left(a^{L+1} - y\right) \\ D^{(\ell+1)} W^{(\ell+1)^t} \delta^{(\ell+1)} & \ell < L \end{cases}$ $n_{\ell+1} \times b$ $\Delta W^{(\ell)} \leftarrow \delta^{(\ell)} a^{(\ell-1)^t}$

权重更新 $n_{\ell+1} \times n_\ell$

$w^{(\ell)} \leftarrow w^{(\ell)} - \Delta W^{(\ell)}$

$\Delta b^{(\ell)} \equiv \delta^{(\ell)}$ 偏置更新 $n_{\ell+1} \times b$

图 13.1：深度学习的前向 / 反向传播过程。

13.2.4 算法

我们现在完整展示图 13.1 中的算法。我们的网络包含层 $\ell = 1$、 …、 L，其中参数 n_ℓ 表示第 ℓ 层的输入维度。

第 1 层输入为 x，第 L 层输出为 y。为适应小批量训练，我们用 x、 y 表示一组大小为 b 的输入 / 输出，因此其维度分别为 $n_1 \times b$ 和 $n_{L+1} \times b$ 。

13.3 计算特性

本节将探讨深度学习的高性能计算特性。从标量运算角度，我们论证矩阵 - 矩阵乘积存在的必要性，该运算能以极高效率执行。

我们还将讨论并行性，重点关注

- 数据并行，其基本策略是分割数据集；
- 模型并行，其基本策略是分割模型参数；以及
- 流水线技术，其中指令可以按照不同于朴素模型中的顺序执行。

13.3.1 权重矩阵乘积

无论是在应用网络（前向传播）还是学习过程（反向传播）中，我们都会执行矩阵乘以向量积的权重矩阵运算。该操作没有太多缓存复用，因此会

不具备高性能；章节 6.10。

另一方面，若我们将多个数据点捆绑处理 —— 这有时被称为 *mini-batch* —— 并对其进行联合运算，基础运算便转化为矩阵间乘积，它能实现更高的性能；参见章节 1.6.1.2。

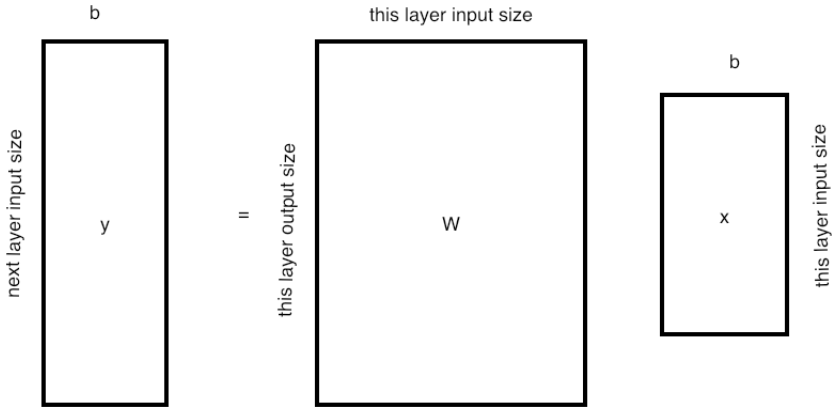


图 13.2：单层中数组的形状。

如图 13.2 所示：

- 输入批次与输出批次包含相同数量的向量；
- 权重矩阵 W 的行数等于输出尺寸，列数等于输入尺寸。

该 *gemm* 核心（参见章节 1.6.1 与 7.4.1 ）的重要性促使人们为其开发专用硬件。

另一种方法是采用特殊形式的权重矩阵。在 [136] 中研究了托普利茨矩阵近似。这种方法既能节省存储空间，又能通过 *FFT* 实现乘积运算。

13.3.2 权重矩阵乘积中的并行性

现在我们可以探讨 $N(x)$ 的高效计算。前文已指出矩阵乘法是重要核心，此外还可利用并行处理。图 13.3 展示了两种并行化策略。

第一种策略中，批次被划分给不同进程处理（即多个进程同时处理独立批次），我们称之为数据并行。

练习 13.1. 考虑以下共享内存场景中的代码：对于 $b = 1, \ldots,$ 批处理大小为 $i = 1, \ldots,$ 输出尺寸 $y_{i,b} \leftarrow \sum_j W_{i,j} \cdot x_{j,b}$

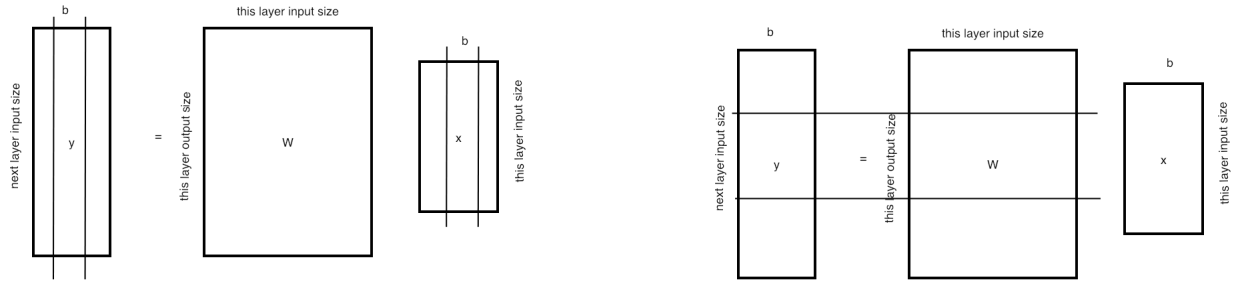


图 13.3：并行层评估的分区策略。

假设每个线程计算 $1, \ldots,$ batchsize 范围的一部分。将其翻译为您喜欢的编程语言。您是按行还是按列存储输入 / 输出向量？为什么？这两种选择各有什么影响？

练习 13.2. 现在考虑分布式内存环境中的数据并行性，每个进程处理批次的一个切片（块列）。您是否看到了一个直接的问题？

还有第二种策略，称为模型并行，其中模型参数（即权重和偏置）是分布式的。如图 13.3 所示，这直接意味着该层的输出向量是分布式计算的。

练习 13.3. 在分布式内存环境中概述这种第二种分区的算法。

策略的选择取决于模型规模是否庞大、权重矩阵是否需要拆分，或输入数据量是否巨大。当然，也可结合两者，实现模型与批处理数据的双重分布式处理。

13.3.3 权重更新

权重更新的计算

$$\Delta W^{(\ell)} \leftarrow \delta^{(\ell)} a^{(\ell-1)^t}$$

是一个秩为 b 的外积运算。该运算接收两个向量，并由此生成一个低秩矩阵。

习题 13.4：讨论该操作中（潜在的）数据复用率，分析其与 n_ℓ 和 b 的关系。为简化起见，假设 $n_{\ell+1} \approx n_\ell$。

习题 13.5：分析图 13.3 所示两种分区策略中涉及的数据传输结构。

除了这些方面，当我们考虑并行处理小批量数据时，这一操作变得更加有趣。在这种情况下，每个批次独立计算一个更新，我们需要对它们进行平均。假设每个进程都计算一个完整的 ΔW，这就变成了一个全规约操作。这种 ‘高性能计算技术’ 的应用被开发成了 *Horovod* 软件 [75, 171, 108]。在一个例子中，展示了涉及 40 个 GPU 的配置下显著的加速效果。

另一种选择是延迟更新，或异步执行更新。

练习 13.6. 讨论在 MPI 中实现延迟或异步更新的方法。

13.3.4 流水线处理

最后一种并行化方式是通过在层间应用流水线技术实现。简述这种方法如何提升训练阶段的效率。

13.3.5 卷积运算

应用卷积运算等价于乘以一个 *Toeplitz* 矩阵。其计算复杂度低于完全通用的矩阵间乘法。

13.3.6 稀疏矩阵

权重矩阵可通过忽略微小项实现稀疏化，这使得稀疏矩阵乘稠密矩阵成为主导运算 [70]。

13.3.7 硬件支持

综上所述，核心的重要性。将常规 CPU 专门用于此目的
是对硅片与电力的极大浪费。至少应使用 GPU 来
a一种高能效解决方案。

然而，通过使用专用硬件还能实现更高效率。以下是概述：
https://blog.inten.to/hardware-for-deep-learning-part-4-asic-96a542fe6a81 从某种意义上说，这些专用处理器是脉动阵列的再生。

13.3.8 降低精度

参见章节 3.8.4.2。

13.4 其他内容

通用逼近定理

设 $\varphi(\cdot)$ 为一个非常值、有界且单调递增的连续函数。令 I_m 表示 m 维单位超立方体 $[0,1]^m$。在 I_m 上的连续函数空间记作 $C(I_m)$。那么，对于任意函数 $f \in C(I_m)$ 和 $\varepsilon > 0$，存在一个整数 N、实数常量 v_i、 $b_i \in \mathbb{R}$ 以及实向量 $w_i \in \mathbb{R}^m$（其中 $i = 1$、 $\cdots$、 N），使得我们可以定义：

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

作为函数 f 的近似实现，其中 f 独立于 φ；即，

$$|F(x) - f(x)| < \varepsilon$$

对所有 $x \in I_m$ 成立。换言之，形如 $F(x)$ 的函数在 $C(I_m$ 中稠密).

神经网络能近似乘法运算吗?

https://stats.stackexchange.com /questions/217703/can-deep-neural-network-approximate

传统神经网络由线性映射和 Lipschitz 激活函数组成。作为 Lipschitz 连续函数的复合，神经网络本身也是 Lipschitz 连续的，但乘法运算并不具备 Lipschitz 连续性。这意味着当 x 或 y 中任一变量过大时，神经网络无法近似乘法运算。

第三部分

APPENDICES

本课程无需高深的数学基础，主要假定读者已掌握线性代数基础知识：理解矩阵与向量的概念及其最常见运算。

后续附录将涵盖一些较为冷门的理论内容，这些内容已从前文各章主线叙述中剥离出来。

第 14 章

线性代数

本课程假定您已了解矩阵和向量的定义、基本算法（如乘法运算）以及矩阵可逆性等性质。本附录将介绍一些通常不属于线性代数初级课程的概念与定理。

14.1 范数

范数是将绝对值概念推广到向量、矩阵等多维对象的方式。范数的定义方法众多，且不同范数之间存在关联理论。此处我们仅介绍基本概念。

14.1.1 向量范数

范数是向量空间 V 上满足以下性质的任意函数 $n(\cdot)$：

- $n(x) \geq 0$ 对所有 $x \in V$ 成立，且仅当 $x = 0$ 时 $n(x) = 0$，
- $n(\lambda x) = |\lambda| n(x)$ 对所有 $x \in V$ 及 $\lambda \in \mathbb{R}$ 成立。
- $n(x + y) \leq n(x) + n(y)$

对于任意 $p \geq 1$，以下定义了一个向量范数：

$$|x|_p = \sqrt[p]{\sum_i |x_i|^p}.$$

常见的范数包括 1- 范数、 2- 范数和无穷范数：

- 1- 范数 $\|\cdot\|_1$ 是绝对值的总和：

$$\|x\|_1 = \sum |x_i|.$$

- 2- 范数 $\|\cdot\|_2$ 是平方和的平方根：

$$\|x\|_2 = \sqrt{\sum x^2}.$$

- 无穷范数 $\|\cdot\|_\infty$ 定义为 $\lim_{p\to\infty} \|\cdot\|_p$，不难看出其等价于

$$\|x\|_\infty = \text{最大值}_i \ |x_i|.$$

14.1.2 矩阵范数

通过将大小为 n 的矩阵视为长度为 n^2的向量，我们可以定义 Frobenius 矩阵范数：

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}.$$

不过，我们主要关注关联矩阵范数：

$$\|A\|_p = \sup_{\|x\|_p=1} \|Ax\|_p = \sup_x \frac{\|Ax\|_p}{\|x\|_p}.$$

根据其定义，可以得出

$$\|Ax\| \leq \|A\|\|x\|$$

对于关联范数成立。

以下内容易于推导得出：

- $\|A\|_1 = \max_j \sum_i |a_{ij}|$ 表示最大列绝对值和；
- $\|A\|_\infty = \max_i \sum_j |a_{ij}|$ 表示最大行绝对值和。

通过观察 $\|A\|_2 = \sup_{\|x\|_2=1} x^t A^t Ax$不难推导出， $\|A\|_2$ 是 A 的最大奇异值，即 A^tA最大特征值的平方根，而 A 的最大特征值为 A （当 A 对称时）。

矩阵条件数 的定义为

$$\kappa(A) = \|A\|\|A^{-1}\|.$$

在对称矩阵情况下使用 2- 范数时，该值为最大与最小特征值之比；一般情况下，则为最大与最小奇异值之比。

14.2 格拉姆 - 施密特正交化

格拉姆 - 施密特（ GS ）算法通过归纳法对一系列向量进行正交化处理。该算法可用于将向量空间的任意基转换为正交基；亦可视为将矩阵 A 转化为具有正交列的矩阵。若 Q 的列已正交，则 Q^tQ 为对角矩阵，这一属性通常极为便利。

GS 算法的基本原理可通过两个向量 u 、 v 来演示。假设我们需要一个向量 v'，使得 u 、 v 张成的空间与 u 、 v' 相同，但 $v' \perp u$。为此我们令

$$v' \leftarrow v - \frac{u^t v}{u^t u} u.$$

显然，这样构造的向量满足需求条件。

假设我们有一组待正交化的向量 u_1 、 … 、 u_n。通过多次应用上述转换即可实现正交化：

For $i = 1, \dots, n$:
    For $j = 1, \dots i-1$:
        let $c_{ji} = u_j^t u_i / u_j^t u_j$
    For $i = 1, \dots, n$:
        update $u_i \leftarrow u_i - u_j c_{ji}$

通常算法中的向量 v 会被归一化；这需要在算法中添加一行

$u_i \leftarrow u_i / \|u_i\|$

采用这种归一化的 GS 正交化方法，当应用于矩阵的列时，也被称为 QR 分解。

习题 14.1. 假设我们将 GS 算法应用于矩形矩阵 A 的列，得到矩阵 Q。证明存在一个上三角矩阵 R 使得 $A = QR$。（提示：观察上述 c_{ji} 系数。）如果我们在算法中对正交向量进行归一化处理，Q 还具有 $Q^tQ = I$的附加属性。请一并证明这一点。

上述 GS 算法在精确算术下能计算出预期结果，但计算机实现时若 v 与某个 u_i 的夹角过小，则可能产生显著误差。此时，改进的 Gram-Schmidt （ MGS ）算法表现更优：

For $i = 1, \dots, n$:
    For $j = 1, \dots i-1$:
        let $c_{ji} = u_j^t u_i / u_j^t u_j$
        update $u_i \leftarrow u_i - u_j c_{ji}$

完整的 GS 算法也被称为经典 Gram-Schmidt （ CGS ）。

为了说明这两种方法之间的差异，考虑矩阵

$$A = \begin{pmatrix} 1 & 1 & 1 \\ \epsilon & 0 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon \end{pmatrix}$$

其中 ϵ 的数量级为机器精度，因此在机器算术中 $1+\epsilon^2 = 1$。CGS 方法按如下步骤进行：

- 第一列在机器算术中的长度为 1，因此

$$q_1 = a_1 = \begin{pmatrix} 1 \\ \epsilon \\ 0 \\ 0 \end{pmatrix}.$$

- 第二列通过 $v \leftarrow a_2 - 1 \cdot q_1$ 进行正交化处理，得到

$$v = \begin{pmatrix} 0 \\ -\epsilon \\ \epsilon \\ 0 \end{pmatrix}, \quad \text{normalized:} \quad q_2 = \begin{pmatrix} 0 \\ -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{pmatrix}$$

- 第三列被正交化为 $v \leftarrow a_3 - c_1 q_1 - c_2 q_2$，其中

$$\begin{cases} c_1 = q_1^t a_3 = 1 \\ c_2 = q_2^t a_3 = 0 \end{cases} \quad \Rightarrow v = \begin{pmatrix} 0 \\ -\epsilon \\ 0 \\ \epsilon \end{pmatrix}; \quad \text{normalized:} \quad q_3 = \begin{pmatrix} 0 \\ \sqrt{2} \\ 2 \\ 0 \\ \sqrt{2} \\ 2 \end{pmatrix}$$

容易看出 q_2 和 q_3 完全不正交。相比之下，MGS 方法在最后一步有所不同：

- 如前所述，$q_1^t a_3 = 1$，因此

$$v \leftarrow a_3 - q_1 = \begin{pmatrix} 0 \\ -\epsilon \\ \epsilon \\ 0 \end{pmatrix}.$$

接着，$q_2^t v = \frac{\sqrt{2}}{2}\epsilon$（注意之前 $q_2^t a_3 = 0$），所以第二次更新得到

$$v \leftarrow v - \frac{\sqrt{2}}{2}\epsilon q_2 = \begin{pmatrix} 0 \\ \frac{\epsilon}{2} \\ -\frac{\epsilon}{2} \\ \epsilon \end{pmatrix}, \quad \text{normalized:} \quad \begin{pmatrix} 0 \\ \frac{\sqrt{6}}{6} \\ -\frac{\sqrt{6}}{6} \\ 2\frac{\sqrt{6}}{6} \end{pmatrix}$$

现在所有 $q_i^t q_j$ 的量级均为 ϵ 对于 $i \neq j$。

14.3 幂方法

向量序列

$$x_i = A x_{i-1},$$

其中 x_0 是某个起始向量，被称为幂方法，因为它计算了连续矩阵幂与向量的乘积：

$$x_i = A^i x_0.$$

在某些情况下，x_i 向量之间的关系较为简单。例如，如果 x_0 是 A 的特征向量，则对于某个标量 λ，我们有

$$Ax_0 = \lambda x_0 \qquad \text{and} \qquad x_i = \lambda^i x_0.$$

然而，对于任意向量 x_0，序列 $\{x_i\}_i$ 很可能由独立向量组成。在某种程度上。

练习 14.2. 设 A 和 x 分别为 $n\times n$ 矩阵与维度 n 向量

$$A=\begin{pmatrix}1&1&&&\\&1&1&&\\&&\ddots&\ddots&\\&&&1&1\\&&&&1\end{pmatrix},\qquad x=(0,\dots,0,1)^t.$$

证明序列 $[x,Ax,\dots,A^ix]$ 是 $i<n$ 的独立集。为何该结论对 $i\geq n$ 不再成立?

现在考虑矩阵 B:

$$B=\left(\begin{array}{ccccc|ccccc}1&1&&&&&&&&\\&\ddots&\ddots&&&&&&&\\&&1&1&&&&&&\\&&&1&&&&&&\\&&&&&1&1&&&\\&&&&&&\ddots&\ddots&&\\&&&&&&&1&1&\\&&&&&&&&1\end{array}\right),\qquad y=(0,\dots,0,1)^t$$

证明集合 $[y,By,\dots,B^iy]$ 是 $i<n/2$ 的独立集，但对任何更大的 i 值均不成立。

虽然通常可以预期向量 x、 Ax、 A^2x、 … 是独立的，但在计算机算术中这一结论不再明确成立。

假设矩阵具有特征值 $\lambda_0>\lambda_1\geq\cdots\lambda_{n-1}$ 及对应的特征向量 u_i，使得

$$Au_i=\lambda_iu_i.$$

设向量 x 可表示为

$$x=c_0u_0+\cdots+c_{n-1}u_{n-1},$$

then

$$A^ix=c_0\lambda_0^iu_i+\cdots+c_{n-1}\lambda_{n-1}^iu_{n-1}.$$

若将其表示为

$$A^ix=\lambda_0^i\left[c_0u_i+c_1\left(\frac{\lambda_1}{\lambda_0}\right)^i+\cdots+c_{n-1}\left(\frac{\lambda_{n-1}}{\lambda_0}\right)^i\right],$$

从数值上看， A^ix 将逐渐趋近于 u_0 的倍数，即主特征向量。因此，任何依赖于 A^ix 向量独立性的计算都可能不准确。章节 5.5.9 讨论了可视为为幂方法向量张成的空间构建正交基的迭代方法。

14.4 非负矩阵；佩龙向量

若 A 为非负矩阵，其最大特征值具有以下属性：对应的特征向量为非负向量 —— 此即佩龙 - 弗罗贝尼乌斯定理。

定理 5 若非负矩阵 A 不可约，其特征值满足

- 模最大的特征值 α_1 为实数且是单重的：

$$\alpha_1 > |\alpha_2| \geq \cdots .$$

- 对应的特征向量为正向量。

该定理最著名的应用是 Googe *PageRank* 算法；参见第 10.4 节。

14.5 格尔施戈林定理

求矩阵的特征值通常很复杂，但有一些工具可以估计特征值。本节将介绍一个定理，在某些情况下能提供关于特征值的有用信息。

设 A 为方阵，x、λ 为其特征对：$Ax = \lambda x$。观察其中一个组件，可得

$$a_{ii}x_i + \sum_{j\neq i} a_{ij}x_j = \lambda x_i.$$

取范数：

$$(a_{ii} - \lambda) \leq \sum_{j\neq i} |a_{ij}| \left|\frac{x_j}{x_i}\right|$$

取使 $|x_i|$ 最大的 i 值，可得

$$(a_{ii} - \lambda) \leq \sum_{j\neq i} |a_{ij}|.$$

该陈述可解读如下：

特征值 λ 位于以 a_{ii} 为中心、半径为 $\sum_{j\neq i}|a_{ij}|$的圆内 .

由于我们无法确定 i $|x_i|$ 取何值时达到最大，只能断言存在某个 i 值使得 λ 落在此类圆内。这便是格尔舒戈林定理。

定理 6 设 A 为方阵，D_i 表示以 a_{ii} 为中心、$\sum_{j\neq i}|a_{ij}|$为半径的圆，则其特征值包含于圆 $\cup_i D_i$ 的并集中。

若常向量不是特征向量，则可得出结论：特征值位于这些圆盘的内部。

14.6 Householder 反射器

在某些情况下，会出现如何将一个子空间转换为另一个子空间的问题。*Householder* 反射器在某种意义上是最小化解。考虑一个单位向量 u，并设

$$H = I - 2uu^t.$$

对于这个矩阵，我们有 $Hu = -u$，如果 $u \perp v$，那么 $Hv = v$。换句话说， u 的倍数子空间被翻转，而正交子空间保持不变。

现在回到将一个空间映射到另一个空间的原始问题。设原始空间由向量 x 张成，结果空间由 y 张成，那么注意

$$\begin{cases} x = (x+y)/2 + (x-y)/2 \\ y = (x+y)/2 - (x-y)/2. \end{cases}$$

换句话说，我们可以基于 $u = (x-y)/$ 的反射器将 x 映射到 y 2.

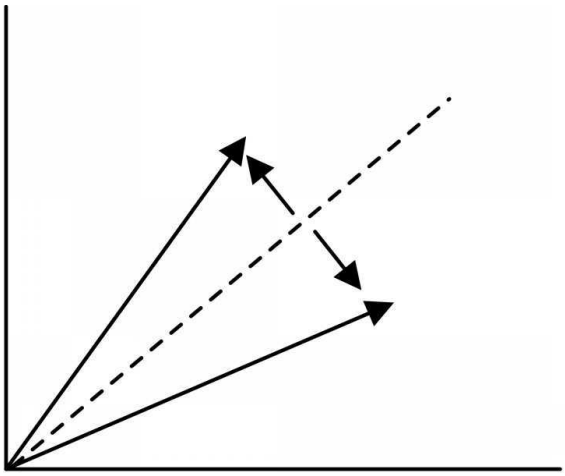


图 14.1：豪斯霍尔德反射器。

我们可以将豪斯霍尔德反射器推广为一种形式

$$H = I - 2uv^t.$$

在 LU 分解中使用的矩阵 L_i （参见第 5.3 节）可被视为具有以下形式 $L_i = I - \ell_i e_i^t$ 其中 e_i 在 i-th 位置有一个单一的非零元素，且 ℓ_i 仅在该位置下方有非零元素。这种形式也使得容易看出 $L_i^{-1} = I + \ell_i e_i^t$:

$$(I - uv^t)(I + uv^t) = I - uv^t uv^t = 0$$

如果 $v^t u = 0$。

第 15 章

复杂度

本书多处探讨了算法所需的操作次数。具体操作类型视上下文而定，但通常我们会计算加法（或减法）和乘法的次数。这被称为算法的**算术复杂度**或**计算复杂度**。例如，对 n 个数求和需要进行 $n-1$ 次加法运算。另一个需要描述的指标是所需的空间量（计算机内存）。有时仅需容纳算法的输入输出空间，但某些算法需要临时空间。总需求空间被称为算法的**空间复杂度**。

算术复杂度和空间复杂度都依赖于输入的某种描述。例如，对数字数组求和时，仅需知道数组长度 n 即可。我们通过表述 ‘对数字数组求和的时间复杂度为 $n-1$ 次加法，其中 n 表示数组长度’ 来表达这种依赖关系。

求和算法所需的时间（或空间）不依赖于其他因素（如数值大小）。相比之下，某些算法（如计算整数数组的最大公约数）可能会依赖于实际数值。在关于排序的第 9 章中，您将看到这两种类型。

练习 15.1. 两个大小为 $n \times n$ 的方阵相乘的时间与空间复杂度是多少？假设加法与乘法耗时相同。

我们常常致力于简化描述时间或空间复杂度的公式。例如，若某算法的复杂度为 $n^2 + 2n$，可观察到当 $n > 2$ 时复杂度低于 $2n^2$，而当 $n > 4$ 时则低于 $(3/2)n^2$。另一方面，对于所有 n 值，复杂度至少为 n^2。显然，二次项 n^2 最为关键，而线性项 n 的占比影响逐渐减弱。我们非正式地表述为：当 $n \to \infty$ 时，复杂度关于 n 呈二次增长 —— 存在常数 c、 C，使得当 n 足够大时，复杂度至少为 cn^2 且至多为 Cn^2。

简言之，这表示复杂度属于阶 n^2，记作 $O(n^2)$ 当 $n \to \infty$。在第 4 章中，你将看到可用趋于零的参数阶数描述的现象。此时我们写作例如 $f(h) = O(h^2)$ 当 $h \downarrow 0$，意指 f 被 ch^2 和 Ch^2 约束，其中存在特定常数 c、 C 与 h 足够小。

15.1 形式化定义

从形式化角度，我们区分以下复杂度类型：

- 大 O 复杂度。这表示存在一个绝对量级的上界：

$$g(x) = O(f(x)) \equiv \exists_{C>0,x_0>0} \forall_{x>x_0} : |g(x)| < Cf(x)$$

- 小 o 复杂度。这表示一个函数本质上比另一个函数小：

$$g(x) = o(f(x)) \equiv \forall_{C>0} \exists_{x_0>0} \forall_{x>x_0} : |g(x)| < Cf(x)$$

- 大 Θ 复杂度。这类似于大 O，但此时我们同时拥有上界和下界：

$$g(x) = \Theta(f(x)) \equiv \exists_{C>c>0,x_0>0} \forall_{x>x_0} : cf(x) < g(x) < Cf(x)$$

- 最后：大 Ω 复杂度表示一个函数至少与另一个函数一样大：

$$g(x) = \Omega(f(x)) \equiv \exists_{C>0,x_0>0} \forall_{x>x_0} : g(x) > Cf(x)$$

15.2 主定理

对于许多操作而言，复杂度很容易以递归形式表达。例如，快速排序的复杂度遵循一个递推关系

$$T(n) = 2T(n/2) + O(n),$$

这表明对包含 n 个数字的数组进行排序，需要先对 $n/2$ 个数字进行两次排序，再加上分割步骤，其时间复杂度与数组大小呈线性关系。

所谓的复杂度主定理允许将这些递归表达式转换为闭式公式。

Suppose the function T satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

其中 a、 $b \geq 1$ 和 $f(\cdot)$ 均为渐近正数。此时存在三种情况：

1. If

$$f(n) = O\left(n^{\log_b(a)-\epsilon}\right) \quad \text{then} \quad T(n) = \Theta\left(n^{\log_b(a)}\right)$$

2. If

$$f(n) = O\left(n^{\log_b(a)}\right) \quad \text{then} \quad T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$$

3. If

$$f(n) = O\left(n^{\log_b(a)+\epsilon}\right) \quad \text{then} \quad T(n) = \Theta\left(f(n)\right)$$

以下是一些示例。

15.2.1 快速排序与归并排序

快速排序 与 归并排序 算法通过递归将工作划分为两个（理想情况下）均等的部分，并在分割 / 合并步骤前后进行线性处理。因此，它们的复杂度满足

$$T(n) = 2T(n/2) + O(n)$$

代入 $a = b = 2$ 和 $f(n) = n$，根据情形 2 可得：

$$T(n) = \Theta(n \log n).$$

15.2.2 矩阵乘法

我们已知普通三重循环实现的矩阵乘法复杂度为 $O(n^3)$。若通过递归二分法将矩阵分割为 2×2 块进行乘法运算，则得到

$$T(n) = 8T(n/2) + O(n^2).$$

代入 $a = 8, b = 2$ 可得 $\log_b(a) = 3$，因此适用情形 1（参数为 $\epsilon = 1$），我们得出

$$T(n) = \Theta(n^3).$$

稍有趣的是，*Strassen* 算法 [179] 具有 $a = 7$ 特性，其复杂度本质更低，为 $O(n^{\log 7}) \approx O(n^{2.81})$。类似 [157], 的其他算法复杂度略低，但仍远高于 $O(n^2)$。

15.3 小 o 复杂度

大多数算法中我们关注大 O 复杂度，计算小 o 复杂度的情况较为罕见。

以大整数分解为例（密码学的重要环节）。朴素方法需遍历 $\sqrt{N}$ 以内的所有数以寻找 N 的因数。最佳算法之一是二次筛法，其复杂度为 $O(e^{\sqrt{\ln n \ln \ln n}})$。

这比 $O(\sqrt{N})$ 更优吗？是的，因为

$$e^{\sqrt{\ln n \ln \ln n}} = o(\sqrt{n}).$$

练习 15.2. 证明这一点。提示：写出 $\sqrt{n} = e^{\ln \sqrt{n}}$。

15.4 摊还复杂度

.

在某些情况下，您可能关注一系列操作，其中每个操作的成本差异很大。此时，考察最坏操作是合理的，但从实际角度来看并非最相关。为此，人们提出了摊还复杂度的概念 [180]。

摊还复杂度的一个常见例子是动态扩展数组。假设我们已为 n 个连续元素分配了空间，现在要向数组中添加一个新元素并保持数据连续性。以下是空间最优的解决方案：

1. 分配 $n+1$ 个元素的空间；
2. 复制现有数据；3. 写入新
元素；4. 释放旧数组。

练习 15.3. 重复执行此操作的复杂度是多少？忽略创建数组的所有成本，仅考虑复制操作，并假设每个元素的复制时间为单位时间。

诸如 C++ 等语言的运行时库采用了一种更高效的方案：

1. 给定已分配 n 个元素的空间，且需要存储第 $n+1$ 个元素时：2. 分配 $2n$
个元素的空间；3. 复制原有数据并写入新元素；4. 释放旧数组；5. 后续
$n-1$ 个元素直接写入新数组的未使用部分。

该方案在空间上并非最优：若始终未填满 $2n$ 个元素，部分内存将被浪费 .
H但根据上述元素写入成本衡量标准，该方案已达到最优。

练习 15.4 证明此方案存储 n 个元素的总成本为 $O(n)$。实际乘数常数是多少？

由于写入 n 个元素的复杂度为 $O(n)$，我们称此方案的摊还复杂度为 $O(1)$，该结果在常数范围内已达最优。，

第 16 章

偏微分方程

偏微分方程是高性能计算（ HPC ）问题的主要来源之一。以下是两个最重要偏微分方程的快速推导。

16.1 偏导数

函数 $u(x)$ 的导数是变化率的度量。偏导数同理，但针对二元函数 $u(x,y)$。记作 u_x 和 u_y，这些偏导数表示仅一个变量变化而另一个保持恒定时函数的变化率。

形式上，我们定义 u_x u_y by:

$$u_x(x,y) = \lim_{h\to 0} \frac{u(x+h,y)-u(x,y)}{h}, \quad u_y(x,y) = \lim_{h\to 0} \frac{u(x,y+h)-u(x,y)}{h}$$

16.2 泊松或拉普拉斯方程

设 T 为某材料的温度，则其热能与之成正比。长度为 Δx 的区段具有热能 $Q = c\Delta x \cdot u$。若该区段内的热能保持恒定

$$\frac{\delta Q}{\delta t} = c\Delta x \frac{\delta u}{\delta t} = 0$$

但它同时也是该区段流入与流出热能的差值。由于热流与温差（即 u_x）成正比，我们可推导出

$$0 = \frac{\delta u}{\delta x}\bigg|_{x+\Delta x} - \frac{\delta u}{\delta x}\bigg|_{x}$$

当 $\Delta x \downarrow 0$ 趋近极限时，得到 $u_{xx} = 0$，称为拉普拉斯方程。若存在源项（如对应外部施加的热量），方程则变为 $u_{xx} = f$，称为泊松方程。

16.3 热方程

设 T 为材料的温度，则其热能与之成正比。长度为 Δx 的片段具有热能 $Q = c\Delta x \cdot u$。该片段中热能的变化率为

$$\frac{\delta Q}{\delta t} = c\Delta x \frac{\delta u}{\delta t}$$

但它也是该片段流入与流出热能的差值。由于热流与温差（即 u_x）成正比，我们可得出

$$\frac{\delta Q}{\delta t} = \frac{\delta u}{\delta x}\Big|_{x+\Delta x} - \frac{\delta u}{\delta x}\Big|_{x}$$

当 $\Delta x \downarrow 0$ 趋近于极限时，可得 $u_t = \alpha u_{xx}$。

16.4 稳态

初边值问题（IBVP）的解是一个函数 $u(x,t)$。当外力函数和边界条件不随时间变化时，解将随时间收敛至一个称为稳态解的函数：

$$\lim_{t\to\infty} u(x,t) = u_{\text{steadystate}}(x).$$

该解满足一个边值问题（BVP），可通过设置 $u_t \equiv 0$ 求得。例如，对于热传导方程

$$u_t = u_{xx} + q(x)$$

稳态解满足 $-u_{xx} = q(x)$。

第 17 章

泰勒级数

泰勒级数展开是一种强大的数学工具。在本课程中，它被多次用于证明数值方法的性质。

泰勒展开定理在某种意义上探讨了函数能被多项式逼近的程度，即对于给定函数 f，我们能否找到系数 c_i，使得 $i = 1$、 ...、 n 满足

$$f(x) \approx c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n.$$

这个问题显然需要进一步明确。‘近似相等’ 具体指什么？这个近似公式不可能对所有函数 f 和所有 x 都成立：例如 $\sin x$ 函数在所有 x 上是有界的，但任何多项式在 $x \to \pm\infty$ 时都是无界的，因此任何对 $\sin x$ 函数的多项式逼近必然无界。显然我们只能在某个区间内进行近似。

我们将证明具有足够多导数的函数 f 可如下近似：如果 n 阶导数 $f^{(n)}$ 在区间 I 上连续，则存在系数 c_0、 ...、 c_{n-1} 使得

$$\forall_{x \in I} : |f(x) - \sum_{i<n} c_i x^i| \leq c M_n \qquad \text{where } M_n = \max_{x \in I} |f^{(n)}(x)|$$

It is easy to get inspiration for what these coefficients should be. Suppose

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots$$

(此处我们暂不讨论收敛性问题及省略号延续的长度) 然后代入

$$x = 0 \text{ gives } c_0 = f(0).$$

接着，取一阶导数

$$f'(x) = c_1 + 2c_2 x + 3c_3 x^2 + \cdots$$

并代入

$$x = 0 \text{ gives } c_1 = f'(0).$$

从二阶导数出发

$$f''(x) = 2c_2 + 6c_3x + \cdots$$

so filling in $x = 0$ gives

$$c_2 = f''(0)/2.$$

类似地，在三阶导数中

$$\text{filling in } x = 0 \text{ gives } c_3 = \tfrac{1}{3!} f^{(3)}(0).$$

现在我们需要更精确一些。柯西形式的泰勒定理指出

$$f(x) = f(a) + \frac{1}{1!} f'(a)(x-a) + \cdots + \frac{1}{n!} f^{(n)}(a)(x-a)^n + R_n(x)$$

where the 'rest term' R_n is

$$R_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-a)^{n+1} \quad \text{where } \xi \in (a,x) \text{ or } \xi \in (x,a) \text{ depending.}$$

若 $f^{(n+1)}$ 有界，且 $x = a + h$，则我们常用泰勒定理的形式为

$$f(x) = \sum_{k=0}^{n} \frac{1}{k!} f^{(k)}(a)h^k + O(h^{n+1}).$$

我们现已用多项式在特定区间内逼近函数 f，其误差随多项式次数的倒数呈几何级数递减。

为证明泰勒定理，我们采用分部积分法。首先写出

$$\int_a^x f'(t)dt = f(x) - f(a)$$

as

$$f(x) = f(a) + \int_a^x f'(t)dt$$

分部积分后可得

$$\begin{aligned} f(x) &= f(a) + [xf'(x) - af'(a)] - \textstyle\int_a^x tf''(t)dt \\ &= f(a) + [xf'(x) - xf'(a) + xf'(a) - af'(a)] - \textstyle\int_a^x tf''(t)dt \\ &= f(a) + x\textstyle\int_a^x f''(t)dt + (x-a)f'(a) - \textstyle\int_a^x tf''(t)dt \\ &= f(a) + (x-a)f'(a) + \textstyle\int_a^x (x-t)f''(t)dt \end{aligned}$$

再次应用分部积分可得

$$f(x) = f(a) + (x-a)f'(a) + \frac{1}{2}(x-a)^2 f''(a) + \frac{1}{2}\int_a^x (x-t)^2 f'''(t)dt$$

通过归纳法，这为我们提供了带余项的泰勒定理

$$R_{n+1}(x) = \frac{1}{n!}\int_a^x (x-t)^n f^{(n+1)}(t)dt$$

根据中值定理可知

$$\begin{aligned} R_{n+1}(x) &= \frac{1}{(n+1)!} f^{(n+1)}(\xi) \int_a^x (x-t)^n f^{(n+1)}(t)dt \\ &= \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-a)^{n+1} \end{aligned}$$

第 18 章

最小化

18.1 下降方法

我们考虑一个多元函数 $f: R^N \to$ 以及寻找使函数达到最小值的向量 x 的问题。即使对于光滑函数，我们也立即面临局部最小值和全局最小值的存在；目前我们满足于寻找一个局部最小值。

与其通过一次大规模计算找到最小值，我们采用迭代策略：从点 x 出发，更新至 $x + h$，并重复此过程。更新向量 h 通过线性搜索策略确定：选定一个搜索方向 h，沿该方向找到步长 τ，然后进行更新

$$\bar{x} \leftarrow \bar{x} + \tau \bar{h}.$$

18.1.1 最速下降法

利用泰勒展开公式的多维形式（第 17 节），我们可以将任何足够光滑的函数表示为

$$f(\bar{x} + \bar{h}) = f(\bar{x}) + \bar{h}^t \nabla f + \cdots.$$

不难看出选择 $h = -\nabla f$ 能实现最大程度的极小化，因此我们设

$$x_{\text{new}} \equiv x - \tau \nabla f.$$

此方法称为梯度下降或最速下降法。

对于新的函数值，这给出了

$$f(\bar{x} - \tau \nabla f) \approx f(\bar{x}) - \tau \|\nabla f\|^2$$

因此当 τ 足够小时，新的函数值既为正，又小于 $f(x)$。

步长 τ 可针对二次函数 f 计算，其他情况下可近似求得：

$$f(\bar{x} + \tau \bar{h}) = f(\bar{x}) + \tau \bar{h}^t \nabla f + \frac{\tau^2}{2} h^t (\nabla \cdot \nabla f) h + \cdots.$$

忽略高阶项并设置 $\delta f/\delta\tau = 0$ 得到

$$\tau = -\bar{h}^t \nabla f / h^t (\nabla \cdot \nabla f) h.$$

另一种寻找合适步长的策略称为回溯法 :

- 从默认步长开始，例如 $\tau \equiv 1$。
- 然后直到 $f(x + \tau h) < f(x)$ 执行

$$\tau \leftarrow \tau/2.$$

18.1.2 随机梯度下降

虽然梯度是每一步中的最优搜索方向，但整体而言它未必是最佳选择。例如，对于二次型问题（包括许多偏微分方程问题），更好的选择是对搜索方向进行正交化处理。另一方面，在机器学习应用中，随机梯度下降法（*SGD*）是一个不错的选择，其中坐标方向被用作搜索方向。在这种情况下

$$f(x + \tau e_i) = f(x) + \tau \frac{df}{de_i} + \frac{\tau^2}{2} \frac{\delta^2 f}{\delta e_i^2}$$

Then:

$$\tau = -(\nabla f)_i / \frac{\delta^2 f}{\delta e_i^2}.$$

18.1.3 代码

18.1.3.1 预备知识

我们声明一个 `vector` 类，它是一个标准向量，其上定义了加法等运算，还包括旋转操作。

存在一个派生类 `unit_vector`，其功能显而易见。

18.1.3.2 框架

我们首先将函数定义为一个纯虚类，这意味着任何函数都需要支持此处提到的方法：

```
// minimlib.h
class function {
public:
  virtual double eval( const valuevector& coordinate ) const = 0;
  virtual valuevector grad( const valuevector& coordinate ) const = 0;
  virtual std::shared_ptr<matrix> delta( const valuevector& coordinate ) const = 0;
  virtual int dimension() const = 0;
};
```

利用此类函数，可以定义多种更新步骤。例如，最速下降法采用梯度：

```
valuevector steepest_descent_step
    ( const function& objective,const valuevector& point ) {
  auto grad = objective.grad(point);
  auto delta = objective.delta(point);

  auto search_direction( grad );
  auto hf = grad.inprod(search_direction);
  auto hfh = search_direction.inprod( delta->mvp(search_direction) );
  auto tau = - hf / hfh;
  auto new_point = point + ( search_direction * tau );
  return new_point;
};
```

另一方面，随机下降法基于单位向量：

```
valuevector stochastic_descent_step
    ( const function& objective,const valuevector& point, int idim,
      double damp) {
  int dim = objective.dimension();
  auto grad = objective.grad(point);
  auto delta = objective.delta(point);

  auto search_direction( unit_valuevector(dim,idim) );
  auto gradfi = grad.at(idim);
  auto hf = gradfi;
  auto hfh = delta->d2fdxi2(idim);
  auto tau = - hf / hfh;
  auto new_point = point + ( search_direction * (tau * damp) );

  return new_point;
};
```

18.1.3.3 健全性测试

在圆形上使用最速下降法可一步收敛：

Code:

```
        ellipse circle
          ( valuevector( {1.,1.} ),valuevector(
        {0.,0.} ) );
        valuevector search_point( { 1.,1. } );
        auto value = circle.eval(search_point);

        auto new_point = steepest_descent_step(
        circle,search_point);

        auto new_value = circle.eval(new_point);
        cout << "Starting point "
             << "(" << search_point.at(0)
             << "," << search_point.at(1) << "); "
             << "with value: " << value << "\n"
             << "  becomes "
             << "(" << new_point.at(0)
             << "," << new_point.at(1) << "); "
             << "with value: " << new_value << endl
        ;
```

Output
[code/minimization] descentcircle:

```
missing snippet
code/minimization/descentcircle.runout
: looking in codedir=code
missing snippet
code/minimization/descentcircle.runout
: looking in codedir=code
```

另一方面，在椭圆上进行最速下降法需要多次迭代：

Code:

```
ellipse circle( valuevector( {1.,.1} ),
  valuevector( {0.,0.} ) );
valuevector search_point( { 1.,1. } );
auto value = circle.eval(search_point);
for (int step=0; step<5 and value>.0001; step
  ++) {

  auto new_point = steepest_descent_step(
  circle,search_point);

  auto new_value = circle.eval(new_point);
  cout << "Starting point " << fixed
       << "(" << setprecision(4) <<
  search_point.at(0)
     << "," << setprecision(4) <<
  search_point.at(1) << "); "
     << "with value: " << value << "\n"
       << "  becomes "
       << "(" << setprecision(4) << new_point
  .at(0)
     << "," << setprecision(4) << new_point.
  at(1) << "); "
       << "with value: " << new_value << endl
  ;
  search_point = new_point; value = new_value
  ;
}
```

Output

```
[code/minimization] descentellipse:

missing snippet
code/minimization/descentellipse.runout
: looking in codedir=code
missing snippet
code/minimization/descentellipse.runout
: looking in codedir=code
```

18.2 牛顿法

牛顿法 (或称牛顿 - 拉夫逊方法) 是一种迭代过程

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

用于寻找函数的零点 f，即满足 x 的数值解，此时 $f(x) = 0$。该方法需要已知函数的导数 f'，并可通过类似图 18.1 的图示加以验证。

迭代过程如下:

Another justification comes from minimization: if a function f is twice differentiable, we can write

$$f(x+h) = f(x) + h^t \nabla f + \frac{1}{2} h^t (\nabla^2 f) h$$

最小值出现于

$$x_{\text{new}} = x - (\nabla^2 f)^{-1} \nabla f.$$

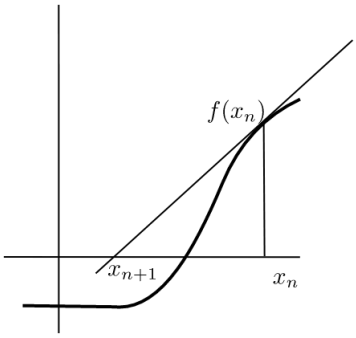


图 18.1: 一维牛顿方法中的一步。

这等价于寻找梯度的零点：

$$0 = \nabla f(x+h) = \nabla f(x) + h^t(\nabla^2 f(x))h.$$

练习 18.1. 设 $f(x_1, x_2) = (10x_1^2 + x_2^2)/2 + 5\log(1 + e^{-x_1-x_2})$。梯度下降法和牛顿方法的收敛速度如何？为了理解它们的不同行为，绘制若干迭代点相对于函数等高线的图。

这个练习给出了梯度下降法的问题所在：它总是沿着当前等高线的垂直方向移动。然而，最小值并不一定在那个方向上。

我们不加证明地陈述：

- 若迭代起始点足够接近零点且函数在该零点可微，牛顿方法将收敛至该零点。
- 在多数情况下，该方法展现出极快的收敛速度：其收敛为二次收敛，亦可描述为每次迭代中正确数字位数翻倍。
- 对于许多函数，牛顿方法可能不收敛，但可通过引入阻尼或进行非精确更新来实现收敛：

  $$x_{n+1} = x_n - \alpha f(x_n)/f'(x_n)$$

  其中 $\alpha < 1$。

练习 18.2. 牛顿方法有可能陷入循环。假设这是一个长度为二的循环：

$$x_0 \to x_1 \to x_2 = x_0.$$

若写出该循环对应的方程，您会发现关于 f 的微分方程。其解为何？为何牛顿方法对此函数不收敛？

在多维情况下，即对于函数 $f: \mathbb{R}^N \to \mathbb{R}$，牛顿方法演变为一个迭代的线性系统解：

$$\bar{x}_{n+1} = \bar{x}_n - F(\bar{x}_n)^{-1} f(\bar{x}_n)$$

其中 F 是 f的雅可比矩阵。

由于牛顿法是一种迭代过程，我们不需要此线性系统的解达到完全精度，因此不精确的解（例如通过迭代求解）通常是可行的。

18.2.1 不精确牛顿法

牛顿法可能因多种原因无法收敛。（实际上，对于合适的函数，绘制其收敛区域会产生漂亮的分形图形。）因此，实践中会使用不精确牛顿法。不精确性体现在两个方面：

- 我们使用一个分数而非 ‘最优’ 步长。这通常通过 ‘回溯法’ 选择，并采用能观察到函数值至少有所下降的选项。
- 我们使用该算子的近似值而非雅可比矩阵的逆。例如，可以计算雅可比矩阵（通常是一个昂贵的过程），然后将其用于多个牛顿步骤。

可以证明，这些技术的组合能够确保收敛 [199]。

第 19 章

随机数

随机数生成在生成随机测试数据或蒙特卡洛模拟中非常有用；参见第 12 章。

此处我们将讨论随机数生成器（ RNGs ）的通用原理、在编程语言中的应用以及并行随机数生成所面临的问题。

19.1 随机数生成

随机数常被用于模拟实验，如下文若干示例所示。获取真正的随机数极为困难：可通过测量量子过程（如放射性粒子）生成。自 *Intel Ivy Bridge* 架构起，英特尔处理器配备了基于热噪声的硬件随机数生成器 [95] ）。

最常见的解决方案是使用伪随机数。这意味着我们采用一种确定性的数学过程，该过程足够不规则，以至于在实际应用中无法发现其规律性。

19.1.1 序列随机数生成器

生成随机数（我们省略 ‘伪’ 限定词）的简单方法是使用线性同余生成器（关于随机数的所有知识，可参阅 Knuth [120] ），其递推公式形式如下

$$x_{k+1} = (ax_k + b) \mod m.$$

该序列具有周期性，因为它由最多 $m - 1$ 的非负整数构成，并在特定条件下周期为 m。典型周期长度为 2_{31}。序列起始点 x_0 被称为 ‘种子’ 。随机数软件通常允许用户指定种子值：若需获得可复现的结果，可多次使用相同种子运行程序；若需程序多次运行时表现随机行为，可从时钟和日历函数中派生种子值。

线性同余生成器的低位可能存在一定相关性。另一种随机数生成原理是滞后斐波那契随机数生成器

$$X_i = X_{i-p} \otimes X_{i-q}$$

其中 p 和 q 为滞后参数，$\otimes$ 为任意二元运算，例如模 M 的加法或乘法。

滞后斐波那契生成器的主要问题在于：

- 它们需要设置最大值 (p,q) 初始值，且其随机性对这些选择非常敏感；
- 该理论不如同余生成器那样成熟，因此更依赖于统计测试来评估其“随机性”。

19.2 编程语言中的随机数

19.2.1 C

有一种简单（但不算特别好）的随机数生成器（RNG），在 C 和 C++ 中的工作方式相同。

```
#include <random>
using std::rand;
float random_fraction =
    (float)rand()/(float)RAND_MAX;
```

函数 *rand* 会生成一个 `int` —— 每次调用时都会不同 —— 其范围从零到 *RAND_MAX*。通过缩放和类型转换，你可以用上述代码生成一个介于零和一之间的分数。

如果你两次运行程序，会得到相同的随机数序列。这对于调试（Debug）程序很有帮助，但如果你希望进行统计分析就不太理想了。因此，你可以通过 *srand* 函数设置随机数种子来指定随机序列的起始点。示例：

```
srand(time(NULL));
```

以当前时间作为随机数生成器的种子。此调用应仅发生一次，通常位于程序主逻辑的高层级位置。

19.2.1.1 C 语言随机数生成器的问题

- 该随机数生成器的周期仅为 2_{15}，可能偏小。
- 仅有一种依赖于具体实现的生成算法，且无法保证其质量。
- 缺乏将序列转换到特定范围的机制。常见做法

  ```
  int under100 = rand() % 100
  ```

  会偏向小数值。图 19.1 展示了周期为 7 的生成器取模 3 时的偏差情况。

19.2.1.2 其他生成器

还有其他源自 Unix 系统调用的 C 语言随机数生成器。

- *drand48*: 在 *SystemV 3* 中已废弃，由 *rand* 取代。
- *rand_r*: *rand* 的线程安全变体。在 *POSIX 1-2008* 中标记为废弃
- *random*: *rand* 的现代升级版本。
- *random_r*: *random* 的线程安全变体。

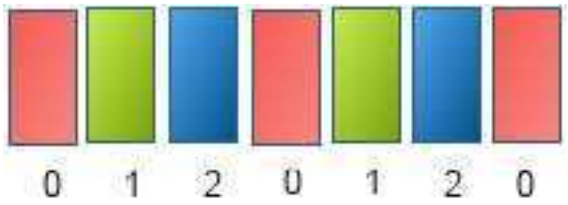


图 19.1：随机数生成器取模后的低数值偏差。

19.2.2 C++

标准模板库（STL）提供了一个随机数生成器，它比 C 语言版本更通用且更灵活。

- 有几种生成器可以产生均匀分布的数字；
- 然后还有一些分布函数可以将这些数字转换为非均匀或离散分布。

First you declare an engine; later this will be transformed into a distribution:

```
std::default_random_engine generator;
```

该生成器每次都会从相同的初始值开始。您可以通过设定种子值来改变这一行为：

```
std::random_device r;
std::default_random_engine generator{ r() };
```

接下来需要声明分布类型。例如，声明一个给定边界内的均匀分布：

```
std::uniform_real_distribution<float> distribution(0.,1.);
```

掷骰子的结果可通过以下方式生成：

```
std::uniform_int_distribution<int> distribution(1,6);
```

19.2.2.1 随机浮点数

```
// seed the generator
std::random_device r;
// std::seed_seq ssq{r()};
// and then passing it to the engine does the same

// set the default random number generator
std::default_random_engine generator{r()};

// distribution: real between 0 and 1
std::uniform_real_distribution<float> distribution(0.,1.);

cout << "first rand: " << distribution(generator) << endl;
```

19.2.2.2 掷骰子

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

19.2.2.3 泊松分布

另一种分布是泊松分布：

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

19.2.3 Fortran

本节我们将简要讨论 Fortran 的随机数生成器。其基本机制是通过库子程序 `random_number` 实现，该子程序有一个类型为 `REAL` 的参数，且 `INTENT(OUT)`：

```
real(4) :: randomfraction
call random_number(randomfraction)
```

结果是一个来自 $[0,1)$ 上均匀分布的随机数。

设置随机种子的过程稍显复杂。存储种子所需的空间量可能因处理器和实现方式而异，因此子程序 `random_seed` 可包含三种命名参数类型，但每次只能指定其中一种。关键字可以是：

- `SIZE` 用于查询种子大小；
- *PUT* 用于设置种子；以及
- *GET* 用于查询种子。

A typical fragment for setting the seed would be:

```
integer :: seedsizeinteger,dimension(:),
allocatable :: seed
call random_seed(size=seedsize)
allocate(seed(seedsize))
seed(:) = ! your integer seed here
call random_seed(put=seed)
```

19.2.4 Python

Python 有一个随机模块：

```
import random
x = random.random()
i = random.randint(lo,hi)
```

19.3 并行随机数生成

在并行计算中，随机数生成存在难题。设想一个并行进程在每个子进程中使用随机数生成器，再考虑用单处理器模拟该并行进程。此时，这个单一进程实际上拥有一个由并行生成器结果交织组成的随机数生成器。这意味着，如果在所有并行进程中使用相同的生成器，整个进程的有效生成器将产生大量重复值序列。

存在多种解决方案。

19.3.1 管理者 - 工作者生成器

我们可以集中生成随机数。在共享内存中，这可能意味着使其操作具有原子性。但这可能引发严重的性能瓶颈。

练习 19.1 通常认为临界区是合理的，如果其消耗的工作量低于并行工作。为何该论点在此不适用？

另一种解决方案是让一个线程或进程生成随机数，并将其分发给其他进程。逐个数分发会产生可观的开销。替代方案是让生成器进程分发数字块。然而，这种分发方式可能再次导致进程间的相关性。

19.3.2 序列分割解决方案

更优的解决方案是建立独立的生成器，并通过参数选择确保统计随机性。这并非易事。例如，若两个序列 $x_i^{(1)}$、$x_i^{(2)}$具有相同的 a、b、m 值，且它们的起始点相近，则序列间会存在强相关性。此外还存在更复杂的相关性案例。

存在多种随机数生成技术，例如使用两个序列：一个序列为另一个实际用于模拟的序列生成起始点。并行随机数生成器的软件可在 http://sprng.cs.fsu.edu/ [144] 找到。

若能在给定 x_i 条件下高效计算 x_{i+k}，可采用蛙跳技术：k 个进程拥有互不重叠的序列 $i \mapsto x_{s_k+ik}$，其中 x_{s_k} 是第 k 个序列的起始点。

19.3.3 分块随机数生成器

某些随机数生成器（参见 [133] ）允许计算距离种子值多次迭代后的结果。随后可将从种子到该迭代的数值块分配给一个处理器。同理，每个处理器将获得生成器连续迭代的数值块。

19.3.4 密钥随机数生成器

部分随机数生成器并非递归实现，而是允许显式公式化

$$x_n = f(n),$$

即第 n 个数值是其 ' 密钥 ' n 的函数。在等式中加入区块密钥后

$$x_n^{(k)} = f_k(n)$$

允许通过 k 索引的进程实现并行化。参见 [167]。

19.3.5 梅森旋转算法

梅森旋转算法随机数生成器经过调整，可生成不相关的并行数字流 [145]。此处进程 ID 被编码至生成器中。

第 20 章

图论

图论是数学的一个分支，研究对象之间的成对关系。图既作为分析高性能计算问题的工具出现，本身也是研究的对象。本附录将介绍基本概念及相关理论。

20.1 定义

图由一组对象及它们之间的关系集合构成。这些对象称为图的节点或顶点，通常构成有限集，因此我们常用连续整数 $1 \dots n$ 或 $0 \dots n-1$ 来标识它们。节点之间的关系由图的边描述：若 i 与 j 相关联，则称 (i,j) 为图的一条边。此关系不必对称，例如 “ 小于 ” 关系即为一例。

形式化地说，图是一个元组 $G = \langle V, E \rangle$，其中 $V = \{1, \dots n\}$ 对于某些 n 成立，且 $E \subset \{(i,j) : 1 \leq i,j \leq n, i \neq j\}$。

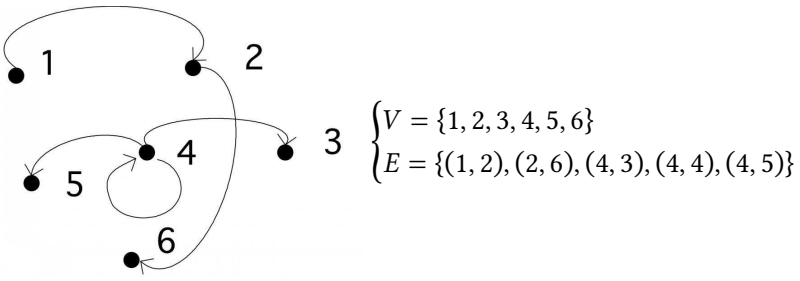


图 20.1：一个简单的图。

若 $(i,j) \in E \Leftrightarrow (j,i) \in E$，则称该图为无向图。另一种是有向图，其中我们用从 i 指向 j 的箭头表示边 (i,j)。

图论中经常出现的两个概念是图的度和直径。

定义 3 度表示连接到任一节点的最大节点数：

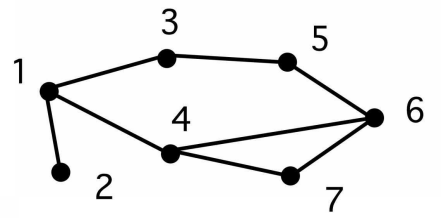
$$d(G) \equiv \max_i |\{j : j \neq i \wedge (i,j) \in E\}| .$$

定义 4 图的直径是指图中最长最短路径的长度，其中路径被定义为一组顶点 $v_1, \ldots, v_{k+1}$，使得 $v_i \neq v_j$ 对于所有 $i \neq j$ 且

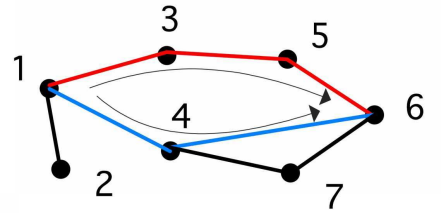
$$\forall_{1 \leq i \leq k} : (v_i, v_{i+1}) \in E.$$

该路径的长度为 k。

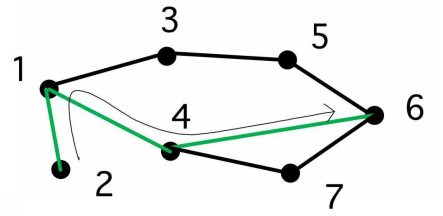
直径的概念如图 20.2 所示



A graph



Two paths from 1 to 6; $\langle 1, 4, 6 \rangle$ is the shorter



The longest shortest path of this graph

图 20.2：最短路径示意图

除 $v_1 = v_{k+1}$ 外所有节点均不重复的路径称为环。

有时我们仅关注边 (i, j) 的存在性，而其他时候则会为该边赋予一个值或 ‘权重’ w_{ij}。带有权重边的图称为加权图。此类图可表示为元组 $G = \langle V、 E、 W \rangle$，其中 E 与 W 具有相同的基数。反之，无权图的所有边权重相同，此时我们省略权重的提及。

20.2 常见图类型

20.2.1 有向无环图

不含环的图称为无环图，其特例是有向无环图（*DAG*）。此类图可用于建模任务间的依赖关系：若存在从 i 指向 j 的边，则意味着任务 i 必须在任务 j 之前完成。

20.2.2 树结构

有向无环图（DAG）的一个特例是树形图，简称树：其中任何节点可有多条出边，但只能有一条入边。没有出边的节点称为叶节点；没有入边的节点称为根节点，其余节点则称为内部节点。

习题 20.1：一棵树能否拥有多个根节点?

20.2.3 可分离图

可分离图是由两组节点构成的图，其中所有连接都位于任一组内部。此类图显然可并行处理，因此多种并行化算法致力于将图转化为可分离形式。若一个图可表示为 $V = V_1 + V_2 + S$，其中 V_1、V_2 中的节点仅与 S 相连而不与另一组相连，则称 S 为分离器。

20.2.4 二分图

如果一个图可以被划分为两个节点集，其中边仅从一个集合指向另一个集合，而不会在同一个集合内部相连，我们称其为二分图。

20.3 图着色与独立集

我们可以为图的节点分配标签，这相当于将节点集划分为不相交的子集。其中一种值得关注的标签类型是图着色：此处选择的标签（或‘颜色’）需满足若节点 i 与 j 颜色相同，则它们之间不存在边相连：$(i, j) \notin E$。

存在一种平凡的图着色方案，即每个节点拥有独特色。更有趣的是，能够用最少数量的颜色对图进行着色，该数量称为图的色数。

练习 20.2. 证明若一个图的度为 d，则其色数至多为 $d + 1$。

一个著名的图着色问题是‘四色定理’：如果一个图表示二维地图上的国家（即所谓的‘平面图’），那么其色数最多为四。一般而言，确定色数是非常困难的（实际上是 NP 难问题）。

图着色的颜色集合也被称为独立集，因为在每个颜色内部，没有节点与同色的其他节点相连。

寻找独立集有一种平凡的方法：为每个节点声明一个唯一的颜色。另一方面，寻找‘最佳’的独立集划分（例如通过确定图的色数）是困难的。然而，通常只需将节点合理地划分为独立集即可，例如在构建并行 ILU 预条件子时（参见第 7.7.5 节）。以下算法实现了这一目标 [113, 142]：

- 为每个节点分配一个唯一的随机数。
- 接着找出所有编号大于其邻居的节点集合，将其称为第一个独立集。

- 将该集合从图中移除，再次找出数值大于所有相邻节点的节点；这将成为第二个独立集合。
- 重复此过程，直到所有节点都被归入某个独立集合。

练习 20.3. 请验证通过此方法找到的集合确实满足独立性。

20.4 图算法

本节仅简要介绍图算法；完整讨论可参阅章节 10.1。

- 距离算法。在交通路线规划等应用中，了解从给定节点到所有其他节点的最短距离（即单源最短路径问题）或任意节点对之间的最短距离（即全源最短路径问题）具有重要意义。
- 连通性算法。在社交网络中，了解两个人是否存在连接关系、图是否可分解为不连通的子图、或某人是否为两个群体间的关键桥梁等问题具有研究价值。

20.5 图与矩阵

图可以通过多种方式呈现。当然可以直接列出节点和边，但这种方式难以获得深刻见解。简单图形可通过绘制顶点和边来可视化，但对于大型图则会显得杂乱。另一种选择是构建图的邻接矩阵。对于图 $G = \langle V 、 E \rangle$，邻接矩阵 M （其尺寸 n 等于顶点数 $|V|$ ）定义为

$$M_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

反之，若你有一个矩阵，尤其是稀疏矩阵，你可以构建其邻接图。图 20.3 展示了稠密矩阵与稀疏矩阵的示例。本例中，这些矩阵在结构上

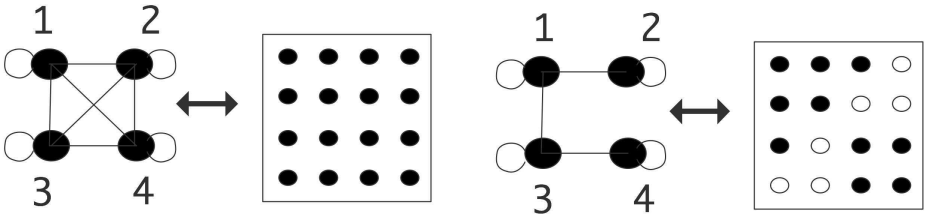


图 20.3：稠密矩阵与稀疏矩阵及其对应的邻接图。

对称，因此在图中我们用线段代替箭头。每个顶点上对应对角线元素的边通常会在图示中省略。

对于带边权重的图，我们将邻接矩阵的元素设置为对应权重：

$$M_{ij} = \begin{cases} w_{ij} & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

若矩阵不含零元素，则其邻接图中每对顶点间都存在边。此类图称为团。若图是无向的，则邻接矩阵对称；反之，若矩阵具有结构对称性，其邻接图必定无向。

20.5.1 排列

图常被用于表示现实世界中对象间的关系，例如 Facebook 中的 ' 好友 ' 关系。在此类场景下，图的节点并无自然编号：它们通过名称标识，任何编号均为人工赋予。因此我们需要思考：若采用不同编号方式，哪些图属性保持不变，哪些会随之改变。

对对象集合重新编号的代数建模方法，可通过将邻接矩阵乘以排列矩阵来实现。

定义 5 置换矩阵是一种方阵，其中每一行和每一列恰好有一个元素为 *1* ；其余元素均为零。

习题 20.4. 给定一组 N 对象 x_1、 ...、 x_N，将它们按 x_1、 x_3、 ...、 x_2、 x_4、 ... 顺序排列的置换矩阵是什么？即，找到满足条件的矩阵 P 使得

$$\begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = P \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix}$$

习题 20.5. 证明矩阵的特征值在置换下保持不变。

20.5.2 不可约性

以邻接矩阵中易于解释的图概念为例，考虑可约性。

定义 6 若对于任意节点对 i, j，存在从 i 到 j 以及从 j 到 i 的路径，则称该（有向）图为不可约的。若图不是不可约的，则称其为可约图。

习题 20.6. 设 A 为一个矩阵

$$A = \begin{pmatrix} B & C \\ \varnothing & D \end{pmatrix} \tag{20.1}$$

其中 B 和 D 为方阵。证明以此作为邻接矩阵的图的可约性。

方程 20.1 中的矩阵是块上三角矩阵。这意味着求解系统 $Ax = b$ 可分为两步完成，每步规模为 $N/2$，若 N 是 A 的大小。

练习 20.7. 证明这使得求解 $Ax = b$ 的算术复杂度低于一般的 $N \times N$ 矩阵。

若对图进行置换，其可约性或不可约性不会改变。然而，此时从邻接矩阵上可能不再直观可见。

20.5.3 图闭包

Here is another example of how adjacency matrices can simplify reasoning about graphs.

定义 7 设 $G = \langle V, E\rangle$ 为一个无向图，则我们定义 G 的闭包为 $G' = \langle V, E'\rangle$，即具有相同顶点但边由以下方式定义的图

$$(i, j) \in E' \Leftrightarrow \exists_k : (i, k) \in E \wedge (k, j) \in E.$$

举一个小例子：

习题 20.8. 若 M 是 G 的邻接矩阵，证明 M^2 是 G' 的邻接矩阵，其中我们对元素采用布尔乘法： $1 \cdot 1 = 1, 1 + 1 = 1$。

20.6 谱图论

对于图 G[1] 及其邻接矩阵 A_G，我们可以通过缩放 A_G 使其行和为 1 来定义一个随机矩阵或马尔可夫矩阵：

$$W_G = D_G^{-1} A_G \qquad \text{where } (D_G)_{ii} = \deg(i).$$

T为了理解其含义，让我们看一个简单例子。假设有一个无权图，其中包含一个
a邻接矩阵

$$A_G = \begin{pmatrix} 1 & & 1 & 1 \\ & & 1 & 1 \\ 1 & & 1 & 1 \\ & 1 & 1 & \end{pmatrix}$$

1. 本节内容很大程度上归功于 Dan Spielman 关于谱图理论的课程 http://www.cs.yale.edu/homes/spielman/561/。

观察第二行，其中显示存在边 $(2,3)$ 和 $(2,4)$。这意味着如果你在节点 2 上，可以前往节点 3 和 4。缩放该矩阵后得到

$$W_G = \begin{pmatrix} 1/3 & & 1/3 & 1/3 \\ & & 1/2 & 1/2 \\ 1/3 & & 1/3 & 1/3 \\ & 1/2 & 1/2 & \end{pmatrix}$$

现在第二行表明，从节点 2 出发，你可以以同等概率到达节点 3 和 4。你也可以通过数学推导得出这一结论：

$$\begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} W_G = \begin{pmatrix} 0 & 0 & 1/2 & 1/2 \end{pmatrix}$$

I很容易推断出：如果 p 是一个向量，其中 i 的组件表示处于
n节点 i 的概率，那么 $(p^t W_G)_i$就是沿着图边再走一步后处于节点 i 的概率。

练习 20.9. 证明 $p^t W_G$确实是一个概率向量。提示：你可以将 p 是概率向量表示为 $p^t e = e$，其中 e 是全 1 向量。

20.6.1 图拉普拉斯矩阵

与图相关联的另一种矩阵是图拉普拉斯矩阵

$$L_G = D_G - A_G.$$

该矩阵具有零行和与正对角元，根据 *Gershgorin* 定理（章节 14.5），其所有特征值均位于复平面右半部分。

习题 20.10 证明全 1 向量是特征值为 1 的特征向量。

该拉普拉斯矩阵给出了一个二次型：

$$x^t L_G x = \sum_{(i,j)\in E} (x_i - x_j)^2.$$

20.6.2 通过拉普拉斯矩阵的域分解

图论中的邻接矩阵和拉普拉斯矩阵关联着各种有趣的定理。这些定理在域分解方面具有非常实际的应用。

我们从椭圆型偏微分方程中获得了灵感。

与拉普拉斯方程 $-\Delta u = f$ 相关联的是一个算子 $Lu = -\Delta u$。在单位区间 $[0,1]$ 上，该算子的特征函数（即满足 $Lu = \lambda u$ 的函数）为 $u_n(x) = \sin n\pi x$ （其中 $n > 0$）。这些函数具有这样的属性：$u_n(x)$ 在区间内部有 $n-1$ 个零点，并将区间划分为 n 个连通区域，函数在这些区域内为正或负。因此，若需将域 Ω 分配到 p 个处理器上，可考虑 Ω 上拉普拉斯算子的第 p 个特征函数，并找出其正负值所在的连通区域。

关于偏微分方程的这一论述在图论中有两个等效版本，即菲德勒定理。（本节不作证明，详见 [175]。）

定理 7 设 G 为具有 n 个顶点的加权路径图，L_P 的特征值为 $0 = \lambda_1 < \lambda_2 \leq ... \leq \lambda_n$，$v_k$ 为 λ_k 的特征向量。则 v_k 会改变符号 $k-1$ 次。

第二个定理更为实用 [62]：

定理 8 设 $G = (V, E, w)$ 为加权连通图，L_G 为其拉普拉斯矩阵。令 $0 = \lambda_1 < \lambda_2 \leq \cdots \leq \lambda_n$ 为 L_G 的特征值，$v_1, \ldots, v_n$ 为对应的特征向量。对于任意 $k \geq 2$，令 $W_k = \{i \in V : v_k(i) \geq 0\}$。则由 G 在 W_k 上诱导的子图至多有 $k-1$ 个连通组件。

这一结论的重要之处在于，首个非平凡特征值对应的特征向量可用于将图划分为两个连通部分——一部分节点对应特征值为正，另一部分为负。该特征向量被称为菲德勒向量。邻接矩阵非负，且此类矩阵存在广泛理论 [13]；参见 14.4 节的佩龙 - 弗罗贝尼乌斯定理。

一般而言，无法保证这种分解在边数比例衡量下的优劣程度，但实践表明其表现相当良好 [176]。

20.6.3 切格尔不等式

前文提到，图拉普拉斯算子的首个非平凡特征值与图的二分划分存在关联。切格尔常数与切格尔不等式将此特征值与划分质量的特定度量联系起来。

设 V 为顶点集，$S \subset V$，则图的切格尔常数定义为

$$C = \min_{S} \frac{e(S, V-S)}{\min \operatorname{vol}(S), \operatorname{vol}(V-S)}$$

其中 $e(S, V-S)$ 表示连接 S 与 $V-S$ 的边数，而节点集的体积定义为

$$\operatorname{vol}(S) = \sum_{e \in S} d(e).$$

切格尔不等式则表明

$$2C \geq \lambda \geq \frac{C^2}{2}$$

其中 λ 是图拉普拉斯算子的第一个非平凡特征值。

第 21 章

自动机理论

自动机是机器的数学抽象。自动机理论内容广泛，此处我们仅探讨基本概念。让我们从一个简单例子开始。

21.1 有限状态自动机

有限状态自动机（*FSA*）是一种非常简单的机器，类似于投入 25 美分硬币就会分发糖果的自动售货机。自动售货机有四种可能操作：投入硬币、按下 ‘ 退币 ’ 键要求返还已投入的钱币、打开窗口取出糖果棒、再次关闭窗口。操作是否可行（尤其是第三种）取决于机器所处的状态。共有三种状态：初始状态、已投入硬币且窗口解锁的 ‘ 待分发 ’ 状态、以及窗口打开的 ‘ 分发中 ’ 状态。

在特定状态下，某些操作不可行。例如，初始状态下无法打开窗口。

这台自动售货机的数学描述包含两部分：1. 状态列表；2. 一个展示可能动作如何使机器从一个状态转移到另一个状态的表格。不过，相较于书写表格，图形化表示通常更具启发性。

21.2 总体讨论

通过自动售货机的例子，你观察到了自动机的一个重要特征：它允许特定动作，但仅在某些条件下执行，最终会达到一个对应 ‘ 成功 ’ 的状态 —— 该状态仅当采取特定动作序列时才能抵达。

该过程的正式描述如下：我们将单个动作称为 ‘ 字母表 ’，基于该字母表的动作序列称为 ‘ 单词 ’。机器的 ‘ 成功 ’ 结果对应于判定某个单词属于该自动机接受的 ‘ 语言 ’。因此，自动机理论与语言理论之间存在对应关系。

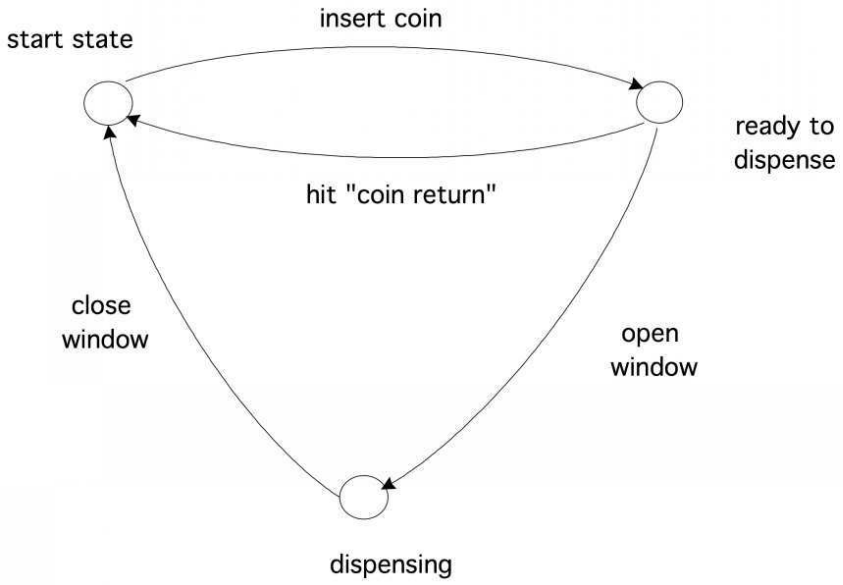


图 21.1：一个简单的现实生活中的自动机。

练习 21.1. 考虑字母表 $\{a, b\}$，即仅包含字母 a 、 b 的语言，并考虑语言 $\{a^m b^n : m, n > 0\}$，即由一个或多个 a 后接一个或多个 b 组成的单词。画出接受该语言的自动机。

有限状态自动机之所以是最简单的类型，是因为它没有记忆功能。大多数自动售货机不会因为你投入超过一枚 25 美分硬币而抱怨：它们除了 ‘已投入一枚硬币’ 外没有其他记忆。更复杂的机器会计算你投入了多少枚硬币，然后允许你打开相应数量的窗口选择不同糖果棒。用上述形式化方式描述，该机器将接受语言 $\{q^n w^n : n \geq 0\}$，即你存入多少枚 ‘q’ 硬币就打开多少个 ‘w’ 窗口的序列。这种语言是所谓的上下文无关语言的一个例子；原始自动售货机的语言则是正则语言。

这两种语言类型属于四层乔姆斯基层级的语言体系。著名的图灵机位于该层级的顶端，它能识别递归可枚举语言类型。缺失的层级对应的是上下文相关语言类型，由线性有界自动机进行识别。

第 22 章

并行前缀

要使操作能够并行执行，它们必须是相互独立的。这使得递归问题在并行评估时变得棘手。递归不仅出现在诸如解三角方程组（见章节 5.3.4 ）等明显场景中，还可能出现在排序及许多其他操作里。

在本附录中，我们将探讨并行前缀操作：通过涉及结合运算符的递推关系定义的并行运算。（另见 7.10.2 节中关于递推关系并行化的 “ 递归倍增 ” 方法。）计算元素数组的和便是此类操作的一个实例（暂时忽略非结合性）。设 $\pi(x,y)$ 为二元求和运算符：

$$\pi(x,y) \equiv x+y,$$

那么我们定义 $n \geq 2$ 项的前缀和为

$$\Pi(x_1,\ldots,x_n) = \begin{cases} \pi(x_1,x_2) & \text{if } n=2 \\ \pi\big(\Pi(x_1,\ldots,x_{n-1}),x_n\big) & \text{otherwise} \end{cases}$$

作为一个非显而易见的例子，我们可以利用前缀操作来计算数组中具有特定属性的元素数量。

练习 22.1. 设 $p(\cdot)$ 为一个谓词，若对 x 成立则 $p(x)=1$ 为 1，否则为 0。定义一个二元运算符 $\pi(x,y)$，使其在数字数组上的归约结果即为满足 p 为真的元素个数。

现在假设存在一个结合运算符 $\oplus$，以及值数组 $x_1,\ldots,x_n$。那么 e
将前缀问题定义为计算 $X_1,\ldots,X_n$，其中

$$\begin{cases} X_1 = x_1 \\ X_k = \oplus_{i\leq k} x_i \end{cases}$$

22.1 并行前缀

并行化前缀问题的关键在于认识到我们可以并行计算部分归约结果：

$$x_1 \oplus x_2, \quad x_3 \oplus x_4, \ldots$$

都是相互独立的。此外，这些归约的部分归约，

$$(x_1 \oplus x_2) \oplus (x_3 \oplus x_4), \quad \ldots$$

也是相互独立的。我们使用记号

$$X_{i,j} = x_i \oplus \cdots \oplus x_j$$

来表示这些部分归约。

你已在第 2.1 节见过此操作：一个包含 n 个数字的数组可在 $\lceil \log_2 n \rceil$ 步骤内完成归约。求和算法若要成为完整的前缀运算，缺失的是对所有中间值的计算。

Observing that, for instance, $X_3 = (x_1 \oplus x_2) \oplus x_3 = X_2 \oplus x_3$, you can now imagine the whole process; see figure 22.1 for the case of 8 elements. To compute, say, X_{13}, you express $13 = 8 + 4 + 1$ and compute

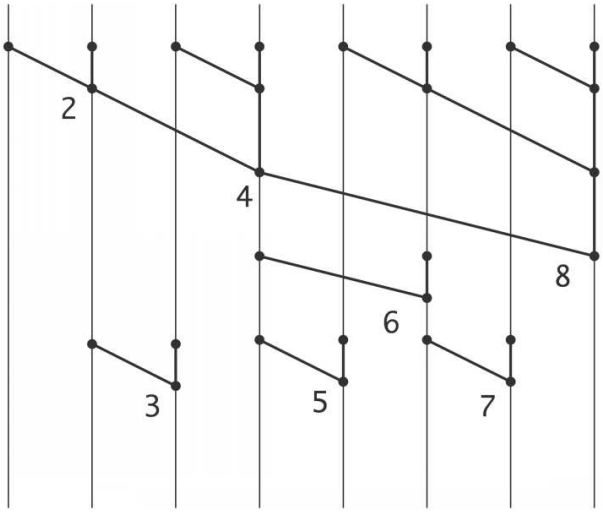


图 22.1：应用于 8 个元素的前缀运算。

$X_{13} = X_8 \oplus X_{9,12} \oplus x_{13}$.

在此图中，相同 ‘距离’ 上的操作已垂直对齐，对应于 SIMD 类型的执行。若执行过程采用任务图推进，某些步骤可比图中所示更早执行；例如 X_3 可与 X_6 同时计算。

无论计算步骤如何排列，不难看出整个前缀计算可在 $2\log_2 n$ 步内完成：$\log_2 n$ 步用于计算最终归约 X_n，再以 $\log_2 n$ 步填充中间值。

22.2 稀疏矩阵向量积作为并行前缀

已有研究表明，稀疏矩阵向量积可被视为前缀操作；参见 [16]。其原理在于先计算所有 $y_{ij} \equiv a_{ij}x_j$，随后用 $y_i = \sum_j y_{ij}$ 计算求和

一个前缀运算。

如上所述的前缀和计算无法得到正确结果。前几个 y_{ij} 项确实求和得到 y_1，但继续计算前缀和会得到 $y_1 + y_2$ 而非 y_2。解决这个问题的技巧在于考虑二元组 $\langle y_{ij}, s_{ij} \rangle$，其中

$$s_{ij} = \begin{cases} 1 & \text{若 } j \text{ 是行中首个} \\ 0 & \text{非零索引 否则} \end{cases}$$

现在我们可以定义每次 $s_{ij} = 1$ 时被 ‘重置’ 的前缀和。

22.3 霍纳法则

Horner's rule 用于评估多项式的算法是简化 e recurrence:

$$y = c_0 x^n + \cdots + c_n x^0 \equiv \begin{cases} t_0 \leftarrow c_0 \\ t_i \leftarrow t_{i-1} \cdot x + c_i \quad i = 1, \ldots, n \\ y = t_n \end{cases} \tag{22.1}$$

或者，更明确地表述为

$$y = (((c_0 \cdot x + c_1) \cdot x + c_2) \cdots).$$

与许多其他递推关系类似，这个看似顺序的操作实际上可以并行化：

$$\begin{array}{cccc} c_0 x + c_1 & & c_2 x + c_3 & & c_4 x + c_5 & & c_6 x + c_7 \\ & \cdot \times x^2 + \cdot & & & & \cdot \times x^2 + \cdot & \\ & & & \cdot \times x^4 + \cdot & & & \end{array}$$

然而，这里我们需要一些巧思：需要 x、 x^2、 x^4 等来对子结果进行乘法运算。

将霍纳法则解释为前缀方案是行不通的：可以轻易看出 ' 霍纳运算符 ' $h_x(a, b) = ax + b$ 不满足结合律。从上述树状计算中可见，我们需要携带并更新 x，而非将其附加到运算符上。

稍加实验表明

$$h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}\right) \equiv \begin{bmatrix} ay + b \\ xy \end{bmatrix}$$

符合我们的目的：

$$h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right) = \left\{ \begin{array}{lcl} h\left(h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}\right), \begin{bmatrix} c \\ z \end{bmatrix}\right) & = & h\left(\begin{bmatrix} ay + b \\ xy \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right) \\ h\left(\begin{bmatrix} a \\ x \end{bmatrix}, h\left(\begin{bmatrix} b \\ y \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right)\right) & = & h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} bz + c \\ yz \end{bmatrix}\right) \end{array} \right\} = \begin{bmatrix} ayz + bz + c \\ xyz \end{bmatrix}$$

据此，我们可以实现霍纳法则如下

$$h\left(\begin{bmatrix} c_0 \\ x \end{bmatrix}, \begin{bmatrix} c_1 \\ x \end{bmatrix}, \ldots, \begin{bmatrix} c_n \\ x \end{bmatrix}\right)$$

值得一提的是，这种特殊形式的 ' 霍纳算子 ' 对应于编程语言 *APL* 中的 'rho' 算子，通常被表述为在可变基数数制中的求值运算。

22. 并行前缀

第四部分

项目与代码

第 23 章

类项目

以下是一些结合编码与分析的期末项目建议。

第 24 章

教学指南

本书内容远超一个学期课程所能涵盖的范围。以下是几种教学策略：可作为独立课程授课，或将其内容整合到其他课程中。

24.1 独立课程

24.2 并行编程类补充材料

建议的教学内容与练习：

并行分析：2.12，2.13，2.16。

架构：2.32，2.33，2.35，2.37。

复杂度：2.38

24.3 教程

24.4 缓存模拟与分析

在本项目中，您将构建一个缓存模拟器，并分析代码（无论是真实还是模拟的）的缓存命中 / 未命中行为。

24.4.1 缓存模拟

模拟缓存是一种简单的数据结构，用于记录每个缓存地址所对应的内存地址以及数据已存在的时间。请设计该数据结构并编写访问例程。（建议使用面向对象语言实现。）

编写代码时需确保缓存支持不同级别的关联性及多种替换策略。

F为简化设计，无需区分读写访问。因此，程序执行过程将转化为
b连续的访问流

```
cache.access_address( 123456 );
cache.access_address( 70543 );
cache.access_address( 12338383 );.....
```

调用时参数为内存地址。您的代码需记录该请求能否从缓存中满足，或是否需要从内存加载数据。

24.4.2 代码模拟

寻找一些示例代码（例如您参与的科学项目中的代码），并将其转换为内存地址序列。

您也可以通过生成如上的访问指令流来模拟代码：

- 部分访问将针对随机位置，对应于标量变量的使用；
- 其他时候访问将针对规律间隔的地址，对应于数组的使用；
- 采用间接寻址的数组操作会导致长时间的随机地址访问序列。

尝试混合上述不同类型的指令。对于数组操作，可尝试不同复用程度的小型及大型数组。

您的模拟代码是否表现出真实代码的特性?

24.4.3 调研分析

首先实现单级缓存并研究缓存命中与未命中的行为。探索不同的关联度设置和不同的替换策略。

24.4.4 分析

对缓存命中 / 未命中行为进行统计分析。您可以从 [164][1]开始。Hartstein [96] 发现了幂律行为。您是否得出了相同结论?

1. 严格来说，该论文讨论的是虚拟内存中的页面交换（章节 1.3.9.2），但所有原理均可转用于主内存中的缓存行交换。

24.5 批量同步编程的评估

在本项目中，你需要从抽象和模拟两个角度分析批量同步并行（ BSP ）模型。

24.5.1 讨论

你的作业需要研究批量同步编程。阅读维基百科文章
http://en.wikipedia.org/wiki/Bulk_synchronous_parallel 和书中 2.6.8 章节。

Consider the following questions.

1. 讨论 BSP 复杂度模型与 α、 β、 γ 模型的关系，该模型你在 7.1 节学过。2. BSP 使用屏障同步。这些在实践中是否必要？分别考虑两类不同算法：偏微分方程求解和大规模图计算。

24.5.2 模拟

在关于 BSP 的定义性论文中，Valiant [182] 主张创建比处理器数量更多的任务，并随机分配它们。编写一个模拟程序并测试该策略是否能解决负载均衡问题。结合不同算法讨论随机分配策略。

24.6 热方程

在本项目中，你将结合在课堂上学到的部分理论和实践技能来解决一个现实世界的问题。除了编写解决该程序的代码外，你还将运用以下软件实践：

- 使用源代码控制，
- 为程序提供检查点 / 重启功能，以及
- 使用基本性能分析和结果可视化。

热传导方程（参见章节 4.3 ）由下式给出：

$$\frac{\partial T(x,t)}{\partial t} = \begin{cases} \alpha \frac{\partial^2 T(x,y)}{\partial x^2} + q(x,t) & \text{1D} \\ \alpha \frac{\partial^2 T(x,y)}{\partial x^2} + \alpha \frac{\partial^2 T(x,y)}{\partial y^2} + q(x,t) & \text{2D} \\ \ldots & \text{3D} \end{cases}$$

其中 $t \geq 0$ 和 $x \in [0,1]$，受限于边界条件

$$T(x,0) = T_0(x), \quad \text{for } x \in [0,1],$$

在更高维度中也是如此，

$$T(0,t) = T_a(t),\ T(1,t) = T_b(t), \quad \text{for } t > 0.$$

你将使用显式欧拉方法和隐式欧拉方法来解决这个问题。

24.6.1 软件

就偏微分方程而言，这是一个相当简单的例子。特别是其规则结构使得使用常规数组存储数据来编写此项目变得容易。不过，你也可以使用 PETSc 库来编写软件。在这种情况下，矩阵向量乘法请使用 `MatMult` 例程，线性系统求解则使用 `KSPSolve` 。异常：欧拉方法需自行编码。

务必使用 Makefile 来构建你的项目（教程教程手册，第 3 节）。

将你的源文件、 Makefile 和作业脚本添加到 git 代码仓库中（教程教程手册，第 5 节）；不要添加二进制文件或输出文件。确保包含一个 README 文件，其中说明如何构建和运行你的代码。

通过将向量数据、时间步长大小和其他必要项写入 `hdf5` 文件来实现检查点 / 重启功能（教程教程手册，第 7 节）。你的程序应能读取该文件并恢复执行。

24.6.2 测试

在单核上执行以下测试。

方法稳定性

运行你的程序处理一维情况，使用

$$\begin{cases} q = \sin \ell \pi x \\ T_0(x) = e^x \\ T_a(t) = T_b(t) = 0 \\ \alpha = 1 \end{cases}$$

空间离散化至少为 $h = 10^{-2}$，但在并行计算中尤其不要害怕尝试大规模问题。尝试不同的时间步长，并展示显式方法可能发散的情况。其保持稳定的最大时间步长是多少?

对于隐式方法，首先使用直接法来求解系统。这对应于 PETSc 选项 `KSPPREONLY` 和 `PCLU` （参见章节 5.5.11 ）。

现在使用迭代方法（例如 `KSPCG` 和 `PCJACOBI` ）；方法是否仍然稳定? 尝试使用较低的收敛容差和大时间步长进行探索。

由于外力函数 q 和边界条件均与时间无关，解 $u(\cdot, t)$ 将随着 $t \to \infty$ 趋近于一个稳态解 $u_\infty(x)$。时间步长对隐式方法收敛至该稳态的速度有何影响?

提示：稳态由 $u_t \equiv 0$ 描述。将其代入偏微分方程中。能否显式求出该稳态解?

使用不同的 ℓ 值运行这些测试。

时序

若使用命令行选项 `-log_summary` 运行代码，将获得各 PETSc 例程的计时表格。利用该功能完成以下计时实验。请确保使用的 PETSc 版本未以调试模式编译。

将系数矩阵构建为稠密矩阵（而非稀疏矩阵）。报告总内存占用的差异，以及单次时间步长的运行时间和浮点运算速率。分别对显式方法和隐式方法进行测试，并对结果进行解释。

针对稀疏系数矩阵，报告单次时间步长的耗时。分别讨论浮点运算次数及由此产生的性能表现。

重启

实现重启功能：每 10 次迭代输出当前迭代值，连同 Δx、 Δt 和 ℓ 的值。在程序中添加标志 `-restart` ，使其能读取重启文件并恢复执行，所有参数均从重启文件中读取。

运行程序 25 次迭代后进行重启，使其从第 20 次迭代重新运行。验证 $20 \ldots 25$ 次迭代中的数值是否匹配。

24.6.3 并行性

执行以下测试以确定代码的并行扩展性。

首先测试显式方法，该方法应具备完美的并行性。报告实际达到的加速比。尝试不同规模的问题（较大和较小），并报告问题规模对结果的影响。

上述隐式方法的设置（ `KSPPREONLY` 和 `PCLU` ）会导致运行时错误。一种解决方案是让系统通过迭代方法求解。查阅 PETSc 手册及网页，了解迭代方法和预条件器的可选方案并进行尝试。报告其有效性。

24.6.4 求解器比较

在隐式时间步进方法中，你需要求解线性系统。在二维（或三维）情况下，使用直接求解器还是迭代求解器可能会产生很大差异。

将你的代码设置为在 $q \equiv 0$ 且边界条件处处为零的情况下运行；从一个非零初始解开始。现在你应该会收敛到一个零稳态，因此当前解的范数就是迭代的范数。

现在进行以下比较；取几个时间步长的值。

直接求解器

如果你的 PETSc 安装包含直接求解器如 *MUMPS*，你可以通过以下方式调用它们

```
myprog -pc_type lu -ksp_type preonly \
    -pc_factor_mat_solver_package mumps
```

运行您的代码，分别采用直接求解器的串行和并行模式，并记录误差降至 10^{-6}所需的时间。

迭代求解器

使用迭代求解器，例如 `KSPCG` 和 `KSPBCGS`。尝试不同的收敛容差：若将迭代方法容差设为 10^{-12}，需多少时间步才能获得 10^{-6} 的误差？若采用更小的容差，结果又如何？

比较直接求解法与迭代方法的耗时 od.

24.6.5 报告撰写

使用 LaTeX 撰写报告（教程教程书籍第 *15* 节）。通过表格与图表呈现实测数据。运用 `gnuplot` （教程教程书籍第 *9* 节）或类似工具绘制图表。

24.7 内存墙

在本项目中，您将探索内存墙的影响；参见第 1.3 节。该项目涉及文献检索，但编程内容较少。

24.7.1 背景

Wulf 和 McKee [194] 观察了处理器平均延迟的趋势。设 t_m 为内存访问延迟，t_c 为缓存访问延迟，p 为缓存命中概率，则平均延迟为

$$t_{\text{avg}} = pt_c + (1 - p)t_m.$$

随着处理器速度与内存速度差距的扩大，除非能降低 p （即减少缓存未命中次数或至少减轻其影响），否则这种延迟将持续增长。

24.7.2 作业

进行文献检索并讨论以下主题。

- 处理器速度和内存速度的趋势如何？多核芯片的引入对这种平衡有何影响？
- 章节 1.3.5.4 讨论了各种类型的缓存未命中。有些未命中与算法更相关，有些则与硬件更相关。硬件设计师采用了哪些策略来减少缓存未命中的影响？
- 强制性缓存未命中似乎是不可避免的，因为它们是算法的属性。然而，如果它们可以被隐藏（参见章节 1.3.2 关于 ‘ 延迟隐藏 ’ ），它们的性能影响就会消失。研究预取流及其与延迟隐藏的关系。
- 缓存行大小如何影响这一行为？
- 探讨减少缓存未命中发生率的编译技术。
- 你能找到算法选项的例子吗？也就是说，那些能计算出相同结果（不一定在算术意义上；比如直接平方根与迭代近似法的对比）但具有不同计算行为的算法？

针对上述所有主题，建议您动手编写模拟程序进行实践。

第 25 章

代码

本节包含几个简单代码示例，用以说明与单 CPU 性能相关的各类问题。具体解释可参阅第 6 章。

25.1 预备知识

25.1.1 硬件事件计数

本章代码示例调用了名为 PAPI （'性能应用程序编程接口'）的库 [23, 158]。这是一组可移植的调用接口，用于查询大多数处理器内置的硬件计数器。由于这些计数器属于处理器硬件的一部分，它们可以测量诸如缓存未命中等细节事件，且不会因测量过程干扰本应观察的现象。

虽然使用硬件计数器相当直接，但它们报告的内容是否确实是您想要测量的，则完全是另一回事。例如，硬件预取流的存在（章节 1.3.6 ）意味着数据可以在未被缓存未命中触发的情况下加载到缓存中。因此，计数器报告的数字在简单解读下可能显得异常，甚至不可能。

25.1.2 测试设置

在接下来的章节中，您将看到几个尝试对某些现象进行精确计时的代码。如果您想确保实际测量的是您感兴趣的内容而非其他，这需要格外小心。

首先，当然您需要多次执行实验以获得良好的平均计时。如果个别计时结果相差过大，您可以决定剔除异常值，或者重新考虑实验设计：可能存在您未考虑到的变量。

假设您的计时结果都在可接受的范围内；接下来您需要补偿因多次重复实验而产生的缓存效应。如果问题规模较小，第一次计时会将所有数据载入缓存，第二次运行时数据仍在缓存中，从而导致 ...

更快的运行时间。您可以通过在任意两次计时之间访问一个大于缓存大小的数组来解决此问题。这会起到清空缓存中所有实验数据的效果。

另一方面，如果您想对缓存中数据的操作进行计时，则需要确保数据已加载到缓存中。例如，您可以在实验前对数组进行写入操作，从而将其加载到缓存中。这有时被称为缓存预热，而包含问题数据的缓存被称为热缓存。未能预热缓存会导致首次运行耗时明显长于后续运行的结果。

25.1.3 内存问题

出于性能考虑，通常需要强制数据实现缓存行边界对齐。最简单的解决方案是使用静态分配，这会将数据放置在缓存行边界上。

要将动态对齐的数据放置在缓存行边界上，以下代码可实现这一目的：

```
double *a;a = malloc( /* some number of bytes */ +8 );
if ( (int)a % 8 != 0 ) { /* it is not 8-byte aligned */
a += 1; /* advance address by 8 bytes, then */
/* either: */a = ( (a>>3) <<3 );/* or: */
a = 8 * ( ( (int)a )/8 );}
```

这段代码会分配一块内存，并在必要时将其右移，以确保起始地址是 8 的倍数。

然而，更好的解决方案是使用 *posix_memalign*：

```
int posix_memalign(void **memptr,  size_t alignment,
 size_t size);
```

该操作会分配 `size` 字节的内存，并按 `alignment` 字节的倍数对齐。例如 e:

```
double x;
posix_memalign( (void**)&x,64,N*sizeof(double) );
```

将为 `x` 分配 64 个双精度浮点数，并在每个缓存行包含 8 个字时对齐数组至缓存行边界。

(若 *Intel* 编译器提示 `posix_memalign` 被隐式声明，请在编译行添加 `-std=gnu99` 标志。)

25.2 缓存大小

此代码验证了一个事实：若数据位于 L1 缓存中，其操作效率高于位于 L2 、 L3 缓存或主内存的情况。为确保不测量任何非预期的数据移动，我们在启动计时器前先执行一次迭代以将数据预加载至缓存。

`size.c`

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */
#define PCHECK(e) \if (e!=PAPI_OK) \
 {printf("Problem in papi call, line %d\n",__LINE__); return 1;}
#define NEVENTS 3define NRUNS 200define L1WORDS 8096
#define L2WORDS 100000nt main(int argc, char **argv)
#int events[NEVENTS] ={PAPI_TOT_CYC,/* total cycles */PAPI_L1_DCM,
#
  /* stalls on L1 cache miss */PAPI_L2_DCM,
i /* stalls on L2 cache miss */};long_long values[NEVENTS];
{PAPI_event_info_t info, info1;const PAPI_hw_info_t *hwinfo = NULL;
 int retval,event_code, m,n, i,j,size, arraysize;
 const PAPI_substrate_info_t *s = NULL;double *array;
 tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
 retval = PAPI_library_init(PAPI_VER_CURRENT);
 if (retval != PAPI_VER_CURRENT)
 test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);{int i;
 for (i=0; i<NEVENTS; i++) {
 retval = PAPI_query_event(events[i]) ; PCHECK(retval);}}
```

```
  /* declare an array that is more than twice the L2 cache size */
  arraysize=2*L2WORDS;
  array = (double*) malloc(arraysize*sizeof(double));

  for (size=L1WORDS/4; size<arraysize; size+=L1WORDS/4) {
    printf("Run: data set size=%d\n",size);

    /* clear the cache by dragging the whole array through it */
    for (n=0; n<arraysize; n++) array[n] = 0.;
    /* now load the data in the highest cache level that fits */
    for (n=0; n<size; n++) array[n] = 0.;

    retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
    /* run the experiment */
    for (i=0; i<NRUNS; i++) {
      for (j=0; j<size; j++) array[j] = 2.3*array[j]+1.2;
```

```
    }retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
  printf("size=%d\nTot cycles: %d\n",size,values[0]);
  printf("cycles per array loc: %9.5f\n",size,values[0]/(1.*NRUNS*size));
  printf("L1 misses:\t%d\nfraction of L1 lines missed:\t%9.5f\n",values[1],
  values[1]/(size/8.));
  printf("L2 misses:\t%d\nfraction of L2 lines missed:\t%9.5f\n",values[2],
  values[2]/(size/8.));printf("\n");}free(array);return 0;}
```

我们还通过反复遍历链表数组来测量性能。该数组的创建方式如下：

```
// hardware/allocation.cpp
std::iota(indices.begin(),indices.end(),0);
std::random_device r;
std::mt19937 g(r());
std::shuffle(indices.begin(), indices.end(), g);
auto data = thecache;
for (size_t i=0; i<indices.size(); i++)
  data[i] = indices[i];
```

并按如下方式遍历：

```
// hardware/allocation.cpp
auto data = thecache;
for (size_t i=0; i<n_accesses; i++) {
  res = data[res];
  /* ... */
}
```

25.3 缓存行

这段代码说明了向量代码中小步长的重要性。主循环对向量进行操作，并以恒定步长推进。随着步长增大，运行时间会增加，因为传输的缓存行数量增多，而带宽成为计算的主要成本。

这段代码存在一些微妙之处：为了防止意外重用缓存中的数据，在计算前会先执行一个循环，该循环访问的数据量至少是缓存容量的两倍。因此，可以确保数组不会留在缓存中。

line.c

```
#include "papi_test.h"
extern int TESTS_QUIET;         /* Declared in test_utils.c */

#define PCHECK(e) \
```

```
  if (e!=PAPI_OK) \
    {printf("Problem in papi call, line %d\n",__LINE__); return 1;}
#define NEVENTS 4
#define MAXN 10000
#define L1WORDS 8096
#define MAXSTRIDE 16

int main(int argc, char **argv)
{
  int events[NEVENTS] =
    {PAPI_L1_DCM,/* stalls on L1 cache miss */
     PAPI_TOT_CYC,/* total cycles */
     PAPI_L1_DCA, /* cache accesses */
     1073872914 /* L1 refills */};
  long_long values[NEVENTS];
  PAPI_event_info_t info, info1;
  const PAPI_hw_info_t *hwinfo = NULL;
  int retval,event_code, m,n, i,j,stride, arraysize;
  const PAPI_substrate_info_t *s = NULL;
  double *array;

   tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
   retval = PAPI_library_init(PAPI_VER_CURRENT);
   if (retval != PAPI_VER_CURRENT)
      test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);
   {
     int i;
     for (i=0; i<NEVENTS; i++) {
       retval = PAPI_query_event(events[i]) ; PCHECK(retval);
     }
   }

   /* declare an array that is more than twice the cache size */
   arraysize=2*L1WORDS*MAXSTRIDE;
   array = (double*) malloc(arraysize*sizeof(double));

   for (stride=1; stride<=MAXSTRIDE; stride++) {
     printf("Run: stride=%d\n",stride);
     /* clear the cache by dragging the whole array through it */
     for (n=0; n<arraysize; n++) array[n] = 0.;

     retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
     /* run the experiment */
     for (i=0,n=0; i<L1WORDS; i++,n+=stride) array[n] = 2.3*array[n]+1.2;
     retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
     printf("stride=%d\nTot cycles: %d\n",stride,values[1]);
     printf("L1 misses:\t%d\naccesses per miss:\t%9.5f\n",
    values[0],(1.*L1WORDS)/values[0]);
     printf("L1 refills:\t%d\naccesses per refill:\t%9.5f\n",
    values[3],(1.*L1WORDS)/values[3]);
     printf("L1 accesses:\t%d\naccesses per operation:\t%9.5f\n",
    values[2],(1.*L1WORDS)/values[2]);
     printf("\n");
```

```
    }
    free(array);

    return 0;
}
```

25.4 缓存关联性

此代码展示了缓存关联性的影响；详见章节 1.3.5.10 和 6.5 中的详细解释。多个向量（取决于内层循环变量 `i`）被同时遍历。其长度经过刻意设置以引发缓存冲突。若向量数量足够低，缓存关联性将解决这些冲突；当 `m` 取值较大时，运行时间会急剧增加。通过分配更大尺寸的向量，缓存冲突即会消失。

assoc.c#include "papi_test.h"

```
extern int TESTS_QUIET;         /* Declared in test_utils.c */
#define PCHECK(e) if (e!=PAPI_OK) \
{printf("Problem in papi call, line %d\n",__LINE__); \return 1;}
#define NEVENTS 2#define MAXN 20000
/* we are assuming array storage in C row mode */#if defined(SHIFT)
#define INDEX(i,j,m,n) (i)*(n+8)+(j)#else
#define INDEX(i,j,m,n) (i)*(n)+(j)#endif
int main(int argc, char **argv){
int events[NEVENTS] = {PAPI_L1_DCM,PAPI_TOT_CYC};
long_long values[NEVENTS];PAPI_event_info_t info,  info1;
const PAPI_hw_info_t *hwinfo = NULL;int retval, event_code,  m, n,  i,
j;const PAPI_substrate_info_t *s = NULL;double *array;
tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
retval = PAPI_library_init(PAPI_VER_CURRENT);
if (retval != PAPI_VER_CURRENT)
test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);{int i;
for (i=0; i<NEVENTS; i++) {
retval = PAPI_query_event(events[i]); PCHECK(retval);}
```

```
    }
    /*
    if (argc<3) {
      printf("Usage: assoc m n\n"); return 1;
    } else {
      m = atoi(argv[1]); n = atoi(argv[2]);
    } printf("m,n = %d,%d\n",m,n);
    */

#if defined(SHIFT)
array = (double*) malloc(13*(MAXN+8)*sizeof(double));#else
array = (double*) malloc(13*MAXN*sizeof(double));#endif
/* clear the array and bring in cache if possible */
for (m=1; m<12; m++) {for (n=2048; n<MAXN; n=2*n) {
printf("Run: %d,%d\n",m,n);#if defined(SHIFT)
printf("shifted\n");#endiffor (i=0; i<=m; i++)
for (j=0; j<n; j++)array[INDEX(i,j,m+1,n)] = 0.;
/* access the rows in a way to cause cache conflicts */
retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);
for (j=0; j<n; j++)for (i=1; i<=m; i++)
array[INDEX(0,j,m+1,n)] += array[INDEX(i,j,m+1,n)];
retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
printf("m,n=%d,%d\n#elements:\t%d\nTot cycles: %d\n",m,n,m*n,
values[1]);
printf("L1 misses:\t%d\nmisses per accumulation:\t%9.5f\n\n",
values[0],values[0]/(1.*n));}}free(array);return 0;}
```

25.5 TLB

此代码展示了 *TLB* 的行为模式；详细解释请参阅章节 1.3.9.2 与 6.4.4 。一个二维数组以列主序（ Fortran 风格）声明。这意味着通过变化 i 坐标来遍历数据时，由于所有元素

当页面上的元素被连续访问时，访问的 TLB 条目数量等于元素数量除以页面大小。若按 j 坐标跨步遍历数组，每个后续元素都会触及新页面，因此当列数超过 TLB 条目数量时，将引发 TLB 缺失。

tlb.c#include "papi_test.h"

```
extern int TESTS_QUIET;         /* Declared in test_utils.c */
#define PCHECK(e) if (e!=PAPI_OK) \
{printf("Problem in papi call, line %d\n",__LINE__); \return 1;}
#define NEVENTS 2
/* we are assuming array storage in Fortran column mode */
#define INDEX(i,j,m,n) i+j*mdouble *array;
void clear_right(int m,int n) {int i, j;for (j=0; j<n; j++)
for (i=0; i<m; i++)array[INDEX(i,j,m,n)] = 0;return;}
void clear_wrong(int m,int n) {int i, j;for (i=0; i<m; i++)
for (j=0; j<n; j++)array[INDEX(i,j,m,n)] = 0;return;}
void do_operation_right(int m,int n) {int i, j;
for (j=0; j<n; j++)for (i=0; i<m; i++)
array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;return;}
void do_operation_wrong(int m,int n) {int i, j;
for (i=0; i<m; i++)for (j=0; j<n; j++)
array[INDEX(i,j,m,n)] = array[INDEX(i,j,m,n)]+1;return;}
#define COL 1#define ROW 2int main(int argc, char **argv)
```

25. 代码

```
{int events[NEVENTS] = {PAPI_TLB_DM,PAPI_TOT_CYC};long_long values[NEVENTS];
int retval, order=COL;PAPI_event_info_t info,  info1;
const PAPI_hw_info_t *hwinfo = NULL;int event_code;
const PAPI_substrate_info_t *s = NULL;
tests_quiet(argc, argv);     /* Set TESTS_QUIET variable */
if (argc==2 && !strcmp(argv[1],"row")) {printf("wrong way\n"); order=ROW;
} else printf("right way\n");retval = PAPI_library_init(PAPI_VER_CURRENT);
if (retval != PAPI_VER_CURRENT)
test_fail(__FILE__, __LINE__, "PAPI_library_init", retval);{int i;
for (i=0; i<NEVENTS; i++) {
retval = PAPI_query_event(events[i]); PCHECK(retval);}}#define M 1000
#define N 2000{int m, n;m = M;array = (double*) malloc(M*N*sizeof(double));
for (n=10; n<N; n+=10) {if (order==COL)clear_right(m,n);elseclear_wrong(m,n);
retval = PAPI_start_counters(events,NEVENTS); PCHECK(retval);if (order==COL)
do_operation_right(m,n);elsedo_operation_wrong(m,n);
retval = PAPI_stop_counters(values,NEVENTS); PCHECK(retval);
printf("m,n=%d,%d\n#elements:\t%d\n",m,n,m*n);
printf("Tot cycles: %d\nTLB misses:\t%d\nmisses per column:\t%9.5f\n\n",
values[1], values[0], values[0]/(1.*n));}free(array);}return 0;}
```

第五部分

INDICES

第 26 章

索引

第 27 章

缩略语列表

TLB 转换后备缓冲区

第 28 章

参考文献

[1] IEEE 754-2019 浮点运算标准。 *IEEE Std 754-2019 (Revisionof IEEE 754-2008)*, 第1–84页，2019年。 [Cited 见于 157、 161 及 167 页。]]

[2 Loyce M. Adams 与 Harry F. Jordan。 SOR 方法是否色盲？ *SIAM J. Sci. Stat. Comput.*, 7:490–506，1986 年。 [Cited 见于 316 页。]

[3] Sarita V. Adve 与 Hans-J. Boehm。 内存模型：重新思考并行语言与硬件的案例。 *Communications of the ACM*, 53:90–101。 [Cited 见于 91 页。]

[4] G. 阿姆达尔。单处理器方法实现大规模计算能力的有效性。载于 *AFIPS* 计算会议论文集，第 30 卷，第 483–485 页，1967 年。 [Cited on 页 65。]

[5] O. Axelsson and A.V. Barker. 有限元边界值问题解法。理论与计算 . Academic Press, Orlando, Fl., 1984. [Cited on page 259.]

[6] Owe Axelsson and Ben Polman. 块预处理与区域分解方法 II. *J.Comp. Appl. Math.*, 24:55–72, 1988. [Cited on page 325.]

[7] David H. Bailey. 向量计算机存储体冲突研究 . *IEEE* 计算机汇刊 , C-36:293–298, 1987 年 . [Cited 见第 37 页]

[8] Josh Barnes 和 Piet Hut。一种分层 $O(N$ 对数 $N)$ 力计算算法。自然 , 324:446–449, 1986 年。 [Cited 见第 389 页]

[9] Richard Barrett 、 Michael Berry 、 Tony F. Chan 、 James Demmel 、 June Donato 、 Jack Dongarra 、 Victor Eijkhout 、 Roldan Pozo 、 Charles Romine 和 Henk van der Vorst。线性系统求解模板：迭代方法的构建模块。SIAM，费城，1994 年。`http://www.netlib.org/templates/`。 [Cited on page 261。]

[10] K.E. Batcher。MPP：一种高速图像处理器。载于算法特化并行计算机。学术出版社，纽约，1985 年。 [Cited 第 76 页。]

[11] Robert Beauwens 和 Mustafa Ben Bouzid。稀疏近似块分解的存在性与条件性质。*SIAM* 数值分析，25:941–956，1988 年。 [Cited 第 260 页。]

[12] Gordon Bell。可扩展并行处理的展望。决策资源公司，1994 年。 [Cited on 第 72 页。]

[13] Abraham Berman and Robert J. Plemmons. 数学科学中的非负矩阵。SIAM, 1994. originally published by Academic Press, 1979, New York. [Cited 见于第 202 和 449 页。]

[14] Petter E. Bjorstad, William Gropp, and Barry Smith. 区域分解：椭圆型偏微分方程的并行多级方法 . Cambridge University Press, 1996. [Cited on page 319。]

[15] 费希尔 · 布莱克与迈伦 ·S· 斯科尔斯。期权与公司负债的定价。政治经济学杂志 , 81(3):637–54, 1973 年 5-6 月。 [Cited on page 397。]

[16] Guy E. Blelloch, Michael A. Heroux, 和 Marco Zagha。面向向量多处理器的稀疏矩阵计算分段操作技术报告。技术报告 CMU-CS-93-173，卡内基梅隆大学，1993 年。 [Cited 见第 453 页]

[17] Guy E. Blelloch 、 Charles E. Leiserson 、 Bruce M. Maggs 、 C. Greg Plaxton 、 Stephen J. Smith 与 Marco Zagha 合著。面向连接机 CM-2 的排序算法比较。收录于第三届 *ACM* 年度并行算法与架构研讨会论文集，SPAA '91，第 3-16 页，美国纽约，1991 年。ACM 出版社。 [Cited 第 370 页]

[18] 马克 · 玻尔。回顾 Dennard MOSFET 缩放论文三十年历程。*IEEE* 固态电路通讯，12(1):11–13，2007 年冬季刊。 [Cited on page 53.]

[19] Mark Bohr. SoC 时代下的缩放新纪元 . 载于 *ISSCC*, 第 23–28 页 , 2009 年 . [Cited on page 55.]

[20] Jeff Bolz, Ian Farmer, Eitan Grinspun, 与 Peter Schröoder. GPU 上的稀疏矩阵求解器 : 共轭梯度法与多重网格法 . *ACM Trans. Graph.*, 22(3):917–924, 2003 年 7 月 . [Cited on page 305.]

[21] BOOST 区间算术库 . http://www.boost.org/libs/numeric/interval/doc/interval.htm. [Cited 第 187 页 .]
W. Briggs. *A*

[22] 多重网格教程 . 费城 SIAM 出版社 , 1977 年 . [Cited 第 2 页 60.]

[23] S. Browne, J Dongarra, N. Garner, G. Ho, 与 P. Mucci. 现代处理器性能评估的可移植编程接口 . *International Journal of High Performance ComputingApplications*, 14:189–204, 2000 年秋季 . [Cited 第 466 页 .]

[24] A. Buluc 和 J. R. Gilbert。关于超稀疏矩阵的表示与乘法。载于 *2008* 年 *IEEE* 国际并行与分布式处理研讨会，第 1–11 页，2008 年 4 月 .
[Cited on 第 387 页。]

[25] A. W. Burks 、 H. H. Goldstine 和 J. von Neumann。电子计算仪器逻辑设计的初步讨论。技术报告，哈佛大学，1946 年。 [Cited on page 26。]

[26] A. Buttari 、 J. Dongarra 、 J. Langou 、 P. Luszczek 和 J. Kurzak。用于稠密线性系统求解的混合精度迭代优化技术。国际高性能计算应用期刊，21:457–466，2007 年。 [Cited 见第 188 和 248 页。]

[27] Alfredo Buttari 、 Victor Eijkhout 、 Julien Langou 和 Salvatore Filippone。分块稀疏核的性能优化与建模。国际高性能计算应用期刊，21:467–484，2007 年。 [Cited on 第 305 和 310 页。]

[28] P.M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree loadbalancing using space-filling curves, 2003. [Cited 见第 146 页]

[29] Ernie Chan, MarcelHeimlich, AviPurkayastha, and Robert van de Geijn. Collective communication:theory,practice,and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 2007. [Cited 见第 282 和 284 页]

[30] A. P. Chandrakasan 、 R. Mehra 、 M. Potkonjak 、 J. Rabaey 和 R. W. Brodersen。优化功耗转换技术。*IEEE Transaction on Computer Aided Design of Integrated Circuitsand Systems* ,
pages 13–32, January 1995. [Cited 见第 55 页]

[31] Chapel programming language homepage. http://chapel.cray.com/。 [Cited 见第 111 页]

[32] BarbaraChapman, Gabriele Jost, 与 Ruudvander Pas. *UsingOpenMP: Portable Shared MemoryParallel Programming*, 《 *Scientific Computation Series* 》第 10 卷 . MIT 出版社 , ISBN 0262533022 ,
9780262533027, 2008. [Cited 于第 96 页。]

[33] A. Chronopoulos and C.W. Gear. *s*- 步迭代方法用于对称线性系统。 *Journal ofComputational and Applied Mathematics*, 25:153–168, 1989. [Cited 见于第 314 和 340 页。]

[34] Barry A. Cipra. 伊辛模型导论。 *The AmericanMathematical Monthly*, p 第 937–959 页，1987. [Cited 见于第 398 页。]

[35] Charles Clos. 非阻塞交换网络研究。 *BellSystem Technical Journal*, 32:406–242, 1953. [Cited 见于第 129 页。]

[36] 英特尔公司 . bfloat16 - 硬件数值定义 . https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf,2018. 文档编号：338302-001US. [Cited 见于第 188 和 189 页。]

[37] E. Cuthill 和 J. McKee。减少稀疏对称矩阵的带宽。收录于 *ACM 第 24 届全国会议论文集*，1969 年。 [Cited 第 240 页。]

[38] G. Cybenko。分布式内存多处理器的动态负载均衡。并行与分布式计算杂志，7(2):279–301，1989 年 10 月。 [Cited 第 144 页。]

[39] Eduardo F. D’Azevedo 、 Mark R. Fahey 和 Richard T. Mills。压缩行存储格式的向量化稀疏矩阵乘法。计算机科学讲义，计算科学？*ICCS 2005*，第 99–106 页，2005 年。 [Cited 第 233 页和 305 页。]

[40] E.F. D’Azevedo 、 V.L. Eijkhout 和 C.H. Romine。共轭梯度法及其部分变体的矩阵框架：减少同步开销。收录于第六届 *SIAM* 科学计算并行处理会议论文集，第 644–646 页，费城，1993 年。SIAM。 [Cited on page 314。]

[41] Jeffrey Dean 和 Sanjay Ghemawat。MapReduce：大型集群上的简化数据处理。载于 *OSDI’04*：第六届操作系统设计与实现研讨会，2004 年。 [Cited 第 148 和 151 页。]

[42] J. Demmel 、 M. Heath 和 H. Van der Vorst。并行数值线性代数。载于 *Acta Numerica1993*。剑桥大学出版社，剑桥，1993 年。 [Cited on page 314。]

[43] James Demmel 、 Mark Hoemmen 、 Marghoob Mohiyuddin 和 Katherine Yelick。稀疏矩阵计算中的通信避免。载于 *IEEE* 国际并行与分布式处理研讨会，2008 年。 [Cited 第 340 页。]

[44] Jim Demmel 、 Jack Dongarra 、 Victor Eijkhout 、 Erika Fuentes 、 Antoine Petitet 、 Rich Vuduc 、 R. Clint Whaley 和 Katherine Yelick。自适应线性代数算法与软件。*IEEE* 会刊，93 卷：293–312 页，2005 年 2 月。 [Cited 第 310 页。]

[45] R.H. Dennard,F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implantedmosfet’s with very small physical dimensions. *Solid-State Circuits,IEEE Journal of*, 9(5):256 – 268,oct 1974. [Cited 见第 53 页]

[46] Ashish Deshpande and Martin Schultz. Efficient parallel programmingwith linda. In *In Supercom-puting ’92 Proceedings*, pages 238–244, 1992. [Cited 见第 113 页]

[47] TimDettmers. 8-bit approximations for parallelism in deep learning. 11 2016. Proceedings of ICLR2016. [Cited 见第 188 页]

[48] E.W.Dijkstra. Cooperating sequential processes. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html. Technological University, Eindhoven,The Nether-lands, September 1965. [Cited 见第 90 页]

[49] Edsger W. Dijkstra. 将编程视为一项人类活动。发表于 EWD:EWD117pub，无日期。 [Cited 第 13 页]

[50] J. Dongarra、 A. Geist、 R. Manchek 与 V. Sunderam。集成 PVM 框架支持 Het- 异构网络计算。 物理中的计算机，7(2):166–75，1993 年 4 月。 [Cited on page 105。]

[51] J. J. Dongarra。 *LINPACK* 基准测试：一种解释，第 297 卷，《超级计算 1987》章节第 456–474 页。 Springer-Verlag 出版社，柏林，1988 年。 [Cited 见第 48 页。]

[52] Dr. Dobbs。复数运算：C 与 C 的交集 ++。 http://www.ddj.com/cpp/184401628。 [Cited 见第 190 页。]

[53] Michael Driscoll、 Evangelos Georganas、 Penporn Koanantakool、 Edgar Solomonik 与 Katherine Yelick。直接相互作用的通信最优 N 体算法。载于 *IEEE* 国际并行与分布式处理研讨会 *(IPDPS)*，2013 年。 [Cited 见第 392 与 393 页。]

[54] I. S. Duff, R. G. Grimes, 与 J. G. Lewis.《哈维尔 - 波音稀疏矩阵集合用户指南》（发布 I）。技术报告 RAL 92-086，卢瑟福 · 阿普尔顿实验室，1992 年。 [Cited 第 231 页。]

[55] C. Edwards, P. Geng, A. Patra, 与 R. van de Geijn。《并行矩阵分布：我们一直做错了吗？》技术报告 TR-95-40，德克萨斯大学奥斯汀分校计算机科学系，1995 年。 [Cited 第 295 页。]

[56] Victor Eijkhout。《不完全分块分解的通用公式》《通信应用》 . 《数值方法》，第 4 卷：161–164 页，1988 年。 [Cited on page 325。]

[57] Victor Eijkhout。《并行计算中数据移动的理论》《 *Procedia* 计算机科学》 , 9(0):236 – 245, 2012. 国际计算科学会议论文集 , ICCS 2012, 同时作为德克萨斯大学奥斯汀分校德克萨斯高级计算中心技术报告 TR-12-03 出版。 [Cited 第 393 页。]

[58] Victor Eijkhout, Paolo Bientinesi, 与 Robert van de Geijn. 迈向 Krylov 求解器库的机械化推导。 *Procedia Computer Science*, 1(1):1805–1813, 2010. ICCS 2010 会议论文集 , http://www.sciencedirect.com/science/publication?issn=18770509&volume=1&issue=1。 [Cited on page 256。]

[59] Paul Erdös 与 A. Rényi. 论随机图的演化。 匈牙利科学院数学研究所出版物 , 5:17?–61, 1960. [Cited on page 383。]

[60] V. Faber 和 T. Manteuffel。共轭梯度方法存在的必要与充分条件。 *SIAM* 数值分析杂志 , 21:352–362, 1984。 [Cited 见第 259 页]

[61] R.D.Falgout、 J.E. Jones 和 U.M. Yang。追求 hypre 概念接口的可扩展性 . 技术报告 UCRL-JRNL-205407，劳伦斯利弗莫尔国家实验室，2004 年。已提交至《 ACM 数学软件汇刊》。 [Cited on page 312。]

[62] M. Fiedler。非负对称矩阵特征向量的一个属性及其在图论中的应用。 捷克斯洛伐克数学杂志 , 25:618–633, 1975。 [Cited 见第 449 页]

[63] D. C. Fisher。您最喜爱的并行算法可能不如您想象的那么快。*IEEE* 汇刊 . 计算机 , 37:211–213, 1988。 [Cited 见第 64 页]

[64] M. Flynn. 某些计算机组织及其效能 . *IEEETrans.Comput.*, C-21:948 , 1972. [Cited 第 75 页 <style id='10'>。

[65] Project fortress 主页 . http://projectfortress.sun.com/Projects/Community. [Cited on 第 112 页 <style id='13'>。

[66] D.Frenkel 和 B. Smit. 理解分子模拟：从算法到应用，第二版 . 2002. [Cited 第 347 和 350 页 <style id='14'>。

[67] Roland W. Freund 和 Noël M. Nachtigal. QMR：非厄米线性系统的拟最小残差方法 . *Numer. Math.*, 60:315–339, 1991. [Cited 第 259 页 <style id='14'>。

[68] M. Frigo, Charles E. Leiserson, H. Prokop, 和 S. Ramachandran. 缓存无关算法 . 收录于第 *40* 届计算机科学基础年会论文集 , 1999. [Cited on page 279.]

[69] Matteo Frigo 和 Volker Strumpen. 缓存无关模板计算的内存行为分析 . 超级计算杂志 , 39(2):93–112, 2007 年 2 月 . [Cited on page 330.]

[70] Trevor Gale, Matei Zaharia, Cliff Young, 和 Erich Elsen. 深度学习稀疏 GPU 内核 . 收录于国际高性能计算、网络、存储与分析会议论文集 , SC ’20. IEEE 出版社 , 2020. [Cited on page 408.]

[71] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, 和 V. Sunderam. *PVM* 网络并行计算用户指南与教程 . MIT 出版社 , 1994. 该书提供电子版，访问地址为 `ftp://www.netlib.org/pvm3/book/pvm-book.ps`. [Cited on page 105.]

[72] David Gelernter. Linda 中的生成式通信 . *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985. [Cited on page 112.]

[73] David Gelernter 与 Nicholas Carriero. 协调语言及其重要性 . *Commun.ACM*, 35(2):97–107, 1992. [Cited 见于第 112 页 .]

[74] Alan George. 规则有限元网格的嵌套剖分 . 10(2):345–363, 1973. [引用自第 318 页 .]

[75] Andrew Gibiansky. 将 HPC 技术引入深度学习领域 . https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/. [Cited on page 407.]

[76] GNU 多精度运算库 . http://gmplib.org/. [Cited on page 187.]

[77] D. Goldberg. 计算机算术。附录于 [100]。 [Cited on page 171。]]

[78 David Goldberg. 每位计算机科学家都应了解的浮点算术知识。 计算概览，1991 年 3 月。 [Cited 见第 157 与 170 页。]

[79] G. H. Golub 与 D. P. O’Leary. 共轭梯度法与 Lanczos 算法简史：1948-1976。31:50–102, 1989. [Cited 见第 253 页。]

[80] Gene H. Golub 与 Charles F. Van Loan. 矩阵计算。约翰霍普金斯大学出版社，巴尔的摩，第二版，1989 年。 [Cited on page 212。]

[81] Google. 谷歌云 TPU。 https://cloud.google.com/tpu/。 [Cited 见第 303 页。]

[82] Kazushige Goto 与 Robert A. van de Geijn. 高性能矩阵乘法的剖析。*ACM* 数学软件汇刊，34(3):1–25, 2008. [Cited 见第 279 与 301 页。]

[83] Ananth Y. Grama, Anshul Gupta, 与 Vipin Kumar. 等效率性：衡量并行算法与架构的可扩展性。*IEEE* 并行与分布式技术 , 第 1 卷第 3 期：12–21 页，1993 年 8 月。 [Cited 见于 72 页]

[84] F. 格雷 . 脉冲编码通信 . 美国专利 2,632,058，1953 年 3 月 17 日（1947 年 11 月提交）。 [Cited on 第 126 页]

[85] Ronald I. Greenberg 与 Charles E. Leiserson 合著。《基于胖树的随机路由算法》，载于计算研究进展，第 345-374 页。JAI 出版社，1989 年。 [Cited on page 128。]W. Gropp 、 E. Lusk 及 A. Skjellum 合著。*MPI* 使用指南。麻省理工学院出版社，1994 年。 [

[86]
[87] 威廉·格罗普、托尔斯滕·霍夫勒、拉吉夫·塔库尔与尤因·勒斯科合著。 引用页码 101。] <style id='3'>《高级 MPI 应用：消息传递接口的现代特性》 <style id='5'>，麻省理工学院出版社，2014 年 11 月出版。 [Cited <style id='8'> 第 <style id='10'>101<style id='12'> 页。]

[88] William Gropp, Steven Huss-Lederman, AndrewLumsdaine, Ewing Lusk, BillNitzberg, WilliamSaphir, and Marc Snir. *MPI: The Complete Reference, Volume 2 - The MPI-2* 扩展 . MIT Press,1998. [Cited onpage 105.]

[89] 威廉 · 格罗普（William Gropp）、尤因 · 卢斯克（Ewing Lusk）和拉吉夫 · 塔库尔（Rajeev Thakur）。使用 *MPI-2*：消息传递接口的高级特性。美国麻省理工学院出版社，剑桥，1999 年。 [Cited on page 101。]

[90] William Gropp、Thomas Sterling 和 Ewing Lusk。*Beowulf Cluster Computing with Linux*，*2nd* 版本。麻省理工学院出版社，2003 年。[Cited 第 78 页。]

[91] I. Gustafsson。一类一阶分解方法。*BIT*，18:142–156，1978 年。[Cited on page 260。]

[92] Diana Guttman、Meenakshi Arunachalam、Vlad Calina 和 Mahmut Taylan Kandemir。预取调优优化。载 James Reinders 和 Jim Jeffers 编，高性能珍珠，第二卷，第 401–419 页。Morgan，2015 年。[Cited 第 35 页。]

[93] Hadoop 维基。http://wiki.apache.org/hadoop/FrontPage。[Cited on page 148。]

[94] Louis A. Hageman 和 David M. Young。应用迭代方法。学术出版社，纽约，1981 年。[Cited 第 246 页。]

[95] Mike Hamburg, Paul Kocher, 与 Mark E. Marson。英特尔 Ivy Bridge 数字随机数生成器分析。Cryptography Research, Inc.，http://www.rambus.com/wp-content/uploads/2015/08/Intel_TRNG_Report_20120312.pdf, 2012 年。[Cited 见于第 436 页。]

[96] A. Hartstein, V. Srinivasan, T.R. Puzak, 及 P.G. Emma。缓存失效行为：成因探究 √2。载于第三届计算前沿会议论文集，CF '06，第 313–320 页，美国纽约州纽约市，2006 年。ACM 出版 [Cited on 第 460 页]。

[97] Michael T.Heath. 科学计算：导论与概览（第二版）. McGraw Hill, 2002 [Cited on 页 192、212 和 216。]。

[98] Don Heller. 数值线性代数中的并行算法综述 . *SIAM Review*, 20:740–777 1978. [Cited 第 64 页。]

[99] B. A. Hendrickson 与 D. E. Womble。面向大规模并行计算机稠密矩阵计算的环面环绕映射。*SIAM J. Sci. Comput.*, 15(5):1201–1226, 1994 年。[Cited on page 290。]

[100] John L. Hennessy 与 David A. Patterson。计算机体系结构：量化研究方法。Morgan Kaufman 出版社，第 3 版，1990 年，第 3 版 2003 年。[Cited 第 12 与 497 页。]

[101] Maurice Herlihy 与 Nir Shavit。异步可计算性的拓扑结构。*J. ACM* 46(6):858–923, 1999 年 11 月。[Cited on page 74。]

[102] M.R. Hestenes 与 E. Stiefel。求解线性系统的共轭梯度方法。*Nat.Bur Stand. J. Res.*，第 49 卷：409–436 页，1952 年。[Cited on page 253。]

[103] Catherine F. Higham 与 Desmond J. Higham. 深度学习：应用数学家入门指南 . *SIAM Review*, 61(4):860–891, 2019. [Cited on page 402.]

[104] Nicholas J. Higham. 数值算法的精度与稳定性 . 美国工业与应用数学学会 , 费城 , 宾夕法尼亚州 , 美国 , 第二版 , 2002. [Cited 见于页码 157,174, 179 及 215.]

[105] 乔纳森 · 希尔，比尔 · 麦科尔，丹 ·C· 斯特凡内斯库，马克 ·W· 古德罗，凯文 · 朗，萨蒂什 ·B· 拉奥，Torsten Suel, Thanasis Tsantilas 与 Rob H. Bisseling. BSPlib：BSP 编程库并行计算 ,24(14):1947–1980, 1998. [Cited 见于页码 115.]

[106] C.A.R. 霍尔 . 通信顺序进程 . 普伦蒂斯霍尔出版社 , 1985. ISBN-10: 0131532715 ISBN-13: 978-0131532717. [Cited 第 74 页]

[107] 托尔斯滕 · 霍夫勒、克里斯蒂安 · 西伯特与安德鲁 · 拉姆斯代恩 . 动态稀疏数据交换的可扩展通信协议 . *SIGPLAN* 通知 , 45(5):159–168, 2010 年 1 月 . [Cited on page 312.]

[108] Horovod 主页 . https://horovod.ai/. [Cited on page 407.]

[109] Y. F. 胡与 R. J. 布莱克 . 动态负载均衡的改进扩散算法 . 并行计算 ,25:417–444, 1999. [Cited 第 144 与 145 页]

[110] IEEE 754: 二进制浮点算术标准 . http://grouper.ieee.org/groups/754. [Cited 第 161 页]

[111] 区间算术。 http://en.wikipedia.org/wiki/Interval_(mathematics). [Cited on p 年龄 187。]
[112] C.R. Jesshope 与 R.W. Hockney 合编。DAP 方法，第 2 卷。第 311–329 页。Infotech 国际有限公司，梅登黑德，1979 年。 [Cited 见第 76 页。]
[113] M. T. Jones 与 P. E. Plassmann 合著。大规模稀疏线性系统的高效并行迭代求解。收录于 A. George 、J.R. Gilbert 和 J.W.H. Liu 合编的图论与稀疏矩阵计算，IMA 第 56 卷。Springer 出版社，柏林，1994 年。 [Cited 见第 328 与 444 页。]
[114] W. Kahan。实用技巧：关于减少截断误差的进一步说明。 *ACM* 通讯，8(1):40–，1965 年 1 月。 [Cited 见第 187 页。]
[115] L. V. Kale 与 S. Krishnan 合著。Charm++：基于消息驱动对象的并行编程。收录于使用 *C* 的并行编程 *++*，*G. V. Wilson* 与 *P. Lu* 合编，第 175–213 页。MIT 出版社，1996 年。 [Cited 见第 113 页。]
[116] L.V. 康托罗维奇与 G.P. 阿基洛夫合著。赋范空间中的泛函分析 <style id='5'>。培格曼出版社，1964 年。 [Cited 见于第 248 页。]
[117] R.M. Karp 与 Y. Zhang 合著。一种随机并行分支定界算法。收录于第二十届 *ACM* 计算理论年会论文集，美国伊利诺伊州芝加哥，*1988* 年 *5* 月 *2-4* 日，第 290–300 页。ACM 出版社，1988 年。 [Cited 见第 143 页]
[118] J. Katzenelson. N 体问题的计算结构 . *SIAM Journal ofScientific andStatistical Computing*, 10 卷 :787–815 页 , 1989 年 7 月 . [Cited on page 391.]
[119] Kendall Square Research. http://en.wikipedia.org/wiki/Kendall_Square_Research. [Cited on page 113.]
[120] 唐纳德 · 克努特。 计算机程序设计艺术 第 *2* 卷：半数值算法。 艾迪生 - 韦斯利出版社，雷丁马萨诸塞州，第 3 版，1998 年。 [Cited 第 436 页。]
[121] Arvind Krishnamurthy 和 Katherine Yelick。优化并行 SPMD 程序。载于 Keshav Pingali 、 Utpal Banerjee 、 David Gelernter 、 Alex Nicolau 与 David Padua 合编的 *Languages and Com-pilersforParallelComputing*，第 331–345 页，柏林 / 海德堡，1995 年。Springer Berlin Heidelberg 出版社。 [Cited on page 92.]
[122] Ulrich Kulisch。极速精确乘积累加算法。 *Computing*，第 91 卷第 4 期，397–405 页，2011 年 4 月。 [Cited 见于第 179 页。]
[123] Ulrich Kulisch 与 Van Snyder。精确点积作为长区间算术的基础工具。*Computing*，第 91 卷第 3 期，307–313 页，2011 年。 [Cited on 第 179 页。]
[124] Milind Kulkarni 、 Martin Burtscher 、 Rajasekhar Inkulu 、 Keshav Pingali 及 Calin Cascaval。不规则应用中的并行度有多少？ 载于 *Principles and Practices of Parallel Program-ming (PPoPP)*，2009 年。 [Cited 第 84 页。]
[125] Vipin Kumar,Ananth Grama, Anshul Gupta, and George Karypis. 并行计算导论 .Benjamin Cummings, 1994. [Cited 第 363 页]
[126] H.T. Kung. Systolic algorithms. In S. Parter, editor, 大规模科学计算 , 第 127–140 页 . Academic Press, 1984. [Cited 第 19 页]
[127] U. Kung, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. In *Proc. Intl. Conf. Data Mining*, 第 229–238 页 , 2009.[Cited on page 381.]
[128] 克里斯托夫 · 拉梅特。NUMA （非统一内存访问）概述。2013 年 11 月。 [Cited on page 94.]
[129] C. Lanczos。通过最小化迭代求解线性方程组。*Journal of Research,Nat. Bu. Stand.*，49 卷：33–53 页，1952 年。 [Cited on page 253。]

[130] 鲁宾 ·H· 兰道，曼努埃尔 · 何塞 · 派斯，克里斯特安 ·C· 博德亚努。 计算物理概览普林斯顿大学出版社，2008 年。. [Cited on page 66。]

[131] J. 朗古。 迭代法求解多右端项线性系统。 博士学位论文，图卢兹国立应用科学学院，2003 年 6 月。CERFACS 技术报告 TH/PA/03/24。 [Cited on page 314。]

[132] 艾米 ·N· 朗维尔与卡尔 ·D· 迈耶。 用于网页信息检索的特征向量方法综述。 *SIAM* 评论，47(1):135-161, 2005。 [Cited on page 384。]

[133] P. 莱库耶。 组合多递归随机数生成器。 运筹学 , 44, 1996。 [Cited on page 441。]

[134] R. B. 勒霍克。 隐式重启迭代法的分析与实现。 博士论文，莱斯大学，休斯顿，德克萨斯州，1995 年。另见计算与应用数学系技术报告 TR95-13。 [Cited 第 261 页。]

[135] Charles E. Leiserson. Fat-Trees: Universalnetworks for hardware-efficient supercomputing. *IEEETrans. Comput*, C-34:892–901, 1985. [Cited 第 129 页。]

[136] Siyu Liao,Ashkan Samiee,Chunhua Deng, Yu Bai, and Bo Yuan. Compressing deep neural net-worksusing toeplitz matrix: Algorithm design and fpga implementation. *ICASSP 2019 - 2019 IEEEInternationalConference on Acoustics,Speech andSignal Processing (ICASSP)*, pages 1443–1447, 2019[Cited on 第 406 页。] .

[137] Stevel Lionel. Improving numerical reproducibility in C/C++/Fortran, 2013. [Cited 第 186 页。]

[138] 理查德 ·J· 利普顿、唐纳德 ·J· 罗斯和罗伯特 · 恩德雷 · 塔尔扬。广义嵌套剖分。 *SIAM J Numer. Anal.*, 16:346–358, 1979. [Cited 第 322 和 324 页。] .

[139] Richard J. Lipton 和 Robert Endre Tarjan。平面图的分隔定理。 *SIAM J. Appl Math.*, 36:177–189, 1979. [Cited 见第 319 、 324 和 378 页。] .

[140] J.D.C. Little。排队公式的证明 $L = \lambda W$。 *Ope. Res.*, 页 383–387, 1961。 [Cited on 见第 36 页。]

[141] Yongchao Liu 、 Douglas L. Maskell 和 Bertil Schmidt。CUDASW++：为支持 CUDA 的图形处理器优化 Smith-Waterman 序列数据库搜索。 *BMCRes Notes*, 2:73 , 2009. PMID- 19416548。 [Cited 见第 333 页。]

[142] M. Luby。最大独立集问题的简单并行算法。 *SIAM Journal onComputing*, 4, 1986。 [Cited 见第 328 和 444 页。]

[143] 格里高兹 · 马莱维奇，马修 ·H· 奥斯特恩，阿尔特 ·J·C· 比克，詹姆斯 ·C· 德纳特，伊兰 · 霍恩，纳蒂 · 莱瑟与 Grzegorz Czajkowski。Pregel：大规模图处理系统。载于第 *28* 届 *ACM* 分布式计算原理研讨会论文集，PODC '09，第 6–6 页，纽约州纽约市 , 美国，2009 年。ACM。 [Cited 第 115 页]

[144] M. Mascagni 与 A. Srinivasan。算法 806：Sprng：可扩展伪随机数生成库，。*ACM* 数学软件汇刊，26:436–461, 2000 年。 [Cited on page 440。]

[145] M. Matsumoto 与 T. Nishimura。伪随机数生成器的动态创建。载于 E.H Niederreiter 与 J. eds. Spanier 编，蒙特卡洛与拟蒙特卡洛方法，第 56–69 页 . Springer, 2000. [Cited 第 441 页] .

[146] J.A. Meijerink 与 H.A. van der Vorst。一种针对系数矩阵为对称 M 矩阵的线性系统的迭代解法。 *Math Comp*, 31:148–162, 1977. [Cited on page 249.]

[147] N. Metropolis 、 A. W. Rosenbluth 、 M. N. Rosenbluth 、 A. H. Teller 和 E. Teller。通过快速计算机进行状态方程计算。 *Journal ofChemicalPhysics*, 21:1087–1092, 1953 年 6 月 . [Cited on 第 398 页]

[148] Gerard Meurant。在 CRAY X-MP/48 上多任务处理共轭梯度法。 *ParallelComputing*, 5:267–280, 1987. [Cited onpage 314.]

[149] Gérard Meurant. 偏微分方程在并行计算机上的区域分解方法。《国际超级计算应用期刊》，2:5–12, 1988 年。[Cited on page 325。]

[150] Kenneth Moreland 与 Ron Oldfield. Formal metrics for large-scale parallel performance. 见 Ju- lian M. Kunkel 和 Thomas Ludwig 编，*High Performance Computing*, 卷 9137 之 *LectureNotes in Computer Science*, 页 488–496. Springer International Publishing, 2015. [Cited on page 74.]

[151] 国家标准与技术研究院。矩阵市场。http://math.nist.gov/MatrixMarket。[Cited 第 231 页]

[152] Andrew T. Ogielski 与 William Aiello. Sparse matrix computations on parallel processor arrays.*SIAM J. Sci. Stat. Comput.* 印刷中．[Cited on page 386.]

[153] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, 及 Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, 1996 年 9 月．[Cited on page 39.]

[154] S. Otto, J. Dongarra, S. Hess-Lederman, M. Snir, 和 D. Walker. 消息传递接口：完整参考．麻省理工学院出版社，1995. [Cited on page 105.]

[155] Michael L. Overton. *IEEE* 浮点算术数值计算．SIAM, 费城 PA, 2001. [Cited 第 157 页.]

[156] Larry Page, Sergey Brin, R. Motwani, 和 T. Winograd. PageRank 引用排名：为网络带来秩序，1998. [Cited 第 384 页.]

[157] V. Ya. Pan. 加速矩阵乘法的新方法组合．计算机与数学应用，7:73–125, 1981. [Cited 第 48 和 422 页.]

[158] 性能应用程序编程接口．http://icl.cs.utk.edu/papi/. [Cited 第 466 页.]

[159] Seymour V. Parter. 线性图在高斯消元法中的应用．*SIAM* 评论，3:119–130, 1961.[Cited on page 235.]

[160] S. Plimpton. 短程分子动力学的快速并行算法。*J. Comput. Phys.*, 117:1–19, 1995. [Cited 见第 350 页]

[161] 克里斯托夫 · 波斯皮奇。高性能计算：第 *30* 届国际会议，*ISC* 高性能 *2015*，德国法兰克福，*2015* 年 *7* 月 *12-16* 日，会议录，章节《追踪负载不均：一个移动的目标》，第 497–505 页。施普林格国际出版社，瑞士查姆，2015 年。[Cited on 第 143 页。]

[162] SiddheshPoyarekar. 基本无害：伪常规浮点数解析 https://developers.redhat.com/blog/2021/05/12/mostly-harmless-an-account-of-pseudo-normal-floating-point-numbers/, 2021.[Cited on page 187.]

[163] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, 与 R. W. Johnson。SPIRAL：面向平台优化的信号处理算法库生成器。国际高性能计算应用期刊，18(1):21–45，2004 年。[Cited 见第 278 页。]

[164] Gururaj S. Rao. 缓存存储器性能分析。*J. ACM*, 25:378–395, 1978 年 7 月。[Cited on 第 460 页。]

[165] J.K. Reid. 论共轭梯度法求解大型稀疏线性方程组。见 J.K. Reid 编，大型稀疏线性方程组，第 231–254 页。学术出版社，伦敦，1971 年。[Cited 见第 254 页。]

[166] Y. Saad. 稀疏线性系统的迭代方法．PWS 出版公司，波士顿，1996 年。[Cited on 第 261 页。]

[167] John K. Salmon, Mark A. Moraes, Ron O. Dror, 与 David E. Shaw. 并行随机数 : As

简单如 1、 2、 3。收录于 *2011* 年国际高性能计算网络、存储与分析会议论文集，SC’ ，11，美国纽约州纽约市，2011 年。计算机器协会。 [Cited 第 441 页。]

[168] 佐藤哲也。地球模拟器：角色与影响。核物理 *B* 辑 - 会议增刊 129-130:102 – 108, 2004 年。格点 2003。 ，[Cited 第 77 页。]

[169] David Schade。Canfar：天文学网络基础设施整合。 https://wiki.bc.net/atl-conf/display/BCNETPUBLIC/CANFAR+-+Integrating+Cyberinfrastructure+for+Astronomy。 [Cited 第 150 页。]

[170] R. Schreiber。稀疏直接求解器的可扩展性。收录于 A. George 、 J.R. Gilbert 和 J.W.-H. Liu 编辑的稀疏矩阵计算：图论问题与算法（*IMA* 研讨会卷） Springer-Verlag，纽约，1993 年，1993 年。另见：技术报告 RIACS TR 92.13，美国宇航局艾姆斯研究中心，加利福尼亚州莫菲特场，1992 年 5 月。 [Cited on page 290。]

[171] AlexanderSergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in Ten-sorFlow. *arXiv preprint arXiv:1802.05799*, 2018. [Cited 第 407 页]

[172] D.E. Shaw. A fast, scalable method for the parallel evaluation of distance-limited pairwise particleinteractions. *J. Comput. Chem.*, 26:1318–1328, 2005. [Cited 第 351 、 356 和 357 页]

[173] D. B. 斯基尔科恩、乔纳森 ·M. D. 希尔与 W. F. 麦科尔。《关于 BSP 的问答》，1996 年 . [Cited on 第 114 页]

[174] 马克 · 斯尼尔、史蒂夫 · 奥托、史蒂文 · 赫斯 - 莱德曼、大卫 · 沃克和杰克 · 唐加拉合著的 *MPI: The CompleteReference, Volume 1, The MPI-1 Core*。麻省理工学院出版社，1998 年第二版。 [Cited on page 101.]

[175] 丹 · 斯皮尔曼 . 谱图理论 , 2009 年秋季 . http://www.cs.yale.edu/homes/spielman/561/<style id='5">. [Cited 第 448 页]

[176] Daniel A. Spielman 和 Shang-Hua Teng。谱划分的有效性：平面图与有限元网格。*Linear Algebra and its Applications*, 421(2):284 – 305, 2007. [Cited on page 449.]

[177] Volker Springel. 宇宙学模拟代码 GADGET-2。 *Mon. Not. R. Astron. Soc.*, 364:1105–1134, 2005. [Cited 第 147 页]

[178] G. W. Stewart. 大型消息传递系统中的通信与矩阵计算。并行计算 , 16:27–40, 1990 年。 [Cited 第 290 页]

[179] V. 斯特拉森。高斯消元法并非最优解。数值数学。，13 卷：354–356 页，1969 年。 [Cited on 页码 48 与 422。]

[180] Robert Endre Tarjan. 摊销计算复杂度 . *SIAM J. Algebraic Discrete Methods* ，6(2):306?318, 1985 年 4 月 . [Cited 第 423 页]

[181] 乔治华盛顿大学通用并行 C. http://upc.gwu.edu/. [Cited 第 109 页]

[182] Leslie G. Valiant. 并行计算的桥接模型 . *Commun. ACM*, 33:103–111, 1990 年 8 月 . [Cited 第 114 和 461 页]

[183] Robert A. van de Geijn 与 Enrique S. Quintana-Ortí. 矩阵计算编程科学 . www.lulu.com, 2008 年 . [Cited 第 219 和 297 页]

[184] Henk van der Vorst. 某些 ICCG 方法的向量化变体 . *SIAM J. Sci.Stat. Comput.* ，3:350–356, 1982. [Cited 第 337 页。]

[185] Henk van der Vorst. Bi-CGSTAB: 一种用于求解非对称线性系统的快速且平滑收敛的 Bi-CG 变体。*SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992. [Cited 第 259 页。]

[186] Richard S. Varga. 矩阵迭代分析 . Prentice-Hall, Englewood Cliffs, NJ, 1962. [Cited 第 244 和 253 页。]

[187] R. Vuduc、 J. Demmel 和 K. Yelikk。Oski：自动调优稀疏矩阵核函数库。载于《*SciDAC 2005* 会议论文集》，*Journal of Physics: Conference Series*，待出版。，2005 年。 [Cited 第 310 页。]

[188] Richard W. Vuduc。稀疏矩阵核函数的自动性能调优。博士论文，大学加州大学伯克利分校，2003 年。 [Cited 第 305 页]

[189] M. S. Warren 和 J. K. Salmon。一种并行哈希八叉树 N 体算法。载于 *1993* 年 *ACM/IEEE* 超级计算会议论文集，超级计算 ’93，第 12–21 页，美国纽约州纽约市，1993 年。ACM。 [Cited 第 147 页]

[190] R. Clint Whaley, Antoine Petitet, 与 Jack J. Dongarra。软件自动化经验优化及 ATLAS 项目。并行计算，27(1–2):3–35，2001 年。另见田纳西大学 LAPACK 工作笔记第 147 号，UT-CS-00-448，2000 年（`www.netlib.org/lapack/lawns/lawn147.ps`）。 [Cited 见第 278 页。]

[191] O. Widlund。关于利用可分离有限差分方程的快速方法求解一般椭圆问题。载于 D.J. Rose 与 R.A. Willoughby 合编的 *Sparse matrices and theirapplications* 一书，第 121–134 页。Plenum 出版社，纽约，1972 年。 [Cited on page 248。]

[192] J.H. Wilkinson. 代数过程中的舍入误差 . Prentice-Hall,Englewood Cliffs, N.J., 1963.[Cited 第 157 、 174 和 215 页。]

[193] Samuel Williams、 Andrew Waterman 和 David Patterson。Roofline：多核架构的直观可视化性能模型。*Commun. ACM*, 52:65–76, 2009 年 4 月。 [Cited on page 48。]

[194] Wm. A. Wulf 和 Sally A. McKee。撞上内存墙：显而易见的启示。 *SIGARCHComput. Archit. News*, 23(1):20–24, 1995 年 3 月。 [Cited 第 22 和 465 页。]

[195] L. T. Yand 和 R. Brent。并行分布式内存架构上大型稀疏非对称线性系统的改进 BICGSTAB 方法。见第五届国际并行处理算法与架构会议论文集 . IEEE, 2002 年。 [Cited 第 314 页。]

[196] 兰迪 · 耶茨。定点计算导论。 `http://www.digitalsignallabs.com/fp.pdf`, 2007 年。[Cited on page 188。]

[197] Andy Yoo、 Edmond Chow、 Keith Henderson、 William McLendon、 Bruce Hendrickson 和 U mit Catalyurek。一种在 BlueGene/L 上可扩展的分布式并行广度优先搜索算法。载于 *2005* 年 *ACM/IEEE* 超级计算会议论文集，SC '05，第 25 页起，美国华盛顿特区，2005 年。IEEE 计算机学会。 [Cited on page 387。]

[198] DavidM. Young. 求解椭圆型偏微分方程的迭代方法 . 博士论文，哈佛大学，剑桥，马萨诸塞州，1950 年。 [Cited 第 253 页。]

[199] L. Zhou 和 H. F. Walker。迭代方法的残差平滑技术。15:297–312, 1994.[Cited on page 435.]

ISBN 978-1-257-99254-6
90000
9 781257 992546