


# 高性能计算导论

## Introduction to High Performance Scientific Computing

作者：Victor Eijkhout

组织：Texas Advanced Computing Center

时间：Jan 15 2022

版本：1.0

分类：高性能计算

ElegantLaTeX Program

竹杖芒鞋轻胜马，谁怕？一蓑烟雨任平生。——苏轼

# 引言

科学计算是一门涉及多个领域，对问题过程进行综合建模，并使用计算机求得定量结果的交叉学科。它将某一门具体的学科转变为数值计算问题，并高效构建数学中的解决办法。这清楚地表明了科学计算涉及的三个科学分支：

- 应用数学：真实世界的数学建模。应用数学常常隐式地描述真实世界的运行规律，例如偏微分方程。为了取得实际的具体成果，我们需要采取一种建设性的办法。
- 数值分析提供了科学模型的算法思考。它提供了一个建设性的方法来解决隐式模型，并分析了成本和稳定性。
- 计算采用数值算法，并分析实际实现它们的效果，而非假设的计算引擎。

有人可能会说，当现实世界现象的数学被要求具有建设性时，“计算”本身就成为了一个领域。也就是说，从证明解的存在到实际求得解，算法本身成为一个研究对象，而不仅仅是一个研究工具。

计算机发明后，对算法的研究变得尤为重要。人们用时间开销重新定义了数学运算，算法的复杂度进而成为了一个新的研究领域；计算不再以“实数”的形式进行，而是以有限位串的形式进行，因此我们还需要研究算法的准确性。事实上，早在机械计算器诞生之初，这种趋势就已经出现了萌芽。

科学计算重点关注的是效率。虽然部分科学家仅仅以：“找到一种解决办法”作为研究目的，但在计算中，解决方案往往需要时效性。基于此，本书中我们将非常具体地讨论算法和硬件的效率——因为高性能计算不仅是硬件上的高效实用，更有对算法本身的研究。

本书涵盖了一名合格的计算科学家需要掌握的基础知识，可以作为研究生或高级本科生的教材；或者您也可以把它作为额外的参考，也可以当作阅读练习的材料。

Victor Eijkhout
Research Scientist
Texas Advanced Computing Center
The University of Texas at Austin

# 目录


**第一部分　理论（Theory）　1**

**1　单处理器运算　2**
1.1　冯诺依曼架构　2
1.2　当代处理器　4
1.3　内存层级结构　9
1.4　多核架构　21
1.5　节点架构和插槽　24
1.6　局部性和数据复用　25

**2　并行计算　32**
2.1　引言　32
2.2　理论概念　35
2.3　并行计算机架构　44
2.4　不同类型的内存访问　47
2.5　并行计算中的粒度　49
2.6　线程并行与 OpenMP　52
2.7　拓扑　78
2.8　多线程架构　90
2.9　GPU 与协处理器　90
2.10　负载均衡　94
2.11　其他话题　98

**3　计算机中的运算　105**
3.1　位运算　105
3.2　整数　105
3.3　实数　108
3.4　舍入误差的例子　116
3.5　编程语言中的计算机运算　120
3.6　更多关于浮点运算的内容　124
3.7　结论　127
3.8　复习题　127

**4　微分方程的数值处理　128**
4.1　初值问题　128
4.2　边界值问题　132
4.3　初始边界值问题　138

**5　数值线性代数　142**
5.1　消除未知数　142
5.2　线性代数在计算机运算中的应用　144
5.3　LU 分解　145
5.4　稀疏矩阵　151
5.5　迭代法　161
5.6　特征值法　174
5.7　延伸阅读　174

**6　高性能线性代数　175**
6.1　集合　175
6.2　并行密集型矩阵-向量乘积　178
6.3　并行 LU 因子化　184
6.4　矩阵-矩阵乘积　186
6.5　稀疏的矩阵-向量乘积　188
6.6　迭代方法的计算问题　194
6.7　并行预处理程序　196
6.8　排序策略和并行性　198
6.9　解除 PDE 线性系统的并行性　207
6.10　并行性和隐式操作　207
6.11　网格更新　211
6.12　多核架构上的块状算法　213

**第二部分　应用（Applications）　216**

**7　分子动力学　217**
7.1　受力计算　217
7.2　并行分解　220
7.3　并行快速傅立叶变换　224
7.4　分子动力学整合　226

**8　组合算法　229**
8.1　排序算法概要　229
8.2　奇偶交换排序　230
8.3　快速排序　231
8.4　基数排序　232
8.5　样本排序　234
8.6　双调排序　234
8.7　素数查找　236

**9　图论分析　237**
9.1　传统图算法　237
9.2　并行化　239
9.3　现实世界的图　242
9.4　超文本算法　242
9.5　大型计算图论　244

**10 N 体问题** **246**
10.1 巴恩斯-胡特算法 . . . . . 247
10.2 快速多极点法 . . . . . 248
10.3 全程计算 . . . . . 248
10.4 执行 . . . . . 248

**11 蒙特卡洛方法** **253**
11.1 奖励 . . . . . 253
11.2 示例 . . . . . 254

**12 机器学习** **255**
12.1 神经网络 . . . . . 255
12.2 深度学习网络 . . . . . 256
12.3 计算层 . . . . . 259
12.4 Stuff . . . . . 261

**第三部分 附录（Appendices）** **262**

**13 线性代数** **263**
13.1 范数 . . . . . 263
13.2 格兰-施密特正交化 . . . . . 264
13.3 幂法 . . . . . 265
13.4 非负矩阵；佩伦向量 . . . . . 266
13.5 格什戈林定理 . . . . . 266
13.6 Householder 反射器 . . . . . 267

**14 复杂度** **268**

**15 偏微分方程** **269**
15.1 偏导数 . . . . . 269
15.2 泊松或拉普拉斯方程 . . . . . 269
15.3 热方程 . . . . . 269
15.4 恒稳态 . . . . . 269

**16 泰勒级数** **270**

**17 最小化** **272**
17.1 下降法 . . . . . 272
17.2 牛顿法 . . . . . 274

**18 图论** **276**
18.1 定义 . . . . . 276
18.2 图的一般类型 . . . . . 277
18.3 图着色和独立集 . . . . . 277
18.4 图论算法 . . . . . 277
18.5 图和矩阵 . . . . . 278
18.6 谱图理论 . . . . . 280

**19 自动机理论** **282**
19.1 有限状态自动机 . . . . . 282
19.2 一般讨论 . . . . . 282

**20 并行前缀** **284**
20.1 并行前缀 . . . . . 284
20.2 并行前缀的稀疏矩阵向量积 . . . . . 285
20.3 Horner 法则 . . . . . 285

**第四部分 HPC Carpentry** **287**

**21 Unix 简介** **289**
21.1 Unix 简介 . . . . . 289
21.2 文件及相关 . . . . . 289
21.3 文本搜索和正则表达式 . . . . . 295
21.4 其他有用的命令：tar . . . . . 297
21.5 命令的执行 . . . . . 297
21.6 输入/输出重定向 . . . . . 301
21.7 Shell 环境变量 . . . . . 302
21.8 控制结构 . . . . . 304
21.9 脚本 . . . . . 305
21.10扩展 . . . . . 307
21.11启动文件 . . . . . 308
21.12交互式 Shell . . . . . 309
21.13系统和其他用户 . . . . . 309
21.14其他系统：SSH 和 SCP . . . . . 310
21.15sed 和 awk 工具 . . . . . 311
21.16复习题 . . . . . 312

**22 编译器和库** **313**
22.1 当我们运行一个程序时，会执行什么? 313
22.2 简单汇编 . . . . . 313
22.3 库 . . . . . 315

**23 Make 与 Makefile** **318**
23.1 使用 Make 命令管理项目 . . . . . 318
23.2 变量和模板规则 . . . . . 321
23.3 杂录 . . . . . 323
23.4 Makefile 中的 shell 脚本 . . . . . 325
23.5 使用 Make 的实用技巧 . . . . . 326
23.6 LaTeX 的 Makefile . . . . . 326
23.7 Cmake . . . . . 327

**24 源码控制** **330**
24.1 源码控制 . . . . . 330
24.2 源码控制系统的工作流程 . . . . . 330
24.3 Mercurial( hg ) 和 Git . . . . . 330

**25 科学数据存储** **338**
25.1 科学数据存储 . . . . . 338
25.2 HDF5 简介 . . . . . 338
25.3 创建一个文件 . . . . . 339
25.4 数据集 . . . . . 340
25.5 写入数据 . . . . . 343
25.6 读取 . . . . . 345

**26 GNU plot** **347**
26.1 使用 GNUplot 绘图 . . . . . 347
26.2 使用方法 . . . . . 347
26.3 绘图 . . . . . 347
26.4 工作流程 . . . . . 348

**27 代码规范** **349**
27.1 良好的代码行为 . . . . . 349
27.2 防御式编程 . . . . . 349
27.3 防范内存错误 . . . . . 352
27.4 测试 . . . . . 354

**28 Debug** **355**
28.1 调试 . . . . . 355
28.2 调用调试器 . . . . . 355
28.3 查找错误 . . . . . 357
28.4 内存调试 . . . . . 359
28.5 逐步调试程序 . . . . . 361
28.6 观察变量值 . . . . . 362
28.7 断点 . . . . . 363
28.8 并行调试 . . . . . 363
28.9 延伸阅读 . . . . . 364

**29 C、Fortran 与 Python** **365**
29.1 C/Fortran 的互用性 . . . . . 365
29.2 链接器规定 . . . . . 365
29.3 数组 . . . . . 368
29.4 字符串 . . . . . 369
29.5 子程序参数 . . . . . 369
29.6 输入/输出 . . . . . 370
29.7 Fortran2003 中的 Fortran/C 互用性 . . . 370
29.8 Python 调用 C 代码 . . . . . 370

**30 位操作** **372**
30.1 位操作 . . . . . 372
30.2 结构和展示 . . . . . 372
30.3 位操作 . . . . . 372

**31 用于科学文献的 LaTeX** **374**
31.1 LaTeX 理念及历史 . . . . . 374
31.2 对 LaTeX 的简单介绍 . . . . . 374
31.3 已完成的例子 . . . . . 379

# 第一部分

# 理论（Theory）

# 第 1 章 单处理器运算

内容提要

- 冯·诺依曼架构
- 当代处理器
- 内存的层级结构
- 多核架构
- 节点架构和插槽
- 局部性与数据复用

了解计算机体系结构对于编写高效、科学的代码具有十分重要的作用。两段基于不同处理器架构编写的代码，其计算结果可能相同，但速度差异可能从几个百分点到几个数量级之间不等。显然，仅仅把算法放到计算机上是不够的，计算机架构也是至关重要的内容。

有些问题可以在单个中央处理单元（CPU）上解决，而有些问题则需要由多个处理器组成的并行计算机解决。我们将在下一章详细介绍并行计算机，但即便是使用并行计算机处理，也需要首先了解单个 CPU 的情况。

在该部分，我们将重点关注 CPU 及其内存系统内部发生的事情。首先讨论指令如何执行，研究处理器核心中的运算；最后，由于内存访问通常比处理器执行指令要慢得多，因此我们将重点关注内存、处理器以及处理器内部的数据移动情况；“flops（每秒浮点操作数）计数”作为预测代码性能的时代已经一去不复返了。这种差异实际上是一个不断增长的趋势，所以随着时间的推移，处理内存流量的问题变得越来越重要，而非逐渐销声匿迹。

这一章中，我们将对 CPU 设计是如何影响性能的，以及如何编写优化性能的代码等问题有一个清晰的认识。想学习更多细节，请参阅关于 PC 架构的在线书籍 [114]，以及关于计算机架构的标准工作，Hennesey 和 Patterson[97]。

## 1.1 冯诺依曼架构

虽然各类计算机在处理器细节上存在很多不同，但也有许多相似之处。总的看来，它们都采用了**冯·诺伊曼架构**（von Neumann architectures）。该架构主要包含：存储程序和数据的内存，以及一个在“获取、执行、存储周期”中对数据进行操作的指令处理单元。

注释 具有指定指令序列的模型也称为**控制流**（control flow），与**数据流**（data flow）相对应。

由于指令和数据共同存储在一个处理器中，这使得冯·诺依曼架构区别于早期或一些其他特殊用途的硬接线当代处理器，能够允许修改或生成其他程序。这给我们提供了编辑器和编译器：计算机可以将程序视为数据进行处理。

注释 存储程序的概念允许一个正在运行的程序修改其源代码。然而，人们很快就意识到这将导致代码变得难以维护，因此在实际中很少见到。

本书将不会讨论编译器将高级语言翻译成机器指令的过程，而是讨论如何编写高质量的程序以确保底层运行的效率。

在科学计算中，我们通常只关注数据在程序执行期间如何移动，而非程序代码具体如何。大多数应用中，程序与数据似乎是分开存储的。与高级语言不同，处理器执行的机器指令通常会指定操作的名称，操作数和结果的位置。这些位置不是表示为内存位置，而是表示为**寄存器**（registers）位置：即在 CPU 中被称作内存的一小部分。

注释 我们很少分析到内存的架构，尽管它们已经存在。20 世纪 80 年代的 Cyber 205 超级计算机可以同时有三个数据流，两个从内存到处理器，一个从处理器到内存。这样的架构只有在内存能够跟上处理器速度的情况下才

可行，而现在已经不是这样了。

下面是一个简单的 C 语言例子：

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

及其 X86 汇编输出，由 gcc -O2 -S -o - store.c:

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

(程序演示的为 64 位系统输出；32 位系统可以添加-m64 指令输出) 这段程序的指令有：

- 从内存加载到寄存器；
- 执行加法操作；
- 将结果写回内存。

每条指令的处理如下：

- 指令获取：根据**程序计数器**（program counter）的指示将下一条指令装入进程。此处我们不考虑这是如何发生以及从哪里发生的问题。
- 指令解码：处理器检查指令以确定操作和操作数。
- 内存获取：必要时，数据将从内存取到寄存器中。
- 执行：执行操作，从寄存器读取数据并将数据写回寄存器。

数组的情况稍微复杂一些：加载（或存储）的元素被确定为数组的基址后会加上一个偏移量。

在某种程度上，现代 CPU 在程序员看来就像冯 · 诺伊曼机器，但也有各种例外。首先，虽然内存看起来为随机寻址，但在实际中存在着**局部性**（locality）的概念：一旦一个数据项被加载，相邻的项将更有效地加载，而重新加载初始项也会更快。

简单数据加载的另一个复杂之处是，当前的 CPU 同时操作多条指令，这些指令被称为“**正在执行**（in flight）”，这意味着它们处于不同的完成阶段。当然，与这些同步指令一起，它们的输入和输出也以重叠的方式在内存和处理器之间移动。这是超标量 CPU 体系结构的基本思想，也被称为指令级并行（Instruction Level Parallelism，ILP）。因此，虽然每个指令可能需要几个时钟周期才能完成，但处理器可以在合适的情况下每个周期完成一条指令；在某些情况下，每个周期可以完成多条指令。

CPU 的处理速度处在千兆赫级（G），意味着处理器的速度是决定计算机性能的主要因素。速度虽然与性能紧密联系，但实际情况却更为复杂：一些算法被 CPU 所限制，此时进程的速度是最重要的制约；另外一些算法受到内存的限制，总线速度和缓存大小等方面是影响该问题的关键。

在科学计算中，第二种情况相当显著，因此在本章中，我们将大量关注将数据从内存转移到处理器的过程，而对实际处理器的关注相对较少。

## 1.2 当代处理器

当代处理器极为复杂，在这一节中，我们将简短地介绍一下其组成部分。下图是 Intel Sandy Bridge 处理器的芯片图。这种芯片大约一英寸大小，包含近十亿个晶体管。

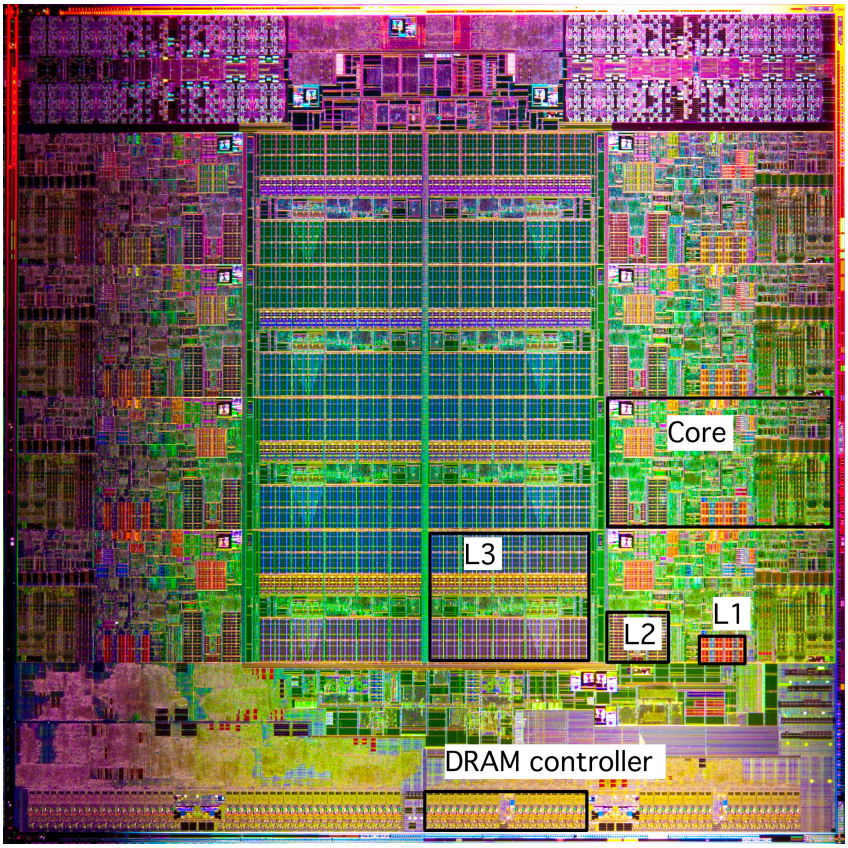



图 1.1: Intel Sandy Bridge 处理器的芯片

### 1.2.1 处理核心

冯 · 诺依曼模型中只有一个执行指令的实体。自 21 世纪初以来，这种情况并没有显著的增长。上图所示的 Sandy Bridge 有 8 个核，每个核都是执行指令流的独立单元。在本章中，我们将主要讨论单个核心的各个方面；后面将讨论多核的集成方面。

#### 1.2.1.1 指令处理

冯 · 诺伊曼模型也是不现实的，因为它假设所有的指令都严格按照顺序执行。在过去的二十年中，处理器越来越多地使用了无序指令处理，即指令可以按照不同于用户程序指定的顺序进行处理。当然，处理器只有在不影响执行结果的情况下才允许对指令重新排序！

在图 1.2 中，你可以看到与指令处理有关的各种单元：这种聪明的做法实际上要花费相当多的能源及大量的晶体管。正因如此，包括第一代英特尔 Xeon Phi 协处理器，Knights Corner 在内的处理器都采取了使用顺序指令处理的策略。然而，在下一代 Knights Landing 中，这种做法却由于性能不佳而被淘汰。

#### 1.2.1.2 浮点单元

在科学计算中，我们最感兴趣的是处理器如何处理浮点数据。而并非整数或布尔值的运算。因此，核心在处理数值数据方面具有相当的复杂性。

例如，过去的处理器只有一个**浮点单元**（FPU），而现在的它们有多个且能够同时执行的浮点单元。

例如，加法和乘法通常是分开的；如果编译器可以找到独立的加法和乘法操作，就可以同时调度它们从而使处理器的性能翻倍。在某些情况下，一个处理器会有多个加法或乘法单元。

另一种提高性能的方法是使用**乘加混合运算**（Fused Multiply-Add，FMA）单元，它可以在与单独的加法或乘法相同的时间内执行指令 $x \leftarrow ax + b$。配合流水线操作，这意味着处理器在每个时钟周期内有几个浮点运算的渐近速度。

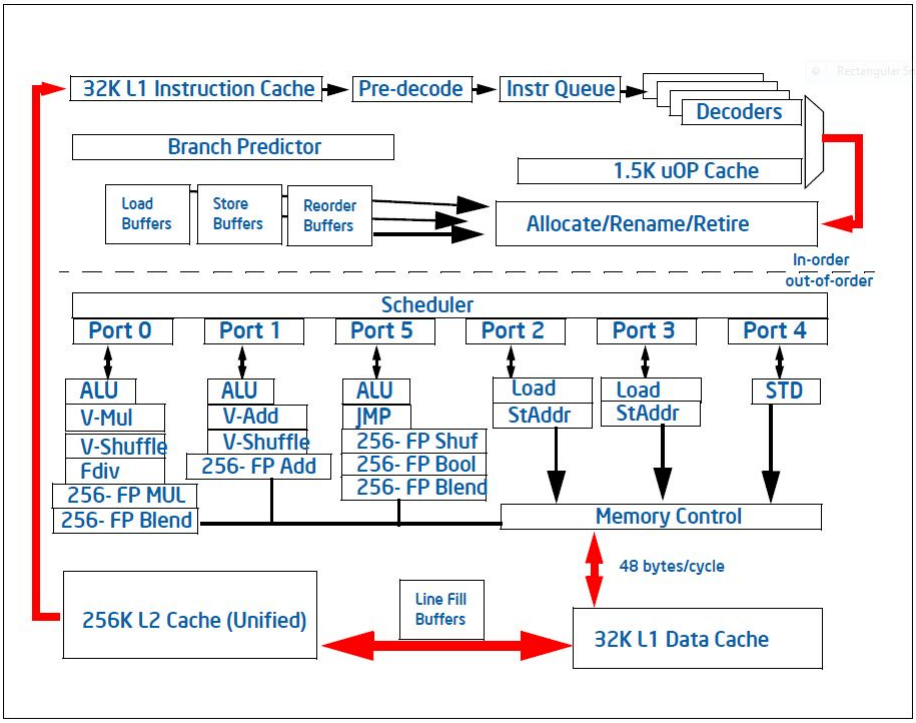



Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

图 1.2: Intel Sandy Bridge 核心框图

顺便说一句，很少有用除法当制约因素的算法。在现代 CPU 中，除法操作的优化程度远不及加法和乘法。除法操作可能需要 10 或 20 个时钟周期，而 CPU 可以有多个加法和/或乘法单元（渐进地）使之每个周期都可以产生一个结果。下表为多个处理器架构的浮点能力 (每个核心)，以及 8 个操作数的 DAXPY 周期数。

表 1.1: 不同处理器架构信息对比

| 处理器 | 年份 | 加/乘/乘加混合单元 (个数 × 宽度) | daxpy cycles(arith vs load/store) |
|---|---|---|---|
| MIPS R10000 | 1996 | $1\times1+1\times1+0$ | 8/24 |
| Alpha EV5 | 1996 | $1\times1+1\times1+0$ | 8/12 |
| IBM Power5 | 2004 | $0+0+2\times1$ | 4/12 |
| AMD Bulldozer | 2011 | $2\times2+2\times2+0$ | 2/4 |
| Intel Sandy Bridge | 2012 | $1\times4+1\times4+0$ | 2/4 |
| Intel Haswell | 2014 | $0+0+2\times4$ | 1/2 |

### 1.2.1.3 流水线

处理器的浮点加乘单元是流水线式的，其效果是独立的操作流可以以每个时钟周期一个结果的渐近速度执行。流水线背后的思想如下：假设一个操作由若干个简单操作组成，并且每个子操作在处理器中都有独立的硬件。如果我们现在有多个操作要执行，我们可以通过让所有的子操作同步执行来获得加速：每个操作将其结果

交给下一个操作，并接受前一个操作的输入。

**注释** 这与排队救火的过程非常类似，是一种收缩算法。

例如，加法指令可以包含以下组件：

- 指令解码，包括查找操作数的位置。
- 将操作数复制到寄存器中（数据获取）。
- 调整指数；加法 $.35 \times 10^{-1} + .6 \times 10^{-2}$ 变成 $.35 \times 10^{-1} + .06 \times 10^{-1}$。
- 执行尾数的加法，在这个例子中是 .41。
- 将结果归一化，在本例中为 $.41 \times 10^{-1}$。（本例中归一化并未执行任何操作，$.3 \times 100 + .8 \times 100$ 和 $.35 \times 10^{-3} + (-.34) \times 10^{-3}$ 做了很大的调整）

这些部分通常被称为流水线的“**深度**（stages）”或“**阶段**（segments）”。

如果按照顺序执行，上文中每个组件设计为 1 个时钟周期，则整个过程需要 6 个时钟周期。然而，如果每个操作都有自己对应的硬件，我们就可以在少于 12 个周期内执行两个操作：

- 第一个操作执行解码；
- 第一个操作获取数据，同时第二个操作执行解码；
- 同时执行第一操作的第三阶段和第二操作的第二阶段；
- 以此类推。

可以看到，第一个操作仍然需要 6 个时钟周期，但第二个操作只需再延后一个周期就能同时完成。

对流水线获得的加速做一个正式的分析：在传统的浮点单元上，产生 $n$ 个结果需要花费时间为 $t(n) = n\ell\tau$，其中 $\ell$ 是状态个数，而 $\tau$ 是时钟周期。结果产生的速率是 $t(n)/n$ 的倒数：$r_{serial} \equiv (\ell\tau)^{-1}$

另一方面，对于流水线的浮点单元，时间是 $t(n) = [s + l + n - 1]t$ 其中 $s$ 是设置成本；第一次执行时必须要经历一个完整的串行阶段，但在此后，处理器将在每个周期获得更多的收益。可以记作公式

$$xt(n) = [n + n_{1/2}]\tau \tag{1.1}$$

表示线性时间，加上偏移量。流水线示意图描述如下：

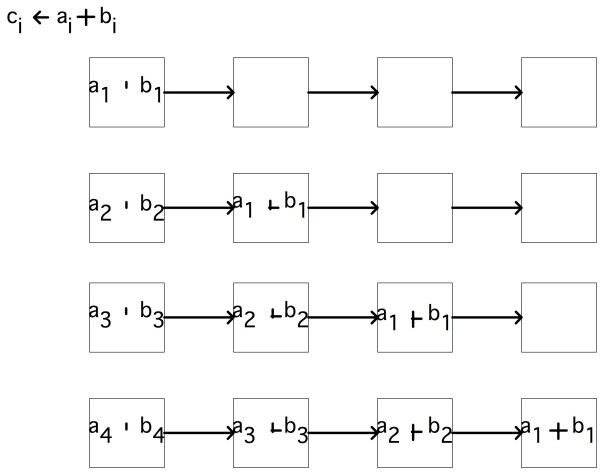



图 1.3: 流水线操作的示意图

**练习 1.1** 请对比传统 FPU 和流水线 FPU 的速度差异。证明结果速率依赖于 $n$：给出 $r(n)$ 和 $r_\infty = \lim_{n\to\infty} r(n)$ 的公式。在非流水线情况下 $r$ 的加速极限是什么样？它需要多长时间才能接近极限情况？注意到 $n = n_{1/2}$，可以得到 $r(n) = r_\infty/2$，这通常被用做 $n_{1/2}$ 的定义。

由于向量处理器同时处理多个指令，因此这些指令必须是独立且相互之间没有依赖关系的。$\forall_i : a_i \leftarrow b_i + c_i$ 有独立的加法运算；$\forall_i : a_{i+1} \leftarrow a_i b_i + c_i$ 将一次迭代 $(a_i)$ 的结果输入到下一次迭代的输入 $(a_{i+1} = ...)$，所以这些操作并非独立。

与传统的 CPU 相比，流水线处理器可以将操作速度提高 4、5 甚至 6 倍。在上世纪 80 年代，当第一台向量计算机成功上市时，这样的加速效率十分常见。现在，CPU 可以有 20 个阶段的流水线，是否意味着它们的速度非常快？这个问题有点复杂。芯片设计者不断提高主频，流水线部分不再能够在一个周期内完成他们的工作，所以他们进一步分裂。有时甚至有一些时间片段什么也没有发生：但这段时间是又是必须的，以确保数据可以及时传输到芯片的不同部分。

人们能从流水线 CPU 得到的改进是有限的，为了追求更高的性能，计算机科学家们尝试了几种不同的流水线设计。例如，Cyber 205 有单独的加法和乘法流水线，可以将一个流水线输入另一个流水线，而无需先将数据返回内存。像 $\forall_i : a_i \leftarrow b_i + cd_i$ 这样的操作被称为“链接三元组（linked triads）”（因为到内存的路径数量，一个输入操作数必须是标量）。

✍ **练习 1.2** 分析链接三元组的加速和 $n_{1/2}$。

另一种提高性能的方法是使用多个相同的流水线。NEC SX 系列完善了这种设计。例如，有 4 条流水线时，$_i : a_i \leftarrow b_i + c_i$ 操作将对模块 4 进行拆分，以便第一条流水线对索引 $i = 4j$ 操作，第二条流水线对索引 $i = 4j + 1$ 操作，以此类推。

✍ **练习 1.3** 分析具有多个并行操作流水线处理器的速度提升情况和 $n_{1/2}$。也就是说，假设有 $p$ 个执行相同指令的独立流水线，每条流水线都可以处理的操作数流。

（你可能想知道我们为什么在这里提到一些相当老的计算机：真正的流水线超级计算机已经不存在了。在美国，Cray X1 是该系列的最后一款，而在日本，只有 NEC 还在生产。然而，现在 CPU 的功能单元是流水线的，所以这个概念仍然很重要。）

✍ **练习 1.4** 如下操作

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

不能由流水线处理，因为在操作的一次迭代的输入和前一次迭代的输出之间存在依赖关系。但是，我们可以将循环转换为数学上等价的循环，并且可能更有效地计算。导出一个表达式，该表达式从 $x[i]$ 中计算 $x[i+2]$ 而不涉及 $x[i+1]$。这就是所谓的**递归加倍**（recursive doubling）。假设有足够的临时存储空间。参考如下：

- 做初步计算；
- 计算 $x[i], x[i+2], x[i+4], \ldots$，并从这些中
- 计算缺失项 $x[i+1], x[i+3], \ldots$

通过给出 $T_0(n)$ 和 $T_s(n)$ 的计算公式，分析了该格式的有效性。你能想到为什么初步计算在某些情况下可能不那么重要吗？

**1.2.1.3.1 收缩计算** 上面描述的流水线操作是**收缩算法**（systolic algorithm）的一种情况。在 20 世纪 80 年代和 90 年代，有研究使用流水线算法并构建特殊硬件——**脉动阵列**（systolic arrays）来实现它们 [125]。这也与**现场可编程门阵列**（Field-Programmable Gate Arrays，FPGA）的计算连接，其中脉动阵列是由软件定义的。

#### 1.2.1.4 峰值性能

现代 CPU 由于流水线的存在，时钟速度和峰值性能之间存在着较为简单的关系。由于每个 FPU 可以在一个周期内产生一个结果，所以峰值性能是时钟速度乘以独立 FPU 的数量。浮点运算性能的衡量标准是“**每秒浮点运算**（floating point operations per second）”，缩写为 flops。考虑到现在计算机的速度，你会经常听到浮点运算被表示为“gigaflops”：$10^9$ 次浮点运算的倍数。

### 1.2.2 8 位，16 位，32 位，64 位

处理器的特征通常是可以处理多大的数据块。这可以联系到

- 处理器和内存之间路径的宽度：一个 64 位的浮点数是否可以在一个周期内加载，还是分块到达处理器。
- 内存的寻址方式：如果地址被限制为 16 位，只有 64,000 字节可以被识别。早期的 PC 有一个复杂的方案，用段来解决这个限制：用段号和段内的偏移量来指定一个地址。
- 单个寄存器中的数值位数，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，例如在 x86 体系结构中是 80 位。) 这也对应于处理器可以同时操作的数据块的大小。
- 浮点数的大小：如果 CPU 的算术单元被设计成有效地乘 8 字节数（“双精度”；见 3.2.2 节），那么一半大小的数字（“单精度”）有时可以以更高的效率处理，而对于更大的数字（“四倍精度”），则需要一些复杂的方案。例如，一个四精度的数字可以由两个双精度的数字来模拟，指数之间有一个固定的差异。

这些测量值不一定相同。例如，原来的奔腾处理器有 64 位数据总线，但有一个 32 位处理器。另一方面，摩托罗拉 68000 处理器（最初的苹果 Macintosh）有一个 32 位 CPU，但 16 位的数据总线。

第一个英特尔微处理器 4004 是一个 4 位处理器，它可以处理 4 位的数据块。如今，64 位处理器正在成为标准。

### 1.2.3 缓存（Caches）：芯片上的内存

计算机内存的大部分是在与处理器分离的芯片中。然而，通常有少量的片上内存（通常是几兆字节），这些被称为**高速缓存**（cache）。

### 1.2.4 图形、控制器、专用硬件

“消费型”和“服务器型”处理器之间的一个区别是，消费型芯片在处理器芯片上花了相当大的空间用于图形处理。手机和平板电脑的处理器甚至可以有专门的安全电路或 mp3 播放电路。处理器的其他部分专门用于与内存或 I/O 子系统通信。我们将不在本书中讨论这些方面。

### 1.2.5 超标量处理和指令级并行性

在冯 · 诺伊曼模型中，处理器通过控制流进行操作：指令之间线性地或通过分支相互跟踪，而不考虑它们涉及哪些数据。随着处理器变得越来越强大，一次可以执行多条指令，就有必要切换到数据流模型。这种超标量处理器分析多个指令以找到数据相关性，并行执行彼此不依赖的指令。这个概念也被称为**指令级并行** (Instruction Level Parallelism，ILP)：

**定义 1.1 (指令级并行)**

指令级并行由各种因素共同完成：

- 多发射（multiple-issue）：独立指令可同时启动；
- 流水线（pipelining）：上文提到，算术单元可以在不同的完成阶段处理多个操作；
- 分支预测和推测执行（branch prediction and speculative execution）：编译器可以“预测”条件指令的值是否为真，然后相应地执行这些指令；

- 无序执行（out-of-order execution）：如果指令之间不相互依赖，并且执行效率更高，则指令可以重新排列；
- 预取（prefetching）：数据可以在实际遇到任何需要它的指令之前被推测地请求（这将在后面进一步讨论）。

♣

在上面我们看到了浮点操作上下文中的流水线操作。事实上，不仅是浮点运算，当代处理器的整个 CPU 都是流水线的，任何类型的指令都将尽快被放入指令流水线中。注意，这个流水线不再局限于相同的指令：现在，流水线的概念被概括为同时" 在执行" 的任何指令流。

随着主频的增加，处理器流水线的长度也在增加，以使分段在更短的时间内可被执行。可以看到，更长的流水线有着更大的 $n_{1/2}$，因此需要更多的独立指令来使流水线以充分的效率运行。当达到指令级并行性的极限时，使流水线变长（或者称为“更深”）将不再有好处。因此，芯片设计者们通常转向多核架构以更高效地利用芯片上的晶体管。

这些较长的流水线的第二个问题是：如果代码到达一个分支点（一个条件或循环中的测试），就不清楚要执行的下一条指令是什么。在该点上，流水线就会会停止。例如，CPU 总是假设测试结果是正确的，因此采取了**分支预测执行**（speculative execution）。如果代码随后接受了另一个分支（这称为分支错误预测），则必须刷新流水线并重新启动。执行流中产生的延迟称为**分支惩罚**（branch penalty）。

## 1.3 内存层级结构

冯 · 诺伊曼体系结构中，数据立即从内存加载到处理器，并在处理器中进行操作。然而这是不现实的，因为这一过程中会有**访存墙**（memory wall）[204] 的存在：内存速度过慢，无法和处理器速度相匹配。具体来说，单次加载可能需要 1000 个周期，而处理器每个周期可以执行若干个操作。(在长时间等待加载之后，下一个加载可能会更快，但对处理器来说仍然太慢）

实际上，在 FPU 和内存之间会有不同的存储器级别：寄存器和缓存，一起称为**存储器层级结构**（memory hierarchy）。它们可以更快速地从内存中读取一些最近使用的数据以缓解访存墙的问题。当然，这是在数据被多次使用的前提下。这类**数据复用**（data reuse）问题将在后面进行更详细的讨论。

寄存器和缓存都在不同程度上比内存快；寄存器的速度越快，其存储量就越小。寄存器的大小和速度之间的矛盾产生了一个有趣的博弈，我们将随后讨论这些问题。

接下来，我们将讨论内存层次结构的各部分并分析其在执行过程所需的理论基础。

### 1.3.1 总线

在计算机中，将数据从处理器移动到 CPU、磁盘控制器或屏幕的线路被称为**总线**（busses）。对我们来说最重要的是连接 CPU 和内存的**前端总线**（Front-Side Bus，FSB）。在当前较为流行的架构中，这被称为“**北桥**（north bridge）”，与连接外部设备（除了图形控制器）的“**南桥**（south bridge）”相对。

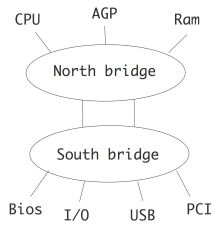



图 1.4: 处理器的总线结构

#### 1.3.1.1 延迟和带宽

寄存器中访问数据几乎是瞬时的，而将数据从内存加载到寄存器是进行任何操作之前的一个必要步骤，加载的过程会导致大量的延迟，下面我们将细化这一过程。

有两个重要的概念来描述数据的移动：**延迟**和**带宽**（latency and bandwidth）。这里的假设是，请求一组数据会引起初始延迟；如果这一项是该组数据的第一个项，则通常是连续的内存地址范围，而该组数据的其余部分将以一个固定的时间周期不再延迟到达。

**定义 1.2 (延迟和带宽)**

**延迟和带宽**（latency and bandwith）是两个描述数据移动的重要概念。请求一组数据会引起初始延迟；如果这一项是该组数据的第一个项，则通常是连续的内存地址范围，而该组数据的其余部分将以一个固定的时间周期不再延迟到达。

**延迟**：为处理器发出内存项请求到内存项实际到达之间的延迟。我们可以区分不同的延迟，比如从内存到缓存的传输，缓存到寄存器的传输，或者将它们总结为内存和处理器之间的延迟。延迟是以（纳米）秒或时钟周期来衡量的。如果处理器按照在汇编代码中的顺序去执行指令，则在从内存中提取数据时经常会停止；这也被称为内存延迟（memory stall）。低延迟非常重要。在实际中，许多处理器都有指令的"无序执行"情况，即允许它们在等待所请求的数据时执行其他操作。程序员可以考虑这一点，并以一种实现延迟隐藏的方式编写代码；图形处理单元（GPU）可以在线程之间快速切换，以实现延迟隐藏。

**带宽**：为克服初始延迟后，数据到达目的地的速率。带宽以字节（千字节 k、兆字节 m、千兆字节 g）/秒或每个时钟周期来衡量。两个存储层之间的带宽通常是通道的周期速度（总线速度）和总线宽度的乘积：总线时钟的每个周期中可以同时发送的比特数。

♣

延迟和带宽的概念通常结合在一个公式中，表示消息从开始到结束所花费的时间：

$$T(n) = \alpha + \beta n \tag{1.2}$$

其中 $\alpha$ 是延迟，$\beta$ 是带宽的倒数：即每字节所用的时间。

通常，距离处理器越远，延迟就越长，带宽就越低。因此我们希望处理器尽量使用缓存中的数据而非内存，例如考虑向量加法

### 1.3.2 寄存器

每个处理器内部都有少量类似内存的结构：**寄存器**（registers）或**寄存器堆**（registers file）。寄存器是处理器实际操作的对象：例如

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

实际过程为

- 将 b 的值从内存中装入寄存器，
- 将 c 的值从内存中装入另一个寄存器，
- 计算和并将其写入另一个寄存器，然后
- 将计算后的总和写回 a 的内存位置。

查看汇编代码（例如编译器的输出）就可以看到显式的加载、计算和存储指令。

像加或乘这样的计算指令只能在寄存器上操作。例如，在汇编语言中我们会看到如下指令

```
addl %eax %edx
```

它将一个寄存器的内容添加到另一个寄存器。正如在这个示例指令中看到的，与内存地址相反，寄存器没有编号，而是具有在汇编指令中引用的不同名称。通常，一个处理器有 16 或 32 个浮点寄存器；英特尔安腾（Intel Itanium）的 128 个浮点寄存器是例外。

寄存器具有高带宽和低延迟，因为它们是处理器的一部分。可以将进出寄存器的数据移动看作是瞬时的。

在本章中，我们会发现从内存中读取数据的时间开销较大。因此，尽可能将数据留在寄存器中是一种简单的优化策略。例如，如果上面的计算后面跟着一个语句

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

a 的计算值就可以留在寄存器中。编译器通常会帮助我们完成这种优化：编译器不会生成存储和重新加载 a 的指令。我们就称 a 停留在寄存器中。

将值保存在寄存器中通常是为了避免重新计算新的变量。例如，在

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

正弦和余弦值可能会保留在寄存器中。我们可以通过显式地引入临时数量来帮助编译器：

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

当然，寄存器的数量是有限制的；试图在寄存器中保留太多的变量被称为**寄存器漫溢**（register spill），这会降低代码的性能。

如果变量出现在内部循环中，那么将该变量保存在寄存器中尤为重要。在计算

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

中，变量 c 可能会被编译器保存在寄存器中，而在

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

最好是显式地引入一个临时变量来保存 $c[k]$。在 C 语言中，你可以通过将变量声明为寄存器变量来提示编译器将变量保存在寄存器中：

```
register double t;
```

### 1.3.3 缓存

在包含指令即时输入和输出数据的寄存器和大量数据可以长期存放的内存之间，有各种级别的高速缓冲存储器，它们比内存具有更低的延迟和更高的带宽，并将数据保存一段时间。

数据从内存通过高速缓存到达寄存器的好处是，如果一组数据在第一次使用后不久就要被复用，由于它仍然在缓存中，因此访问的速度比从内存中引入数据要快得多。从历史的角度来看，存储器层次结构的概念早在 1946 年就已经被讨论过了 [25]，当时提出的原因是内存技术发展较为缓慢。

#### 1.3.3.1 示例

例如，假设一个变量 *x* 被使用了两次，它的两次使用间隔较大，以至于会留在寄存器中：

```
... = ... x ..... //执行x的指令
.........       //不涉及到x的一些指令
... = ... x ..... //执行x的指令
```

汇编代码为：

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

使用缓存，汇编代码保持不变，但内存系统的实际行为现在变成：

- 将 *x* 从内存加载到缓存中，并从缓存加载到寄存器中，执行操作；
- 执行中间其他的指令；
- 从内存中请求 *x*，但由于它仍然在缓存中，因此直接从缓存中加载，继续操作。

由于从缓存加载比从主存加载要快，因此计算速度将会更快。缓存的容量较小，所以数值不能无限期地保存在那里。我们将在下面的讨论中看到它的含义。

缓存和寄存器之间有一个重要的区别：虽然数据是通过显式的汇编指令移入寄存器的，但从内存到缓存的移动完全是由硬件完成的。因此，缓存的使用和重用不在程序员的直接控制范围之内。稍后，特别是在 1.6.2 和 1.7 节中，我们将看到如何间接影响缓存的使用。

#### 1.3.3.2 缓存标签

上面没有提及在缓存中找到某个项的机制，但每个缓存位置都有一个标记，以便我们有足够多的信息来找到某个缓存项与内存的映射。

#### 1.3.3.3 缓存级别、速度和大小

缓存通常被称为“level 1”和“level 2”（简称 L1 和 L2）缓存；有些处理器可能有 L3 缓存。L1 和 L2 缓存位于处理器的芯片上；L3 缓存则在芯片外部。L1 缓存的容量很小，通常在 16Kbyte 左右。相比之下，第 2 级 (如果有，则是第 3 级) 缓存容量则较大，最多可达几兆字节，但速度也随之下降。与可扩展的内存不同，缓存的大小是固定的。如果某个版本的处理器芯片上附带了一个较大的缓存，那么它的价格通常相当昂贵。

某些操作所需的数据在传送到处理器的过程中被复制到不同的缓存中。如果在一些指令之后，又需要一个数据项，计算机会首先在 L1 缓存中搜索它；如果没有找到，就在 L2 缓存中搜索；如果没有找到，就从内存中加载。在缓存中找到数据称为**缓存命中**（cache hit），没有找到数据则称为**缓存失效**（cache miss）。

图 1.5 展示了缓存层次结构的基本情况，本例是针对 Intel Sandy Bridge 芯片：缓存越接近 FPU，其速度越快，容量越小。

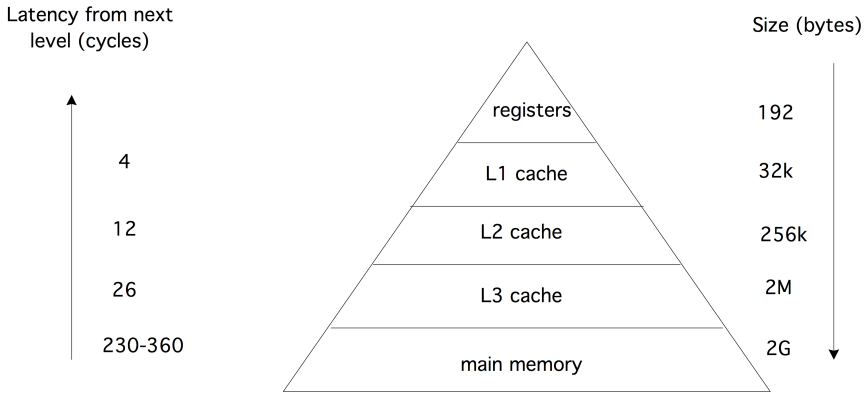



图 1.5: 英特尔 Sandy Bridge 的内存层次结构（速度与容量）

- 从寄存器加载数据是如此之快，以致于它不会成为阻碍算法执行速度的限制。另一方面，寄存器的数量很少。每个核心有 16 个通用寄存器和 16 个 SIMD 寄存器。
- L1 缓存很小，但是每个周期保持了 32 字节的带宽，即 4 个双精度数。这足以为两个操作分别加载两个操作数，但请注意，内核实际上每个周期可以执行 4 个操作。因此，为了达到峰值速度，某些操作数需要留在寄存器中：通常，L1 带宽足以满足大约一半的峰值性能。
- L2 和 L3 缓存的带宽名义上与 L1 相同。然而，部分带宽将在缓存一致性问题上浪费。
- 主内存访问带宽大于 100 个周期，带宽为 4.5 字节/周期，约为 L1 带宽的 1/7。然而，这个带宽是由一个处理器芯片的多个核心共享的，因此有效的带宽是这个数字的一个小数。大多数集群每个节点也有多个**插槽**（socket，即处理器芯片），通常是 2 或 4 个，因此一些带宽花费在维持缓存一致性上（参见 1.4 节），再次减少了每个芯片的可用带宽。

在 L1 上，指令和数据有单独的缓存；L2 和 L3 的缓存则同时包含数据和指令。可以看到，越来越大的缓存无法足够快地向处理器提供数据。因此，有必要以这样一种方式编码，即数据尽可能地保存在最高缓存级别。我们将在本章的其余部分详细讨论这个问题。

✍ **练习 1.5** L1 缓存比 L2 缓存小，如果有 L3 缓存，则 L2 要比 L3 小。给出一个实际的和理论上的原因。

#### 1.3.3.4 缓存失效的类型

缓存失效有三种类型。

正如在上面的例子中看到的，第一次引用数据时，总是会导致缓存丢失，这被称为**强制失效**（compulsory miss），因为这些是不可避免的。这是否意味着在第一次需要数据项时，我们要一直等待它？不一定：后面我们会解释硬件如何通过预测下一步需要什么数据来帮助运行。

第二种缓存失效是由于工作集的大小造成的：**容量失效**（capacity miss）是由于数据被覆盖，因为缓存不能包含所有问题数据。如果想要避免这种类型的失误，需要将问题划分为足够小的块，以便数据可以在缓存中停留相当长的时间。当然，这是在假设数据项被多次操作的前提下，所以把数据项保存在缓存中是有意义的。

最后，由于一个数据项被映射到与另一个相同的缓存位置而导致的**冲突失效**（conflict miss），而这两个数据项仍然是计算所需要的，并且可能有更好的候选者需要被驱逐。

在多核上下文中还有另外一种类型的缓存丢失：**无效失效**（invalidation miss）。如果缓存中的某个项因为另一个内核改变了相应内存地址的值而失效，就会发生这种情况。内核将不得不重新加载这个地址。

#### 1.3.3.5 复用是关键

设置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，以及如何充分利用缓存。第一次引用一个项时，它被从内存复制到缓存，并通过处理器的寄存器。缓存的存在并没有以任何

方式减少延迟和带宽。当同一项第二次被引用时，它可能在缓存中被找到，因此在延迟和带宽方面的成本大大降低：缓存比主存有更短的延迟和更高的带宽。

我们的结论是，程序员设计的算法必须有数据复用的机会。如果每个数据项只被使用一次（就像除了两个向量之外），不存在复用，则缓存的存在在很大程度上是无关紧要的。只有当缓存中的项被多次引用时，代码才会从缓存增加的带宽和减少的延迟中受益；例如，矩阵向量乘法 $y = Ax$，其中的每个元素 $x$ 都在操作中使用 $n$，$n$ 是矩阵维数。

其次，算法理论上可能有复用的机会，但需要以较为明显的复用方式进行编码，后者尤其重要。

有些问题很小，可以完全放在 L3 缓存中。这是在进行**基准测试**（benchmarking）时需要注意的一点，因为它对处理器性能的描述过于乐观。

#### 1.3.3.6 替换策略

高速缓存和寄存器中的数据仅由硬件决定，而非由程序员控制。同样地，当缓存或寄存器中的数据在一段时间内没有被引用，并且其他数据需要放在那里时，系统就会决定什么时候覆盖这些数据。下面，我们将详细介绍缓存如何做到这一点，但在整合一个总体原则，一个**最近最少使用**（Least Recently Used，LRU）替换：如果缓存已满，需要放入新数据，最近最少使用的数据从缓存中刷新，这意味着它是覆盖在新项目，因此不再访问。LRU 是目前最常见的替换策略；其他的策略还有：**先进先出**（First In First Out，FIFO）或**随机替换**。

✍ **练习 1.6** LRU 替换策略与直接映射缓存和关联缓存有什么关系？

✍ **练习 1.7** 举例论证 LRU 比 FIFO 更适合作为替代策略，并给出（伪）代码。

#### 1.3.3.7 缓存线

在内存和高速缓存之间或多个高速缓存之间的数据移动不是用单个字节，甚至字来完成的。相反，移动数据的最小单位称为**高速缓存线**（cache line），有时也称为**高速缓存块**（cache block）。一个典型的缓存行是 64 或 128 字节长，在科学计算中意味着 8 或 16 个双精度浮点数。移动到 L2 缓存的数据的缓存线大小可能比移动到 L1 缓存的数据大。

高速缓存线的设计初衷是便于实际应用：它简化了所涉及的电路。由于许多代码显示出空间局部性，缓存线的存在有着非同寻常的意义。

反之，我们现在需要利用数据的局部性编写高质量代码，因为任何内存访问都需要传输几个字符。在加载缓存线后，我们希望尽可能地使用一同加载进来的其他数据以实现高效利用资源，因为同一缓存线上的内容访实际上是自由且方便的。这种现象在通过**跨步访问**（stride access）数组的代码中是十分常见的：以规则的时间间隔读取或写入元素。

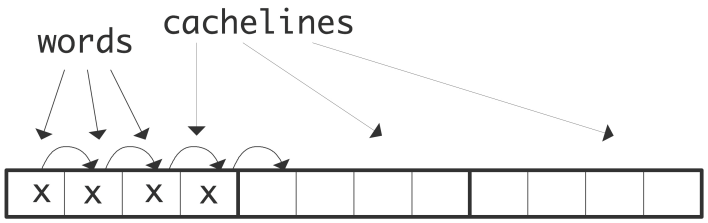



图 1.6: 在 Stride 1 中跨步访问 4 个元素

**Stride 1** 连续访问数组中的元素：

```
for (i=0; i<N; i++)
```

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

让我们用一个例子来说明：每个缓存线上有 4 个字。请求第一个元素将包含它的整个缓存线加载到缓存中。然后，对第 2、3 和 4 个元素的请求可以从缓存中得到满足，这意味着实现了高带宽和低延迟。**Stride 1** 跨步访问数组中的元素：

```
for (i=1; i<N; i+=stride)
  ... = ... x[i] .
```

意味着在每个缓存线中只有某些元素被使用。我们用 Stride = 3 来说明这一点：第一个元素加载一个缓存线，该也包含第二个元素。然而，第三个元素在下一个缓存上，因此加载它会引起主内存的延迟和带宽。第四个元素也是如此。加载四个元素现在需要加载三条缓存线而不是一条，这意味着三分之二的可用带宽被浪费了。(如果没有注意到常规访问模式的硬件机制，并先发制人地加载进一步的缓存线，第二种情况也会导致三倍于第一种情况的延迟)

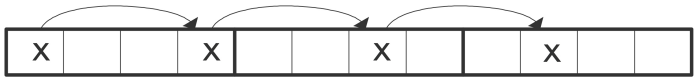



图 1.7: 在 Stride 3 中跨步访问 4 个元素

有些应用程序自然会导致大于 1 的进步，例如，只访问一个复数数组的实数部分。另外，使用递归加倍的方法通常具有非单位步长的代码结构

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

在这个关于缓存线的讨论中，我们隐式地假设缓存线的开头也是一个单词的开头，不管是整数还是浮点数。实际情况中往往并非如此：一个 8 字节的浮点数可以放置在两个缓存线之间的边界上。可以想象，这将极大程度上影响程序性能。

### 1.3.3.8 缓存映射

越接近 FPU 的缓存速度越快，其容量也更小。但即便最大的缓存也比内存容量小的多。在第 1.3.4.6 节中，我们将讨论如何做出保留哪些元素和替换哪些元素的决定。

现在我们将讨论缓存映射的问题，也就是“如果一个条目被放在缓存中，它会被放在哪里”的问题。这个问题通常是通过将项目的（主存）地址映射到缓存中的地址来解决的，这就导致了“如果两个项目映射到同一个地址会怎么样”的问题。

### 1.3.3.9 直接映射缓存

最简单的缓存映射策略是**直接映射**（direct mapping）。假设内存地址是 32 位长，因此它们可以寻址 4G 字节；进一步假设缓存有 8K 个字，也就是 64K 字节，需要 16 位来寻址。直接映射从每个内存地址取最后 (“最低有效”)16 位，并使用这些作为缓存中的数据项的地址；参见图 1.8。

直接映射的效率非常高，因为它的地址计算速度非常快，导致了较低的延迟，但在实际应用中存在一个问题。如果两个被 8K 字分隔的条目被寻址，它们将被映射到相同的缓存位置，这将使某些计算效率低下。例如：

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

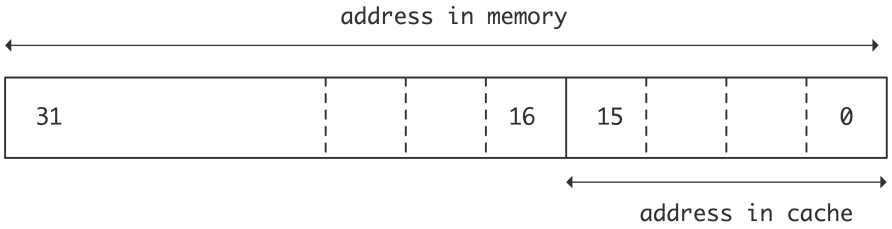



图 1.8: 32 位地址直接映射到 64K 高速缓存中

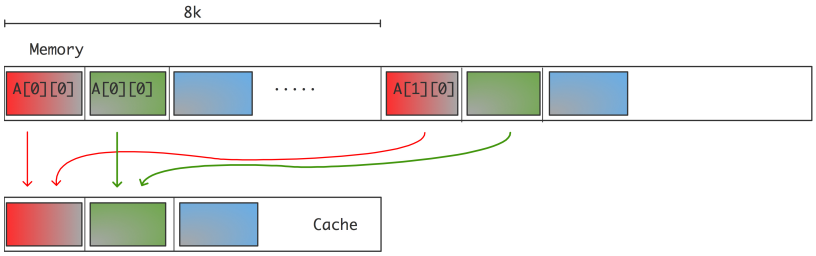



图 1.9: 直接映射缓存中的映射冲突

或在 Fortran 中：

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

此处，[0][$i$]，[1][$i$] 和 [2][$i$] (或者 $a(i,1)$，$a(i,2)$，$a(i,3)$) 的位置对于每个 $i$ 来说是 8K 的，所以它们地址的最后 16 位是相同的，因此它们将被映射到缓存中的相同位置；参见图 1.9。

循环的执行情况将如下：

- $a$[0][0] 处的数据被带入高速缓存和寄存器中，这产生了一定的延时。和这个元素一起，整个高速缓存行被转移。
- 在 [1][0] 处的数据被带入缓存（和寄存器中），连同其整个缓存行，以一些延迟为代价。由于这个缓存行和第一个缓存行被映射到相同的位置，所以第一个缓存行被覆盖。
- 为了写入输出，包含 $a$[2][0] 的缓存行被带入内存。这又被映射到同一位置，导致刚刚加载的 $a$[1][0] 的缓存行被刷新。
- 在下一次迭代中，需要 $a$[0][1]，它和 $a$[0][0] 在同一个缓存行。然而，这个缓存行已经被刷新了，所以它需要从主内存或更深的缓存层中被重新带入。在这样做的时候，它覆盖了保存 $a$[2][0] 的缓存行。
- $a$[1][1] 的情况类似：它在 $a$[1][0] 的缓存行上，不幸的是，它已经被上一步覆盖了。

如果一个缓存行有四个字，我们可以看到，循环的每四次迭代都涉及到八个 $a$ 元素的传输的元素，而如果不是因为缓存冲突，两个元素就足够了。

**练习 1.8** 在直接映射高速缓存的例子中，从内存到高速缓存的映射是通过使用 32 位内存地址的最后 16 位作为高速缓存地址完成的。如果使用前 16 位（"最有意义的"）作为缓冲区地址，那么这个例子中的问题就会消失。为什么在一般情况下这不是一个好的解决方案？

注释 到目前为止，我们一直假装缓存是基于虚拟内存地址的。实际上，缓存是基于内存中数据的物理地址，这取决于将虚拟地址映射到内存页的算法。

### 1.3.3.10 关联式缓存

如果任何数据项目都可以进入任何缓存位置，那么上一节中概述的缓存冲突问题就会得到解决。在这种情况下，除了缓存被填满之外，不会有任何冲突，在这种情况下，缓存替换策略（第 1.3.4.6 节）会刷新数据，为新来的项目腾出空间。这样的缓存被称为**全关联映射**（fully associative mapping）的，虽然这是最好的办法，但它的构建成本很高，而且在使用中比直接映射的缓存慢得多。

正因如此，最常见的解决方案是建立一个 **k-路关联缓存**（$k$-way associative mapping），其中 $k$ 至少是两个。在这种情况下，一个数据项可以进入任何一个 $k$ 缓存位置。代码必须要有 $k+1$ 路冲突，才会像上面的例子那样过早地刷新数据。在这个例子中，$k=2$ 的值就足够了，但在实践中经常会遇到更高的值。图 1.10 展示了一个直接映射的和一个三向关联的缓冲区的内存地址与缓冲区位置的映射情况。两个缓冲区都有 12 个元素，但是它们的使用方式不同。直接映射的高速缓存（左边）在内存地址 0 和 12 之间会有冲突，但是在三向关联高速缓存中，这两个地址可以被映射到三个元素中的任何一个。

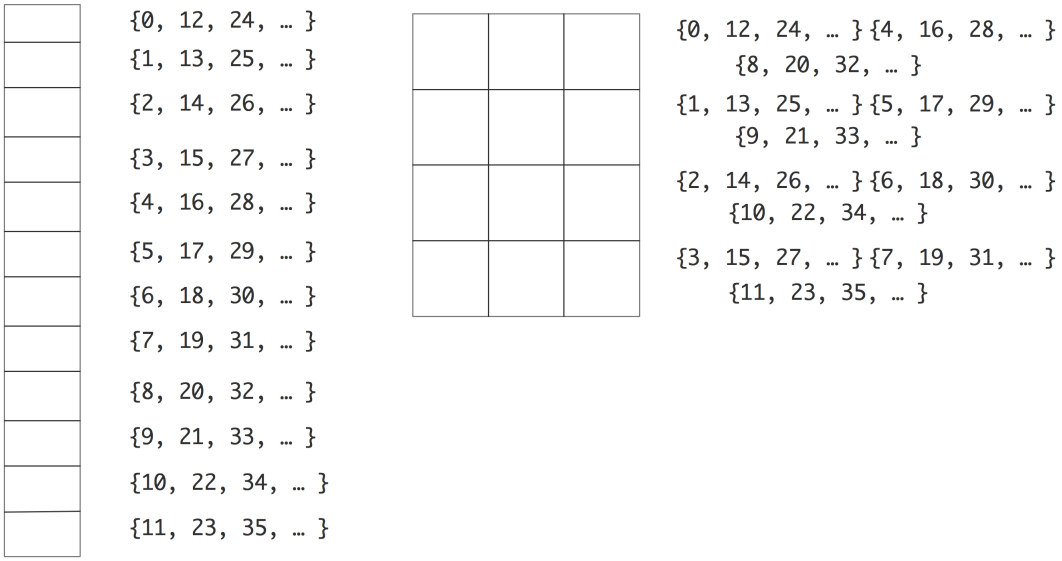



图 1.10: 两个 12 个元素的缓存：直接映射（左）和 3 路关联（右）

英特尔 Woodcrest 处理器有一个 32K 字节的 L1 缓存，它是 8 路设置的关联性，缓存行大小为 64 字节，L2 缓存为 4M 字节，是 8 路设置的关联性，缓存行大小为 64 字节。另一方面，AMD Barcelona 芯片的 L1 缓存是 2 路关联，L2 缓存是 8 路关联。更高路关联性显然是可取的，但是会使处理器变得更慢，因为确定一个地址是否已经在缓存中变得更加复杂。由于这个原因，在速度最重要的地方，L1 高速缓存的关联性通常比 L2 低。

练习 1.9 用你喜欢的语言写一个小型的高速缓存模拟器。假设一个有 32 个条目的 $k$ 方式的同构缓存和一个 16 位地址的架构。对 $k = 1, 2, 4, \ldots$ 进行以下实验。

1. 让 $k$ 模拟高速缓存的关联性。
2. 写下从 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，并模拟将其存储在缓存中。由于高速缓存有 32 个条目，最佳情况下这 32 个地址都可以存储在高速缓存中。这种情况实际发生的几率很小，往往一个地址的数据会在另一个地址与之冲突时被驱逐出缓存（意味着它被覆盖）。记录在模拟结束时，32 个地址中，有多少地址被实际存储在缓存中。将步骤 3 做 100 次，并绘制结果；给出中位数和平均值，以及标准偏差。观察一下，增加关联性可以提高存储地址的数量。其极限行为是什么？（为了获得奖励，请做一个正式的统计分析）

#### 1.3.3.11 缓存内存与普通内存的对比

那么，缓冲存储器有什么特别之处；为什么我们不把它的技术用于所有的存储器?

缓存通常由静态随机存取存储器（SRAM）组成，它比用于主存储器的动态随机存取存储器（DRAM）更快，但也更昂贵，每一位需要 5-6 个晶体管，而不是一个，而且耗电量更大。

#### 1.3.3.12 加载与存储

在上述描述中，在程序中访问的所有数据都需要在使用这些数据的指令执行之前被移入高速缓存。这对读取的数据和写入的数据都适用。然而，已经写入的数据，如果不再需要（在一定的合理时间内），就没有理由留在缓存中，可能会产生冲突或驱逐仍然可以重复使用的数据。出于这个原因，编译器通常支持流式存储：一个纯粹输出的连续数据流将被直接写入内存，而不被缓存。

### 1.3.4 预取流

在传统的冯 · 诺依曼模型中（第 1.1 节），每条指令都包含其操作数的位置，所以实现这种模型的 CPU 会对每个新的操作数进行单独请求。在实践中，往往后续的数据项在内存中是相邻的或有规律的间隔。内存系统可以通过查看高速缓存的数据模式来检测这种数据模式，并请求**预取数据流**（prefetch data stream）；

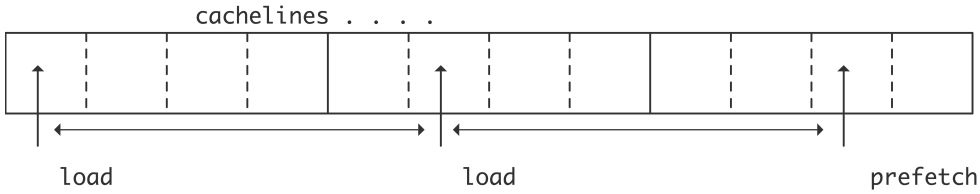



图 1.11: 由等间隔请求产生的预取流

在最简单的形式下，CPU 会检测到来自两个连续的高速缓存行的连续加载，并自动发出对接下来的高速缓存行的请求。如果代码对第三条高速缓存线发出实际请求，这个过程可以重复或扩展。由于这些高速缓存行现在是在需要提前从内存中取出，所以预取有可能消除除前几个数据项之外的所有延迟。

现在我们需要重新审视一下缓存失效的概念。从性能的角度来看，我们只对缓存失效的停顿感兴趣，也就是说，在这种情况下，计算必须等待数据被带入。不在缓存中的数据，但在其他指令还在处理的时候可以被带入，这不是一个问题。如果"L1 缺失" 被理解为只是" 缺失时的停顿"，那么术语"L1 缓存重新填充" 被用来描述所有的缓存线负载，无论处理器是否在它们上面停顿。

由于预取是由硬件控制的，所以它也被描述为**硬件预取**（hardware prefetch）。预取流有时可以从软件中控制，例如通过**源语**（intrinsic）。

由程序员引入预取是对一些因素的谨慎平衡 [94]。其中最重要的是预取距离：从预取开始到需要数据时的周期数。在实践中，这通常是一个循环的迭代次数：预取指令请求未来迭代的数据。

### 1.3.5 并发和内存传输

在关于内存层次的讨论中，我们提出了一个观点：内存比处理器慢。如果这还不够糟糕的话，利用内存提供的所有带宽甚至不是小事。换句话说，如果你不仔细编程，你会得到比你根据可用带宽所期望的更少的性能。让我们来分析一下。

内存系统的带宽通常为每周期一个以上的浮点数，所以你需要每周期发出那么多请求来利用可用的带宽。即使在零延迟的情况下也是如此；由于存在延迟，数据从内存中出来并被处理需要一段时间。因此，任何基于第一个数据的计算而请求的数据都必须在延迟至少等于内存延迟的情况下请求。

**定理 1.1 (Little 定理)**

为了充分利用带宽，在任何时候都必须有相当于带宽乘以延迟的数据量在运行。由于这些数据必须是独立的，我们得到了

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

维护并发性的问题并不是程序没有这种并发性，而是程序要让编译器和运行时系统识别它。例如，如果一个循环遍历了一个长的数组，编译器就不会发出大量的内存请求。预取机制会提前发出一些内存请求，但通常不够。因此，为了使用可用的带宽，多个数据流需要同时进行。因此，我们也可以将 **Little 定理**表述为

$$\text{有效吞吐量} = \text{并发性}/\text{延迟}$$

这在图1.12中得到了说明。

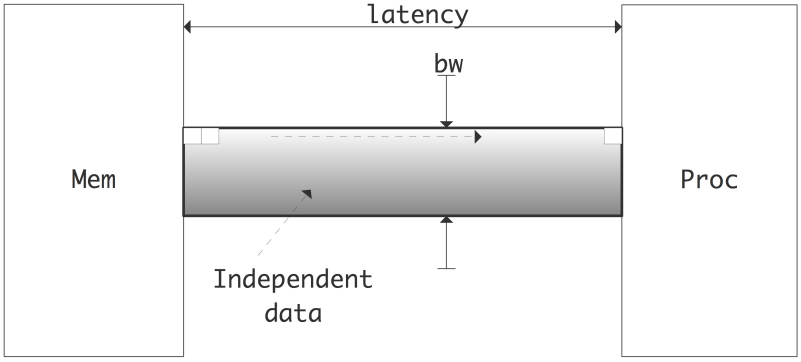



图 1.12: Little 定律的说明，该定律规定了运行中需要多少独立的数据

### 1.3.6 内存 bank

上面，我们讨论了与带宽有关的问题。可以看到：内存以及在较小程度上的缓存，其带宽低于处理器可以最大限度地吸收的带宽。这种情况实际上比上面的讨论看起来还要糟糕。由于这个原因，内存通常被分为交错的内存组：在四个内存组中，字 0、4、8... 在 0 组，字 1、5、9... 在 1 组，依此类推。

假设我们现在按顺序访问内存，那么这样的 4 路交错式内存可以维持 4 倍于单个内存组的带宽。不幸的是，按跨度 2 访问将使带宽减半，而更大的跨度则更糟糕。如果两个连续的操作访问同一个内存 bank，我们就会说到内存 bank 冲突 [7]。在实践中，内存库的数量会更多，因此，小跨度的内存访问仍然会有完整的广告带宽。例如，Cray-1 有 16 个 banks，而 Cray-2 有 1024 个。

✍ **练习 1.10** 证明在有质数的 bank 时，任何达到该质数的跨步都是无冲突的。你认为为什么这个解决方案没有在实际的内存架构中被采用?

在现代处理器中，DRAM 仍然有 bank，但由于有缓存的存在，其影响较小。然而，GPU 有内存组，没有缓存，所以它们遭受了与老式超级计算机相同的一些问题。

✍ **练习 1.11** 下面为对一个数组的元素进行求和的递归加倍算法：

```
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 bank 冲突。假设 $n = 2^p$，bank 有 $2^k$ 元素，其中 $k < p$。同时考虑到这是一个并行算法，内循环的所有迭代都是独立的，因此可以同时进行。另外，我们可以使用递归减半法。

```
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 bank 的混乱情况。这种算法更好吗？在并行情况下呢？

缓存存储器也可以使用 bank。例如，AMD 巴塞罗那芯片的 L1 缓存中的缓存线是 16 个字，分为两个 8 个字的交错库。这意味着对高速缓存线的元素进行顺序访问是有效的，但串联访问的性能就会下降。

### 1.3.7 TLB、页和虚拟内存

一个程序的所有数据可能不会同时出现在内存中。这种情况可能由于一些原因而发生：

- 计算机为多个用户服务，所以内存并不专门用于任何一个用户。
- 计算机正在运行多个程序，这些程序加起来需要的内存超过了物理上可用的内存。
- 一个单一的程序所使用的数据可能超过可用的内存。

基于这些原因，计算机使用**虚拟内存**：如果需要的内存比可用的多，某些内存块会被写入磁盘。实际上，磁盘充当了真实内存的延伸。这意味着一个数据块可以出现在内存的任何地方，事实上，如果它被换入和换出，它可以在不同时间出现在不同位置。交换不是作用于单个内存位置，而是作用于内存页：连续的内存块，大小从几千字节到几兆字节（在早期的操作系统中，将内存移至磁盘是程序员的责任。互相替换的内存页被称为覆盖层）。

由于这个原因，我们需要一个从程序使用的内存地址到内存中实际地址的翻译机制，而且这种翻译必须是动态的。一个程序有一个" 逻辑数据空间"（通常从地址 0 开始），是编译后的代码中使用的地址，在程序执行过程中需要将其翻译成实际的内存地址。出于这个原因，有一个页表，指定哪些内存页包含哪些逻辑页。

#### 1.3.7.1 大页

在非常不规则的应用中，例如数据库，页表会变得非常大，因为更多或更少的随机数据被带入内存。然而，有时这些页面显示出某种程度的集群，这意味着如果页面大小更大，需要的页面数量将大大减少。由于这个原因，操作系统可以支持大的页面，通常大小为 2Mb 左右。(有时会使用' 巨大的页面'；例如，英特尔 Knights Landing 有 Gigabyte 页面）

大页面的好处取决于应用：如果小页面没有足够的集群，使用大页面可能会使内存过早地被大页面的未使用部分填满。

#### 1.3.7.2 TLB

然而，通过查找该表进行地址转换是很慢的，所以 CPU 有一个**转译后备缓冲器**（Translation Look-aside Buffer，TLB）。TLB 是一个经常使用的页表项的缓存：它为一些页提供快速的地址转换。如果一个程序需要一个内存位置，就会查询 TLB，看这个位置是否真的在 TLB 所记忆的页面上。如果是这样，逻辑地址就被翻译成物理地址；这是一个非常快速的过程。在 TLB 中没有记住该页的情况被称为 TLB 缺失，然后查询页面查找表，如果有必要，将需要的页面带入内存。TLB 是（有时是完全）关联的，使用 LRU 策略。

一个典型的 TLB 有 64 到 512 个条目。如果一个程序按顺序访问数据，它通常只在几个页面之间交替进行，而且不会出现 TLB 缺失。另一方面，一个访问许多随机内存位置的程序可能会因为这种错过而出现速度下降。目前正在使用的页面集被称为" 工作集"。

[这个故事有一些复杂的情况。例如，通常有一个以上的 TLB。第一个与 L2 缓存相关，第二个与 L1 相关。在 AMD Opteron 中，L1 TLB 有 48 个条目，并且是完全（48 路）关联的，而 L2 TLB 有 512 个条目，但只是 4 路关联的。这意味着实际上可能存在 TLB 冲突。在上面的讨论中，我们只谈到了 L2 TLB。之所以能与 L2 缓存而不是主内存相关联，是因为从内存到 L2 缓存的转换是确定性的]。

使用大页也可以减少潜在的 TLB 缺失次数，因为可以减少工作页的集合。

## 1.4 多核架构

传统的处理器芯片设计已经达到了性能的极限。主要由于

1. 主频已经不能再增加，因为这会增大功耗，使芯片发热过大；
2. 从代码中提取更多的**指令级并行**（Instruction Level Parallelism，ILP）变得困难，要么是由于编译器的限制，要么是由于内在可用的并行量有限，要么是由于分支预测使之无法实现。

从单个处理器芯片中获得更高的利用率的方法之一是，从进一步完善单个处理器的策略，转向将芯片划分为多个处理" 核心"。这些独立的内核可以在不相关的任务上工作，或者通过引入数据的并行性，以更高的整体效率协作完成一个共同的任务 [163]。

注释 另一个解决方案是英特尔的超线程（hyperthreading），它可以让一个处理器混合几个指令流的指令。这方面的好处在很大程度上取决于具体情况。然而，这种机制在 GPU 中也得到了很好的利用，后面我们会讨论。

这就解决了上述两个问题：

1. 两个主频较低的内核可以拥有与主频较高的单个处理器相同的吞吐量；因此，多个内核的能效更高。
2. 指令级并行现在被明确的任务并行化所取代，由程序员管理。

虽然第一代多核 CPU 只是在同一个芯片上的两个处理器，但后来的几代 CPU 都加入了 L3 或 L2 缓存，在两个处理器核心之间共享；见图 1.13。

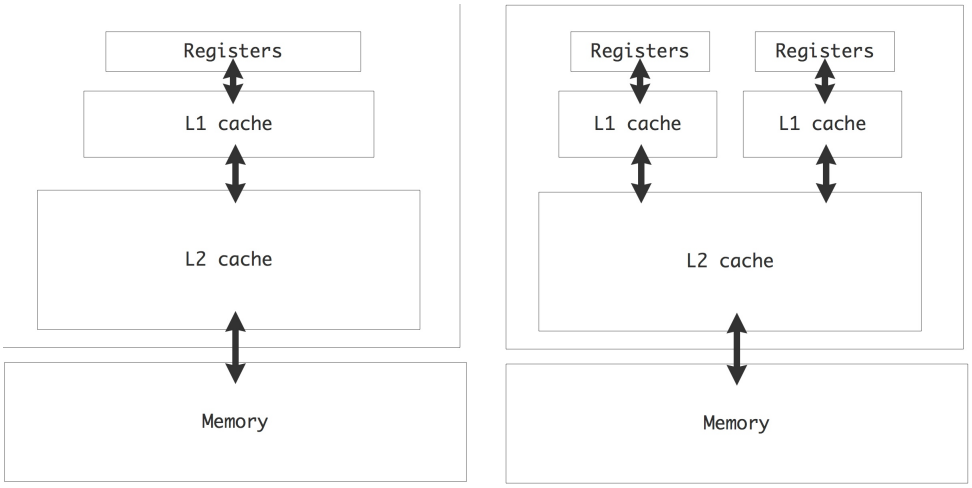



图 1.13: 单核和双核芯片的高速缓存层级结构

这种设计使内核能够有效地联合处理同一问题。内核仍然会有自己的 L1 高速缓存。而这些独立的高速缓存造成了高速缓存的一致性问题。

我们注意到，" 处理器" 这个词现在是模糊的：它可以指芯片，也可以指芯片上的处理器核心。由于这个原因，我们大多谈论的是整个芯片的**插槽**（socket）和包含一个算术和逻辑单元并有自己的寄存器的部分的**核心**（core）。目前，具有 4 或 6 个内核的 CPU 很常见，甚至在笔记本电脑中也是如此，英特尔和 AMD 正在销售 12 个内核的芯片。核心数量在未来可能会上升。英特尔已经展示了一个 80 核的原型，它被开发成 48 核的" 单芯片云计算机"，如图 1.14 所示。这个芯片的结构有 24 个双核" 瓦片"，通过一个二维网状网络连接。只有某些瓦片与内存控制器相连，其他的瓦片除了通过片上网络外，无法到达内存。

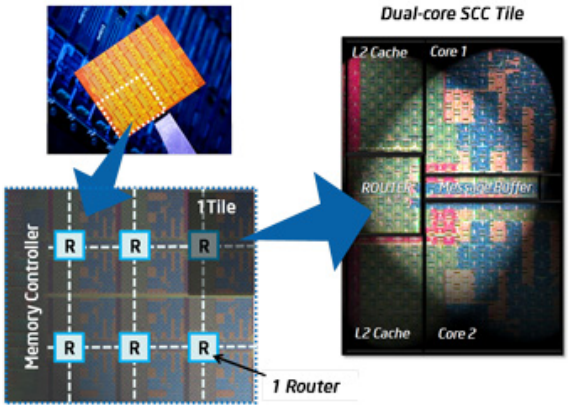



图 1.14: 英特尔单芯片云计算机芯片的结构

通过这种共享和私有缓存的混合，多核处理器的编程模型正在成为共享和分布式内存的混合体。

- 核心（Core）：各个内核都有自己的私有 L1 缓存，这是一种分布式内存。上面提到的英特尔 80 核原型，其核心以分布式内存的方式进行通信。
- 插槽（Socket）：在一个插座上，通常有一个共享的二级缓存，这是内核的共享内存。
- 节点（Node）：在一个" 节点" 或主板上可以有多个插座，访问同一个共享内存。
- 网络（Network）：需要分布式内存编程（见下一章）来让节点进行通信。

从历史上看，多核结构在多处理器共享内存设计中已有先例（第 2.4.1 节），如 Sequent Symmetry 和 Alliant FX/8。从概念上讲，程序模型是相同的，但现在的技术允许将多处理器板缩小到多核芯片上。

### 1.4.1 缓存一致性

在并行处理中，如果一个以上的处理器拥有同一个数据项的副本，就有可能发生冲突失效。确保所有的缓存数据都是主内存的准确副本，这个问题被称为**缓存一致性**（cache coherence）：如果一个处理器改变了它的副本，则另一个副本也需要更新。

在分布式内存架构中，数据集通常在处理器上被不连续地分割，所以只有在用户知道的情况下才会出现数据的冲突副本，而处理这个问题则是由用户决定的。共享内存的情况更微妙：由于进程访问相同的主内存，似乎冲突实际上是不可能的。然而，处理器通常有一些私有缓存，包含来自内存的数据副本，所以冲突失效的副本可能发生。这种情况特别出现在多核设计中。

假设两个核在它们的（私有）L1 高速缓存中有一个相同数据的副本，其中一个修改了它的副本。现在另一个核心的缓存数据不再是其对应的准确副本：处理器将使该项目副本失效，事实上也是其整个缓存线失效。当该进程需要再次访问该项目时，它需要重新加载该缓存线。另一种方法是，任何改变数据的内核都要将该缓存线发送给其他内核。这个策略的开销可能更大，因为其他内核不可能有一个缓存线的副本。

这个更新或废止缓存线的过程被称为**维护缓存一致性**（maintaining cache coherence），它是在处理器的一个非常低的层次上完成的，不需要程序员参与。(这使得更新内存位置成为一个原子操作；关于这一点，见 2.6.1.5 节)。然而，这将减慢计算速度，并且浪费了核心的带宽，而这些带宽本来是可以用来加载或存储操作数的。

缓存行相对于主存中的数据项的状态通常被描述为以下几种情况之一。

- Scratch：缓存行不包含该项目副本。
- Valid：缓存行是主内存中数据的正确拷贝。
- Reserved：缓存行是该数据的唯一副本。
- Dirty：缓存行已被修改，但尚未写回主内存。
- Invalid：缓存线上的数据在其他处理器上也存在（它没有被保留），并且另一个进程修改了它的数据副本。

一个更简单的变体是修改后的共享无效（MSI）一致性协议，在一个给定的核心上，一个缓存线可以处于以

下状态。

- Modified：缓存线已经被修改，需要写到备份仓库。这个写法可以在行被驱逐时进行，也可以立即进行，取决于回写策略。
- Shared：该行至少存在于一个缓存中且未被修改。
- Invalid：该行在当前缓存中不存在，或者它存在但在另一个缓存中的副本已被修改。

这些状态控制着高速缓存线在内存和高速缓存之间的移动。例如，假设一个核对一个缓存线进行读取，而这个缓存线在该核上是无效的。然后，它可以从内存中加载它，或者从另一个高速缓存中获取它，这可能更快。(找出一个行是否存在于另一个缓冲区（状态为 M 或 S）被称为**监听**（snooping）；另一种方法是维护**标签目录**（tag directory）。如果该行是共享的，现在可以简单地复制；如果它在另一个高速缓存中处于 M 状态，该核心首先需要把它写回内存。

✍ **练习 1.12** 考虑两个处理器，内存中的一个数据项 $x$，以及这两个处理器的私有缓存中的缓存线 $x_1, x_2$，这两个缓存线被映射到这两个处理器。描述在两个处理器上读写 $x_1$ 和 $x_2$ 的状态之间的转换。同时指出哪些行为会导致内存带宽被使用（这个过渡列表是一个有限状态自动机（FSA），见第 19 节）。

MSI 协议的变种增加了一个" 独占" 或" 拥有" 状态，以提高工作效率。

#### 1.4.1.1 缓存一致性的解决方案

有两种实现缓存一致性的基本机制：**监听**（snooping）和基于**标签目录**（tag directory）的方案。

在监听机制中，任何对数据的请求都会被发送到所有的缓冲区，如果数据存在于任何地方，就会被返回；否则就会从内存中检索。这个方案的一个变形为，一个核心" 监听" 所有的总线流量，这样当另一个核心修改它的拷贝时，它就可以使自己的缓存线拷贝失效或更新。缓存失效比更新的代价要小，因为它是一个位操作，而更新涉及到复制整个高速缓存线。

✍ **练习 1.13** 什么条件下更新才是更优方案？写一个简单的缓存模拟器来评估这个问题。

由于监听通常涉及到向所有核心广播信息，所以它的规模不能超过最少的核心数量。一个可以更好地扩展的解决方案是使用一个标签目录：一个中央目录，它包含了一些缓存中存在的数据的信息，以及具体在哪个缓存中。对于拥有大量内核的处理器（如英特尔 Xeon Phi），该目录可以分布在各个内核上。

#### 1.4.1.2 伪共享

如果内核访问不同的项目，就可能会出现缓存一致性问题。例如，

```
double x,y;
```

可能会在内存中紧挨着分配 x 和 y，所以它们很有可能落在同一个缓存线上。现在，如果一个核心更新 x，另一个更新 y，这个缓存线就会在核心之间不断移动。这就是所谓的**伪共享**（false sharing）。最常见的伪共享情况发生在线程更新一个数组的连续位置时。例如，在下面的 OpenMP 片段中，所有线程都在更新自己在部分结果数组中的位置。

```
  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)
```

虽然没有实际的竞争条件（如果线程都更新 *global_result* 变量就会有），但这段代码的性能会很低，因为带有 *local_result* 数组的缓存线会不断被废止。

#### 1.4.1.3 标签目录

在具有分布式但连贯的缓存的多核处理器中（如英特尔 Xeon Phi），标签目录本身也可以是分布式的。这增加了缓存查找的延迟。

### 1.4.2 在多核芯片上进行计算

多核处理器可以通过各种方式提高性能。首先，在桌面情况下，多个内核实际上可以运行多个程序。更重要的是，我们可以利用并行性来加快单个代码的执行速度。这可以通过两种不同的方式实现。MPI 库通常用于通过网络连接的处理器之间的通信。然而，它也可以在单个多核处理器中使用：然后通过共享内存拷贝实现 MPI 调用。另外，我们可以使用共享内存和共享缓存，并使用线程系统，如 OpenMP（第 2.6.2 节）进行编程。这种模式的优点是并行性可以更加动态，因为运行时系统可以在程序运行过程中设置和改变线程和内核之间的对应关系。我们将比较详细地讨论多核芯片上线性代数操作的调度。

### 1.4.3 TLB shootdown

前文解释了 TLB 是如何用于缓存从逻辑地址，也就是逻辑页到物理页的转换。TLB 是插槽内存单元的一部分，所以在多插槽设计中，一个插槽上的进程有可能改变页面映射，这使得其他插槽的映射不正确。解决这个问题的一个办法叫做 TLB shootdown：改变映射的进程会产生一个处理器间中断，从而导致其他处理器重建他们的 TLB。

## 1.5 节点架构和插槽

在前面的章节中，我们已经通过内存层次结构，访问了寄存器和各种缓存级别，以及它们可以被私有或共享的程度。在内存层次结构的最底层是所有内核共享的内存。它的范围从低级别的笔记本电脑的几千兆字节到一些超级计算机中心的几兆字节。

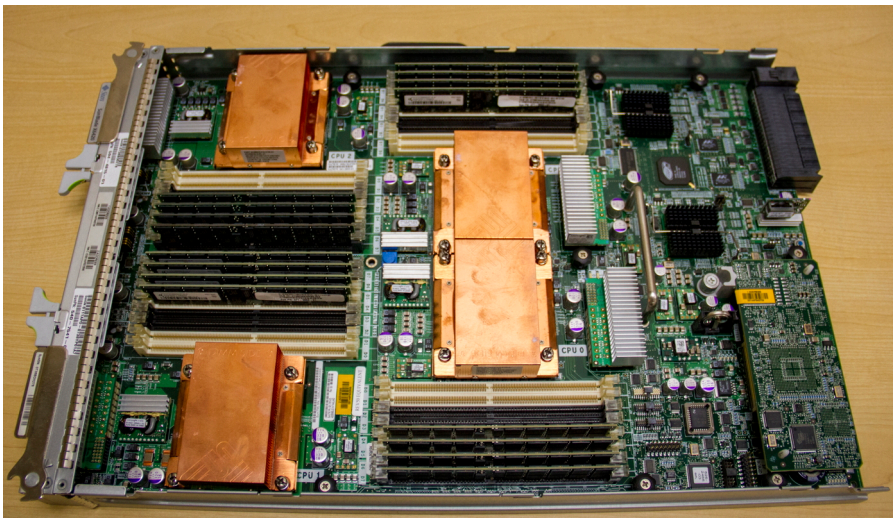


图 1.15: 四插槽设计

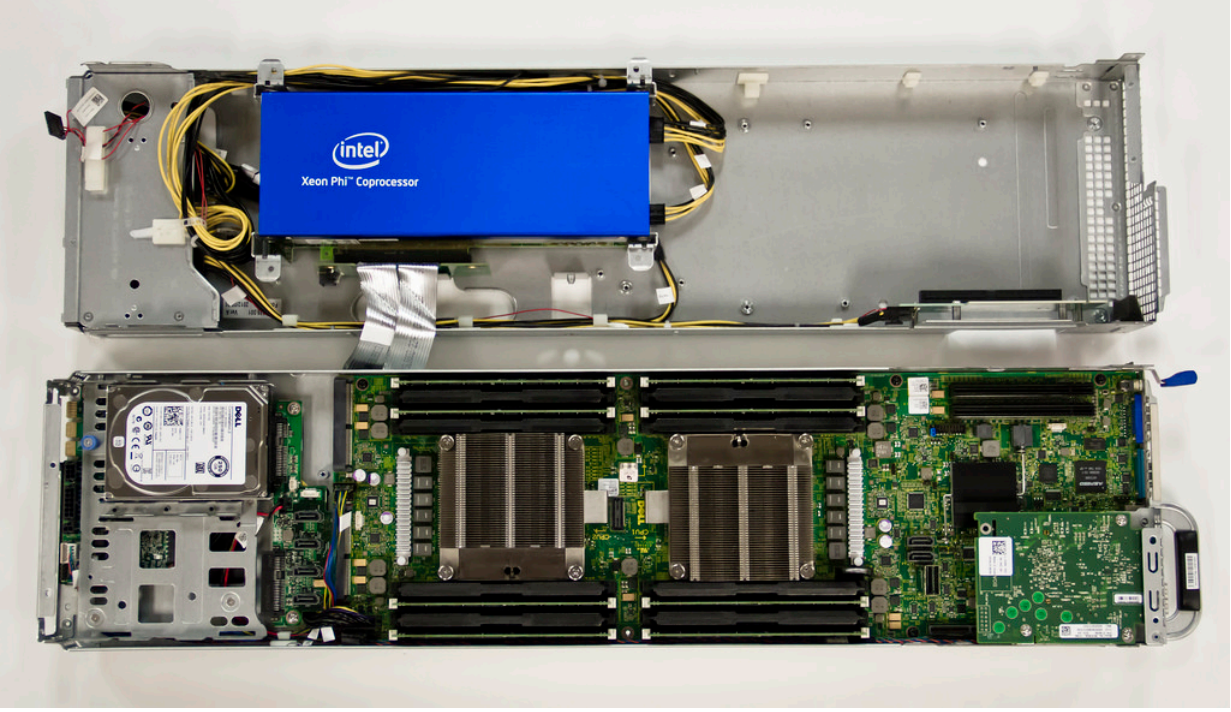



图 1.16: 带有协处理器的双插槽设计

虽然这个内存是在所有核心之间共享的，但它有一些结构。这源于一个事实，即集群节点可以有一个以上的插槽，即处理器芯片。节点上的共享内存通常分布在直接连接到一个特定插槽的库中。例如，图 1.15 显示了 TACC Ranger 集群超级计算机（已停产）的四插槽节点和 TACC Stampede 集群超级计算机的两插槽节点，后者包含一个英特尔至强 Phi 协处理器。在这两个设计中，你可以清楚地看到直接连接到插槽上的内存芯片。

这是一个**非一致内存访问**（Non-Uniform Memory Access，NUMA）架构的例子：对于在某个核心上运行的进程，连接到其插槽上的内存比连接到另一个插座上的内存访问速度略快。

这方面的一个结果就是 First-touch 现象。动态分配的内存在第一次被写入之前实际上并没有被分配。现在考虑下面的 OpenMP 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] ...
```

由于 First-touch，数组被完全分配到插槽的主线程内存上。在随后的并行循环中，其他插槽的核心将对它们操作的内存有较慢的访问。

这里的解决方案是将初始化循环也做成并行的，即使其中的工作量几乎可以忽略不计。

## 1.6 局部性和数据复用

算法的执行不仅包含计算操作，也包含数据传输部分，事实上，数据传输可能是影响算法效率的主要因素。由于缓存和寄存器的存在，数据传输量可以通过编程的方式最小化，使数据尽可能地留在处理器附近。这部分是一个巧妙编程的问题，但我们也可以看看理论上的问题：算法是否一开始就允许这样做。

事实证明，在科学计算中，数据往往主要与在某种意义上靠近的数据互动，这将导致数据的局部性；1.6.2 节。通常这种局部性来自于应用的性质，就像第四章看到的 PDEs 的情况。在其他情况下，如分子动力学（第 7

章），没有这种内在的局部性，因为所有的粒子都与其他粒子相互作用，为了获得高性能，需要相当的编程技巧。

#### 1.6.0.1 数据复用和计算密度

在前面的章节中，我们了解到处理器的设计有些不平衡：加载数据比执行实际操作要慢。这种不平衡对于主存储器来说是很大的，而对于各种高速缓存级别来说则较小。因此，我们有动力将数据保存在高速缓存中，并尽可能地保持数据的复用量。我们首先需要确定计算是否允许数据复用，为此我们定义计算密度如下：

**定义 1.3 (计算密度)**

如果 $n$ 是一个算法所操作的数据项的数量，而 $f(n)$ 是它所需要的操作的数量，则计算密度为

$$f(n)/n$$

我们可以用浮点数或字节来衡量数据项。后者更容易与处理器的硬件规格相关联。计算密度也与延迟隐藏有关：即可以减轻计算活动背后的数据加载对性能的负面影响的概念。要做到这一点，我们需要比数据加载更多的计算来使这种隐藏有效。而这正是计算密度的定义：每一个字节/字/数字加载的高比率操作。 ♣

#### 1.6.0.2 示例：向量操作

考虑到向量加法

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

这涉及到三次内存访问（两次加载和一次存储）和每次迭代的一次操作，给出的算术强度为 1/3。axpy（表示"$a$ 乘以 $x$ 加 $y$ "）操作

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

有两个操作，但内存访问的数量相同，因为 $a$ 的一次性负载被摊销了。因此，它比简单的加法更有效率，重用率为 2/3。因此，它比简单的加法更有效，重用率为 2/3。

内积计算

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

在结构上类似于 axpy 操作，每次迭代涉及一个乘法和加法，涉及两个向量和一个标量。然而，现在只有两个加载操作，因为 $s$ 可以保存在寄存器中，只在循环结束时写回内存。这里的重用是 1。

#### 1.6.0.3 示例：矩阵操作

考虑矩阵乘法

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

这涉及 $3n^2$ 个数据项和 $2n^3$ 个运算，属于高阶运算。算术强度为 $O(n)$，每个数据项将被使用 O(n) 次。这意味着，通过适当的编程，这种操作有可能通过将数据保存在快速缓存中来克服带宽/时钟速度的差距。

✍ **练习 1.14** 根据上述定义，矩阵-矩阵乘积作为一种操作，显然具有数据重用性。矩阵-矩阵乘积显然具有数据复用。请你论证一下，这种复用并不是自然形成的，是什么决定了初始算法中国呢缓存是否对数据进行复用?

[在这次讨论中，我们只关心某个特定实现的操作数，而不是数学操作。例如，有一些方法可以在少于 $O(n^3)$ 的操作中执行矩阵-矩阵乘法和高斯消除算法 [189, 167]。然而，这需要不同的实现方式，在内存访问和重用方面有自己的分析]。

矩阵与矩阵乘积是 LINPACK 基准 [51] 的核心；见 2.11.4 节。如果将其作为评价计算机从能的标准则结果可能较为乐观：矩阵与矩阵的乘积是一个具有大量数据复用的操作，因此这对内存带宽并不敏感，对于并行计

算及而言，这对网络通信也并不敏感。通常情况下，计算机在 Linpack 基准测试中会达到其峰值性能的 60-90%，而其他测试标准得到的数值则可能较低。

#### 1.6.0.4 Roofline 模型

有一种评价计算及性能的理想模型，就是所谓的**屋脊线**（roofline model）模型 [202]，该模型指出：性能受两个因素的制约，如图 1.16 的第一个图所示。

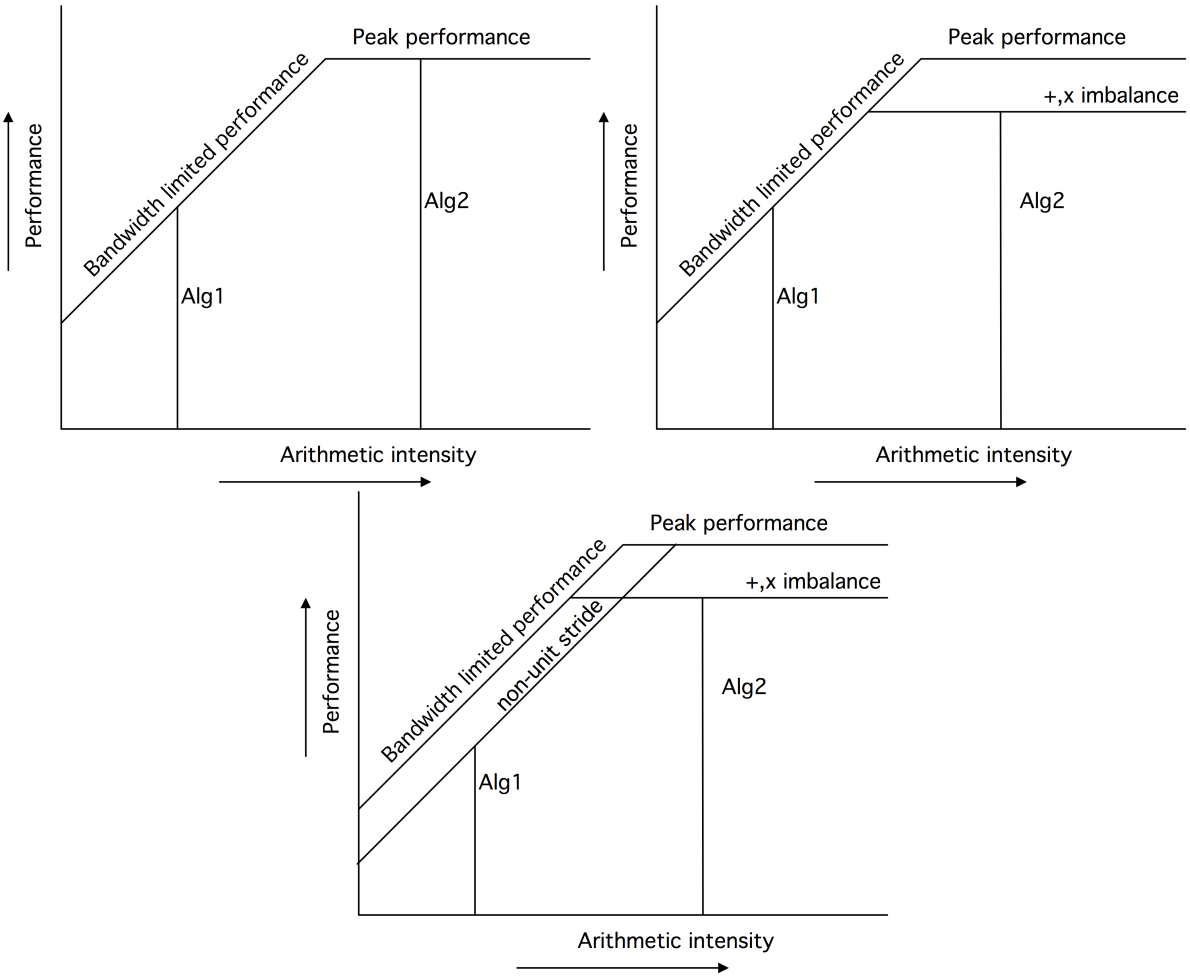



**图 1.17:** 决定性能因素的 Roofline 模型

1. 图中顶部的横线所表示的峰值性能是对性能的绝对约束，只有在 CPU 的各个方面（流水线、多个浮点单元）都完美使用的情况下才能达到。这个数字的计算纯粹是基于 CPU 的特性和时钟周期；假定内存带宽不是一个限制因素。
2. 每秒的操作数受限于带宽、绝对数和计算密度的乘积。

$$= \frac{\text{数据项}}{\text{数据项}}$$

Roofline 模型优雅地指出了影响性能的因素。例如，如果一个算法没有使用全部的 SIMD 宽度，这种不平衡会降低可达到的峰值。图 1.16 中的第二张图显示了降低上限的各种因素。还有各种降低可用带宽的因素，比如不完善的数据隐藏。这在第三张图中由倾斜的屋顶线的降低表示。

**定义 1.4 (计算约束与带宽约束)**

对于一个给定的计算密度，其性能是由其垂直线与 Roofline 相交的位置决定的。如果这是在水平部分，那么该计算被称为受**计算约束**（compute-bound）：性能由处理器的特性决定，而带宽不是问题。另一方面，如果这条垂直线与屋顶的倾斜部分相交，那么计算被称为受**带宽约束**（bandwidth-bound）：性能由内存子系统决定，处理器的全部能力没有被使用。

♣

✍ **练习 1.15** 如何确定一个给定的程序内核是受到带宽约束还是计算约束的?

#### 1.6.0.5 局部性

由于从缓存中读取数据的时间开销要小于从内存中读取，我们当然希望以这种方式进行编码，进而使缓存中的数据最大程度上得到复用。虽然缓存中的数据不受程序员的控制，甚至编写汇编语言也无法控制（在 Cell 处理器和一些 GPU 中，低级别的内存访问可以由程序员控制），但在大多数 CPU 中，知道缓存的行为，明确什么数据在缓存中，并在一定程度上控制它，还是有可能的。

这里的两个关键概念是**时间局部性**（temporal locality）和**空间局部性**（spatial locality）。时间局部性是最容易解释的：即数据使用一次后短时间内再次被使用。由于大多数缓存使用 LRU 替换策略，如果在两次引用之间被引用的数据少于缓存的大小，那么该元素仍然会存在缓存之中，进而实现快速访问。而对于其他的替换策略，例如随机替换，则不能保证同样结果。

#### 1.6.0.6 时间局部性

下面为时间局部性的例子，考虑重复使用一个长向量：

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

$x$ 的每个元素将被使用 10 次，但是如果向量（加上其他被访问的数据）超过了缓存的大小，每个元素将在下一次使用前被刷新。因此，$x[i]$ 的使用并没有表现出时间局部性：再次使用时的时间间隔太远，使得数据无法停留在缓存中。

如果计算的结构允许我们交换循环。

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

$x$ 的元素现在被反复使用，因此更有可能留在缓存中。这个重新排列的代码在使用 $x[i]$ 时显示了更好的时间局部性。

#### 1.6.0.7 空间局部性

空间局部性的概念要稍微复杂一些。如果一个程序引用的内存与它已经引用过的内存" 接近"，那么这个程序就被认为具有空间局部性。经典的冯 · 诺依曼架构中只有一个处理器和内存，此时空间局部性并不突出，因为内存中的一个地址可以像其他地址一样被快速检索。然而，在一个有缓存的现代 CPU 中，情况就不同了。上面我们已经看到了两个空间局部性的例子。

- 由于数据是以缓存线而不是单独的字或字节为单位移动的，因此以这样的方式进行编码，因此使缓存线所有的元素都得到应用是有所裨益的，在下列循环中

```
for (i=0; i<N*s; i+=s){
  ... x[i] ...
}
```

空间局部性体现为函数所进行的跨步递减 $s$。设 $S$ 为缓存线的大小，那么当 $s$ 的范围从 1....$S$，每个缓存线使用的元素数就会从 S 下降到 1。相对来说，这增加了循环中的内存流量花销：如果 $s = 1$，我们为每个元素加载 $1/S$ 的缓存线；如果 $s = S$，我们为每个元素加载一个缓存线。这个效果在 1.7.4 节中得到了证明。

- 第二个值得注意的空间局部性的例子是 TLB（1.3.8.2 节）。如果一个程序引用的元素距离很近，它们很可能在同一个内存页上，通过 TLB 的地址转换会很迅速。另一方面，如果一个程序引用了许多不同的元素，它也将引用许多不同的页。由此产生的 TLB 缺失是时间花销十分庞大；另见 1.7.5 节。

✍ **练习 1.16** 请考虑以下对 $n$ 数字 $x[i]$ 进行求和的算法的伪码，其中 $n$ 是 2 的倍数。

```
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]
```

分析该算法的空间和时间局部性，并将其与标准算法进行对比

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

✍ **练习 1.17** 考虑以下代码，并假设 nvectors 相比于缓存很小，而长度很大。

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

以下概念与该代码的性能有什么关系。

- 复用（Reuse）
- 缓存尺寸（Cache size）
- 关联性（Associativity）

下面这段交换了循环的代码的性能是更好还是更差，为什么?

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

#### 1.6.0.8 局部性示例

让我们看一个实际的例子。矩阵与矩阵的乘法 $C \leftarrow AB$ 可以用几种方法计算。我们比较两种实现方式，假设所有的矩阵都是按行存储的，且缓存大小不足以存储整个行或列。

```
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]
```

这些实现如图 1.17 所示。第一个实现构建了 $(i, j)$ 元素

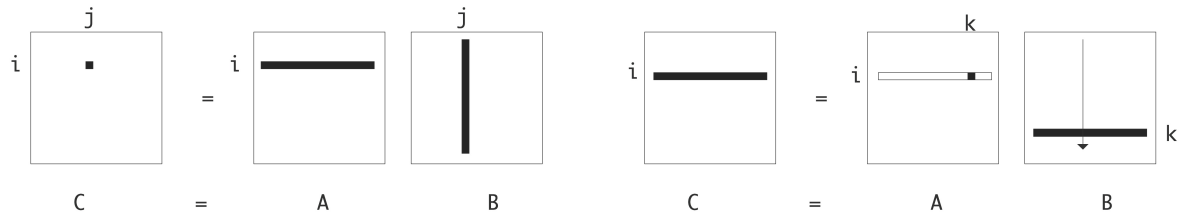



图 1.18: 矩阵-矩阵乘积的两个循环排序

$A$ 的一行与 $B$ 的一列的内积来更新 $C$，在第二行中，$B$ 的一行是通过对 $A$ 的元素进行缩放来更新。$A$ 的元素来更新 $B$ 的行数。

我们的第一个观察结果是，这两种实现都确实计算了 $C \leftarrow C + AB$，并且它们都花费了大约 $2n^3$ 的操作。然而，它们的内存行为，包括空间和时间的局部性是非常不同的。

- $c[i, j]$ ：在第一个实现中，$c[i, j]$ 在内部迭代中是不变的，这构成了时间局部性，所以它可以被保存在寄存器中。因此，$C$ 的每个元素将只被加载和存储一次。
  在第二个实现中，$c[i, j]$ 将在每个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，比第一次实现多了 $n$。
- $a[i, k]$：在这两种实现中，$a[i, k]$ 元素都是按行访问的，所以有很好的空间局部性，因为每个加载的缓存线都会被完全使用。在第二个实现中，$a[i, k]$ 在内循环中是不变的，这构成了时间局部性；它可以被保存在寄存器中。因此，在第二种情况下，$A$ 只被加载一次，而在第一种情况下则是 $n$ 次。
- $b[k, j]$：这两种实现方式在访问矩阵 $B$ 的方式上有很大不同。首先，$b[k, j]$ 从来都是不变的，所以它不会被保存在寄存器中，而且 $B$ 在两种情况下都会产生 $n^3$ 的内存负载。但是，访问模式不同。
  在第二种情况下，$b[k, j]$ 是按行访问的，所以有很好的空间局部性性：缓存线在被加载后将被完全利用。
  在第一种实现中，$b[k, j]$ 是通过列访问的。由于矩阵的行存储，一个缓存线包含了一个行的一部分，所以每加载一个缓存线，只有一个元素被用于列的遍历。这意味着第一个实现对 $B$ 的加载量要比缓存线长度的系数大。也有可能是 TLB 的影响。

请注意，我们并没有对这些实现的代码性能做任何绝对的预测，甚至也没有对它们的运行时间做相对比较。这种预测是很难做到的。然而，上面的讨论指出了与广泛的经典 CPU 相关的问题。

✍ **练习 1.18** 乘积 $C \leftarrow AB$ 的实现算法较多。请考虑以下情况。

```
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，并表明它比上面给出的两种算法更糟糕。

#### 1.6.0.9 核心局部性

上述空间和时间局部性的概念主要是程序的属性，尽管诸如高速缓存线长度和高速缓存大小这样的硬件属性在分析局部性的数量方面发挥了作用。还有第三种类型的局部性与硬件有更密切的联系：**核心局部性**（core locality）。

如果空间上或时间上接近的写访问是在同一个核心或处理单元上进行的，那么代码的执行就会表现出核心局部性。这里的问题是缓存一致性的问题，两个核心在他们的本地存储中都有某个缓存线的副本。如果它们都从该缓存中读取，那就没有问题了。但是，如果他们中的一个对它进行了写操作，一致性协议就会把这个缓存线复制到另一个核心的本地存储中。这需要占用宝贵的内存带宽，所以要避免这种情况。

核心局部性不仅仅是一个程序的属性，而且在很大程度上也是程序的并行执行方式。

# 第 2 章 并行计算

内容提要

- 并行计算引言
- 理论概念
- 并行计算机架构
- 不同类型的内存访问
- 并行计算中的粒度
- 线程并行与 OpenMP
- 处理器拓扑
- 多线程架构
- GPU 与协处理器
- 负载均衡
- 其他话题

规模最大且运算能力最强的计算机被称为" 超级计算机"。在过去的二十年里，超级计算机的概念无一例外地指向拥有多个 CPU 且可同时处理一个问题的机器——并行计算机。

我们很难精确定义并行的概念，因为它在不同的层面上有着不同的含义。在上一章中，同一个 CPU 内部可以有若干条指令同时“执行”，这是所谓的**指令级并行**（instruction-level parallelism，ILP）。指令级并行并不在用户的控制范围内，而是由编译器和 CPU 共同决定。另一种并行的概念为多个处理器同时处理一条以上的指令，每个处理器都在自己所在的电路板上，这种并行可以由用户显式地调度。

这一章中，我们将分析这种更明确的并行类型，支持它的硬件以及分析其概念。

## 2.1 引言

在科学计算中，我们常常需要处理大量且有规律的操作。有没有一种并行计算机可以加快这项工作？假设我们需要执行 $n$ 步操作，且每一步操作在单处理器上需要花费的时间为 $t$，那么使用 $p$ 个处理器，我们能否在 $t/p$ 的时间内完成这些工作？

让我们先从一个简单的例子开始。假设需要将两个长度为 $n$ 的向量相加：

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

最多可以用 $n$ 个处理器来完成。下图所示中，每个处理器都存储一个 a,b,c，且各自执行单一指令：a=b+c。

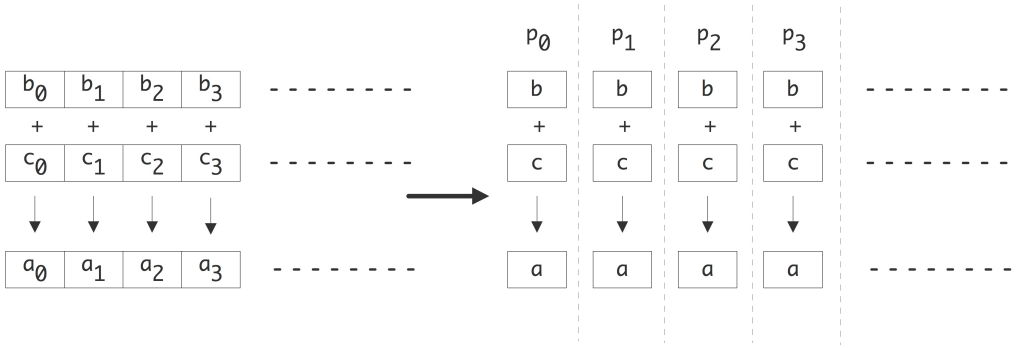



图 2.1: 向量加法的并行化

一般情况下，每个处理器执行的指令类似于

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

程序执行的时间随着处理器数量的增加而线性减少。若定义每步操作为单位时间，原始算法耗时 $n$，而在 $p$ 个处理器上的并行执行需要的时间为 $n/p$，并行后的速度是原来的 $p$ 倍。[1] 输入一个向量而得到一个标量的操作通常称为**规约**（reduction）

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

这段代码的并行情况并不明显，但如果我们将循环改写成：

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

则可以找到相应的方法将其并行化：外循环的每一次迭代现在都是一个可以由 $n/s$ 处理器并行完成的循环。由于外循环将经过 $\log_2 n$ 次迭代，我们可以看到新算法的运行时间缩短为 $n/p \log_2 n$。并行算法的速度比原来快了 $p/\log_2 n$ 倍。

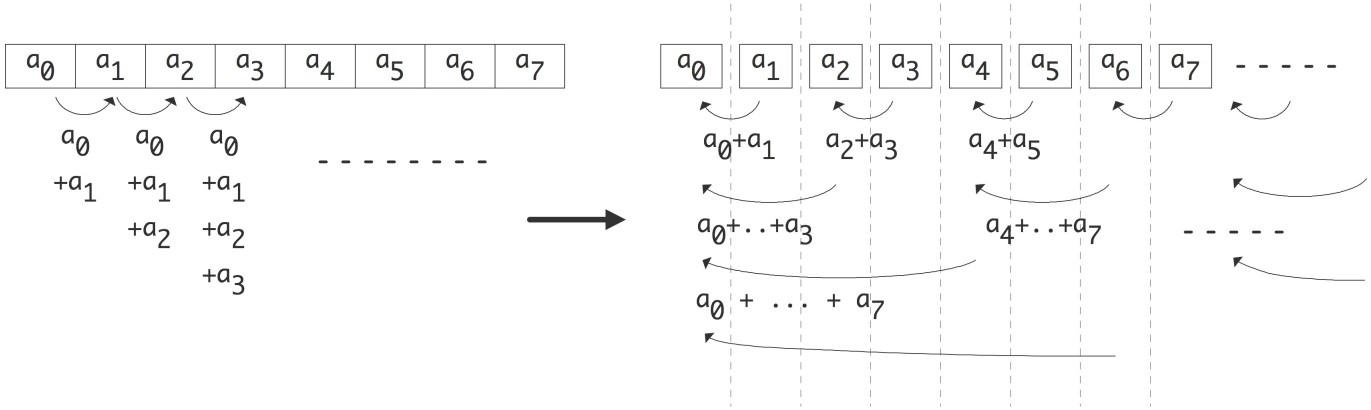



图 2.2: 向量规约的并行化

从这两个简单的例子中可以看到并行计算的一些特点：

- 算法被稍加改写后可以成为并行算法。
- 并行算法并不一定能达到理想加速效果。

此外，第一种情况下每个处理器的 $x_i, y_i$ 都在本地存储，这不会造成额外的时间开销；但在第二种情况下，处理器之间需要进行数据通信，这又会造成大量的开销。

下面我们讨论通信。我们可以把上图右半部分的并行算法变成一个树状图，把输入定义为树的节点，将所有的加和操作视为内部节点，总和作为根节点。如果一个节点是另一个节点的（部分）和的输入，则有一条从一个节点到另一个节点的边。在这个树状图中，可同时计算的元素放置在同一层级；每一级有时被称为**超步计算**（superstep）。垂直排列的节点意味着计算是在同一处理器上完成的，从一个处理器到另一个处理器的箭头对应着一次通信。树状图中的排列顺序并非唯一。如果节点在一个超步或水平层级上重新排列，就会出现不同的通信模式。

[1]输入一个向量而得到一个标量的操作通常称为规约（reduction）

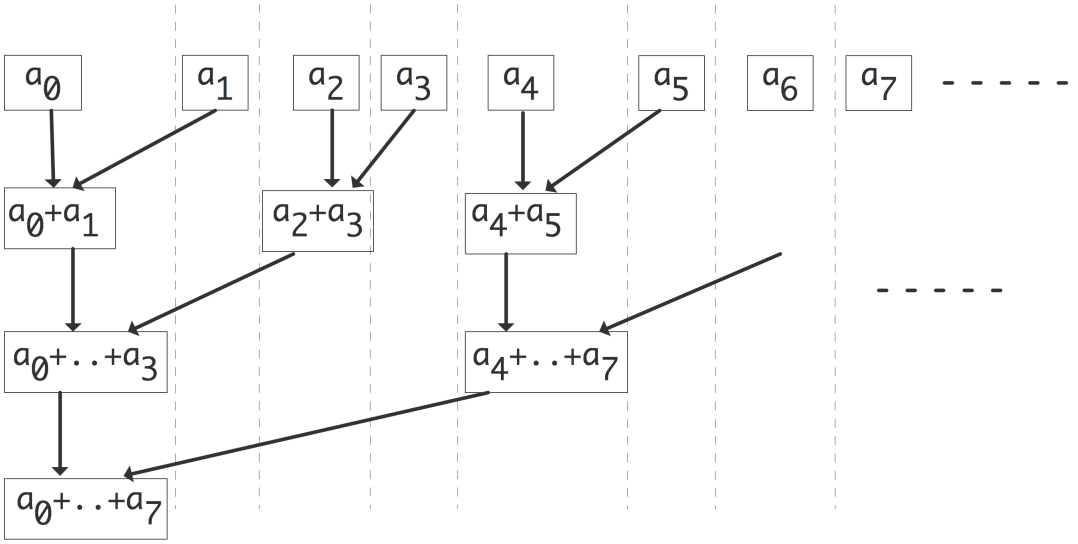



图 2.3: graphics/parallel-sum-graph.jpeg

✍ **练习 2.1** 考虑将超步内的节点放在随机处理器上。表明如果没有两个节点出现在同一个处理器上，那么最多只能进行两倍于图中的通信数量。

✍ **练习 2.2** 你能画出在每个处理器上留下总和结果的计算图吗？有一种解决方案需要两倍的超步，也有一种解决方案需要相同的数量。在这两种情况下，图不再是一棵树，而是一个更普遍的**有向无环图**（Directed Acyclic Graph，DAG）。

处理器之间通常通过网络连接，由于网络之间传输数据需要时间，我们引入处理器之间距离的概念。在上文树状图中，处理器的排列是线性的，这与它们在排序中的等级有关。如果网络只连接一个处理器和它的邻节点，外循环的每一次迭代都会增加通信的距离。

✍ **练习 2.3** 假设一个加法操作需要一个单位时间，而把一个数字从一个处理器移到另一个处理器也需要同样的单位时间。证明通信时间等于计算时间。

现在假设从处理器 $p$ 发送一个数字到 $p \pm k$ 需要时间 $k$。说明现在并行算法的执行时间与顺序时间是一样的。求和的例子做了一个不现实的假设，即每个处理器最初只存储一个向量元素：实际上我们会有 $p < n$，每个处理器都会存储一些向量元素。明显的策略是给每个处理器一个连续的元素序列，但有时明显的策略并不是最好的。

✍ **练习 2.4** 考虑用 4 个处理器对 8 个元素求和的情况。表明图 2.3 中的一些边不再对应于实际通信。现在考虑用 4 个处理器对 16 个元素进行求和。这次通信边的数量是多少？

这些关于算法适应性、效率和通信的问题，对所有的并行计算都是至关重要的。在本章中，我们将以各种形式回到这些问题上。

### 2.1.1 功能性并行与数据级并行

从上面的介绍中，我们可以将并行概念定义为：在程序的执行过程中寻找独立的操作。这些独立的操作往往其执行逻辑相同，只是用于不同的数据项。我们把这种情况称为**数据级并行**（data parallelism）：同一操作被并行地应用于许多数据元素，这在科学计算中是十分常见的。并行性往往源于这样一个事实：一个数据集（向量、

矩阵、图.....）被分散到许多处理器上，每个处理器都在处理其数据的一部分。

如果是单指令操作，传统上多采用数据级并行；如果是在子程序下处理，则通常称为**任务并行**（task parallelism）。

我们一定可以找到这样一种场景，使得指令之间相互独立且无依赖关系。一般情况下，编译器根据指令级并行来分析、运行代码：一条独立的指令可以被赋予给一个独立的浮点单元，也可以在优化寄存器时被重新排列。

指令级并行是功能性并行的一种情况；功能并行可以通过连接相互独立的子程序来获得。在更高层次上，功能并行可以通过包含独立的子程序来获得，通常称为任务并行。

功能性并行的一些例子是蒙特卡洛模拟，以及其他穿越参数化搜索空间的算法。一个参数化的搜索空间，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

有时程序可以直接并行化处理，例如上文讨论的向量加法；有时我们很难找到简易的并行策略，例如在后面将要讨论线性递归；而有些情况下，代码看起来可能没办法并行，但我们可以从理论上进行并行化处理。

✍ **练习 2.5** 回答下列有关双循环 $i, j$ 的问题。

```
for i in [1:N]:
          x[0,i] = some_function_of(i)
          x[i,0] = some_function_of(i)
```

```
for i in [1:N]:
          for j in [1:N]:
             x[i,j] = x[i-1,j]+x[i,j-1]
```

1. 内循环的迭代是否独立，也就是说，它们是否可以同时执行?
2. 外循环的迭代是否独立?
3. 如果 $x[1,1]$ 是已知的，说明 $x[2,1]$ 和 $x[1,2]$ 可以独立计算。
4. 这是否让你对并行化策略有了一个想法?

我们将在第 6.10.1 节讨论这个难题的解决方案。总的来说，第 6 章的全部内容都将是关于科学计算算法中内在的并行性数量。

## 2.2 理论概念

使用并行计算机有两个重要原因：获得更多的内存或者获得更高的性能。用更多的内存的原因很容易解释，因为总内存是各个内存的总和；而并行计算机的速度则较难描述。本节将对采用并行架构后的措施和理论速度进行拓展讨论。

### 2.2.1 定义

#### 2.2.1.1 加速比和效率

对比同一个程序在单处理器上运行的时间与 $p$ 个处理器上的运行时间可以得到加速比，设 $T_1$ 是在单个处理器上的执行时间，$T_p$ 是在 $p$ 个处理器上的运行时间，则加速比为 $S_p = T_1/T_p$(有时 $T_1$ 被定义为" 在单个处理器上解决问题的最短时间"，这允许在单个处理器上使用不同于并行的算法)。在理想情况下，$T_p = T_1/p$，但在实际中往往难以达到，因此 $S_p \leqslant p$。为了衡量我们离理想的加速有多远，我们引入了效率 $E_p = S_p/p$。显然，$0 < E_p \leqslant 1$。

上面的定义会产生一个问题：某个需要并行解决的问题由于规模太大，无法在任何一个单独的处理器上运

行；反之，将单处理器上的问题拆解在多处理器上，由于每个处理器上的数据非常少，因此可能会得到一个十分扭曲的结果。下面我们将讨论更现实的速度提升措施。

**定义 2.1 (负载不均衡与高度并行)**

有各种原因导致实际速度低于 p。首先，使用多个处理器意味着额外的通信开销；其次，如果处理器并未分配到完全相同的工作量，则会产生一部分的闲置，就会造成**负载不均衡**（load unbalance），再次降低实际速度；最后，代码运行可能依赖其原有顺序。

处理器之间的通信是效率损失的一个重要来源。显然，一个不用通信就能解决的问题是非常有效的。这类问题实际上由许多完全独立的计算组成被称为**高度并行**（embarrassingly parallel），它们拥有接近完美的加速比和效率。

♣

✍ **练习 2.6** 加速比大于处理器数量被称为**超线性加速**（superlinear speedup）。请给出一个理论上的论据，为什么这种情况不会发生。

在实践中，超线性加速可能发生。例如，假设一个问题太大，无法装入内存，一个处理器只能通过交换数据到磁盘来解决。如果同一个问题适合在两个处理器的内存中解决，那么速度的提升很可能大于 2，因为磁盘交换不再发生。拥有更少或更局部的数据也可以改善代码的缓存行为。

#### 2.2.1.2 代价最优

在达不到理想加速比的情况下，我们可以将理想加速比与实际加速之间的差异定义为**额外开销**（overhead）：

$$T_0 = pT_p - T1$$

我们也可以把它解释为在单个处理器上模拟并行算法，与实际的最佳串行算法之间的差异。

我们以后会看到两种不同类型的开销：

1. 并行算法可以与串行算法有本质上的不同。例如，排序算法的复杂度为 $O(n \log n)$，但并行双调排序（8.6 节）的复杂度为 $O(n \log^2 n)$。
2. 并行算法可以有来自于过程或并行化的开销，比如发送消息的成本。我们在 6.2.2 节中分析了矩阵与向量乘积中的通信开销。

如果一个并行算法与串行算法达到了数量级差距，那么该算法就被称为**代价最优**（cost-optimal）。

✍ **练习 2.7** 上面的额外开销定义隐含地假定开销是不可并行的。在上述两个例子的背景下讨论这个假设。

### 2.2.2 渐近论

如果我们忽略一些限制，比如处理器的数量，或者它们之间的互连的物理特性，我们可以就推导出关于并行计算效率极限的理论结果。本节将简要介绍这些结果，并讨论它们与现实中高性能计算的联系。

例如，考虑矩阵与矩阵乘法 $C = AB$，它需要 $2N$ 步操作，其中 $N$ 是矩阵规模的大小。由于对 $C$ 元素的操作之间没有依赖性，我们可以并行地执行。如果我们有 $N^2$ 处理器，我们可以将每个处理器分配给 中的 $(i, j)$ 坐标，并让它在 $2N$ 时间内计算 $c_{ij}$。因此，这个并行操作的效率为 1，是最优的。

✍ **练习 2.8** 证明这个算法忽略了关于内存的严重问题。

- 如果矩阵被保存在共享内存中，那么从每个内存位置同时读多少次？内存位置进行多少次读取？
- 如果处理器将输入和输出都保存在本地存储器中，那么有多少重复？

将 N 数字 $\{x\}_{i=1...N}$ 相加，可以在对数 N 时间内由 N/2 个处理器完成。作为一个简单的例子，考虑 $n$ 数之和：$s = \sum_{i=1}^{n} a_i$。如果我们有 $n/2$ 个处理器，我们可以计算：

1. 定义 $s_i^{(0)} = a_i$
2. 迭代 $j = 1, ..., log_2 n$：
3. 计算 $n/2^j$ 部分和 $s_i^{(j)} = s_{2i}^{(j-1)} + s_{2i+1}^{(j-1)}$

我们看到，$n/2$ 个处理器在 $\log_2 n$ 的时间内总共完成了 $n$ 的操作（应该如此）。这个并行方案的效率是 $O(1/\log_2 n)$，是一个缓慢下降的 $n$ 的函数。

✍ **练习 2.9** 请指出，使用刚才的并行加法方案，用 $N^3/2$ 个处理器在对数 $\log_2 N$ 时间内完成两个矩阵的相乘所得的效率是多少？

现在，我们可以提出一个合理的理论问题

- 如果我们有无限多的处理器，矩阵与矩阵乘法的最低时间复杂度是多少？
- 是否有更快的算法仍然具有 O(1) 的效率？

事实上，前人早已经探讨过这类问题（例如，见 [100]），但它们对高性能计算没有什么影响。对这些理论界线的第一个反对意见是，它们隐含地假定了某种形式的共享内存。事实上，这种算法模型被称为 **PRAM 模型**（Parallel Random Access Machine），是 **RAM 模型**（Random Access Machine）在共享内存系统上的扩展。该模型假设所有处理器共享一个连续的内存空间。此外，模型还允许同一位置上同时进行多个访问。这在实际应用中，特别是在扩大问题规模和处理器数量的情况下是不可能的。对 PRAM 模型的另一个反对意见是，即使在单个处理器上，它也忽略了内存的层次结构；1.3 节。

但是，即使我们把分布式内存考虑在内，这个结果仍然是不现实的。上述求和算法确实可以在分布式内存中不变地工作，只是我们必须考虑随着进一步迭代，活跃处理器之间的距离会增加。如果处理器是由一个线性数组连接起来的，那么活跃处理器之间的" 跳跃" 次数就会增加一倍，渐进地，迭代的计算时间也会随之增加。总的执行时间变为 $n/2$，考虑到我们在问题上投入了这么多处理器，这个结果显然令人感到失望。

如果处理器是以超立方体拓扑结构连接的呢（2.7.5 节）？不难看出，求和算法确实可以在 $\log_2 n$ 的时间内完成。然而，当 $n \to \infty$ 时，我们能否建立一个由 $n$ 节点组成的超立方体序列，并保持两个连接的通信时间不变？由于通信时间取决于延迟，而延迟部分取决于总线的长度，所以我们必须担心最近的邻居之间的物理距离。

这里的关键问题是，超立方体（$n$ 维的物体）是否可以被嵌入到三维空间中，同时保持连接的邻居之间的距离（以米计算）不变。很容易看出，3 维网格可以任意放大，同时保持总线的单位长度，但对于超立方体来说，这个问题并不明确。在这里，导线的长度可能会随着 $n$ 的增加而增加，这就与电子的有限速度相抵触。

我们草拟了一个证明（详见 [65]），即在我们的三维世界和有限的光速下，对于 $n$ 处理器上的问题，无论互连方式如何，速度都被限制在 $\sqrt[4]{n}$。该论点如下。考虑一个涉及在一个处理器上收集最终结果的操作。假设每个处理器占用一个单位体积的空间，在单位时间内产生一个结果，并且在单位时间内可以发送一个数据项。那么，在一定的时间内，最多只有半径为 $t$ 的球中的处理器，即 $O(t^3)$ 处理器可以对最终结果做出贡献；所有其他处理器都离得太远。那么，在时间 $T$ 内，能够对最终结果做出贡献的操作数 $\int_0^T t^3 dt = O\left(T^4\right)$. 在时间 $T$ 内，这意味着，最大的可实现的速度提升是串行时间的四次方根。

最后，" 如果我们有无限多的处理器怎么办" 这个问题本身并不现实，但请先不要急着将它抛弃，我们在后面提出弱可扩展性问题时还会讨论它（第 2.2.5 节）。" 如果我们让问题的大小和处理器的数量成比例增长怎么办"。这个问题是合理的，因为它与购买更多的处理器是否可以运行更大的问题，以及如果可以的话，有什么" 好处" 等非常实际的考虑相一致。

### 2.2.3 阿姆达尔定律

无法达到理想加速比的一个原因是，部分代码依赖固有顺序执行。假设有 5% 的代码必须串行执行，那么这部分的时间将不会随着处理器的数量增加而减少。因此，对该代码的提速被限制在 20 的系数。这种现象被称为

**阿姆达尔定律**（Amdahl's Law）[4]，下面我们将对其进行表述。

**定理 2.1 (阿姆达尔定律)**

令 $F_s$ 分别为代码的串行部分，$F_p$ 为代码的并行部分（更严格地说法应该为：" 可并行" 部分）。那么 $F_p + F_s = 1$。在 $p$ 个处理器上的并行执行时间 $T_p$ 是串行执行的部分 $T_1 F_s$ 和可并行化的部分 $T_1 F_p / P$ 之和。

$$T_P = T_1 \left(F_s + F_p / P\right)$$

随着处理器数量的增加，当 $P \to \infty$ 时，现有的并行执行时间已经接近代码的串行部分的时间。$T_P \downarrow T_1 F_s$。我们的结论是，加速受限于 $S_P \leqslant 1/F_s$，效率是一个递减函数 $E \sim 1/P$。

代码的串行部分可以由 I/O 操作等内容组成。然而，并行的代码中也有一些部分实际上是串行的。考虑一个执行单个循环的程序，其中的所有迭代都可以独立计算。显然，这段代码没有提供任何的并行化障碍。然而，通过将循环分割成许多部分（每个处理器一个），每个处理器现在必须处理循环开销：计算边界和完成测试。只要有处理器，这种开销就会被复制很多次。实际上，循环开销是代码的一个串行部分。

**练习 2.10** 我们来做一个具体的例子。假设一段代码的执行需要 1 秒，可并行部分在单个处理器上需要 1000 秒。如果该代码在 100 个处理器上执行，其速度和效率是多少？对于 500 个处理器来说，速度和效率又是多少？请保留最多两位有效数字。

**练习 2.11** 调查阿姆达尔定律的含义：如果处理器的数量 $P$ 增加，代码的并行部分必须如何增加才能保持固定的效率？

### 2.2.3.1 有通信开销的阿姆达尔定律

尽管阿姆达尔定律十分准确的指出了并行后速度的提升情况，但由于通信开销的存在，实际的性能相比于理论性能仍有所降低。让我们细化一下上述方程的模型：

$$T_p = T_1(F_s + F_p/P) + T_c \tag{2.1}$$

其中 $T_c$ 是一个固定的通信时间。为了评估这种通信开销的影响，我们假设代码是完全可并行的，即 $F_p = 1$。可以发现：

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

为了使之接近 $p$，我们需要 $T_c << T_1/p$ 或 $p << T_1/T_c$。换句话说，处理器的数量增长不应超过标量执行时间和通信开销的比例。

### 2.2.3.2 古斯塔法森定律

阿姆达尔定律认为只增加处理器数量并不会对并行加速结果有明显的提升。其隐含假设是：越来越多的处理器上执行同一个固定计算。然而在实际中情况并非如此：通常有一种方法可以扩大问题的规模（后面我们将学习" 离散化" 的概念），人们根据可用处理器的数量来调整问题规模的大小。

一个更现实的假设是，有一个独立于问题大小的串行部分，以及可以任意复制的并行部分。为了正式说明这一点，让我们从并行程序的执行时开始：

$$T_p = T(F_s + F_p), \quad \text{其中} F_s + F_p = 1 \tag{2.3}$$

现在我们有两种可能的 $T_1$ 的定义。首先是在 $T$ 中设置 $p = 1$ 得到的 $T_1$（说服自己这实际上与 $T_p$ 相同）。然而，我们需要的是 $T_1$，它描述的是完成并行程序所有操作的时间。(见图 2.4)。

$$T_1 = F_s T + p F_p T. \tag{2.4}$$

这给我们提供了一个加速比

$$S_p = \frac{T_1}{T_p} = \frac{F_s + pF_p}{F_s + F_p} = F_s + pF_p = p - (p-1)F_s \tag{2.5}$$

从这个公式我们可以看出

- 加速仍以 $p$ 为界。
- 它仍是一个正数。
- 对于一个给定的 p，它又是顺序分数的一个递减函数。

✍ **练习 2.12** 重写方程 (2.3)，用 $p$ 和 $F_p$ 表示速度的提高。效率 $E_p$ 的渐近行为是什么?

与阿姆达尔定律一样，如果我们把通信开销包括在内，我们可以研究古斯塔法森定律的行为。回到一个完全可并行问题的方程，并将其近似为

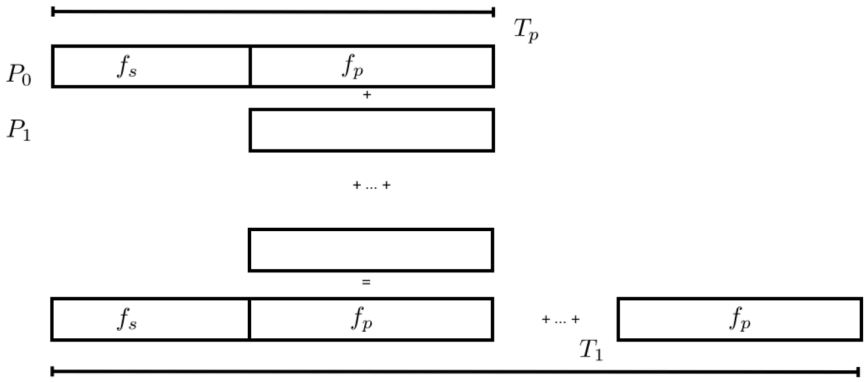



图 2.4: 古斯塔夫森分析中的串行和并行时间

现在，在问题逐渐放大的假设下，$T_c, T_1$ 成为 $p$ 的函数。我们看到，如果 $T_1(p) \sim pT_c(p)$，我们得到的线性加速是远离 1 的恒定分数。一般来说，我们不能进一步进行这种分析；在 6.2.2 节，你会看到一个例子的详细分析。

$$S_p = p(1 - \frac{T_c}{T_1}p) \tag{2.6}$$

现在，在问题逐渐放大的假设下，$T_c, T_1$ 成为 $p$ 的函数。我们看到，如果 $T_1(p) \sim pT_c(p)$，我们得到的线性加速是远离 1 的恒定分数。一般来说，我们不能进一步进行这种分析；在 6.2.2 节，你会看到一个例子的详细分析。

### 2.2.3.3 阿姆达尔定律和混合结构

上文我们已经认识了分布式和共享内存式的混合结构，这导致阿姆达尔定律的一种新型变式：

假设我们有 $p$ 节点，每个节点有 $c$ 核，$F_p$ 描述了使用 $c$ 路线程并行的代码的比例。我们假设整个代码在 $p$ 节点上是完全并行的。理想的速度是 $p_c$，理想的并行运行时间是 $T_1/(pc)$，但实际运行时间是

$$T_{p,c} = T_1(\frac{F_s}{p} + \frac{F_p}{pc})$$

✍ **练习 2.13** 证明加速 $T_1/T_p$，$c$ 可以用 $p/F_s$ 近似。在最初的阿姆达尔定律中，提速被顺序部分限制在一个固定的数字 $1/F_s$，在混合结构中，它被任务并行部分限制在 $p/F_s$。

## 2.2.4 关键路径和布伦特定理

上面关于加速和效率的定义，以及对阿姆达尔定律和古斯塔法森定律的讨论都隐含了一个假设，即并行工作可以被任意细分。然而事实情况并不总是这样：操作之间可能存在依赖关系，这限制了可以采用的并行量。

**定义 2.2 (关键路径)**

我们将**关键路径**（critical path）定义为最长度的依赖关系链（可能是非唯一的），这个长度有时被称为**跨度**（span)。由于关键路径上的任务需要一个接一个地执行，关键路径的长度是并行执行时间的一个下限。为了使这些概念准确，我们定义了以下概念。**定义**

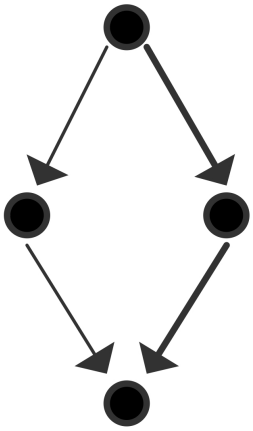
- $T_1$：计算在单个处理器上花费的时间
- $T_p$：计算在 $p$ 处理器上花费的时间
- $T_\infty$：如果有无限的处理器，计算所需的时间。
- $P_\infty$：$T_p = T_\infty$ 的 $p$ 值。

有了这些概念，我们可以将算法的**平均并行度**（average parallelism）定义为 $T_1/T_\infty$，而关键路径的长度为 $T_\infty$。

♣

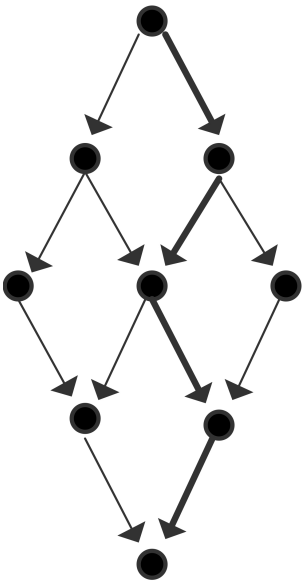
现在我们将通过展示一个任务及其依赖关系的图来进行一些说明。为了简单起见，我们假设每个节点是一个单位时间的任务。

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



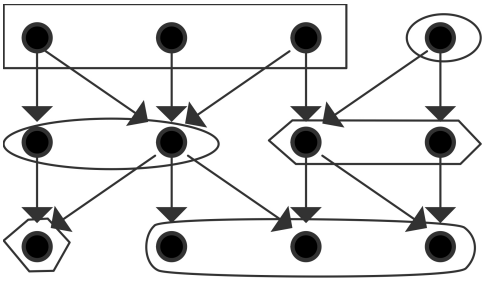
可以使用的最大处理器数量为 3，平均并行度为 9/5；效率最大的是 $p = 2$。

$$T_1 = 9, T_\infty = 5 \Rightarrow T_1/T_\infty = 9/5 T_2 = 6, S_2 = 3/2, E_2 = 3/4 T_3 = 5, S_3 = 9/5, E_3 = 3/5 P_\infty = 3$$



可以使用的最大处理器数量是 4，这也是平均并行度；图中说明了一个效率为 ≡ 1 的 $P = 3$ 的并行化。

$$T_1 = 12, T_\infty = 4 \Rightarrow T_1/T_\infty = 3T_2 = 6, S_2 = 2, E_2 = 1T_3 = 4, S_3 = 3, E_3 = 1T_4 = 3, S_4 = 4, E_4 = 1P_\infty = 4$$



从这些例子我们会发现两种极端情况：如果每个任务都恰好依赖于其他任务，则会得到一个依赖链。

- 如果每个任务都精确地依赖于其他任务，你会得到一个依赖链，并且对于任何 $T_p = T_1$。
- 另一方面，如果所有的任务都是独立的（并且 $p$ 除以它们的数量），你会得到 $T_p = T_1/p$，对于任何 $p$。
- 在一个比上一个稍微不那么琐碎的情况下，考虑关键路径的长度为 $m$，在这些 $m$ 的每一步中，有 $p-1$ 个独立的任务，或者至少：只依赖于前一步的任务。这样，每一个 $m$ 步骤中都会有完美的并行性，我们可以表达为 $T_p = T_1/T_p = m + T_1 - m/p$。

最后这句话实际上在一般情况下是成立的。这被称为" **布伦特定理**（Brent's Theorem）"。

**命题 2.1**

设 $m$ 为任务总数，$p$ 为处理器数量，$t$ 为关键路径的长度。那么计算可以在

$$T_p \leqslant t + \frac{m-t}{p}$$

内完成.

**证明** 将计算分成几步，使 $i+1$ 各步中的任务相互独立，而只依赖于步骤 $i$。设步骤中的任务数为 $s_i$，则该步骤的时间为 $\lceil \frac{s_i}{p} \rceil$。将其相加得出

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

**练习 2.14** 考虑一棵深度为 $d$ 的树，即有 $2^d - 1$ 的节点，以及一个搜索 $\max_{n \in \text{nodes}} f(n)$。假设所有的节点都需要被访问：我们对它们的值没有任何了解或排序。分析在 $p$ 处理器上的并行运行时间，你可以假设 $p = 2q$，其中 $q < d$。这与你从布伦特定理和阿姆达尔定律得到的数字有什么关系?

### 2.2.5 可扩展性

上文说过，使用越来越多数量的处理器处理一个给定的问题是没有意义的：每个节点上的处理器都没有足够有效地运行。在实践中，并行计算的用户要么选择与问题规模相匹配的处理器数量，要么在相应增加的处理器数量上解决一系列越来越大的问题。在这两种情况下，都很难谈及速度提升。因此我们使用了**可扩展性**（scalability）的概念。

我们区分了两种类型的可扩展性。所谓的**强可扩展性**（strong scalability）实际上与上面讨论的加速相同。我们说，如果一个程序在越来越多的处理器上进行分割，它显示出完美或接近完美的速度，也就是说，执行时间随着处理器数量的增加而线性下降，那么这个程序就显示出强大的可伸缩性。在效率方面，我们可以将其描述为

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

通常情况下，人们会遇到类似" 这个问题可以扩展到 500 个处理器" 的说法，这意味着在 500 个处理器以下，速度不会明显低于最佳状态。这个问题不一定要在一个处理器上解决：通常使用一个较小的数字，如 64 个处理

器，作为判断可扩展性的基线。

更有趣的是，**弱可扩展性**（weak scalability）是一个定义更模糊的术语。它描述了执行的行为，当问题的大小和处理器的数量都在增长时，但每个处理器的数据量却保持不变。由于操作数和数据量之间的关系可能很复杂，所以诸如加速等措施很难报告。如果这种关系是线性的，可以说每个处理器的数据量保持不变，并报告说随着处理器数量的增加，并行执行时间是不变的。(任务和数据之间何时是线性关系？何时不是？)

✍ **练习 2.15** 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。

$$t = c/p$$

证明在对数图上，也就是将运行时间的对数与处理器数量的对数相比较，我们会得到一条斜率为-1 的直线。你能提出一种处理不可并行部分的方法吗，也就是说，运行时间 $t = c_1 + c_2/p$？

✍ **练习 2.16** 假设我们正在研究一个代码的弱可扩展性。在运行了几种规模和相应数量的进程之后，发现在每一种情况下，翻转率都是大致相同的。论证该代码确实是弱可扩展的。

✍ **练习 2.17** 在上面的讨论中，我们总是隐含地比较一个串行算法和该算法的并行形式。然而，在第 2.2.1 节中，我们注意到，有时提速被定义为一个并行算法与同一问题的最佳顺序算法的比较。考虑到这一点，请将运行时间为 $(\log)^2$ 的并行排序算法（例如双调排序；第 8 节）与运行时间为 $n \log n$ 的最佳串行算法进行比较。证明在 $n = p$ 的弱可扩展情况下，速度提升为 $p/\log p$。证明在强可扩展情况下，加速是 $n$ 的一个递减函数。

注释 一则历史轶事.

Message: 1023110, 88 lines Posted: 5:34pm EST, Mon Nov 25/85, imported: .... 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.

这一点通过扩大问题的规模得到了满足。

#### 2.2.5.1 等效性

在上述弱可扩展性的定义中，我们指出在问题规模 $N$ 和处理器数量 $P$ 之间的某种关系下，效率将保持不变。我们可以使之精确化，并将等效率曲线定义为 $N$、$P$ 之间的关系，使效率恒定 [86]。

#### 2.2.5.2 可扩展性到底是什么意思?

在工业界的说法中，" 可扩展性"[2]一词有时被应用于架构或整个计算机系统。

在科学计算中，可扩展性是一个算法的属性，以及它在一个架构上的并行化方式，特别是注意到数据的分布方式。在第 6.2.2 节中，我们会发现对矩阵与向量乘积操作的分析：按块行分布的矩阵原来是不可扩展的，但按二维分布的子矩阵是可以的。

[2]在科学计算中，可扩展性是一个算法的属性，以及它在一个架构上的并行化方式，特别是注意到数据的分布方式。在第 6.2.2 节中，我们会发现对矩阵与向量乘积操作的分析：按块行分布的矩阵原来是不可扩展的，但按二维分布的子矩阵是可以的。

### 2.2.6 缩放模拟

在大多数关于弱扩展的讨论中，我们都假设工作量和存储量是线性关系，这并不总是如此。例如，对于 $N^2$ 的数据，矩阵-矩阵乘积的操作复杂度为 $N^3$。如果线性地增加处理器的数量，并保持每个进程的数据不变，工作可能会随着功率的增加而上升。

如果模拟随时间变化的偏微分方程，也会有类似的效果。(这里，总功是每个时间步骤的功和时间步骤数的乘积。这两个数字是相关的；在第 4.1.2 节中，你看到时间步长有一定的最小尺寸，是空间离散化的一个函数。因此，时间步数将随着每个时间步数的工作上升而上升。

在本节中，我们不是从算法运行的角度来研究可扩展性，而是研究模拟时间 $S$ 和运行时间 $T$ 是恒定的情况，我们看看这对我们需要的内存量有何影响。这相当于我们有一个模拟，在一定的运行时间内模拟了一定量的现实世界的时间；现在你买了一台更大的计算机，你想知道在相同的运行时间和保持相同的模拟时间内，你能解决多大的问题。换句话说，如果你能在一天内计算出两天的天气预报，你不希望当你买了更大的电脑后，它开始花费三天时间。

让 $m$ 为每个处理器的内存 $P$ 为处理器的数量，得出

$$M = Pm, \quad \text{总内存}$$

如果 d 是问题的空间维数，通常是 2 或 3，我们可以得到

$$\Delta x = 1/M^{1/d} \quad \text{网格间距}$$

为了稳定起见，这将时间步长 $\Delta t$ 限制为

$$\Delta t = \begin{cases} \Delta x = 1/M^{1/d} & \text{双曲情况} \\ \Delta x^2 = 1/M^{2/d} & \text{抛物线情况} \end{cases}$$

（注意到第四章没有讨论双曲的情况）在模拟时间 $S$ 的情况下，我们发现

$$k = S/\Delta t \quad \text{时间步数}$$

如果我们假设各个时间步骤是完全可并行的，也就是说，我们使用显式方法，或带有最优求解器的隐式方法，我们发现运行时间为

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

令 $T/S = C$，则

$$m = C\Delta t$$

也就是说，每个处理器的内存量随着处理器数量的增加而减少。(最后一句话中缺少的步骤是什么？)

进一步分析这个结果，我们发现

$$m = C\Delta t = c \begin{cases} 1/M^{1/d} & \text{双曲情况} \\ 1/M^{2/d} & \text{抛物线情况} \end{cases}$$

代入 $M = Pm$，我们最终发现

$$m = C \begin{cases} 1/P^{1/(d+1)} & \text{双曲情况} \\ 1/P^{2/(d+2)} & \text{抛物线情况} \end{cases}$$

也就是说，我们可以使用的每个处理器的内存会随着处理器数量的高次方而减少。

### 2.2.7 其他的缩放措施

上面的阿姆达尔定律是以在一个处理器上的执行时间来表述的。在许多实际情况下，这是不现实的，因为并行执行的问题对任何一个处理器来说都太大了。一些公式的处理给了我们在某种程度上等同的数量，但不依赖于这个单处理器的数量 [159]。

首先，将定义 $S_p(n) = \frac{T_1(n)}{T_p(n)}$ 应用于强可扩展，我们发现 $T_1(n)/n$ 是每次操作的串行时间。它的倒数 $n/T_1(n)$ 可以称为串行计算率，表示为 $R_1(n)$。同样可以定义" 并行计算率"

$$R_p(n) = n/T_p(n) \tag{2.7}$$

我们发现

$$S_p(n) = R_p(n)/R_1(n) \tag{2.8}$$

在强可扩展中，$R_1(n)$ 将是一个常数，所以我们做一个加速的对数图，纯粹是基于测量 $T_p(n)$。

### 2.2.8 并发；异步和分布式计算

即使在非并行的计算机上，也有一个同时执行多个进程的问题。操作系统通常有一个时间切片的概念，在这个概念中，所有活动的进程都被赋予 CPU 的指令，在一小段时间内轮流执行。通过这种方式，串行可以模拟一个并行机器；当然，这种做法效率较低。

然而，即使不运行并行程序，时间切片也是有用的。操作系统会有一些独立的进程（例如编辑器，收到的邮件，等等），它们都需要保持活跃，或多或少地运行。这种独立进程的困难在于，它们有时需要访问相同的资源。两个进程都需要相同的两个资源，而每个进程都得到一个，这被称为**死锁**（deadlock）。资源争夺的一个著名的形式化被称为**哲学家进餐**（dining philosophers）问题。

研究这种独立进程的领域有多种说法，如**并发性**（concurrency）、**异步计算**（asynchronous computing）或**分布式计算**（distributed computing）。并发这一术语描述了我们正在处理同时活动的任务，它们的行动之间没有时间串行。分布式计算这一术语来源于数据库系统等应用，在这些应用中，多个独立的客户需要访问一个共享数据库。

本书将不多讨论这个话题。第 2.6.1 节讨论了支持时间切片的线程机制；在现代多核处理器上，线程可以用来实现共享内存并行计算。《Communicating Sequential Processes》一书对并发进程之间的交互进行了分析 [109]。其他作者使用拓扑结构来分析异步计算 [103]。

## 2.3 并行计算机架构

相当一段时间以来，超级计算机都是某种并行计算机，即允许同时执行多个指令或指令序列的架构。福林（Flynn）[66] 提出了一种描述这种架构的各种形式的方法。Flynn 的分类法通过数据流和控制流是共享的还是独立的来描述结构。结果有以下四种类型（也见图 2.5）。

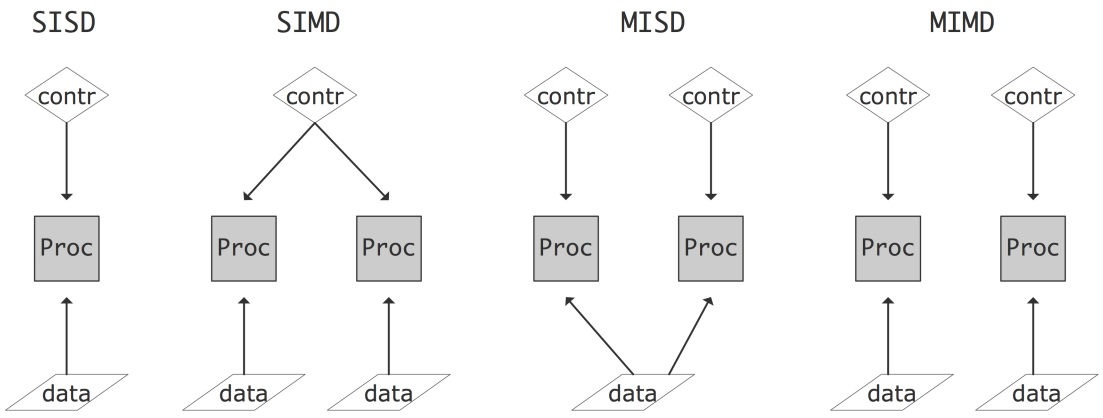



图 2.5: 福林分类法的四个等级

**定义 2.3 (福林分类)**

- **单指令单数据**（Single Instruction Single Data，SISD）：这是传统的 CPU 结构：在任何时候只有一条指令被执行，对一个数据项进行操作。
- **单指令多数据**（Single Instruction Multiple Data，SIMD）：在这种计算机类型中，可以有多个处理器，每个处理器对自己的数据项进行操作，但它们都在对该数据项执行相同的指令。向量计算机（2.3.1.1 节）通常也被定性为 SIMD。
- **多指令单数据**（Multiple Instruction Single Data，MISD）：目前还没有符合这种描述的架构；人们可以说，安全关键应用的冗余计算就是 MISD 的一个例子。
- **多指令多数据**（Multiple Instruction Multiple Data，MIMD）：这里有多个 CPU 对多个数据项进行操作，每个都执行独立的指令。目前大多数并行计算机都属于这种类型。

♣

现在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一些数据项进行相同的操作。这种计算机的 CPU 的设计可以相当简单，因为算术单元不需要单独的逻辑和指令解码单元：所有的 CPU 都是锁步执行相同的操作。这使得 SIMD 计算机在对数组的操作上表现出色，如

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

而且，由于这个原因，它们也经常被称为**阵列处理器**（array processors）。科学代码通常可以写得很好，使很大一部分时间花在阵列操作上。

另一方面，有些操作不能在阵列处理器上有效执行。例如，评估递归的若干项 $x_{i+1} = ax_i + b_i$ 涉及许多加法和乘法，但它们是交替进行的，因此每次只能处理一种类型的操作。这里没有同时作为加法或乘法输入的数字阵列。为了允许对数据的不同部分进行不同的指令流，处理器会有一个" 屏蔽位"，可以被设置来阻止指令的执行。在代码中，这通常看起来像

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

将相同的操作同时应用于一些数据项的编程模型，被称为**数据并行**（data parallelism）。

这种数组操作可以在物理模拟中出现，但另一个重要来源是图形应用。对于这种应用，阵列处理器中的处理器可能比 PC 中的处理器弱得多：通常它们实际上是位处理器，一次只能对一个位进行操作。按照这种思路，ICL 在 20 世纪 80 年代有 4096 个处理器的 DAP[115]，固特异在 20 世纪 70 年代制造了一个 16K 处理器的 MPP[10]。

后来，连接机（CM-1、CM-2、CM-5）相当流行。虽然第一台连接机有位处理器（16 个到一个芯片），但后来的型号有能够进行浮点运算的传统处理器，并不是真正的 SIMD 架构。所有这些都是基于超立方体互连网络。另一家拥有商业上成功的阵列处理器的制造商是 MasPar；图 2.6 说明了该架构。你可以清楚地看到一个方形阵列处理器的单一控制单元，加上一个做全局操作的网络。

基于阵列处理的超级计算机已经不存在了，但是 SIMD 的概念以各种形式存在着。例如，GPU 是基于 SIMD 的，通过其 CUDA 编程语言强制执行。另外，英特尔 Xeon Phi 有一个强大的 SIMD 组件。早期的设计 SIMD 架构的初衷是尽量减少必要的晶体管数量，而这些现代协处理器则是考虑到电源功率。与浮点运算相比，处理指令（称为指令问题）在时间、能源和所需的芯片地产方面实际上是昂贵的。因此，使用 SIMD 是在后两项措施上节约成本的一种方式。

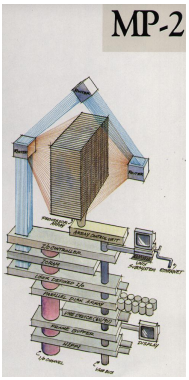



图 2.6: MasPar 2 阵列处理器的结构

#### 2.3.1.1 流水线

许多计算机都是基于向量处理器或流水线处理器的设计。第一批商业上成功的超级计算机，Cray-1 和 Cyber205 都属于这种类型。近来，Cray-X1 和 NEC SX 系列都采用了向量流水线。在 TOP 500 中领先 3 年的" 地球模拟器" 计算机 [178]，就是基于 NEC SX 处理器的。

虽然基于流水线处理器的超级计算机明显是少数，但流水线现在在作为集群基础的超标量 CPU 中是主流。一个典型的 CPU 有流水线的浮点单元，通常有独立的加法和乘法单元。

然而，现代超标量 CPU 的流水线与更老式的向量单元的流水线有一些重要区别。这些向量计算机中的流水线单元并不是 CPU 中的集成浮点单元，而是可以更好地看作是附属于本身具有浮点单元的 CPU 的向量单元。向量单元有向量寄存器，其长度一般为 64 个浮点数；通常没有" 向量缓存"。向量单元的逻辑也比较简单，通常可以通过明确的向量指令来寻址。另一方面，超标量 CPU 完全集成在 CPU 中，面向利用非结构化代码中的数据流。

### 2.3.2 CPU 和 GPU 中的真 SIMD

真正的 SIMD 阵列处理可以在现代 CPU 和 GPU 中找到，在这两种情况下，都是受到图形应用中需要的并行性的启发。

英特尔和 AMD 的现代 CPU，以及 PowerPC 芯片，都有向量指令，可以同时执行一个操作的多个实例。在英特尔处理器上，这被称为 **SIMD 流扩展**（Streaming Extensions，SSE）或**高级矢量扩展**（Advanced Vector Extensions，AVX）。这些扩展最初是用于图形处理的，在这种情况下，往往需要对大量的像素进行相同的操作。通常情况下，数据必须是总共 128 位，这可以分为两个 64 位实数，四个 32 位实数，或更多更小的块，如 4 位。

AVX 指令是基于高达 512 位宽的 SIMD，也就是说，可以同时处理 8 个浮点数。就像单次浮点运算对寄存器中的数据进行操作一样，向量运算使用**向量寄存器**（vector registers）。向量寄存器中的位置有时被称为 **SIMD 流水线**（SIMD lanes）。

SIMD 的使用主要是出于功耗的考虑。解码指令实际上比执行指令更耗电，所以 SIMD 并行是一种节省功耗的方法。

目前的编译器可以自动生成 SSE 或 AVX 指令；有时用户也可以插入 pragmas，例如英特尔的编译器。

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

这些扩展的使用通常要求数据与缓存行边界对齐，所以有一些特殊的 allocate 和 free 调用可以返回对齐的内存。

OpenMP 的第 4 版还有指示 SIMD 并行性的指令。

更大规模的阵列处理可以在 GPU 中找到。一个 GPU 包含大量的简单处理器，通常以 32 个一组的形式排列。每个处理器组只限于执行相同的指令。因此，这是 SIMD 处理的真正例子。

### 2.3.3 MIMD/SPMD 计算机

到目前为止，现在最常见的并行计算机结构被称为多指令多数据（MIMD）：处理器执行多条可能不同的指令，每条指令都在自己的数据上。说指令不同并不意味着处理器实际上运行不同的程序：这些机器大多以**单程序多数据**（Single Program Multiple Data，SPMD）模式运行，即程序员在并行处理器上启动同一个可执行文件。由于可执行程序的不同实例可以通过条件语句采取不同的路径，或执行不同数量的循环迭代，它们一般不会像 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，那就叫做**负载不均衡**（load unbalance,），它是导致速度不完美的一个主要原因。

MIMD 计算机有很大的多样性。其中一些方面涉及到内存的组织方式，以及连接处理器的网络。除了这些硬件方面，这些机器还有不同的编程方式。我们将在下面看到所有这些方面。现在的许多机器被称为**集群**（clusters）。它们可以由定制的或商品的处理器组成（如果它们由 PC 组成，运行 Linux，并通过以太网连接，它们被称为 Beowulf 集群 [93]）；由于处理器是独立的，它们是 MIMD 或 SPMD 模型的例子。

### 2.3.4 超级计算机的商品化

在 20 世纪 80 年代和 90 年代，超级计算机与个人计算机和迷你或超迷你计算机（如 DEC PDP 和 VAX 系列）有着本质的区别。SIMD 向量计算机有一个（CDC Cyber205 或 Cray-1），或最多几个（ETA-10、Cray-2、Cray X/MP、Cray Y/MP）极其强大的处理器，通常是一个向量处理器。大约在 20 世纪 90 年代中期，拥有数千个较简单（微型）处理器的集群开始取代拥有相对较少数量向量流水线的机器（见 http://www.top500.org/lists/1994/11）。起初这些微处理器（IBM Power 系列、Intel i860、MIPS、DEC Alpha）仍然比" 家用电脑" 处理器强大得多，但后来这种区别也在一定程度上消失了。目前，许多最强大的集群是由消费市场上的英特尔至强和 AMD Opteron 芯片提供动力的。其他人则使用 IBM Power 系列或其他" 服务器" 芯片。关于 1993 年以来的这段历史，见 2.11.4 节的说明。

## 2.4 不同类型的内存访问

在介绍中，我们将并行计算机定义为多个处理器共同处理同一问题的设置。除了最简单的情况，这意味着这些处理器需要访问一个联合的

数据池。在上一章中，你看到了即使是在单个处理器上，内存也很难跟上处理器的需求。对于并行机器来说，可能有几个处理器想要访问同一个内存位置，这个问题变得更加糟糕。我们可以通过它们在协调多个进程对联合数据池的多次访问问题上所采取的方法来描述并行机器。

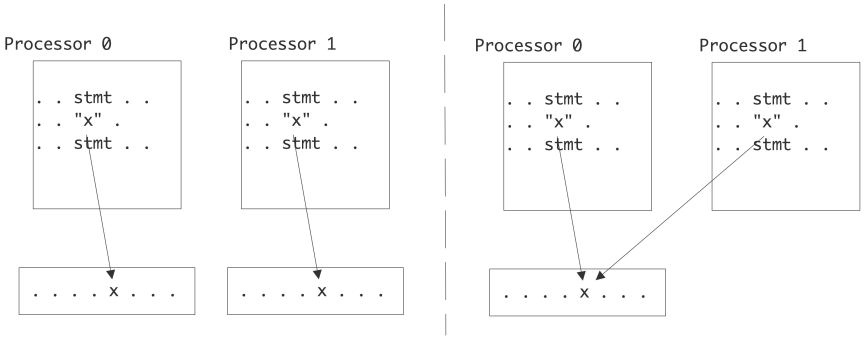



图 2.7: 在分布式和共享内存情况下对相同命名的变量的引用

这里的主要区别在于**分布式内存**（distributed memory）和**共享内存**（shared memory）之间的区别。在分布式内存中，每个处理器有自己的物理内存，更重要的是有自己的地址空间。因此，如果两个处理器引用一个变量 $x$，他们会访问自己本地内存中的一个变量。另一方面，在共享内存中，所有处理器都访问相同的内存；我们也说它们有一个**共享地址空间**（shared address space）。因此，如果两个处理器都引用一个变量 $x$，它们就会访问同一个内存位置。

### 2.4.1 对称多核处理器：统一内存访问

如果任何处理器都可以访问任何内存位置，并行编程就相当简单。由于这个原因，制造商有很大的动力来制造架构，使处理器看不到一个内存位置和另一个内存位置之间的区别：每个处理器都可以访问任何内存位置，而且访问时间没有区别。这被称为**统一内存访问**（Uniform Memory Access，UMA），基于这一原则的架构的编程模型通常被称为**对称多处理**（Symmetric Multi Processing，SMP）。

有几种方法可以实现 SMP 架构。目前的台式电脑可以有几个处理器通过一条内存总线访问一个共享的内存；例如，苹果公司在市场上销售一种带有 2 个六核处理器的机型。处理器之间共享的内存总线只适用于少量的处理器；对于更多的处理器，可以使用连接多个处理器和多个内存 bank 的横梁。

在多核处理器上，有一种不同类型的统一内存访问：各核通常有一个共享的高速缓存，通常是 L3 或 L2 高速缓存。

### 2.4.2 非统一内存访问

基于共享内存的 UMA 方法显然只限于少量的处理器。十字架网络是可以扩展的，所以它们似乎是最好的选择。然而，在实践中，人们将具有本地内存的处理器放在一个具有交换网络的配置中。这导致了一种情况，即一个处理器可以快速访问自己的内存，而其他处理器的内存则较慢。这就是所谓的 NUMA 的一种情况：一种使用物理分布式内存的策略，放弃统一的访问时间，但保持逻辑上的共享地址空间：每个处理器仍然可以访问任何内存位置。

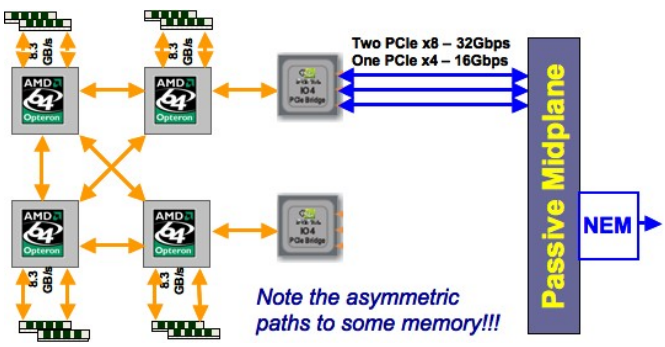



图 2.8: 四插槽主板中的非均匀内存访问

该图说明了 TACC Ranger 集群的四插槽主板的 NUMA 情况。每个芯片都有自己的内存（8Gb），但是主板的行为就像处理器可以访问一个 32Gb 的共享池。很明显，访问另一个处理器的内存比访问本地内存要慢。此外，请注意，每个处理器有三个连接，可以用来访问其他内存，但最右边的两个芯片使用一个连接来连接网络。这意味着访问对方的内存只能通过一个中间处理器进行，减缓了传输速度，并占用了该处理器的连接。

虽然 NUMA 方法对程序员来说很方便，但它为系统提供了一些挑战。想象一下，两个不同的处理器在其本地（缓存）内存中都有一个内存位置的副本。如果一个处理器改变了这个位置的内容，这个变化必须被传播到其他处理器上。如果两个处理器都试图改变一个内存位置的内容，程序的行为会变得不确定。

**定义 2.4 (缓存一致性)**

保持一个内存位置的副本同步被称为**缓存一致性**（cache coherence）；使用这种方法的多处理器系统有时被称为" 缓存一致性的 NUMA " 或 ccNUMA 架构。

将 NUMA 发挥到极致，有可能有一个软件层，使网络连接的处理器看起来在共享内存上运行。这被称为**分布式共享内存**（distributed shared memory）或**虚拟共享内存**（virtual shared memory）。在这种方法中，管理程序提供了一个共享内存 API，通过翻译系统调用到分布式内存管理。这种共享内存 API 可以被 Linux 内核所利用，它可以支持 4096 个线程。

♣

在目前的供应商中，只有 SGI（UV 系列）和 Cray（XE6）的市场产品具有大规模的 NUMA。两者都对分区全局地址空间（PGAS）语言提供了强有力的支持。有一些厂商，如 ScaleMP，为普通集群上的分布式共享内存提供了软件解决方案。

### 2.4.3 逻辑上和物理上的分布式内存

对内存访问问题最极端的解决方案是提供不仅在物理上，而且在逻辑上也是分布式的内存：处理器有自己的地址空间，不能直接看到其他处理器的内存。这种方法通常被称为" 分布式内存"，但这个术语是不明显的，因为我们必须分别考虑内存是否是分布式的和是否是分布式的问题。请注意，NUMA 也有物理上的分布式内存；它的分布式性质对于程序员来说并不明显。

在逻辑和物理的分布式内存中，一个处理器与另一个处理器交换信息的唯一方式是通过网络明确传递信息。

这种类型的架构有一个显著的优势，即它可以扩展到大量的处理器：IBM 蓝色基因已经建立了超过 20 万个处理器。另一方面，这也是最难编程的一种并行系统。

存在上述类型之间的各种混合体。事实上，大多数现代集群会有 NUMA 节点，但节点之间是分布式内存网络。

## 2.5 并行计算中的粒度

一个程序有多少并行度？事实上，大部分指令都可以并行执行，但我们要考虑并行后的代价：并行后程序是否变得简单？以及并行后加速效率是否明显等问题。

本节的讨论主要是在概念层面上进行的；后面将详细介绍如何对并行进行实际编程。

### 2.5.1 数据并行化

对于有简单主体循环的程序来说，遍历大数据集的操作相当常见。

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

这样的代码被认为是**数据并行**（data parallelism）或**细粒度并行**（fine-grained parallelism）的一个实例。如果我们有和数组元素一样多的处理器，那么并行后的代码将非常简单：每个处理器将在其本地数据上执行

```
a = 2*b
```

如果代码主要由数组的循环组成，它可以在所有处理器锁步的情况下有效执行。基于这种思想设计的并行架构早已存在，事实上处理器只能以锁步方式工作。这种数组上的完全并行操作出现在计算机图形学中，图像的每个像素都被独立处理。因此，GPU 的并行就是基于数据并行的。

继续上面的例子，考虑以下操作

$$\textbf{for} \quad 0 \leqslant i \leqslant max \quad \textbf{do} i_{left} = mod(i-1, max) i_{right} = mod(i+1, max) a_i = (b_{i_{left}} + b_{i_{right}})/2$$

在数据并行机器上，可以实现为

$$bleft \leftarrow \textbf{shiftright(b)} bright \leftarrow \textbf{shiftleft(b)} a \leftarrow (bleft + bright)/2$$

其中 shiftleft/right 指令导致一个数据项被发送到数字较低或较高为 1 的处理器。为了使第二个例子有效，有必要使每个处理器能够与其近邻快速通信，并使第一个和最后一个处理器彼此通信。

在各种情况下，如图形中的" 模糊" 操作，对二维数据的操作是有意义的。

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

因此，处理器必须能够将数据移动到二维网格中的相邻处。

### 2.5.2 指令级并行

在 ILP 中，并行性仍然是在单个指令的层面上，但这些指令不一定是相似的。例如，在

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

这两个赋值是独立的，因此可以同时执行。编译器可以帮助我们处理这种并行。事实上，识别 ILP 对于从现代超标量 CPU 中获得良好的性能至关重要。

### 2.5.3 任务并行

数据和指令级并行的另一种应用为**任务并行**（task parallelism），是指可以并行执行的整个子程序。例如，在树形数据结构中的搜索可以按以下方式实现。

```
if optimal (root) then
    exit
else
    parallel: SearchInTree (leftchild),SearchInTree (rightchild)
         Procedure SearchInTree(root)
```

这个例子中的搜索任务是不同步的，而且任务数量也不固定。在实际应用中，任务过多并不是一个很好的策略，因为处理器只在一个任务上工作时其效率才最高。上面的例子可以略加改写为：

```
while there are tasks left do
    wait until a processor becomes inactive;
    spawn a new task on it
```

（之前的两个伪代码之间有一个微妙的区别。在第一个代码中，任务是自我调度的：每个任务都会衍生出两个新的任务。第二个代码是一个 **Manager-Worker Paradigm** 的例子：一个贯穿整个程序执行过程的中心任务负

责派生和分配节点任务。）

与数据并行不同，该方案中数据对处理器的分配不是事先确定的。因此，这种并行模式最适合于线程编程，例如通过 OpenMP 库的并行。下面考虑另一个高度任务并行的例子：

在最简单的情况下，一个有限元网格是覆盖二维物体的三角形的集合。由于应该避免过于尖锐的角度，**Delauney 网格细化**（Delauney mesh refinement）过程可以选择某些三角形，用形状更好的三角形取代它们。图 2.9 说明了这一点：黑色的三角形违反了一些角度条件，所以要么它们自己被细分，要么它们与一些相邻的三角形（呈现为灰色）连接，然后共同被重新细分。

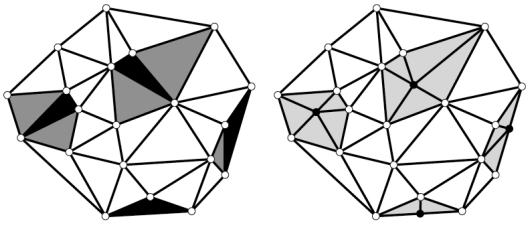



图 2.9: 细化前和细化后的网格

这可以在伪代码中实现：

```
Mesh m = /* read in initial mesh */
WorkList wl;
wl.add(mesh.badTriangles());
while (wl.size() != 0) do
    Element e = wl.get(); //get bad triangle
    if (e no longer in mesh) continue;
    Cavity c = new Cavity(e);
    c.expand();
    c.retriangulate();
    mesh.update(c);
    wl.add(c.badTriangles());
```

很明显，该算法是由一个必须在所有进程之间共享的工作列表（或任务队列）数据结构驱动的。再加上动态分配数据给进程，这意味着这种不规则的并行性适合于共享内存编程，而在分布式内存中则较难做到。

### 2.5.4 高度并行

单处理器的计算通常需要在众多不同的输入上进行。如果计算的数据不存在相关依赖，且不需要任何特定情况，则被称为**高度并行**（embarrassingly parallel）或**便捷并行**（conveniently parallel）计算。这种并行可以发生在几个层面。在诸如计算 Mandelbrot set 或评估国际象棋游戏中的棋子的例子中，一个子程序级别的计算被调用了许多参数值。在一个更粗略的层面上，可能是一个简单的程序需要对许多输入进行运行。在这种情况下，整体计算被称为**参数扫描**（parameter sweep）。

### 2.5.5 中粒度的数据并行化

上述数据并行假定了有与数据元素同样多的处理器。在实际中，处理器内存通常会很大，且处理的数据数量要远远大于处理器数量。因此，数组被分组到子数组的处理器上。伪代码如下

```
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
```

这种模式有数据并行的特点，因为在大量的数据项上执行的操作是相同的。它也可以被看作是任务并行，因为每个处理器执行的代码部分较大，而且不一定对同等大小的数据块进行操作。

### 2.5.6 任务粒度

在前面的小节中，我们考虑了寻找并行工作的不同层次，或者说划分工作的不同方式，以便找到并行性。还有另一种方法：我们将并行方案的**粒度**（granularity）定义为一个处理元素在不得不与其他处理元素进行通信或同步之前可以执行的工作量（或任务大小）。

在 ILP 中，我们处理的是非常细粒度的并行，就像一条指令或几条指令一样。在真正的任务并行中，颗粒度要粗得多。

有趣的是，我们可以自行选择数据并行中的任务大小。SIMD 机器上，我们选择的是单指令粒度，但操作可以被分为中等大小的任务。因此，在处理器数量和总问题规模之间的适当平衡下，数据并行的操作可以在分布式内存集群上执行。

✍ **练习 2.18** 讨论为一个数据并行操作选择合适的粒度，如在二维网格上进行平均化。表明存在一个表面到体积的效应：通信量比计算量低一阶。这意味着，即使通信比计算慢得多，增加任务量仍然会得到一个平衡的执行。

如果试图加大任务规模以减小通信开销，则会导致另一个问题：集合操作时可能会有不同运行时间的任务，导致负载不均衡。一种解决办法是使用过度分解：创建比处理元素更多的任务，并将多个任务分配给一个处理器（或动态分配任务）以平衡不规则的运行时间。这就是所谓的**动态调度**（dynamic scheduling）。

## 2.6 线程并行与 OpenMP

并行编程远比串行编程更为复杂。尽管大部分编程语言的原理是相似的，但其并行方式却多种多样。

现阶段我们很难做出自动并行的机器，其中的难点在于：编译器不仅要弄清哪些操作之间是相互独立的，还要解决如何在并行环境中定位数据的棘手问题。

编译器需要考虑的是整个代码，而非一个又一个的子程序。

共享内存中一种并行方式是：用户编写普通程序，并借助 OpenMP 向编译器发布何时并行或如何并行的指令。这种指明数据分布，并将并行问题留给编译器的方法是后面要介绍的 PGAS 语言的基础。而在分布式内存中，我们则通常采取最难同时也是最好的办法：让程序员管理一切。

### 2.6.1 线程并行

为了解释什么是**线程**（thread），我们首先需要从技术上了解什么是**进程**（process）。在 Unix 中，一个进程就是单个程序的执行过程。它在内存中拥有：

- 程序代码，以机器语言指令的形式存在。
- **堆**（heap），包含 malloc 创建的数组。
- **栈**（stack），包含快速变化的信息，如**程序计数器**（program counter，PC），它显示了当前正在执行的结构。

一个进程执行过程中可以有多个线程，这些线程的共同之处在于：它们共享程序代码和堆，但各自都有独立的栈。因此，线程是程序执行中的独立分支。

进程可以属于不同的用户，或一个用户并发运行的不同程序，因此它们有自己的数据空间。另一方面，线程是一个进程的一部分，因此它们共享进程堆。线程可以有一些私有数据，例如通过拥有自己的数据栈，但它们的主要特征是它们可以在相同的数据上进行协作。

### 2.6.1.1 叉形连接机制

线程是动态的，它们可以在程序执行过程中被创建。(这与 MPI 模型不同，在 MPI 模型中，每个处理器运行一个进程，它们都是在同一时间创建和销毁的)。当程序启动后，处于活跃状态的线程称为**主线程**（main thread），其他线程通过主线程**生成**（thread spawning）创建，主线程需等待其完成，称为**生成-汇合模型**（fork-join）。从同一个线程生成出来并同时活动的一组线程被称为**线程组**（thread team）。

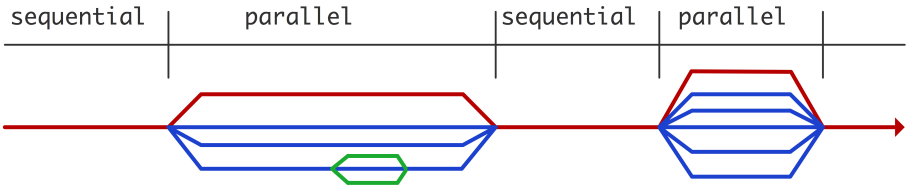



图 2.10: 并行执行期间线程的创建和删除

### 2.6.1.2 线程的硬件支持

上面所描述的线程是一种软件结构。在并行计算机出现之前，线程是可能的；例如，它们被用来处理操作系统中的独立活动。在没有并行硬件的情况下，操作系统将通过多任务或时间切片来处理线程：每个线程将定期使用 CPU 的一小部分时间。(从技术上讲，Linux 内核通过任务的概念来处理进程和线程；任务被保存在一个列表中，并定期被激活或取消)

这可以提高处理器的利用率，因为一个线程可以在另一个线程等待数据时被处理。(在传统的 CPU 上，线程之间的切换是有时间开销的（超线程机制是个例外），但在 GPU 上则不然，事实上，它们需要许多线程才能达到高性能)

在现代多核处理器上，有一种明显的支持线程的方法：每个核有一个线程，可以有效地使用硬件的并行执行。共享内存允许线程看到相同的数据，但这也会导致问题。

### 2.6.1.3 线程实例

下面基于 Unix 的例子清楚地说明了 fork-join 的模型。它使用 pthreads 库来生成一些任务，这些任务都会更新一个全局计数器。由于线程共享相同的内存空间，它们确实看到并更新相同的内存位置。

```
#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;}
```

事实上，这段代码给出了正确的结果，但这是一个巧合：它之所以发生，只是因为更新变量比创建线程要快得多。(在多核处理器上，出错的机会将大大增加)。如果人为地增加更新的时间，我们将不再得到正确的结果。

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

现在所有的线程都读出了 sum 的值，等待一段时间（估计是在计算什么），然后再更新。这可以通过在应该是" 互斥" 的代码区域上设置一个锁来解决。

```
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);
```

锁定和解锁命令保证了没有两个线程可以干扰对方的更新。关于 pthreads 的更多信息，请参见例如 https://computing.llnl.gov

### 2.6.1.4 上下文

在上面的例子和它的 sleep 命令版本中，我们忽略了一个事实，即有两种类型的数据参与其中。首先，变量 *s* 是在线程生成部分之外创建的。因此，这个变量是**共享**（shared）的。另一方面，变量 *t* 是在每个生成的线程中创建一次的。我们称其为**私有**（private）数据。

**定义 2.5 (上下文与上下文切换)**

一个线程可以访问的所有数据的总和被称为其**上下文**（context）。它包含了私有和共享数据，以及线程正在进行的计算的临时结果。(还包含程序计数器和堆栈指针。)

创建的线程比处理器的内核多是很有可能的，所以处理器可能需要在不同线程的执行之间进行切换。这就是所谓的**上下文切换**（context switch）。

普通的 CPU 进行上下文切换会造成时间开销，所以只有在线程工作的粒度足够高时，我们才会这样执行。下面几种情况则较为常见

- 有硬件支持多线程的 CPU，通过**超线程**（hyperthreading）或 Intel Xeon Phi 来实现
- GPU，它实际上依赖于快速上下文切换。
- 某些其他" 奇特" 的架构，如 Cray XMT。

### 2.6.1.5 竞争条件、线程安全和原子操作

共享内存使程序员的工作变得简单，因为每个处理器都可以访问所有的数据：处理器之间不需要明确的数据通信。另一方面，多个进程/处理器也可以写到同一个变量，这是潜在问题的来源。

假设两个进程都试图递增一个整数变量 $I$。

进程 1：$I = I + 2$

进程 2：$I = I + 3$

如果该变量是一个由独立进程计算的累加，这是一个合法的活动。这两个更新的结果取决于处理器读取和写入变量的顺序。

| scenario 1. | | scenario 2. | | scenario 3. |
|---|---|---|---|---|
| I = 0 | | | | |
| read I = 0<br>compute I = 2<br>write I = 2 | read I = 0<br>compute I = 3<br>write I = 3 | read I = 0<br>compute I = 2<br>write I = 2 | read I = 0<br>compute I = 3<br>write I = 3 | read I = 0<br>compute I = 2<br>write I = 2<br>read I = 2<br>compute I = 5<br>write I = 5 |
| I = 3 | | I = 2 | | I = 5 |

图 2.11: 数据竞争的三种执行方式

图 2.11 说明了三种情况。这种情况下，最终结果取决于哪个线程先执行，被称为**竞争条件**（race condition）或**数据竞争**（data race）。一个正式的定义是：如果有两个语句 $S_1$，$S_2$，数据竞争为

- 两个语句之间不存在因果关系
- 都是访问一个位置 $L$；并且
- 至少有一个访问是写操作。

这种冲突性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];
```

这里的乘积是真正独立的，所以我们可以选择让循环迭代并行进行，例如由它们自己的线程进行。然而，所有的线程都需要更新同一个变量的总和。

无论是串行执行还是线程执行，代码的行为都是一样的，这叫做**线程安全**（thread safe）。从上面的例子可以看出，缺乏线程安全通常是由于对共享数据的处理。这意味着程序越是使用本地数据，它是线程安全的机会就越大。不幸的是，有时线程需要写到共享/全局数据，例如，当程序进行**规约**（reduction）时。

解决这个问题的方法基本上有两种。一种是，我们将共享变量的这种更新宣布为代码的**临界区**（critical section）。这意味着临界区的指令（在内积的例子中，" 从内存中读取和，更新，写回内存"）一次只能由一个线程来执行。特别是，它们需要完全由一个线程执行，然后其他线程才能启动它们，所以上面的模糊问题不会出现。当然，上述代码片段非常常见，以至于像 OpenMP 这样的系统有专门的机制来处理它，把它声明为一个减少操作。

例如，临界区可以通过信号机制 [47] 来实现。在每个临界区的周围，会有两个原子操作控制着一个信号灯，即一个信号柱。第一个遇到信号灯的进程将降低信号灯，并开始执行临界区。其他进程看到已经降低的信号灯，并等待。当第一个进程完成临界区时，它执行第二条指令，提高信号灯，允许其中一个等待的进程进入临界区。

解决共享数据的共同访问的另一种方法是在某些内存区域设置一个临时**锁**（lock）。如果对临界区的共同执行是可能的，例如，如果它实现了对数据库或哈希表的写入，那么这种解决方案可能是比较好的。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，即使他们可能是写入不同的位置；那么锁定被访问的特定数据项是一个更好的解决方案。

锁的问题是，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。由于我们希望上述内积循环的迭代能以浮点单元的速度执行，或者至少以内存总线的速度执行，所以这是不可接受的。

这方面的一个实现是**事务内存**（transactional memory），硬件本身支持原子操作；这个术语来自于数据库事务，它有一个类似的完整性问题。在交易型内存中，一个进程将执行正常的内存更新，除非处理器检测到与另一个进程的更新有冲突。在这种情况下，更新（" 事务"）被取消并重新尝试，一个处理器锁定内存，另一个处理器等待锁定。这是一个优雅的解决方案；然而，取消事务可能会带来一定的**流水线冲洗**（pipeline flushing）和缓存线失效的代价。

### 2.6.1.6 内存模型和串行一致性

上面提到的竞争条件现象意味着一些程序的结果可能是非确定性的，这取决于指令的执行顺序。还有一个因素在起作用，它被称为处理器和/或语言使用的**内存模型**（memory model）[2]。内存模型控制一个线程或内核的活动如何被其他线程或内核看到。

例如，考虑

初始：$A = B = 0$，然后

进程 1：$A = 1$；$x = B$。

进程 2：$B = 1$；$y = A$。

如上所述，我们有三种情况，我们通过给出一个全局性的语句序列来描述这些情况。

(在第二种情况下，语句 1,2 和 3,4 都可以颠倒过来，但结果不会改变。)

这三种不同的结果可以被描述为是由尊重局部排序的状态要素的全局排序来计算的。这被称为**串行一致性**（sequential consistency）：并行的结果与顺序执行是一致的，该顺序执行将并行计算交错进行，尊重它们的本地语句排序。

保持串行一致性的代价是很昂贵的：它意味着对一个变量的任何改变都需要立即在所有其他线程上可见，或者对一个线程上的变量的任何访问都需要咨询所有其他线程。

在一个**松弛内存模型**（relaxed memory model）中，有可能会得到一个不符合顺序的结果。假设在上面的例子中，编译器决定对两个进程的语句重新排序，因为读写是独立的。实际上，我们得到了第四种情况。

导致结果 $x = 0, y = 0$，这在上面的串行一致模型下是不可能的。(有寻找这种依赖关系的算法 [127])。串行一致意味着

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

效果应该与下述相同

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

有了串行一致性，就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强烈的要求，所以可能导致代码的低效。

### 2.6.1.7 亲和性

线程编程非常灵活，可以根据需要有效地创建并行性。然而，本书的很大一部分内容是关于科学计算中数据移动的重要性，在线程编程中不能忽视这一方面。

在多核处理器的背景下，任何线程都可以被安排到任何核上，这没有什么直接的问题。然而，如果你关心的是高性能，这种灵活性会带来意想不到的代价。你想让某些线程只在某些核心上运行，有各种原因。由于操作系统允许迁移线程，可能你只是想让线程留在原地。

**定义 2.6 (亲和性)**

- 如果一个线程迁移到不同的核心，而该核心有自己的缓存，就会失去原来的缓存内容，不必要的内存转移就会发生。
- 如果一个线程迁移了，没有什么可以阻止操作系统把两个线程放在一个核心上，而让另一个核心完全不使用。这显然导致了不太完美的速度提升，即使线程的数量等于核心的数量。

我们称亲和性为**线程亲和性**（thread affinity）或**进程亲和性**（process affinity）与核心之间的映射。亲和性通常表示为一个掩码：对允许一个线程运行的位置的描述。例如，考虑一个双插槽的节点，每个插槽有四个核心。有了两个线程和插槽的亲和力，我们就有了以下的**关联掩码**（affinity mask）。

对于核心亲和性，面具取决于亲和力类型。典型的策略是" 接近" 和" 扩散"。在亲和关系密切的情况下，掩码可以是在同一个插槽上有两个线程意味着它们可能共享一个二级缓存，所以如果它们共享数据，这种策略是合适的。

另一方面，随着**亲和性扩散**（spread affinity），线程被进一步分开。这种策略对于带宽受限的应用来说更好，因为现在每个线程都拥有一个插槽的带宽，而不是在" 关闭" 的情况下不得不分享它。

如果分配了所有的内核，关闭和分散策略会导致不同的安排。相对于亲和性也可以被认为是一种将执行与数据绑定的策略。

考虑下段代码：

```
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，将内存带入高速缓存或页表。第二个循环以同样的顺序访问元素，所以为了性能，固定的亲和性是正确的决定。

在其他情况下，固定的映射不是正确的解决方案。

```
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] ...
```

在这第二个例子中，要么程序必须被改造，要么程序员必须实际维护一个任务队列。

- 第一次接触：从" 把执行放在数据所在的地方" 的角度来考虑亲和性是很自然的。然而，在实践中，相反的观点有时是有意义的。例如，图 2.8 显示了一个集群节点的共享内存实际上是如何分布的。因此，一个线程可以连接到一个插槽，但数据可以由操作系统分配到任何一个插槽上。操作系统经常使用的机制被称为 **first-touch 策略**。
- 当程序分配数据时，操作系统实际上并不创建数据。
- 相反，数据的内存区域是在线程第一次访问它时创建的。
- 因此，第一个接触该区域的线程实际上导致数据被分配到其插槽的内存中。

✍ **练习 2.19** 用下面的代码解释一下这个问题。

```
// serial 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，见 [134]。

#### 2.6.1.8 Cilk Plus

还有其他基于线程的编程模型存在。例如，英特尔 Cilk Plus(http://www.cilkplus.org/) 是一套 C/C++ 的扩展，程序员可以用它创建线程。

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

在这个例子中，变量 rst 被两个可能独立的线程更新。这种更新的语义，也就是如何解决同时写入等冲突的精确定义，是由串行一致性定义的。

#### 2.6.1.9 超线程与多线程的比较

在上面的例子中，你看到在一个程序运行过程中产生的线程基本上都是执行相同的代码，并且可以访问相同的数据。因此，在硬件层面上，一个线程是由少量的局部变量唯一决定的，比如它在代码中的位置（程序计数器）和它所参与的当前计算的中间结果。

超线程是英特尔的一项技术，让多个线程真正同时使用处理器，这样处理器的一部分将得到最佳利用。

如果一个处理器在执行一个线程和另一个线程之间切换，它将保存一个线程的本地信息，并加载另一个线程的信息。与运行整个程序相比这样做的成本并不高，但与单条指令的成本相比可能很昂贵。因此，超线程不一定能带来性能的提高。

某些架构有对多线程的支持。这意味着硬件实际上对多个线程的本地信息有明确的存储，而且线程之间的切换可以非常快。GPU 和英特尔 Xeon Phi 架构就是这种情况，每个内核可以支持多达四个线程。

### 2.6.2 OpenMP

OpenMP 是对编程语言 C 和 Fortran 的一个扩展。它的主要并行方法是循环的并行执行：基于**编译器指令**（compiler directives），预处理器可以安排循环迭代的并行执行。

由于 OpenMP 是基于线程的，它的特点是**动态并行**（dynamic parallelism）：在代码的一个部分和另一个部分之间，并行运行的执行流的数量可以变化。并行性是通过创建并行区域来声明的，例如表明一个循环嵌套的所有迭代都是独立的，然后运行时系统将使用任何可用的资源。

OpenMP 不是一种语言，而是对现有的 C 和 Fortran 语言的一种扩展。它主要通过在源代码中插入指令来操作，由编译器进行解释。与 MPI 不同，它也有少量的库调用，但这些不是重点。最后，还有一个运行时系统来管理并行的执行。

与 MPI 相比，OpenMP 的一个重要优势在于它的可编程性：可以从一个串行代码开始，通过**增量并行化**（incremental parallelization）来改造它。相比之下，将串行代码转化为分布式内存 MPI 程序是一个全有或全无的事情。

许多编译器，如 gcc 或 Intel 编译器，支持 OpenMP 扩展。在 Fortran 中，OpenMP 指令被放在注释语句中；在 C 中，它们被放在 #pragma CPP 指令中，用来表示编译器特定的扩展。因此，对于不支持 OpenMP 的编译器来说，OpenMP 代码看起来仍然像合法的 C 或 Fortran 语句。程序需要链接到 OpenMP 运行库，其行为可以通过环境变量来控制。关于 OpenMP 的更多信息，见 [31] 和 http://openmp.org/wp/。

#### 2.6.2.1 OpenMP 示例

OpenMP 使用的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];
}
```

很明显，所有的迭代都可以独立执行，并且以任何顺序执行。然后，pragma CPP 指令将这个事实传达给编译器。

有些循环在概念上是完全并行的，但在实现上不是。

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

这里看起来好像每个迭代都在向一个共享变量 t 写和读。然而，t 实际上是一个临时变量，是每个迭代的局部。应该是可并行的代码，但由于这样的结构而不能并行，这被称为非线程安全。

OpenMP 指出，临时变量对每个迭代都是私有的，如下所示。

```
 #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);
}
```

如果一个标量确实是共享的，OpenMP 有各种机制来处理这个问题。例如，共享变量通常出现在规约操作中。

```
s = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<ProblemSize; i++) {
```

```
    s = s + a[i]*b[i];
}
```

正如我们所看到的，串行代码可以通过最小的努力被并行化。

迭代到线程的分配是由运行时系统完成的，但用户可以指导这种分配。我们主要关注迭代次数多于线程的情况：如果有 $P$ 个线程和 $N$ 个迭代，并且 $N > P$，如何将迭代 $i$ 分配给线程？最简单的分配是使用 **Round-robin 任务调度**（round-robin task scheduling, a static scheduling），这是一种静态的调度策略，线程 $p$ 获得迭代 $p \times (N/P), ..., (p+1) \times (N/P) - 1$。这样做的好处是，如果一些数据在迭代之间被重复使用，它将留在执行该线程的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，进程可能会遭受静态调度的负载不均衡。在这种情况下，动态调度策略的效果会更好，每个线程在完成当前迭代后就开始对下一个未处理的迭代进行工作。参见 2.10.2 节中的例子。

我们可以用 schedule 关键字来控制 OpenMP 对循环迭代的调度，它的值包括静态和动态。也可以指出一个 chunksize，它可以控制一起分配给线程的迭代块的大小。如果你省略了 chunksize，OpenMP 将把迭代分成和线程数量一样多的块。

✍ **练习 2.20** 假设有 $t$ 个线程，代码为

```
for (i=0; i<N; i++) {
    a[i] = // 执行部分计算
}
```

如果你指定 chunksize 为 1，那么迭代 0、t、2t..... 进入第一个线程，1、1+t、1+2t..... 进入第二个线程，依此类推。讨论一下为什么从性能的角度看这是一个糟糕的策略。提示：查一下**伪共享**（false sharing）的定义。什么是一个好的 chunksize?

### 2.6.2.2 通过消息传递的分布式内存编程

虽然 OpenMP 程序和使用其他共享内存范式编写的程序看起来仍然非常像串行程序，但对于消息传递代码来说，情况并非如此。在我们详细讨论消息传递接口（MPI）库之前，我们先来看看并行代码编写方式的这种转变。

### 2.6.2.3 分布式编程中的全局视野与局部视野

在观察者看来，一个并行算法与它的实际编程方式之间可能存在明显的差异。考虑这样的情况：我们有一个处理器 $\{P_i\}_{i=0...p-1}$ 的数组，每个处理器包含数组 x 和 y 中的一个元素，并且 $P_i$ 计算

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

这方面的全局描述可以是

- 每个处理器 $P_i$（最后一个除外）都将其 Pi 元素发送给 $P_{i+1}$。
- 除了第一个之外，每个 $P_i$ 处理器都从他们的邻居 $P_{i-1}$ 那里收到一个 $x$ 元素，并且
- 他们将其添加到自己的 $y$ 元素中。

然而，在一般情况下，我们不能用这些全局术语来编码。在 SPMD 模型中，每个处理器执行相同的代码，而整体算法是这些单独行为的结果。本地程序只能访问本地数据--其他一切都需要用发送和接收操作来沟通--而且处理器知道自己的编号。

一种可能的写法是

- 如果是第 0 个处理器，什么都不做；否则从左边接收一个元素，增加一个 $x$ 元素。
- 如果是最后一个处理器，什么都不做。否则，将我的 $y$ 元素发送到右边。

首先，我们看一下发送和接收是所谓的**阻塞通信**（blocking communication）的情况：发送指令在实际收到发送的项目之前不会结束，而接收指令则等待相应的发送。这意味着处理器之间的发送和接收必须被仔细配对。现在我们将看到，这可能导致在通往高效代码的路上出现各种问题。

图 2.12 展示了上述解决方案，我们展示了描述本地处理器代码的局部时间线，以及由此产生的全局行为。你可以看到，处理器不是在同一时间工作的：我们得到的是序列化的执行。

如果我们把发送和接收操作倒过来呢?

- 如果不是最后一个处理器，就把我的 $x$ 元素发送到右边。
- 如果不是第一个处理器，从左边接收一个 $x$ 元素，并将其添加到 $y$ 元素中。

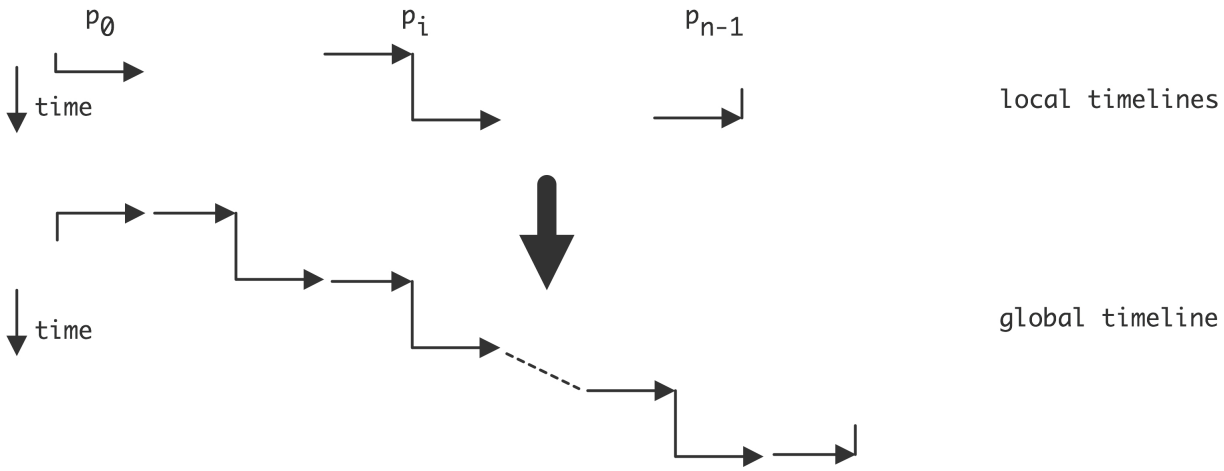



**图 2.12:** 向右侧发送数据的算法的局部和结果的全局视图

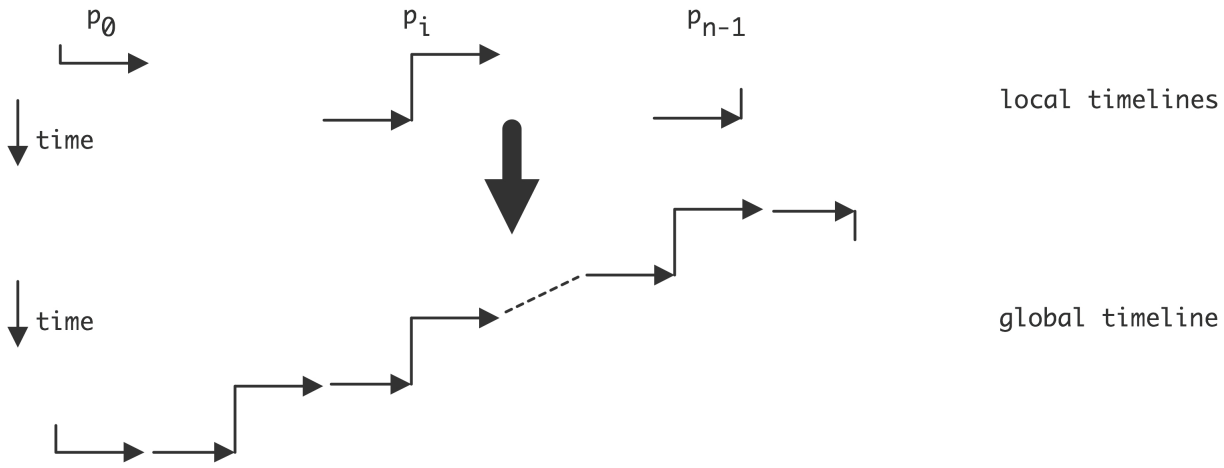



**图 2.13:** 向右侧发送数据的算法的局部和结果的全局视图

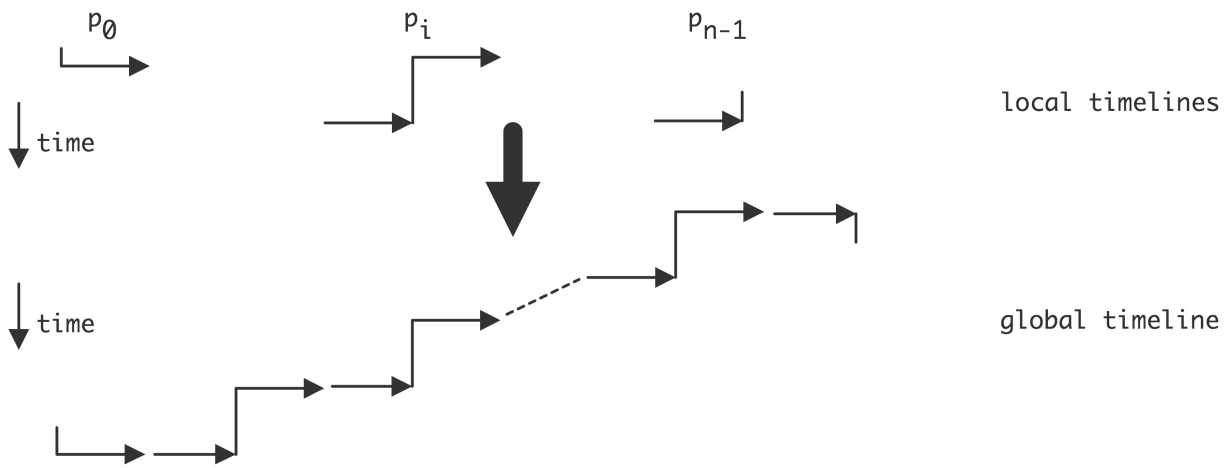



图 2.14: 向右侧发送数据的算法的局部和结果的全局视图

图 2.13 说明了这一点，你可以看到我们再次得到一个序列化的执行，只不过现在处理器是从右到左激活的。如果方程中的算法是循环的：

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

问题会更加严重。现在，最后一个处理器无法开始接收，因为它被阻止向 0 号处理器发送 $x_{n-1}$。这种情况下，程序无法进展，因为每个处理器都在等待另一个处理器，这被称为**死锁**（deadlock）。获得高效代码的解决方案是使尽可能多的通信同时发生。毕竟，在算法中没有串行的依赖性。因此，我们对算法的编程如下

- 奇数处理器，先发后收。
- 偶数处理器，先收后发。

图 2.14 说明了这一点，我们看到现在的执行是并行的。

练习 2.21 再看一下图 2.3 中的并行规约。其基本动作是 - 接收来自邻居的数据

- 将其添加到自己的数据中
- 将结果发送出去。

正如在图中看到的，至少有一个处理器不发送数据，其他的处理器在发送结果之前可能会做不同次数的接收。编写节点代码，使 SPMD 程序实现分布式规约。提示：用二进制写每个处理器的编号。该算法使用的步骤数等于该位串的长度。

- 假设一个处理器收到一条消息，用步数表示到该消息的原点的距离。
- 每个处理器最多发送一条消息。用二进制处理器编号来表示发生这种情况的步骤。

### 2.6.2.4 阻塞和非阻塞通信

阻断指令的原因是为了防止网络中的数据积累。如果一条发送指令在相应的接收指令开始之前完成，网络将不得不在这段时间内将数据储存在某个地方。考虑一个简单的例子：

```
buffer = ... ; // 生成一些数据
send(buffer,0); // 发送给 0 处理器
buffer = ... ; // 生成更多数据
send(buffer,1); // 发送给 1 处理器
```

在第一次发送后，我们开始覆盖缓冲区。如果其中的数据还没有被收到，那么第一组数值就必须在网络的某个地方被缓冲，这是不现实的。通过发送操作的阻断，数据会一直留在发送方的缓冲区中，直到它被保证复制到接收方的缓冲区。

解决由阻塞指令引起的顺序化或死锁问题的一个方法是使用**非阻塞通信**（non-blocking communication）指令，其中包括明确的数据缓冲区。使用非阻塞式发送指令，用户需要为每次发送分配一个缓冲区，并检查何时可以安全地覆盖缓冲区。

```
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.
```

### 2.6.2.5 MPI 库

如果说 OpenMP 是对共享内存进行编程的方式，那么消息传递接口（MPI）[184] 则是对分布式内存进行编程的标准解决方案。MPI（'Message Passing Interface'）是一个库接口的规范，用于在不共享数据的进程之间移动数据。MPI 例程可以大致分为以下几类。

- 进程管理。这包括查询并行环境和构建处理器的子集。
- 点对点通信。这是一组调用，其中两个进程进行交互。这些大多是发送和接收调用的变种。
- 集体调用。在这些程序中，所有的处理器（或整个指定的子集）都参与其中。例如，**广播**（broadcast）调用，一个处理器与其他所有处理器分享它的数据，或者收集调用，一个处理器从所有参与的处理器收集数据。

让我们考虑如何在 MPI4 中对 OpenMP 的例子进行编码。首先，我们不再分配

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

而是分配

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

其中，局部尺寸大约是全局尺寸的 1/P 部分。(实际的考虑决定了是让这个分布尽可能的均匀，还是在某种程度上有偏向)

并行循环是琐碎的并行，唯一的区别是它现在只对一部分数组进行操作。

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

然而，如果循环涉及基于迭代数的计算，我们需要将其映射到全局值。

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

(我们将假设每个进程都以某种方式计算了 LocalProblemSize 和 MyFirstVariable 的值)。本地变量现在自动成为本地变量，因为每个进程都有自己的实例。

```
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，因此必须明确地组装本地计算。

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

“规约”操作将所有的本地值 s 汇总到一个变量 globals 中，该变量在每个处理器上都收到一个相同的值。这就是所谓的**集合操作**（collective operation）。

让我们把这个例子变得稍微复杂些

```
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
}
```

如果有共享内存，我们可以写出以下的并行代码。

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

为了将其转化为有效的分布式内存代码，首先我们要说明，对于 $i == 0$ (bleft) 和 $i == LocalProblemSize - 1$ (bright)，bleft 和 bright 需要从不同的处理器获得。我们通过与我们的左邻右舍处理器进行交换操作来做到这一点。

```
// get bfromleft and bfromright from neighbor processors, then
    for (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
}
```

获得邻居值的方法如下。首先，我们需要询问我们的处理器编号，这样我们就可以与编号高一低一的处理器开始通信。

```
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, /* ... */ );
```

这段代码仍有两个问题。首先，sendrecv 操作需要对第一个和最后一个处理器进行异常处理。这可以通过以下方式优雅地完成。

```
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);
```

✍ **练习 2.22** 这段代码还存在一个问题：没有考虑到原版、全局的边界条件。请给出解决这个问题的代码。

如果不同的进程需要采取不同的行动，例如，如果一个进程需要向另一个进程发送数据，MPI 就会变得复杂。这里的问题是每个进程执行的是同一个可执行文件，所以它需要包含发送和接收指令，根据进程的等级来执行。

```
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,
       /* note explained here: */&status,MPI_COMM_WORLD);
}
```

#### 2.6.2.6 阻塞

尽管 MPI 有时被称为" 并行编程的汇编语言"，因为它被认为是困难的和明确的，但它并不是那么难学，大量使用它的科学代码就证明了这一点。使 MPI 使用起来有些复杂的主要问题是缓冲区管理和阻塞语义。

这些问题是相关的，源于这样一个事实：理想情况下，数据不应该同时出现在两个地方。让我们简单考虑一下如果处理器 1 向处理器 2 发送数据会发生什么。最安全的策略是处理器 1 执行发送指令，然后等待处理器 2 确认数据被成功接收。这意味着处理器 1 被暂时阻断，直到处理器 2 实际执行其接收指令，并且数据已经通过网络。这是 MPI_Send 和 MPI_Recv 调用的标准行为，据说是使用**阻塞通信**（blocking communication）。

另外，处理器 1 可以把它的数据放在一个缓冲区里，告诉系统确保它在某个时间点被发送出去，然后再检查缓冲区是否可以重新使用。这第二种策略被称为**非阻塞通信**（non-blocking communication），它需要使用一个临时缓冲区。

#### 2.6.2.7 集合操作

在上面的例子中，你看到了 MPI_Allreduce 调用，它计算了一个全局和，并将结果留在每个处理器上。还有一个本地版本 MPI_Reduce，它只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例子。集合运算有

- **规约**（reduction）：每个处理器都有一个数据项，这些数据项需要用加法、乘法、最大或最小操作进行算术组合。其结果可以留在一个处理器上，也可以留在所有处理器上，在这种情况下，我们称之为 allreduce 操作。
- **广播**（broadcast）：一个处理器有一个数据项，所有处理器都需要接收。
- **收集**（gather）：每个处理器都有一个数据项，这些数据项需要被收集到一个数组中，而不需要通过加法等操作将其合并。其结果可以留在一个处理器上，也可以留在所有处理器上，在这种情况下，我们称其为 allgather。

- **散发**（scatter）：一个处理器有一个数据项的数组，每个处理器接收该数组的一个片段。
- **全局**（all-to-all）：每个处理器都有一个项目数组，将被分散到所有其他处理器。

集合操作是阻塞的，尽管 MPI 3.0（目前只是一个草案）将有非阻塞的集合操作。我们将在第 6.1 节详细分析集体操作的成本。

#### 2.6.2.8 非阻塞通信

传统的计算机程序中，指令执行的方式取决于处理器中正在进行的操作，而在并行程序中，情况则较为复杂。一个简单发送操作，例如发送某个缓冲区的数据会导致程序执行停止，直至该缓冲区被另一个处理器安全发送和接收时结束。这种操作被称为**非本地操作**（non-local operation），因为它依赖于其他进程的行动；这也被称为**阻塞通信**（blocking communication）操作，因为执行将停止以等待某个事件的发生。

阻塞操作的缺点是它们可能导致死锁。在消息传递的上下文中表现为：一个进程正在等待一个从未发生的事件；例如，它可能正在等待接收一个消息，而该消息的发送者正在等待其他事情。如果两个进程互相等待，或者更普遍的情况是，如果有一个进程的循环，每个进程都在等待循环中的下一个进程，就会发生死锁。例如

```
if ( /* 为处理器 0 */ )
   // 等待来自处理器 1 的消息
else if ( /* 为处理器 1 */ )
   // 等待来自处理器 0 的消息
```

这里的块接收会导致死锁。即使没有死锁，处理器在等待时并没有执行任何操作，也会使其产生大量闲置时间。其优点是可以明确缓冲区何时可以被重用：在操作完成后，可以保证数据在另一端被安全地接收。

可以通过使用非阻塞通信操作来避免阻塞行为，但代价是使缓冲区语义复杂化。一个非阻塞的发送（MPI_Isend）声明需要发送一个数据缓冲区，但随后并不等待相应的接收完成。有第二个操作 MPI_Wait，它实际上会阻塞，直到接收完成。这种发送和阻塞的解耦的好处是，现在有可能进行写入。

```
MPI_ISend(somebuffer,&handle); // 开始发送，且
   // 掌握这个特殊的通信
{...} // 做一些对本地数据做有用的工作
MPI_Wait(handle); // 锁住直至通信完成
{...} // 做一些对输入的数据进行有用的工作
```

运气好的话，本地操作所花的时间比通信的时间多，这样就完全消除了通信时间。

除了非阻塞的发送，还有非阻塞的接收。一个典型的例子如下：

```
 MPI_ISend(sendbuffer,&sendhandle);
MPI_IReceive(recvbuffer,&recvhandle);
{...} // 做一些对本地数据有用的工作
MPI_Wait(sendhandle); Wait(recvhandle);
{...} // 做一些对输入的数据进行有用的工作
```

✍ **练习 2.23** 再看一下上述方程，给出使用非阻塞发送和接收解决问题的伪代码。与阻塞式解决方案相比，这个代码的缺点是什么？

#### 2.6.2.9 三种版本的 MPI 对比

第一个 MPI 标准 [164] 有一些明显的遗漏，这些遗漏包括在 MPI 2 标准 [91] 中。其中之一是关于并行输入/输出：没有为多个进程访问同一个文件提供设施，即使底层硬件允许这样做。一个单独的项目 MPI-I/O 现在已经被纳入 MPI-2 标准。我们将在本书中讨论并行 I/O。

MPI 中缺少的第二个设施是进程管理，尽管它在 MPI 之前的 PVM[50, 73] 中就已经存在了：没有办法创建新的进程并让它们成为并行运行的一部分。最后，MPI-2 支持单边通信：一个进程将数据放入另一个进程的内存中，而接收进程不做实际接收指令。我们将在下面进行简短的讨论。

在 MPI-3 中，该标准获得了一些新的特性，如非阻塞集合体、邻接集合体和剖析接口。单边机制也得到了更新。

#### 2.6.2.10 单边通信

MPI 编写匹配发送和接收指令的方式并不理想。首先，它要求程序员两次给出相同的数据描述，一次发送，一次接收调用。其次，如果要避免死锁，它需要对通信进行相当精确的协调；如果使用异步调用的替代方法，程序将会十分繁琐，且需要程序管理大量的缓冲区。最后，它要求接收处理器知道要等待多少个传入的消息，这在不规则的应用中可能很棘手。如果有可能从另一个处理器中提取数据，或者反过来把数据放在另一个处理器上，而不需要另一个处理器明确参与，过程就会轻松很多。

一些硬件上存在的远程直接内存访问（RDMA）支持进一步鼓励了这种编程风格。一个早期的例子是 Cray T3E。如今，通过在 MPI-2 库中的整合，单边通信被广泛使用。

让我们简单看一下 MPI-2 中的单边通信，以数组值的平均化为例：

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

MPI 并行代码为

```
// 做一些转换
a_local = (a_local+left+right)/3
```

转换要完成的任务很清楚：a_local 变量需要在等级较高的处理器上成为左边的变量，而在等级较低的处理器上成为右边的变量。

首先，处理器需要明确声明哪些内存区域可用于单边传输，即所谓的" 窗口"。在这个例子中，这包括处理器上的 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);
```

该代码现在有两个选择：可以将数据推送出去

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

或将其放入

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

如果 Put 和 Get 调用是阻塞的，上述代码将具有正确的语义。然而，单边通信的部分吸引力在于它使通信的表达更加容易，为此，我们假设了一个非阻塞语义。

非阻塞的单边调用的问题是，有必要明确地确保通信成功完成。例如，如果一个处理器在另一个处理器上做了一个单边的 put 操作，另一个处理器就没有办法检查数据是否已经到达，或者是否已经开始传输。因此，有

必要在程序中插入一个全局屏障，每个包都有自己的实现。在 MPI-2 中，相关调用是 MPI_Win_fence。这些屏障实际上是将程序的执行分为**超步**（supersteps）。

另一种形式的单边通信在 Charm++ 包中使用，我们将在后面进行讨论。

### 2.6.3 混合共享/分布式内存计算

现代架构通常是共享和分布式内存的混合体。例如，一个集群在节点层面上是分布式的，但节点上的插槽和内核为共享内存。再往上一层，每个插槽可以有一个共享的 L3 缓存，但有独立的 L2 和 L1 缓存。直观地说，共享和分布式编程技术的混合似乎很清楚，可以提供与架构最匹配的代码。在这一节中，我们将讨论这种混合编程模型，并讨论其功效。

一个常见的集群设置使用分布式内存节点，每个节点包含几个彼此之间共享内存的插槽。这建议使用 MPI 在节点之间进行通信（节点间通信），使用 OpenMP 在节点上进行并行化（节点内通信）。在实践中，这实现了以下几点

- 在每个节点上启动一个 MPI 进程（而不是每个核心一个）。
- 这一个 MPI 进程然后使用 OpenMP（或其他线程协议）来产生尽可能多的线程，这些线程在节点上有独立的套接字或核心。
- 然后，OpenMP 线程可以访问节点的共享内存。

注释 由于亲和性的原因，我们希望每个插槽启动一个 MPI 进程，而不是每个节点。这并没有实质性地改变上述论点。

这种混合策略听起来是个好主意，但事实上却很复杂。

尽管 MPI 进程之间消息传递看起来比共享内存的通信开销更大，但当 MPI 的优化版本检测进程在同一个节点时，就会采取时间开销更小的数据拷贝以代替通信。不使用 MPI 的唯一理由是：每个进程都有自己的数据空间，这会因为每个进程都要为缓冲区和被复制的数据分配空间而造成内存开销。

线程更加灵活：如果代码的某一部分需要每个进程有更多的内存，那么 OpenMP 方法可以限制这一部分的线程数量。另一方面，对线程的灵活处理会产生一定的操作系统开销，而 MPI 的固定进程是没有这种开销的。

共享内存编程在概念上很简单，但也会有意想不到的性能隐患。例如，现在两个进程的性能可能会因为需要维持缓存一致性和虚假共享而受到阻碍。

另一方面，混合方法提供了一些优势，因为它捆绑了消息。例如，如果一个节点上的两个 MPI 进程分别向另一个节点上的两个进程发送消息，就会有四条消息；在混合模型中，这些消息将被捆绑成一条消息。

练习 2.24 分析上面最后一项的讨论。假设两个节点之间的带宽只够一次维持一条消息。与纯分布式模型相比，混合模型的成本节约是多少？提示：分别考虑频带宽度和延时。

这种 MPI 进程的捆绑可能有一个更深层次的技术原因的优势。为了支持握手协议，每个 MPI 进程需要为每个其他进程提供少量的缓冲空间。在进程数量较多的情况下，这可能是一个限制，因此在英特尔 Xeon Phi 等高核数处理器上，捆绑是有吸引力的。

MPI 库中明确指出了其支持的线程类型：是否完全支持多线程、是否所有的 MPI 调用都必须来自一个线程或一次一个线程，或者在从线程进行 MPI 调用时是否有完全的自由。

### 2.6.4 并行语言

缓解并行编程困难的一个方法是设计出对并行性提供明确支持的语言。下面列举了一些方法：

- 一些语言反映了科学计算中的许多操作是数据并行的。诸如 High Performance Fortran（HPF）等语言有一个数组语法，其中数组的加法等操作可以表示为 A=B+C。这种语法简化了编程，但更重要的是，它在一个抽象的层次上指定了操作，这样下层就可以对如何处理并行作出具体决定。然而，HPF 中表达的数据并行

只是最简单的一种，即数据包含在常规数组中。不规则的数据并行比较困难；而 Chapel 语言试图解决这个问题。

- 并行语言中的另一个概念，不一定与前者正交，是分区全局地址空间（PGAS）模型：只有一个地址空间（与 MPI 模型不同），但这个地址空间是分区的，每个分区与线程或进程有亲和力。因此，这个模型包含了 SMP 和分布式共享内存。一种典型的 PGAS 语言，Unified Parallel C（UPC），允许我们编写看起来像普通的 C 代码。然而，通过指出主要阵列在处理器上的分布方式，程序可以被并行执行。

### 2.6.4.1 讨论

并行语言有希望使并行编程变得更容易，因为它们使通信操作看起来像简单的复制或算术操作。然而，通过这样做，它们邀请用户编写可能并不高效的代码，例如，通过导出许多小信息。

作为一个例子，考虑将数组 a,b 在处理器上进行水平分割，并进行移位（见图 2.15）。

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

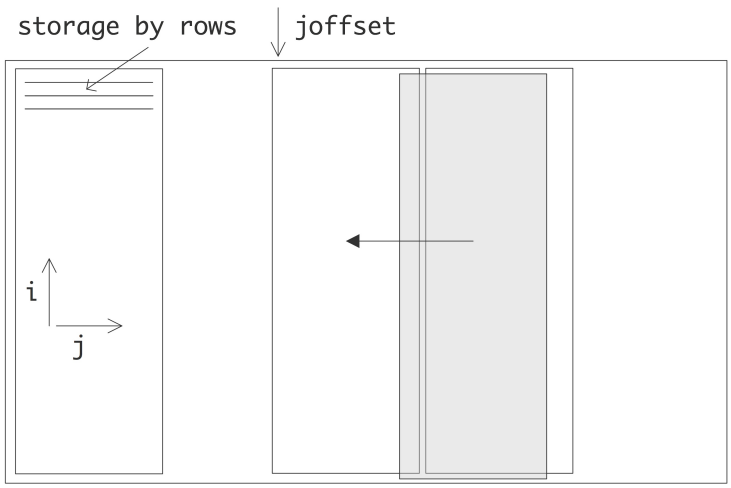



图 2.15: 需要通信的数据转移

如果这段代码在共享内存机器上执行，它将是高效的，但在分布式情况下的原始翻译将在 $i$ 循环的每个迭代中传达一个数字。显然，这些都可以结合在一个缓冲区的发送/接收操作中，但编译器通常无法进行这种转换。因此，用户被迫，实际上，重新实现了需要在 MPI 实现中完成的阻塞。

```
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]
      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，PGAS 语言可以比显式消息传递更有效率，即使是物理分布式内存。

### 2.6.4.2 Unified Parallel C

统一并行 C（UPC）[191] 是 C 语言的一个扩展。它的主要并行来源是数据并行，编译器发现了数组上操作的独立性，并将其分配给不同的处理器。该语言有一个扩展的数组声明，允许用户指定数组是按块划分，还是以轮流方式划分。

下面的 UPC 程序执行了一个向量与向量加法。

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

同样的程序有一个明确的并行循环结构

```
//vect_add.c
    #include <upc_relaxed.h>
    #define N 100*THREADS
    shared 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];
}
```

在含义上与 UPC 相当，但基于 Java 而不是 C。

### 2.6.4.3 High Performance Fortran

高性能 Fortran5（HPF）是 Fortran90 的一个扩展，具有支持并行计算的结构，由高性能 Fortran 论坛（HPFF）发布。HPFF 由莱斯大学的 Ken Kennedy 召集并担任主席。HPF 报告的第一个版本发表于 1993 年。

在 Fortran 90 引入的数组语法的基础上，HPF 使用数据并行计算模型来支持将单个数组计算的工作分散到多个处理器上。这使得在 SIMD 和 MIMD 风格的架构上都能有效地实现。HPF 的特点包括。

- 新的 Fortran 语句，如 FORALL，以及创建 PURE（无副作用）程序的能力。
- 使用编译器指令来推荐阵列数据的分布。
- 用于与非 HPF 并行程序接口的外在程序接口，如那些使用消息传递。
- 额外的库例程，包括环境查询、并行前缀/后缀（例如，’ 扫描’）、数据散射和排序操作。

Fortran 95 整合了几个 HPF 功能。虽然一些供应商在 20 世纪 90 年代确实将 HPF 纳入了他们的编译器中，但有些方面被证明是难以实现的，而且用途值得怀疑。从那时起，大多数供应商和用户都转向了基于 OpenMP 的并行处理。然而，HPF 仍然有影响。例如，为即将到来的 Fortran-2008 标准提出的 BIT 数据类型包含了许多直接来自 HPF 的新的内在函数。

#### 2.6.4.4 Co-array Fortran

Co-array Fortran（CAF）是 Fortran 95/2003 语言的一个扩展。支持并行的主要机制是对数组声明语法的扩展，其中一个额外的维度表示并行分布。例如，在

```
Real,dimension(100),codimension[*] :: X
Real :: Y(100)[*]
Real :: Z(100,200)[10,0:9,*]
```

数组 $X, Y$ 在每个处理器上有 100 个元素。数组 $Z$ 的行为就像可用的处理器在一个三维网格上，其中两边是指定的，第三边可以调整以适应可用的处理器。

现在处理器之间的通信是通过沿着描述处理器网格的（共）维度的拷贝来完成的。Fortran 2008 的标准包括共同数组。

#### 2.6.4.5 Chapel

Chapel[30] 是一种新的并行编程语言，由 Cray 公司开发，是 DARPA 领导的高生产率计算系统计划（HPCS）的一部分。Chapel 旨在提高高端计算机用户的生产效率，同时也是一个可移植的并行编程模型，可用于商品集群或桌面多核系统。Chapel 致力于极大地提高大规模并行计算机的亲和力，同时匹配或击败当前编程模型（如 MPI）的性能和可移植性。

Chapel 通过对数据并行、任务并行、并发和嵌套并行的高级抽象支持多线程执行模型。Chapel 的 locale 类型使用户能够指定并重新确定数据和任务在目标架构上的位置，以便对位置进行调整。Chapel 支持具有用户定义实现的全局视图数据聚合，允许以自然方式表达对分布式数据结构的操作。与许多以前的高级并行语言相比，Chapel 是围绕多分辨率哲学设计的，允许用户最初编写非常抽象的代码，然后逐步增加细节，直到他们接近机器的需要。Chapel 通过面向对象的设计、类型推理和通用编程的功能，支持代码重用和快速原型设计。

Chapel 是根据第一原则设计的，而不是通过扩展现有的语言。它是一种 im-perative 块状结构的语言，旨在使 C、C++、Fortran、Java、Perl、Matlab 和其他流行语言的用户易于学习。虽然 Chapel 建立在许多以前的语言的概念和语法上，但它的并行功能最直接地受到 ZPL、高性能 Fortran（HPF）和 Cray MTA 对 C 和 Fortran 的扩展的影响。下面是 Chapel 中的向量与向量加法：

```
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;
```

#### 2.6.4.6 Fortress

Fortress[67] 是由 Sun Microsystems 开发的一种编程语言。Fortress7 的目的是通过几种方式使平行主义更容易操作。首先，并行性是默认的。这是为了推动工具设计、库设计和程序员技能向并行化方向发展。第二，语言被设计成对并行更友好。不鼓励副作用，因为副作用需要同步化以避免错误。Fortress 提供了事务，这样程序员就不会面临确定锁定顺序的任务，或者调整他们的锁定代码，以便有足够的正确性，但又不至于妨碍性能。Fortress 的循环结构，连同库，把" 迭代" 变成了内部；而不是循环指定如何访问数据，数据结构指定如何运行循环，聚合数据结构被设计成可以有效地安排并行执行的大型部分。Fortress 还包括来自其他语言的功能，旨在普遍地帮助提高生产力--测试代码和方法，与被测试的代码相联系；合同，可以在代码运行时选择检查；以及属性，可能运行成本太高，但可以反馈给定理验证器或模型检查器。此外，Fortress 还包括安全的语言特性，如检查数组边界、类型检查和垃圾收集，这些在 Java 中已经被证明是有用的。Fortress 的语法被设计为尽可能地类似于数学语法，因此任何人在解决其规范中的数学问题时，都可以写出一个与原始规范明显相关的程序。

#### 2.6.4.7 X10

X10 是一种实验性的新语言，目前正在 IBM 与学术伙伴合作开发。X10 工作是 DARPA 高生产率计算机系统计划中的 IBM PERCS 项目（生产性易使用的可靠计算机系统）的一部分。PERCS 项目专注于硬件-软件联合设计方法，以整合芯片技术、架构、操作系统、编译器、编程语言和编程工具方面的进展，提供新的可适应、可扩展的系统，在 2010 年之前将并行应用的开发效率提高一个数量级。

X10 旨在通过开发新的编程模型，结合集成到 Eclipse 中的一套新的工具和新的实现技术，在可管理的运行环境中提供优化的可扩展的并行性，为提高生产率作出贡献。X10 是一种类型安全的、现代的、并行的、面向对象的分布式语言，旨在让 Java(TM) 程序员能够使用。它的目标是未来的低端和高端系统，其节点由多核 SMP 芯片构成，具有非统一的内存层次，并以可扩展的集群配置互连。作为分区全局地址空间（PGAS）语言家族中的一员，X10 强调以地方的形式明确地重新定义位置；体现在 async、future、foreach 和 attach con-结构中的轻量级活动；用于终止检测（finish）和分阶段计算（clocks）的结构；使用无锁同步（原子块）；以及对全局数组和数据结构进行操作。

#### 2.6.4.8 Linda

现在应该很清楚了，数据的处理是迄今为止并行编程最重要的方面，远比算法方面的考虑更重要。编程系统 Linda[74, 75]，也被称为协调语言，旨在明确地解决数据处理问题。琳达不是一种语言，但是可以，而且已经被纳入其他语言。

琳达的基本概念是元组空间：通过给数据添加一个标签，将其添加到一个全局可访问的信息池中。然后，进程通过标签值来检索数据，而不需要知道是哪个进程将数据添加到元组空间中的。

Linda 主要针对的是与高性能计算（HPC）不同的计算模型：它解决的是异步通信进程的需求。然而，它已经被用于科学计算 [45]。例如，在热方程的并行模拟中，处理器可以将他们的数据写入元组空间，而相邻的进程可以检索他们的幽灵重区，而不必知道它的出处。因此，Linda 成为实现单边通信的一种方式。

#### 2.6.4.9 The Global Arrays library

The Global Arrays library（http://www.emsl.pnl.gov/docs/global/）是另一个单边通信的例子，事实上它早于 MPI。这个库的主要数据结构是笛卡尔积数组 8，分布在相同或更低维度的处理器网格上。通过库的调用，任何处理器都可以通过放或取的操作访问阵列中的任何子砖。这些操作是非集体的。与任何单边协议一样，屏障同步是必要的，以确保发送/接收的完成。

### 2.6.5 基于操作系统的方法

可以设计一个具有共享地址空间的架构，并让数据移动由操作系统处理。Kendall Square 计算机 [124] 有一个名为" 全缓存" 的架构，其中没有数据与任何处理器直接相关。相反，所有的数据都被认为是缓存在一个处理器上，并根据需要通过网络移动，就像数据从主内存移动到普通 CPU 的缓存中一样。这个想法类似于目前 SGI 架构中的 NUMA 支持。

### 2.6.6 活跃通信

MPI 范式传统上是基于双侧操作的：每个数据传输都需要一个明确的发送和接收操作。这种方法对于相对简单的代码来说效果很好，但是对于复杂的问题来说，就很难协调所有的数据移动。简化的方法之一是使用**活跃通信**（active message）。这在 Charm++[119] 包中被使用。

通过主动消息，一个处理器可以向另一个处理器发送数据，而不需要第二个处理器做明确的接收操作。相反，接收者声明处理传入数据的代码，用客观方向的说法是" 方法"，而发送处理器则用它想发送的数据调用这个

方法。由于发送处理器实际上是激活了另一个处理器上的代码，这也被称为**远程调用**（remote method invocation）。这种方法的一个很大的优点是，通信和编译的重叠变得更容易实现。

作为一个例子，考虑用一个三对角矩阵进行矩阵与向量乘法

$$\forall_i : y_i \leftarrow 2x_i - x_{i+1} - x_{i-1}.$$

假设每个处理器正好有一个索引 $i$，MPI 代码可以是这样的。

```
 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
}
```

有了活跃通信，这看起来就像

```
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);
```

### 2.6.7 批量同步并行

MPI 库可以带来非常高效的代码。这样做的代价是，程序员需要非常详细地说明通信的内容。在光谱的另一端，PGAS 语言对程序员的要求很低，但却没有带来多少性能回报。一种试图找到中间地带的方法是**批量同步并行**（Bulk Synchronous Parallel，BSP）模型 [192, 183]。在这里，程序员需要写出通信，但不是它们的顺序。

BSP 模型将程序排列成一个超步的序列，每个步骤都以一个障碍物同步结束。在一个超步中开始的通信都是异步的，并依靠屏障来完成。这使得编程更容易，并消除了死锁的可能性。

此外，所有通信都是单边通信类型。

✍ **练习 2.25** 考虑 2.1 节中的并行求和例子。论证 BSP 的实现需要 $log_2 n$ 超步。

由于其通过障碍物完成超级步骤的处理器的同步，BSP 模型可以对并行算法做一个简单的成本分析。

BSP 模型的另一个方面是它对问题的**过度分解**（overdecomposition），即把多个进程分配给每个处理器，以及**随机放置**（random placement）数据和任务。这是以统计学的论点为依据的，表明它可以补救负载的不均衡。如果有 $p$ 个处理器，如果在一个超步中进行了 $p$ 次远程访问，那么很可能有些处理器会收到 $\log p / \log \log p$ 次访问，而其他处理器则没有收到。因此，负载不均衡的问题会随着处理器数量的增加而变得更加严重。另一方面，如果有 $p \log p$ 的访问，例如因为每个处理器上有 $\log p$ 的进程，最大的访问次数是 $3 \log p$，而且概率很大。这意味着负载平衡是在一个完美的恒定系数内。

BSP 模型是在 BSPlib[107] 中实现的。其他系统可以说是类似 BSP 的，因为它们使用了超步的概念；例如，谷歌的 Pregel[150]。

### 2.6.8 数据依赖

如果两个语句引用了相同的数据项，我们就说这些状态之间存在着**数据依赖**（data dependency）关系。这种依赖关系限制了语句的执行可以被重新安排的程度。对这一主题的研究可追溯到 20 世纪 60 年代，当时处理器可以不按串行执行语句以提高吞吐量。语句的重新排序受到了限制，因为执行必须遵守程序的**串行语义**（program order）：结果必须像语句严格按照它们在程序中出现的串行执行一样。

语句排序以及因此而产生的数据依赖性的问题，以几种方式出现：

- 并行化编译器必须对资源进行分析，以确定允许哪些转换。
- 如果你用 OpenMP 指令并行化一个顺序代码，你必须自己进行这样的分析。

这里有两种需要进行这种分析的活动：

- 当一个循环被并行化时，迭代不再按其程序顺序执行，所以我们必须检查依赖关系。
- 引入任务是指程序的某些部分可以按照与顺序执行不同的顺序执行。

依赖性分析的最简单的情况是检测循环迭代是否可以独立执行。如果一个数据项在两个不同的迭代中被读取，迭代当然是独立的，但是如果同一个项目在一个迭代中被读取，在另一个迭代中被写入，或者在两个不同的迭代中被写入，我们需要做进一步分析。

数据依赖性的分析可以由编译器来执行，但是编译器必须采取一种保守的方法。这意味着迭代可能是独立的，但不能被编译器所识别。因此，OpenMP 把这个责任转移给了程序员。

现在，我们将详细讨论数据依赖的细节。

#### 2.6.8.1 数据依赖类型

数据依赖类型

- **流依赖**（flow dependencies），或" 读后写"。
- **反依赖**（anti dependencies），即" 读后写"；以及
- **输出依赖**（output dependencies），即" 写完再写"。

这些依赖关系可以在标量代码中进行研究，事实上编译器也是这样做的，以确定语句是否可以重新排列，但是我们将主要关注它们在循环中的出现，因为在科学计算中很多工作都出现在这里。

- 流依赖：如果读和写发生在同一个循环迭代中，那么流量依赖，或者说读-写，就不是一个问题。

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

另一方面，如果读取发生在后来的迭代中，就没有简单的方法来并行化或向量化循环。

```
for (i=0; i<N; i++) {
  .... = ... x[i] ... ;
  x[i+1] = .... ;
```

```
}
```

这通常需要重写代码。

**练习 2.26** 考虑如下代码

```
for (i=0; i<N; i++) {
  a[i] = f(x[i]);
  x[i+1] = g(b[i]);
}
```

其中 $f()$ 和 $g()$ 表示没有进一步依赖 x 或 i 的算术表达式。

- 反依赖性：反依赖性或读后写的最简单情况是减少。

```
for(i=0; i<N; i++){
  t =t+ ...
}
```

这可以通过明确声明循环是一个减法来处理，或者使用任何其他策略。

如果读和写是在一个数组上，情况就更复杂了。这个片段中的迭代

```
for (i=0; i<N; i++) {
  x[i] = ... x[i+1] ... ;
}
```

不能像这样以任意顺序执行。然而，从概念上讲，这并不存在依赖性。我们可以通过引入一个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的例子，因为它可能会大大影响程序的内存需求。因此，这就留给了程序员。

- 输出依赖：输出依赖或写后依赖的情况本身不会发生：如果一个变量被依次写了两次，中间没有读，那么第一次写可以被删除而不改变程序的意义。因此，这种情况会减少为流动依赖。

其他的输出依赖也可以被移除。在下面的代码中，$t$ 可以被声明为私有，从而消除了依赖性。

```
for (i=0; i<N; i++) {
  t = f(i)
  s += t*t;
}
```

如果想要 $t$ 的最终值，可以在 OpenMP 中使用 lastprivate。

### 2.6.8.2 嵌套循环的并行化

在上述例子中，如果在一个循环的迭代 $i$ 中出现了不同的指数，如 $i$ 和 $i+1$，那么数据的依赖性就是非实质性的。反之，循环如

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

是简单的并行化。然而，嵌套的循环则需要更多的思考。OpenMP 有一个" 折叠" 指令，用于诸如以下的循环

```
for (int i=0; i<M; i++)
  for (int j=0; j<N; j++)
    x[i][j] = x[i][j] + y[i] + z[j];
```

这里，整个 $i, j$ 迭代空间是并行的。这是怎么回事?

```
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 2.27** 对这个循环做一个重用分析。假设 $a, b, c$ 不能一起放进缓存。现在假设 $c$ 和 $b$ 的一行可以放入缓存，并且还有一点空间。你能找到一个能使性能大大提高的循环交换吗? 写一个测试来证实这一点。

分析这个循环嵌套的并行性，你会发现 $j$ 循环是一个减法，而 $n$ 循环有流量依赖：每个 $a[i]$ 在每个 $n$ 次迭代中被更新。结论是，你只能合理地并行化 $i$ 环路。

✍ **练习 2.28** 这个并行性分析与练习 2.27 中的循环交换有什么关系? 交换后的循环是否仍然是可并行的?

如果你会说 OpenMP，请通过编写将 a 的元素相加的代码来确认你的答案，无论交换和引入 OpenMP 并行性，你都应该得到同样的答案。

### 2.6.9 并行程序设计

很久以前，人们认为编译器和运行时系统的某种神奇组合可以将现有的串行程序转化为并行程序。这种希望早已破灭，所以现在的并行程序从一开始就被写成了并行程序。当然，有不同类型的并行性，它们对你如何设计你的并行程序都有各自的影响。在这一节中，我们将简要地探讨其中的一些问题。

#### 2.6.9.1 并行数据结构

并行程序设计中的一个问题是使用**数组结构**（Array-Of-Structures，AoS）与**结构化数组**（Structure-Of-Arrays，SoA）。在正常的程序设计中，我们经常定义一个结构

```
struct {int number; double xcoord,ycoord;} _Node;
struct {double xtrans,ytrans} _Vector;
typedef struct _Node* Node;
typedef struct _Vector* Vector;
```

而如果需要许多这样的结构，我们就创建一个这样的结构数组。

```
Node *nodes = (Node*) malloc( n_nodes*sizeof(struct _Node) );
```

这就是 AoS 的设计。

现在，假设我们想将一个操作并行化

```
void shift(Node the_point,Vector by) {
  the_point->xcoord += by->xtrans;
  the_point->ycoord += by->ytrans;
}
```

这是在一个循环中完成的

```
for (i=0; i<n_nodes; i++) {
  shift(nodes[i],shift_vector);
}
```

这段代码具有 MPI 编程的正确结构，每个处理器都有自己的本地节点数组。这个循环也很容易用 OpenMP 并行化。

然而，在 20 世纪 80 年代，人们意识到 AoS 的设计并不适合向量计算机，因此不得不对代码进行大幅重写。在这种情况下，我们的操作数需要是连续的，所以代码必须采用 SoA 设计。

```
node_numbers = (int*) malloc( n_nodes*sizeof(int) );
node_xcoords = // et cetera
node_ycoords = // et cetera
```

而将迭代

```
for (i=0; i<n_nodes; i++) {
  node_xoords[i] += shift_vector->xtrans;
  node_yoords[i] += shift_vector->ytrans;
}
```

最初的 SoA 设计最适合于分布式内存编程吗，这意味着在向量计算机时代的 10 年后，每个人都必须为集群重新编写他们的代码。当然，如今随着 SIMD 宽度的增加，我们也需要部分地回到 AoS 的设计。（在英特尔的 ispc 项目中，有一些实验性的软件支持这种转变，http: //http://ispc.github.io/，它将 SPMD 代码翻译成 SIMD）

### 2.6.9.2 延迟隐蔽性

处理器之间的通信通常很慢，比单个处理器上的内存数据传输要慢，而且比对数据的操作要慢得多。因此在设计一个并行程序时，最好考虑到网络流量与" 有用" 操作的相对数量。每个处理器必须有足够的工作来抵消通信。

应对通信相对缓慢的另一种方法是安排程序，使通信实际发生在一些计算正在进行的时候。这被称为**计算通信重叠**（overlapping computation with communication）或**延迟隐藏**（latency hiding）。

例如，考虑矩阵与向量乘积 $y = A$ 的并行执行。假设向量是分布式的，那么每个处理器 $p$ 都会执行

$$\forall_{i\in I_p} : y_i = \sum_j a_{ij}x_j$$

由于 $x$ 也是分布式的，我们可以将其写为

$$\forall_{i\in I_p} : y_i = \left( \sum_{j\text{local}} + \sum_{j\text{note local}} \right) a_{ij}x_j$$

如图 2.16 所示。我们现在可以按以下方式进行：

- 开始转移 $x$ 的非本地元素。
- 在数据传输过程中，对 $x$ 的本地元素进行操作。
- 确保传输完成。
- 对 $x$ 的非本地元素进行操作。

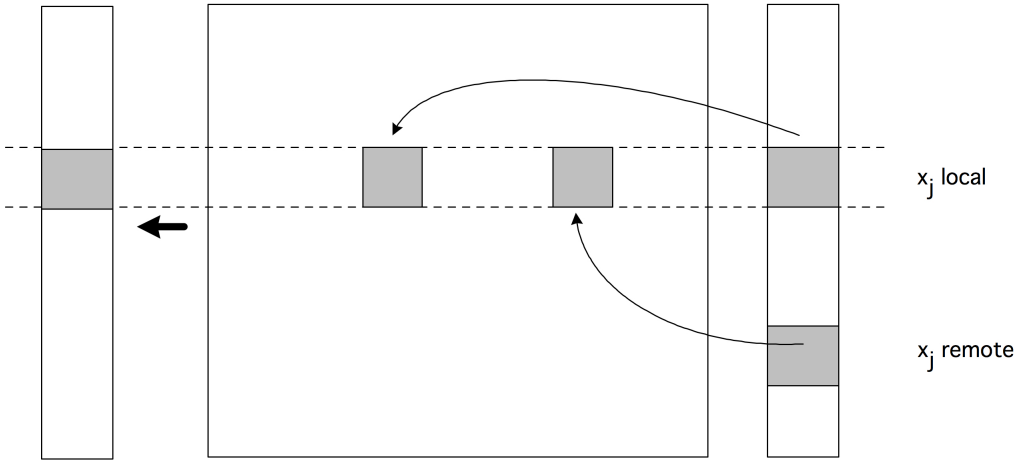



图 2.16: 块排分布的并行矩阵-向量乘积

练习 2.29 你能从计算通信重叠中获得多少好处？提示：考虑计算耗时为零且只有通信的边界情况，以及相反的情况。现在考虑一般情况。

## 2.7 拓扑

当然，这种情况的前提是有软件和硬件对这种重叠的支持。MPI 允许这样做，通过所谓的异步通信或非阻塞通信例程。这并不立即意味着重叠将实际发生，因为硬件支持是一个完全独立的问题。

工作在同一个任务上的不同处理器常常需要交流，这需要有一种方法能够帮助其交换数据。本节中我们将讨论一些常见的处理机连接方式，这被称为（处理器）**拓扑**（topology）。

为了明确这里的问题，请考虑两个不能" 扩展" 的简单方案。

- 以太网是一种连接方案，网络上的所有机器都在一条电缆上（见下文注释）。如果一台机器在电线上放了一个信号来发送信息，而另一台机器也想发送信息，那么后者将检测到唯一可用的通信通道被占用，它将等待一段时间后再重新进行发送操作。在以太网上接收数据是很简单的：信息包含了目标接收者的地址，所以一个处理器只需要检查电线上的信号是否是为它准备的。这个方案的问题应该很清楚。通信通道的容量是有限的，所以当更多的处理器连接到它时，每个处理器可用的容量将下降。由于解决冲突的方案，信息开始前的平均延迟也会增加。
- 在完全连接的配置中，每个处理器都有一条与其他处理器通信的线路。其他处理器。这种方案是完美的，因为消息可以在最短的时间内发送，而且两个消息永远不会互相干扰。一个处理器可以发送的数据量不再是处理器数量的递减函数；事实上，它是一个递增函数，如果网络控制器可以处理，一个处理器甚至可以同时进行多次通信。当然，这个方案的问题是，一个处理器的网络接口的设计不再是固定的：随着更多的处理器被添加到并行机器上，网络接口得到更多的连接线。网络控制器也同样变得更加复杂，机器的成本增加速度超过了处理器数量的线性增长。

注释 以上对以太网的描述是对原始设计的描述。随着交换机的使用，特别是在 HPC 的背景下，这种描述已经不再真正适用。

最初人们认为，信息碰撞意味着以太网将不如其他解决方案，如 IBM 的令牌环网，它明确地防止碰撞。需要相当复杂的统计分析来证明，以太网的工作原理比朴素预期好得多。

在本节中，我们将看到一些可以增加到大量处理器的方案。

### 2.7.1 图论

互联并行计算机中的处理器的网络可以方便地用一些基本的图论概念来描述。我们用一个图来描述并行机器，每个处理器都是一个节点，如果两个节点之间有直接的联系，那么这两个节点就是相连的。(我们假设连接是对称的，所以网络是一个无向图)

下面分析图的两个重要概念。

首先，图中一个节点的程度是它所连接的其他节点的数量。节点代表处理器，边代表导线，很明显，高度不仅是计算效率所希望的，而且从工程的角度来看也是昂贵的。我们假设所有处理器都有相同的度。

其次，从一个处理器到另一个处理器的信息，通过一个或多个中间节点，很可能在节点之间路径的每个阶段产生一些延迟。由于这个原因，图的直径很重要。直径被定义为任何两个节点之间的最大最短距离，包括链接的数量。

$$d(G) = \max_{i,j} \mid \text{shortest path between} i \text{and} j \mid$$

如果 $d$ 是直径，如果在一条线上发送一个信息需要单位时间，这意味着一个信息总是在最多 $d$ 时间内到达。

✍ **练习 2.30** 找出处理器的数量、它们的程度和连接图的直径之间的关系。

除了" 一个消息从处理器 A 到处理器 B 需要多长时间" 的问题外，我们还经常担心两个同时进行的消息之间的冲突：是否存在两个同时进行的消息需要使用同一网络链接的可能性？在图 2.18 中，我们说明了如果每个处理器 $p_i$ 在 $i < n/2$ 的情况下向 $p_{i+n/2}$ 发送消息会发生什么：会有 $n/2$ 的消息试图通过 $p_{n/2-1}$ 和 $p_n$ 之间的线路。这种冲突被称为**拥堵**（congestion）或**争夺**（contention）。显然，一台并行计算机的链接越多，发生拥堵的机会就越小。

描述拥堵可能性的一个精确方法是看**二分宽度**（bisection width）。这被定义为将处理器图分割成两个非连接图所必须移除的最小链接数。例如，考虑处理器连接成一个线性阵列，即处理器 $P_i$ 与 $P_{i-1}$ 和 $P_{i+1}$ 连接。在这种情况下，分界线宽度为 1。

二分宽度 $w$ 描述了在一台并行计算机中可以保证有多少信息同时进行。证明：采取 $w$ 发送和 $w$ 接收处理器。这样定义的 $w$ 路径是不相交的：如果不相交，我们只需去除 $w-1$ 个链接就可以把处理器分成两组。

当然，在实践中，超过 $w$ 条信息可以同时进行。例如，在一个线性阵列中，$w = 1$，如果所有的通信都是在邻居之间，如果一个处理器在任何时候都只能发送或接收，而不能同时发送和接收，则可以同时发送和接收 $P/2$ 条信息。如果处理器可以同时发送和接收，那么网络中可以有 $P$ 个信息正在进行。

二分宽度也描述了网络中的**冗余度**（redundancy ）：如果一个或多个连接出现故障，信息是否仍能从发送方找到接收方？

虽然二分宽度是一种表示导线数量的措施，但实际上我们关心的是通过导线的容量。这里的相关概念是二**分带宽**（bisection bandwidth）：横跨分节宽度的带宽，是分节宽度与导线容量（以每秒比特为单位）的乘积。二分带宽可以被认为是衡量任意一半处理器与另一半处理器进行通信所能达到的带宽。二分带宽是一个比有时引用的总带宽更现实的衡量标准，它被定义为每个处理器都在发送时的总数据率：处理器的数量乘以连接的带宽乘以一个处理器可以执行的同时发送的数量。这可能是一个相当高的数字，而且它通常不能代表实际应用中实现的通信速率。

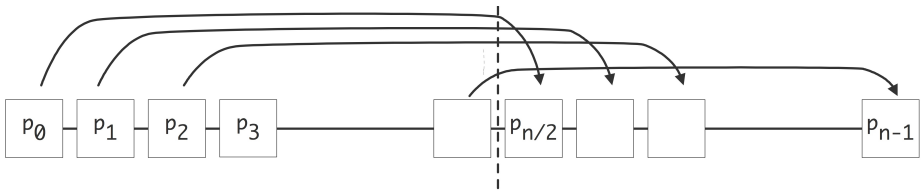



**图 2.17:** 由于同时发出的信息，对网络链接的争夺

### 2.7.2 总线

我们考虑的第一个互连设计是让所有的处理器位于同一内存总线上。这种设计将所有处理器直接连接到同一个内存池，因此它提供了一个 UMA 或 SMP 模型。

使用总线的主要缺点是可扩展性有限，因为每次只有一个处理器可以进行内存访问。为了克服这个问题，我们需要假设处理器的速度比内存慢，或者处理器有缓存或其他本地内存来操作。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，维持缓存一致性是很容易的，这个过程被称为**监听**（snooping）。

### 2.7.3 线性阵列和环状网络

连接多个处理器的一个简单方法是将它们连接成一个**线性阵列**（linear array）：每个处理器都有一个编号 $i$，处理器 $P_i$ 与 $P_{i-1}$ 和 $P_{i+1}$ 相连。第一个和最后一个处理器是可能的例外情况：如果它们相互连接，我们称该架构为环状网络（ring network）。

这个方案要求每个处理器有两个网络连接，所以设计相当简单。

✍ **练习 2.31** 线性阵列的二分宽度是什么？环状网络的二分宽度是什么？

✍ **练习 2.32** 由于线性数组的连接有限，你可能必须对并行算法进行巧妙的编程。例如，考虑一个" 广播" 操作：处理器 0 有一个数据项需要发送给其他每个处理器。

我们做了以下简化的假设。

- 一个处理器可以同时发送任意数量的信息。
- 但一条线一次只能携带一条信息；然而。
- 任何两个处理器之间的通信都需要单位时间，不管它们之间有多少个处理器。

在一个**全连接**（fully connected）的网络或一个**星型**（star）网络中，你可以很容易地写出：

```
for   = 1 ...   - 1:
    send the message to processor
```

假设一个处理器可以发送多个消息，即操作是一步到位的。现在考虑一个线性阵列。说明即使有这种无限的发送能力，上述算法也会因为拥堵而遇到麻烦。

请你尝试找到一个更好的方法来组织发送操作。提示：假设处理器是以二叉树的形式连接的。假设有 $N = 2^n - 1$ 个处理器。证明广播可以在对数 $N$ 个阶段内完成，并且处理器只需要能够同时发送一条信息即可。

这个练习是一个将" 逻辑" 通信模式嵌入物理模式的例子。

### 2.7.4 二维和三维阵列、环面

一种流行的并行计算机设计是将处理器组织在一个二维或三维的**笛卡尔网状**（Cartesian mesh）网络中。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都与邻居相连。处理器的设计还是相当简单的：网络连接的数量（连接图的度数）是网络的空间维数（2 或 3）的两倍。

拥有二维或三维网络是一个相当自然的想法，因为我们周围的世界是三维的，而且计算机经常被用来模拟现实生活的现象。如果我们暂时接受物理模型需要近邻型通信（我们将在第 4.2.3 节看到这种情况），那么网状计算机是运行物理模拟的自然候选者。

✍ **练习 2.33** $n \times n \times n$ 处理器的三维立方体的直径是多少？二分宽度是多少？如果增加环绕环状的连接，会有什么变化？

✍ **练习 2.34** 你的并行计算机的处理器被组织成一个二维网格。芯片制造商推出了一种具有相同时钟速度的新芯片，它是双核的，而不是单核的，而且可以装在现有的插槽上。批评以下论点：" 每秒钟可以完成的工作量（不涉

及通信）增加了一倍；由于网络保持不变，二分带宽也保持不变，所以我可以合理地期望我的新机器变得两倍快"。

基于网格的设计通常有所谓的环绕或环形连接，它连接二维网格的左右两边，以及顶部和底部。这在图 2.18 中有所说明。

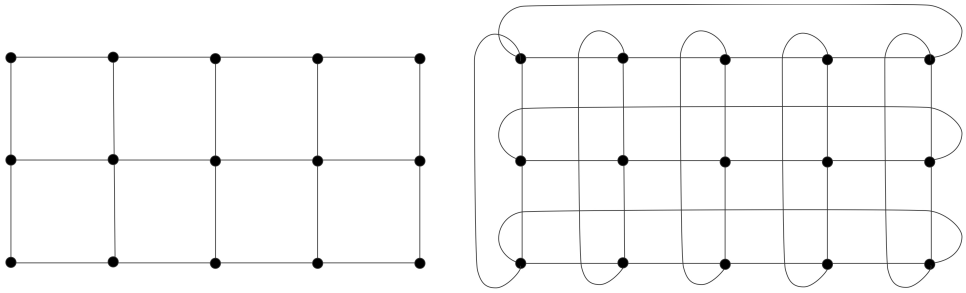


图 2.18: 带有环形连接的二维网格

一些计算机设计声称是高维度的网格，例如 5D，但这里并不是所有的维度都是平等的。例如，一个 3D 网格，其中每个节点是一个四插槽四核，可以被认为是一个 5D 网格。然而，最后两个维度是完全相连的。

### 2.7.5 超立方体

上面我们根据近邻通信的普遍性，对网状组织处理器的适用性做了一个挥手的论证。然而，有时会发生在随机处理器之间的发送和接收。这方面的一个例子就是上面提到的广播。由于这个原因，它希望有一个比网状网络直径小的网络。另一方面，我们希望避免全连接网络的复杂设计。

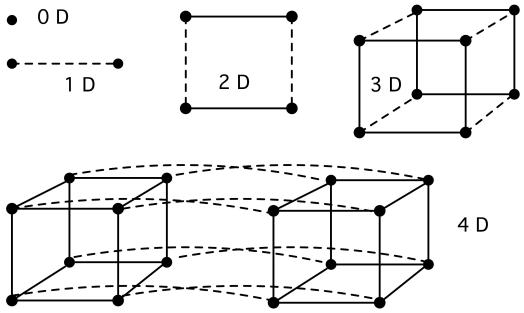



图 2.19: 超立方体

一种不错的解决方案是**超立方体**（hypercube）设计。一个 $n$ 维的超立方体计算机有 $2^n$ 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；见图 2.19。

一个简单的描述方法是给每个处理器一个由 $d$ 位组成的地址：我们给超立方体的每个节点一个数字，这个数字是描述它在立方体中的位置的比特模式；见图 2.20。

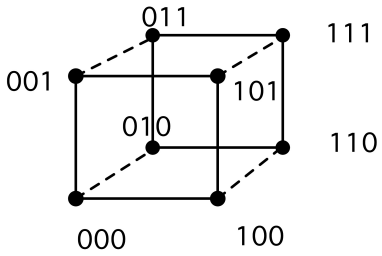



图 2.20: 超立方体的节点编号

有了这个编号方案，一个处理器就会与其他所有地址正好相差一位的处理器连接起来。这意味着，与网格不同的是，一个处理器的邻居的号码并不是相差 1 或 $\sqrt{P}$，而是相差 1,2,4,8,....。

超立方体设计的最大优点是直径小，通过网络的流量容量大。

✍ **练习 2.35** 超立方体的直径是多少？二分宽度是多少？

该方案的一个缺点是，处理器的设计取决于机器的总尺寸。在实践中，处理器会被设计成可能的最大连接数，而购买较小机器的人就会为未使用的容量买单。另一个缺点是，扩展一台给定的机器只能通过加倍来实现：$2^p$ 以外的其他尺寸是不可能的。

✍ **练习 2.36** 考虑第 2.1 节中的并行求和例子，并给出在超立方体上并行实现的执行时间。证明在超立方体上的执行可以达到该例子的理论速度（最多一个系数）。在超立方体中嵌入网格上面我们提出了一个论点，即网格连接的处理器是许多物理现象建模应用的合理选择。超立方体看起来不像网格，但它们有足够的连接，可以通过忽略某些连接来简单地假装成网格。

比方说，我们想要一个一维数组的结构：我们想要有编号的处理器，这样处理器可以直接向 $i-1$ 和 $i+1$ 发送数据。我们不能像图 2.20 中那样使用明显的节点编号。例如，节点 1 与节点 0 直接相连，但与节点 2 的距离为 2。节点 3 在一个环中的右邻，节点 4，甚至在这个超立方体中的最大距离为 3。显然，我们需要以某种方式对节点重新编号。我们将展示的是，有可能在超立方体中行走，精确地触摸每个角落，这相当于在超立方体中嵌入一个一维网格。

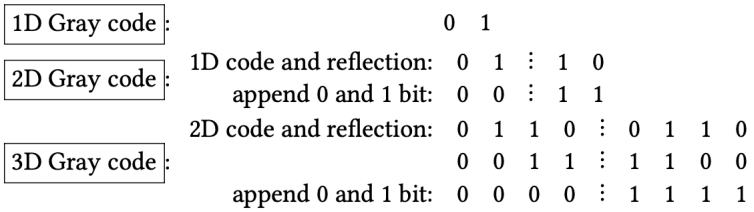



图 2.21: 格雷编码

这里的基本概念是一个（二进制反映的）**格雷编码**（Gray code）[87]。这是一种将二进制数 $0...2^{d-1}$ 排序为 $g_0, ...g_{2d-1}$ 的方法，即 $g_i$ 和 $g_{i+1}$ 只相差一个比特。显然，普通的二进制数并不满足这一点：1 和 2 的二进制表示已经有两个比特的差异。为什么格雷编码能帮助我们？因为 $g_i$ 和 $g_{i+1}$ 只相差一位，这意味着它们是超立方体中直接相连的节点数。

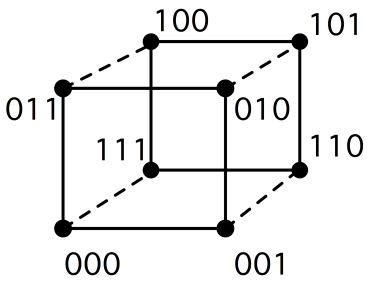



图 2.22: 超立方体的节点的格雷编码

图 2.21 说明了如何构建一个格雷编码。这个过程是递归的，可以正式描述为" 将立方体分为两个子立方体，对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相反顺序对其节点进行编号"。二维立方体的结果如图 2.22 所示。

由于格雷编码为我们提供了一种将一维" 网状" 嵌入超立方体的方法，我们现在可以继续往上做。

✍ **练习 2.37** 显示如何将一个 $2^{2d}$ 节点的正方形网格嵌入到一个超立方体中，方法是将两个 2d 节点的立方体嵌入的比特模式相加。你如何容纳一个 $2^{d_1+d_2}$ 节点的网格？一个由 $2^{d_1+d_2+d_3}$ 个节点组成的三维网格？

### 2.7.6 交换机网络

上面我们简要地讨论了完全连接的处理器。然而，通过在所有处理器之间制作大量的总线来进行连接是不切实际的。然而，还有另一种可能性，即通过将所有处理器连接到一个**交换机**（switch）或**交换机网络**（switch network）。一些流行的网络设计是**交叉开关**（Cross bar）、**蝶形交换**（butterfly exchange）和**胖树**（fat tree.）。

交换机网络是由交换元件组成的，每个交换元件都有少量（最多十几个）的入站和出站链接。通过将所有的处理器连接到一些交换元件上，并有多个交换阶段，那么就有可能通过网络的路径连接任何两个处理器。

#### 2.7.6.1 交叉开关

最简单的开关网络是一个交叉开关，由 $n$ 水平线和垂直线组成，每个交叉点上都有一个开关元件，决定这些线是否连接在一起；见图 2.24。如果我们把横线指定为输入，竖线指定为输出，这显然是让 $n$ 输入映射到 $n$ 输出的一种方式。每一个输入和输出的组合（有时称为" 排列组合"）都是允许的。

这种类型的网络的一个优点是，没有任何连接可以阻挡另一个连接。主要的缺点是开关元素的数量是 $n^2$，是处理器数量 $n$ 的一个快速增长的函数。

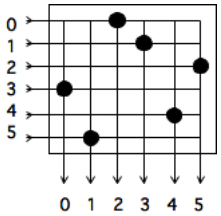



图 2.23: 连接 6 个输入和 6 个输出的简单横杆

#### 2.7.6.2 蝶形交换

蝶形交换网络是由小型交换元件构成的，它们有多个阶段：随着处理器数量的增加，阶段的数量也随之增加。图 2.25 显示了连接 2、4 和 8 个处理器的蝶形网络，每个处理器有一个本地存储器。(另外，你可以把所有的处理器放在网络的一边，而把所有的存储器放在另一边)。

正如在图 2.26 中所示，蝶形交换允许几个处理器模拟访问内存。而且，它们的访问时间是相同的，所以交换网络是实现 UMA 结构的一种方式；见 2.4.1 节。有一台基于 Butterfly 交换网络的计算机是 BBN Butterfly (http://en.wikipedia.org/wiki/BBN_Butterfly)。在 2.7.7.1 节中，我们将看到这些想法是如何在一个实际的集群中实现的。

✍ **练习 2.38** 对于简单的交叉开关和蝶形交换，随着处理器数量的增加网络需要扩展。给出两种情况下连接处理器和存储器所需的导线数量（某种单位长度）和交换元件的数量。一个数据包从存储器到处理器所需的时间，用穿越单位长度的导线和穿越开关元件的时间表示是多少？

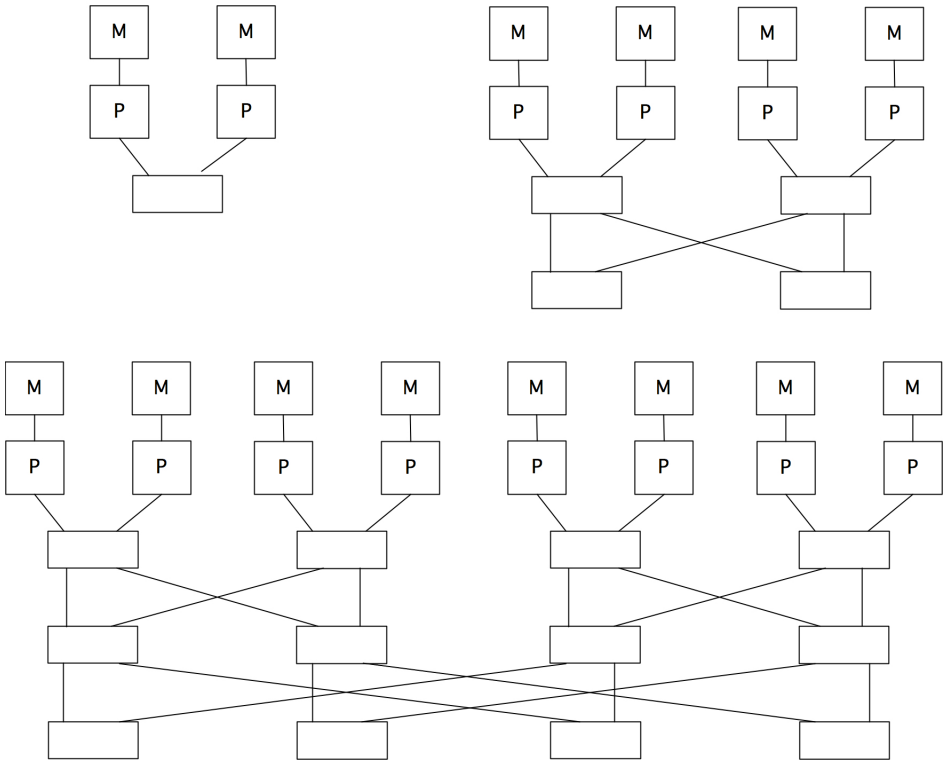



图 2.24: 2,4,8 个处理器的蝴蝶交换网络，每个处理器有一个本地存储器

通过蝶形交换网络的数据包路由是基于考虑目的地地址的位数来完成的。在第 $i$ 层，考虑第 $i$ 个数字；如果是 1，则选择开关的左出口，如果是 0，则选择右出口。这在图 2.27 中有所说明。如果我们把存储器连接到处理器上，如图 2.26 所示，我们只需要两个比特（到最后一个开关），但还需要三个比特来描述反向路线。

### 2.7.6.3 胖树

如果我们像树一样连接交换节点，那么在靠近根部的地方就会出现很大的拥堵问题，因为只有两根线连接到根注。假设我们有一棵 $k$ 级树，所以有 $2^k$ 个叶子节点。如果左边子树上的所有叶子节点都试图与右边子树上的节点通信，我们就有 $2^{k-1}$ 条信息通过一条线进入根部，同样也通过一条线出去。胖树是一个树状网络，每一级都有相同的总带宽，这样就不会出现这种拥堵问题：根部实际上会有 $2^{k-1}$ 条进线和出线连接 [88]。图 2.28 在左边显示了这种结构；右边显示了 Stampede 集群的一个机柜，机柜的上半部和下半部有一个叶子开关。

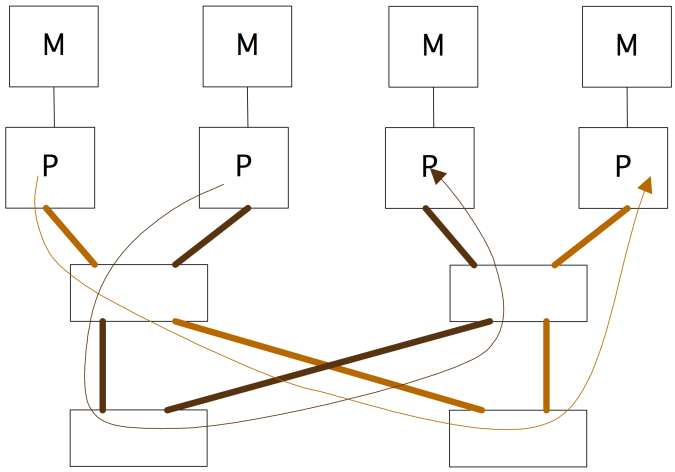



图 2.25: 通过蝴蝶交换网的两条独立路线

第一个成功的基于胖树的计算机结构是连接机 CM5。

在胖树中，就像在其他交换网络中一样，每个信息都带有自己的路由信息。由于在胖树中，选择仅限于上升一级，或者切换到当前级别的其他子树上，因此一条信息需要携带的路由信息的位数与级别相同，对于 $n$ 处理器来说是 $\log_2 n$。

✍ **练习 2.39** 证明胖子树的分叉宽度是 $P/2$，其中 $P$ 是亲子叶子节点的数量。提示：说明只有一种方法可以将胖树连接的处理器集合分割成两个连接的子集。

[142] 中对胖树的理论阐述表明，胖树在某种意义上是最优的：它可以像任何其他需要相同空间来构建的网络一样快速传递信息（最多对数因素）。这个说法的基本假设是，离根更近的开关必须连接更多的线，因此需要更多的组件，相应地也就更大。这个论点虽然在理论上很有趣，但没有实际意义，因为网络的物理尺寸在目前最大的使用胖树互连的计算机中几乎没有起到作用。例如，在德克萨斯大学奥斯汀分校的 TACC Frontera 集群中，只有 6 个核心交换机（即容纳胖树最高层的机柜），连接 91 个处理器机柜。

如上图所示，胖树的建设成本很高，因为每下一级都必须设计一个新的、更大的交换机。因此，在实践中，一个具有胖子树特征的网络是由简单的开关元件构成的；见图 2.29。这个网络的带宽和路由可能性与胖树相当。路由算法会稍微复杂一些：在胖树中，一个数据包只能以一种方式上升，但在这里，一个数据包必须知道要路由到两个较高的交换机中的哪个。

这种类型的交换网络是 Clos 网络的一种情况 [34]。

#### 2.7.6.4 超额订购和争夺

在实践中，胖树网络不使用 2 进 2 出的元件，而是使用 20 进 20 出的开关。这使得网络中的层数有可能被限制在 3 或 4 个。(顶层交换机被称为脊柱卡)。

在这种情况下，网络分析的一个额外的复杂因素是超额订购的可能性。网卡中的端口可以配置为输入或输出，而只有总数是固定的。因此，一个 40 端口的交换机可以被配置为 20 进 20 出，或者 21 进 19 出，等等。当然，如果所有连接到交换机的 21 个节点同时发送，19 个出端口将限制带宽。

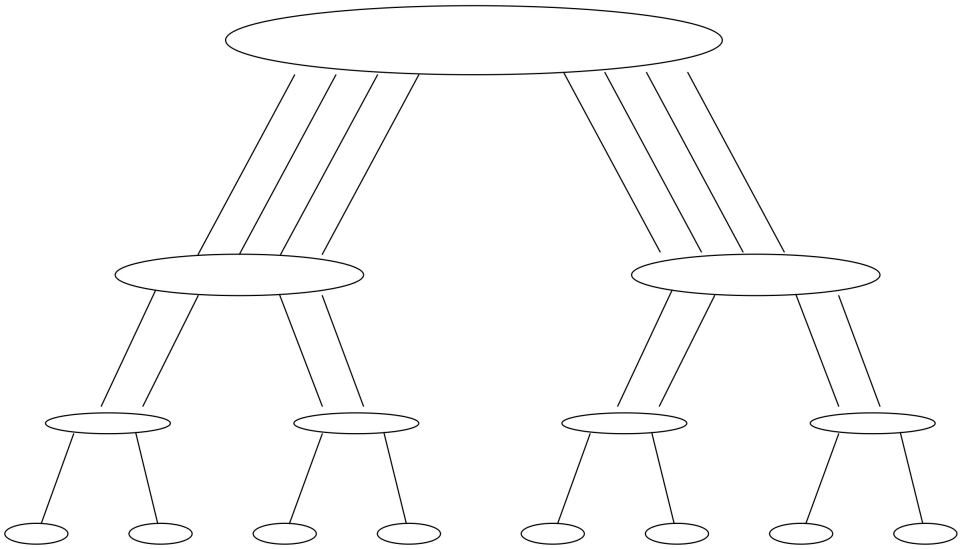


图 2.26: 通过三级蝶形交换网络的路由

还有一个问题。让我们考虑建立一个小型集群，交换机配置为有 $p$ 入端口和 $w$ 出端口，这意味着我们有 $p+w$ 端口交换机。图 2.30 描述了两个这样的开关，总共连接了 $2^p$ 个节点。如果一个节点通过交换机发送数据，它对 $w$ 条可用导线的选择由目标节点决定。这就是所谓的输出路由。

显然，我们只能期望 $w$ 个节点能够在有信息碰撞的情况下进行发送，因为这就是交换机之间可用导线的数量。然而，对于许多 $w$ 目标的选择，无论如何都会有对导线的争夺。这就是生日悖论的一个例子。

**练习 2.40** 考虑上述架构，$p$ 个节点通过切换间的 $w$ 线发送。编一个模拟代码，其中 $p$ 个节点中的 $w' \leqslant w$ 向一个随机选择的目标节点发送一个信息。作为 $w'$、$w$、$p$ 的函数，碰撞的概率是多少？找到一种方法来制表或绘制数据。

作为反馈，请给出简单情况下 $w' = 2$ 的统计分析。

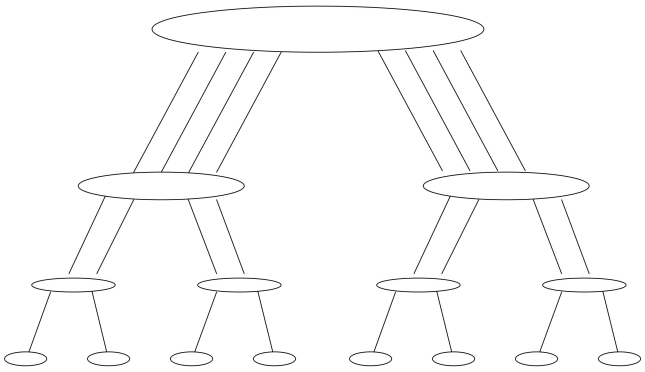


图 2.27: 具有三级互连的胖树

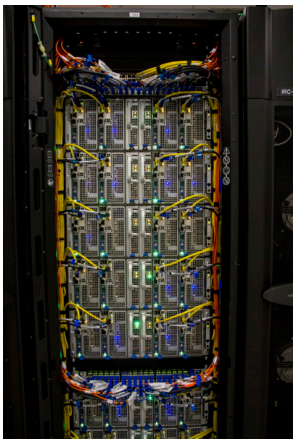


图 2.28: Stampede 集群的机柜中的叶子开关

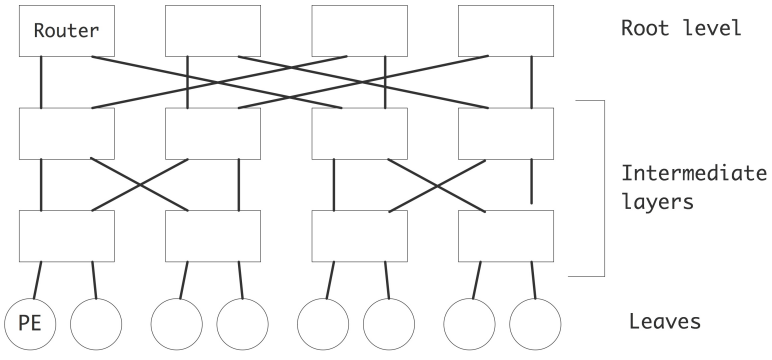



图 2.29: 由简单的开关元件构建的胖树

### 2.7.7 集群网络

上面的讨论有些抽象，但在现实生活中的集群中，你可以实际看到网络设计的体现。例如，肥大的树形集群网络会有一个中央机柜，对应树形中的最高层。图 2.31 显示了 TACC Ranger（已不再使用）和 Stampede 集群的交换机。在第二张图片中可以看出，实际上有多个冗余的胖树网络。

另一方面，像 IBM BlueGene 这样基于环状网络的集群，看起来将是一个相同机柜的集合，因为每个机柜都包含网络的一个相同部分；见图 2.32。

### 2.7.7.1 案例研究: Stampede

作为实践中联网的一个例子，让我们考虑一下德克萨斯高级计算机中心的 Stampede 集群，它是一个多根多级的胖树。

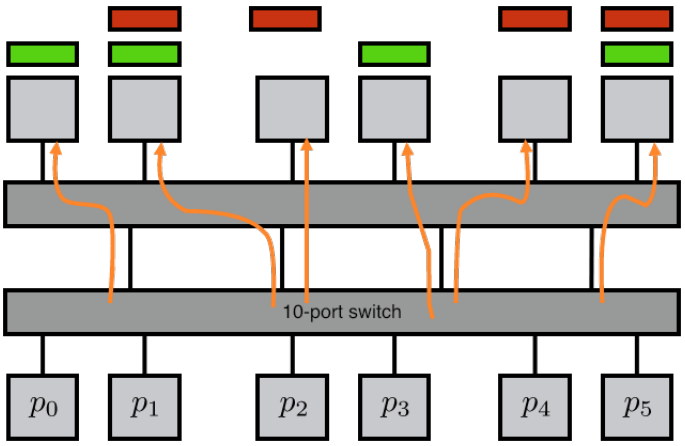



图 2.30: 超额订阅的交换机中的端口争夺

- 每个机架由 2 个机箱组成，每个机箱有 20 个节点。
- 每个机箱都有一个叶子开关，它是一个内部的横杆，使机箱中的节点之间有完美的连接性。
- 叶子交换机有 36 个端口，其中 20 个连接到节点，16 个向外。这种超额订阅意味着最多只有 16 个节点在机箱外通信时可以拥有完美的带宽。
- 有 8 个中心交换机，作为 8 个独立的胖树根发挥作用。每个机箱通过两个连接到每个中央交换机的" 叶卡"，正好占用了 16 个出站端口。
- 每个中心交换机有 18 个针卡，每个针卡有 36 个端口，每个端口连接到不同的叶卡。
- 每台中央交换机有 36 个叶卡，18 个端口连接到叶子交换机，18 个端口连接到脊柱卡。这意味着我们可以支持 648 个机箱，其中 640 个被实际使用。

网络中的一个优化是，与同一叶卡的两个连接进行通信，没有较高树级的延迟。这意味着，一个机箱中的 16 个节点和另一个机箱中的 16 个节点可以有完美的连接。

然而，对于静态路由，如 Infiniband 中使用的路由，有一个与每个目的地相关的固定端口。(目的地到端口的这种映射在每个交换机的路由表中)。因此，对于 20 个可能的目的地中的 16 个节点的某些子集，将有完美的带宽，但其他子集将看到两个目的地的流量通过同一个端口。

### 2.7.7.2 案例研究：Cray Dragonfly 网络

Cray 的蜻蜓网络是一个有趣的实际妥协。上面我们说过，一个完全连接的网络将太过昂贵，无法扩大规模。然而，如果数量保持有限的话，拥有一个完全连接的处理器集合是可能的。蜻蜓设计使用小的完全连接的组，然后将这些组组成一个完全连接的图。

这引入了一个明显的不对称性，因为一个组内的处理器拥有更大的带宽，而组与组之间则没有。然而，由于动态路由，信息可以采取非最小路径，通过其他组进行路由。这可以缓解争夺问题。






图 2.31: TACC Ranger 和 Stampede 集群的网络交换机

### 2.7.7.3 带宽和延迟

上面所说的发送信息可以被认为是一个单位时间的操作，当然是不现实的。一个大的信息比一个短的信息需要更长的时间来传输。有两个概念可以对传输过程进行更现实的描述；我们已经看到了在处理器的缓存层之间传输数据的情况。

- **延迟**在两个处理器之间建立通信需要花费大量时间，这与信息大小无关。这所花费的时间被称为信息的延时。造成这种延迟的原因有很多。
- 两个处理器进行" 握手"，以确保收件人已经准备好，并且有适当的缓冲空间来接收信息。
- 信息需要由发送方进行编码传输，并由接收方进行解码。
- 实际传输可能需要时间：并行计算机通常足够大，即使在光速下，信息的第一个字节也需要数百个周期来穿越两个处理器之间的距离。
- **带宽**在两个处理器之间的传输开始后，主要的数字是每秒可通过通道的字节数。这就是所谓的带宽。带宽通常可由信道速率（物理链路可传送比特的速率）和信道宽度（链路中物理线的数量）决定。信道宽度通常是 16 的倍数，通常为 64 或 128。这也可以表示为，一个通道可以同时发送一个或两个 8 字节的字。



图 2.32: 一台 BlueGene 计算机

带宽和延迟被正式定义为

$$T(n) = \alpha + \beta n \tag{2.9}$$

为一个 $n$ 字节的信息的传输时间。这里，$\alpha$ 是延迟，$\beta$ 是每字节的时间，也就是带宽的倒数。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；见 6.1 节。然后我们将传输时间公式扩展为

$$T(n) = \alpha + \beta n + \gamma n \tag{2.10}$$

其中 $\gamma$ 是每次操作的时间，也就是计算率的倒数。也可以将这个公式细化为

$$T(n,p) = \alpha + \beta n + \delta p \tag{2.11}$$

其中 $p$ 是所穿越的网络" **跳**（hops）"。然而，在大多数网络中，$\delta$ 的值远远低于 $\alpha$ 的值，所以我们在这里将忽略它。另外，在胖树网络中，跳数是 $\log P$ 的数量级，其中 $P$ 是处理器的总数，所以它无论如何都不可能很大。

### 2.7.8 并行计算中的局部性

在前面我们发现了关于单处理器计算中的位置性概念的讨论。并行计算中的位置性概念包括所有这些以及更多的层次。

- 核心之间：私有缓存现代处理器上的核心有私有相干缓存。这意味着我们似乎不必担心局部性问题，因为无论数据在哪个缓存中都可以访问。然而，维持一致性需要花费带宽，所以最好是保持访问的本地化。
- 内核之间：共享高速缓存内核之间共享的高速缓存是一个不需要担心位置性的地方：这是处理核心之间真正对称的内存。
- 在插槽之间：节点（或主板）上的插槽在程序员看来是共享内存的，但这实际上是 NUMA 访问，因为内存与特定的插槽相关。
- 通过网络结构：有些网络有明显的位置效应。我们在第 2.7.1 节中看到了一个简单的例子，一般来说，很明显，任何网格型网络都会有利于" 附近" 处理器之间的通信。基于胖树的网络似乎不存在这样的争论问题，

但是层次引起了不同形式的定位性。比节点上的局域性高一级，小群的节点通常由一个叶子开关连接，它可以防止数据进入中央开关。

## 2.8 多线程架构

当代 CPU 架构模式很大程度上取决于：机器从内存中获取数据要远比处理这些数据要慢得多。因此，由更快更小的存储器组成的层次结构试图通过靠近处理单元以缓解内存的长延迟和低带宽。此外，在处理单元中进行指令级并行也有助于隐藏延迟、充分利用带宽。

然而，寻找指令级并行属于编译器的工作范畴，该处可执行空间有限；另一方面，科学计算中的代码往往更加适合数据并行，这对于编译器来说较为困难但对程序员说却显而易见。能否让程序员明确地指出这种并行性？并让处理器使用它？

前面我们看到 SIMD 架构可以以明确的数据并行方式进行编程。如果我们有大量的数据并行，但没有那么多的处理单元该怎么办？在这种情况下，我们可以把指令并行变成线程并行，让多个线程在每个处理单元上执行。每当一个线程因为未完成的内存请求而停滞不前时，处理器可以切换到另一个线程，因为所有必要的输入都是可用的。这就是所谓的**多线程**（multi-threading）。虽然这听起来像是一种防止处理器等待内存的方法，但也可以被看作是一种保持内存最大限度被占用的方法。

✍ **练习 2.41** 把内存的长延迟和有限的带宽看作是两个独立的问题，多线程是否同时解决了这两个问题？

这里的问题是，大多数 CPU 并不擅长在线程之间快速切换。上下文切换（在一个线程和另一个线程之间切换）需要大量的周期，与等待主内存的数据相当。在所谓的**多线程架构**（Multi-Threaded Architecture，MTA）中，上下文切换是非常有效的，有时只需要一个周期，这使得一个处理器可以同时在许多线程上工作。

多线程的概念在 Tera Computer MTA 机器中得到了探索，该机器演变成了目前的 Cray XMT9。

MTA 的另一个例子是 GPU，其中处理器作为 SIMD 单元工作，同时本身也是多线程的。

## 2.9 GPU 与协处理器

当前，CPU 在处理各种计算中都参与了一定程度的作用，也就是说，如果限制处理器处理的功能，有可能提高其专注效率，或降低其功耗。因此我们试图在主机进程中加入一个辅助处理器如，英特尔的 8086 芯片，为第一代的 IBM PC 提供动力；可以添加一个数字协处理器，即 80287，这种处理器在超越函数方面非常有效，而且它还采用了 SIMD 技术；使用独立的图形功能也很流行，导致了 X86 处理器的 SSE 指令，以及独立的 GPU 单元被连接到 PCI-X 总线。

进一步的例子是使用数字信号处理（DSP）指令的协处理器，以及可以重新配置以适应特定需求的 FPGA 板。早期的阵列处理器，如 ICL DAP 也是协处理器。

在本节中，我们将简要介绍这一理念的一些现代化身，特别是 GPU。

### 2.9.1 追溯历史

协处理器可以用两种不同的方式进行编程：有时它是无缝集成的，某些指令在协处理器中自动执行，而不是在" 主" 处理器上执行。另一方面，也有可能需要明确调用协处理器功能，甚至有可能将协处理器功能与主机功能重叠。从效率的角度看，后一种情况可能听起来很有吸引力，但它提出了一个严重的编程问题。程序员现在需要确定两个工作流：一个用于主机处理器，一个用于协处理器。

采用协处理器的并行机器有：

- 英特尔 Paragon（1993）每个节点有两个处理器，一个用于通信，另一个用于计算。这些处理器实际上是相同的，即英特尔 i860 英特尔 i860 处理器。在后来的修订中，有可能将数据和函数指针传递给通信处理器。

- 洛斯阿拉莫斯的 IBM Roadrunner 是第一台达到 PetaFlop 的机器。(葡萄计算机更早达到了这一点，但那是一台用于分子动力学计算的特殊用途机器)。它通过使用 Cell 协处理器达到了这个速度。顺便说一下，Cell 处理器实质上是索尼 Playstation3 的引擎，再次显示了超级计算机的商品化。
- 中国的" 天河一号" 在 2010 年登上了 Top500 榜单，由于使用了 NVIDIA GPU，达到了约 2.5PetaFlop。
- 天河二号和 TACC Stampede 集群使用英特尔 Xeon Phi 协处理器。

Roadrunner 和 Tianhe-1A 是协处理器的例子，它们非常强大，需要独立于主机 CPU 进行明确编程。例如，在天河-1A 的 GPU 上运行的代码是用 CUDA 编程并单独编译的。

在这两种情况下，由于协处理器不能直接通过网络交流，可编程性问题进一步加剧。要把数据从一个协处理器发送到另一个协处理器，必须先传到一个主处理器，再从那里通过网络传到另一个主处理器，然后才移到目标协处理器。

### 2.9.2 瓶颈问题

协处理器通常有自己的内存，英特尔 Xeon Phi 可以独立运行程序，但更多时候存在如何访问主处理器内存的问题。一个流行的解决方案是通过 PCI 总线连接协处理器。这样访问主机内存的速度比主机处理器的直接连接要慢。例如，Intel Xeon Phi 的带宽为 512 位宽，每秒 5.5GT（我们将在第二部分讨论这个"GT"），而它与主机内存的连接为 5.0GT/s，但只有 16 位宽。

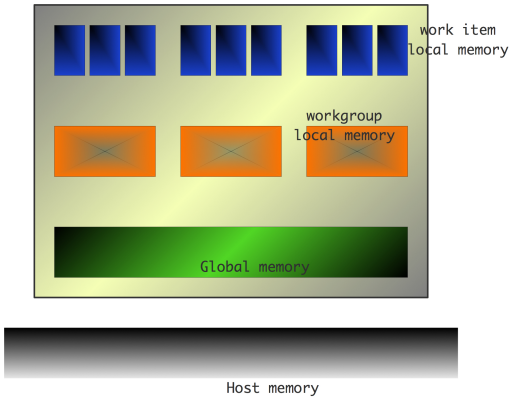
**GT 测量**我们习惯于看到以千兆位/秒为单位的带宽。对于 PCI 总线，人们经常看到 GT 测量。这代表了千兆传输，它衡量了总线在零和一之间改变状态的速度。通常情况下，每个状态转换都对应一个比特，但总线必须提供自己的时钟信息，如果你发送一个相同的比特流，时钟会被混淆。因此，为了防止这种情况，通常将每 8 位编码为 10 位。然而，这意味着有效带宽比理论数字要低，在这种情况下是 4/5 的系数。

由于制造商喜欢讨论事情的积极一面，因此他们报告的数字会更高。

### 2.9.3 GPU 计算

**图形处理单元**（Graphics Processing Unit，GPU），有时也称为**通用图形处理单元**（General Purpose Graphics Processing Unit，GPGPU），是一种特殊用途的处理器，是为快速图形处理而设计的。然而，由于为图形所做的操作是一种算术形式，GPU 已经逐渐发展出对非图形计算也很有用的设计。GPU 的一般设计是由" 图形流水线" 激发的：在数据并行的形式下，对许多数据元素进行相同的操作，并且许多这样的数据并行块可以在同一时间激活。

CPU 的基本限制也适用于 GPU：对内存的访问会产生很长的延迟。在 CPU 中解决这个问题的方法是引入各级缓存；在 GPU 中则采取不同的方法。GPU 关注的是吞吐量计算，以高平均速率提供大量数据，而不是尽可能快地提供任何单一结果。这是通过支持许多线程 = 并在它们之间快速切换而实现的。当一个线程在等待内存中的数据时，另一个已经拥有数据的线程可以继续进行计算。



### 2.9.3.1 用内核进行 SIMD 型编程

当前的 GPU 的一个架构中结合了 SIMD 和 SPMD 并行性。线程并非完全独立，而是在线程块中排列，所有的线程块中执行相同的指令以实现 SIMD。也有可能将同一指令流（CUDA 术语中的" 内核"）安排在一个以上的线程块上。在这种情况下，线程块可以不同步，这有点类似于 SPMD 上下文中的进程。然而，由于我们在这里处理的是线程，而不是进程，所以使用了**单指令多线程**（Single Instruction Multiple Thread，SIMT）这一术语。

这种软件设计在硬件中很明显；例如，NVidia GPU 有 16-30 个流式多处理器（SMs），一个 SMs 由 8 个流式处理器（SPs）组成，对应于处理器内核；见图 2.34。SPs 以真正的 SIMD 方式行动。GPU 中的内核数量通常比传统的多核处理器要多，但内核的数量却更加有限。因此，这里使用了多核这个术语。

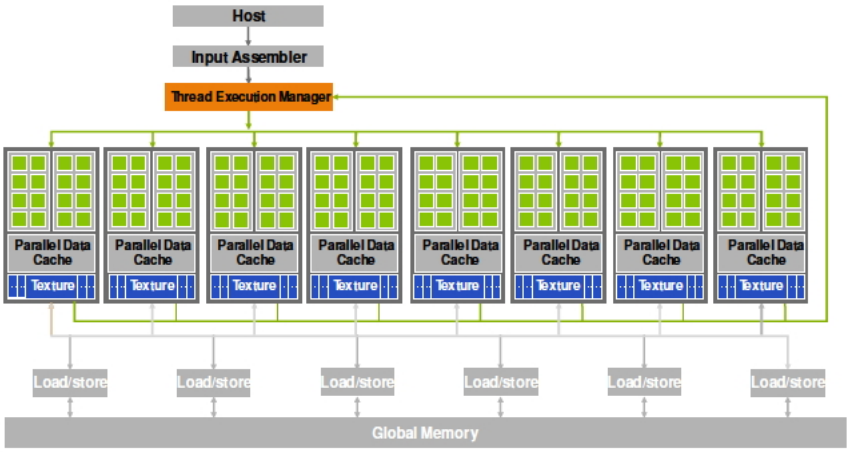



图 2.33: GPU 示意图

GPU 的 SIMD（即数据并行）性质在 CUDA 启动进程的方式中变得很明显。内核，即一个将在 GPU 上执行的函数，在 *mn* 核上启动。

```
KernelProc<< m,n >>(args)
```

执行内核的 mn 内核的集合被称为**网格**（grid），它的结构为 $m$ 线程块，每个线程块有 $n$ 线程。一个线程块最多可以有 512 个线程。

回顾一下，线程共享一个地址空间（见第 2.6.1 节），所以它们需要一种方法来识别每个线程将对哪一部分数据进行操作。为此，线程中的区块用 $x, y, z$ 坐标编号，而区块中的线程用 $x, y, z$ 坐标编号。每个线程都知道自己在块中的坐标，以及其块在网格中的坐标。

我们用一个向量加法的例子来说明这一点。

```
// 每个线程执行一次加法
__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() {
    // 运行N/256块的网格，每块256个线程
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

这显示了 GPU 的 SIMD 性质：每个线程都在执行相同的标量程序，只是在不同的数据上执行。

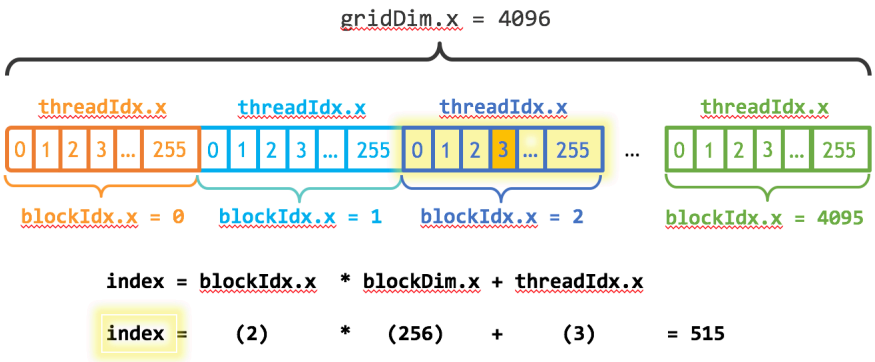



图 2.34: CUDA 中的线程索引

线程块中的线程是真正的数据并行：如果有一个条件，使一些线程走真分支，其他线程走假分支，那么一个分支将首先被执行，另一个分支的所有线程都被停止。随后，而不是同时，另一个分支上的线程将执行他们的代码。这可能会引起严重的性能损失。

GPU 依赖于大量的数据并行性和快速上下文切换的能力。这意味着它们将在有大量数据并行的图形和科学应用中茁壮成长。然而，它们不太可能在" 商业应用" 和操作系统中表现良好，因为那里的并行性是指令级并行性（ILP）类型，通常是有限的。

### 2.9.3.2 GPU 与 CPU 的对比

这些是 GPU 和普通 CPU 之间的一些区别。

- 截至 2010 年底，GPU 是附加的处理器，例如通过 PCI-X 总线，所以它们操作的任何数据都必须从 CPU 传输。由于这种传输的内存带宽很低，至少比 GPU 的内存带宽低 10 倍，因此必须在 GPU 上做足够的工作来克服这种开销。
- 由于 GPU 是图形处理器，所以它强调的是单精度浮点运算的 metic。为了适应科学计算界，对双精度的支持正在增加，但双精度速度通常是单精度翻转率的一半。这种差异可能会在未来几代中得到解决。
- CPU 被优化为处理单一的指令流，这些指令可能具有很强的异质性；而 GPU 是明确为数据并行化而制造的，在传统代码上表现很差。
- CPU 是为了处理一个线程，或最多是少量的线程。GPU 需要大量的线程，远远大于计算核心的数量，才能有效地执行。

### 2.9.3.3 GPU 的预期收益

GPU 在实现高性能、高成本效益方面已经迅速获得了声誉。关于用最小的努力将代码移植到 CUDA 上的故事比比皆是，由此带来的速度提升有时达到 400 倍。GPU 真的是如此神奇的机器吗？原有的代码是否编程不当？如果 GPU 这么厉害，为什么我们不把它用于所有的事情呢？

事实有几个方面。

首先，GPU 并不像普通 CPU 那样具有通用性。GPU 非常擅长做数据并行计算，而 CUDA 则擅长优雅地表达这种细粒度的并行性。换句话说，GPU 适用于某种类型的计算，而对许多其他类型的计算则不适合。

相反，普通的 CPU 不一定擅长数据并行化。除非代码写得非常仔细，否则性能会从最佳状态下降，大约有以下几个因素：

- 除非使用明确的并行结构指令，否则编译后的代码将永远使用可用内核中的一个，例如 4 个。
- 如果指令没有流水线，那么浮点流水线的延迟又会增加 4 个系数。
- 如果内核有独立的加法和乘法管线，如果不同时使用，又会增加 2 个因素。
- 如果不使用 SIMD 寄存器，就会在峰值性能方面增加更多的减慢。

- 编写计算内核的最佳 CPU 实现往往需要使用汇编程序，而直接的 CUDA 代码将以相对较少的努力实现高性能，当然，前提是计算有足够的数据并行性。

### 2.9.4 英特尔 Xeon Phi

英特尔 Xeon Phi，也被其架构设计称为多集成核心（MIC），是一种专门为数值计算设计的设计。最初的设计，即 Knight's Corner 是一个协处理器，而第二次迭代，即 Knight's Landing 是自带主机的。

作为一个协处理器，Xeon Phi 与 GPU 既有区别又有相似之处。

## 2.10 负载均衡

在本章的大部分内容中，我们都假设的是一个问题可以被完美分配到各个处理器上，即一个处理器总是在进行有效的工作且只会因为通信延迟而空闲。然而在实际中，处理器的空闲可能因为其正在等待消息，而发送处理器甚至还没有达到其代码中的发送指令。这种情况下，一个处理器在工作，另一个处理器在闲置，被描述为**负载不均衡**（load unbalance）：一个处理器闲置没有内在的原因，如果我们以不同的方式分配工作负载，它本来可以工作。

在处理器有太多的工作和没有足够的工作之间存在着不对称性：有一个处理器提前完成任务，比有一个超负荷的处理器使所有其他处理器都在等待它要好。

✍ **练习 2.42** 将这个概念精确化。假设一个并行任务在所有处理器上都需要时间 1，但只有一个处理器。

- 假设 $0 < \alpha < 1$，而一个并行任务花费的时间是 $1 + \alpha$，那么速度提升和效率与处理器数量的关系是什么？在 Amdahl 和 Gustafsson 的意义上考虑这个问题。
- 如果一个处理器需要时间 $1 - \alpha$，请回答同样的问题。

负载均衡的代价往往是昂贵的，因为它需要移动大量的数据。例如，后面将有一个分析表明，在稀疏矩阵与向量乘积过程中的数据交换比存储在处理器上的数据要低一阶。然而，我们不会去研究移动的实际成本：我们在这里主要关注的是均衡工作均衡，以及保留原始负载分布中的任何位置性。

### 2.10.1 负载均衡与数据分配

工作和数据之间存在着双重性：在许多应用中，数据的分布意味着工作的分布，反之亦然。如果一个应用程序更新一个大的数组，数组的每个元素通常" 生活" 在一个唯一确定的处理器上，该处理器负责该元素的所有更新。这种策略被称为**拥有者计算**（owner computes）。

因此，数据和工作之间存在着直接的关系，相应地，数据分配和负载均衡也是相辅相成的。例如，在第 6.2 节中，我们将谈论数据分布如何影响效率，但这立即转化为对负载分布的关注。

- 负载需要被均匀地分配。这通常可以通过均匀地分配数据来实现，但有时这种关系并不是线性的。
- 任务需要被放置，以尽量减少它们之间的流量。在矩阵-向量乘法的情况下，这意味着二维分布要优于一维分布；关于空间填充曲线的讨论也是类似的动机。

将剩余的 $r = N - pN/p$ 分摊到 $r$ 处理器上比一个更好。这可以通过给第一个或最后一个 r 处理器提供一个额外的数据点来实现。这可以通过给进程 $p$ 分配范围来实现

$$[p \times \lfloor (N + p - 1)/p \rfloor, (p + 1) \times \lfloor (N + p - 1)/p \rfloor) \tag{2.12}$$

虽然这个方案是很均衡的，但例如计算一个给定的点属于哪个处理器是很棘手的。下面的方案使这种计算更容易：让 $f(i) = iN/p$，那么处理器 i 得到的点 $f(i)$ 直到 $f(i + 1)$。

✍ **练习 2.43** 证明 $N/p \leqslant f(i+1) - f(i) \leqslant N/p$。

根据这个方案，拥有索引 i 的处理器是 $(p(i+1)-1)/N$。

### 2.10.2 负载调度

有些情况下，负载可以比较自由的进行分配，可以通过负载调度实现负载均衡。例如在共享内存的背景下，所有处理器都可以访问所有的数据。在这种情况下，我们可以考虑使用预先确定的工作分配给处理器的**静态调度**（static scheduling），或在执行期间确定分配的**动态调度**（dynamic scheduling）之间的区别。

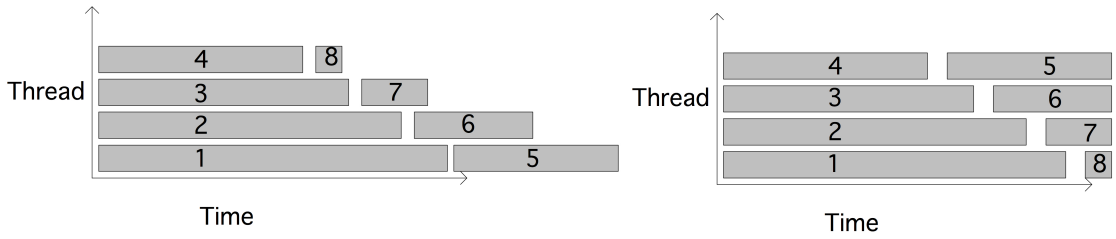



图 2.35: 静态或轮回（左）与动态（右）线程调度；已标明任务编号

为了说明动态调度的优点，考虑在 4 个线程上调度 8 个运行时间递减的任务（图 2.36）。在静态调度中，第一个线程得到任务 1 和 4，第二个线程得到 2 和 5，依此类推。在动态调度中，任何完成其任务的线程都会得到下一个任务。在这个特定的例子中，这显然给出了一个更好的运行时间。另一方面，动态调度可能会有更高的开销。

### 2.10.3 独立任务的负载均衡

在其他情况下，工作负荷不是由数据直接决定的。如果有一个待完成的工作池，而每个工作项目的处理时间不容易从其描述中计算出来，就会发生这种情况。在这种情况下，我们可能希望在给流程分配工作时有一些灵活性。

让我们首先考虑这样一种情况：一项工作可以被划分为不相通的独立任务。一个例子是计算 Mandelbrot 集图片的像素，其中每个像素都是根据一个不依赖于周围像素的数学函数来设置的。如果我们能够预测绘制图片的任意部分所需的时间，我们就可以对工作进行完美的划分，并将其分配给处理器。这就是所谓的**静态负载均衡**（static load balancing）。

更现实的是，我们无法完美地预测工作的某一部分的运行时间，于是我们采用了**过度分解**（overdecomposition）工作的方法：我们将工作分成比处理器数量更多的任务。然后，这些任务被分配到一个**工作池**（work pool）中，每当处理器完成一项工作，就从工作池中抽取下一项工作。这就是所谓的**动态负载均衡**（dynamic load balancing）。许多图形和组合问题都可以用这种方式来解决。

有结果表明，随机分配任务到处理器在统计学上接近于最优 [122]，但这忽略了科学计算中的任务通常是频繁交流的方面。

✍ **练习 2.44** 假设你有任务 $\{T_i\}_{i=1,\ldots,N}$，运行时间为 $t_i$，处理器数量不限。查阅 2.2.4 节中的 Brent 定理，并从中推导出任务的最快执行方案可以被描述为：有一个处理器只执行具有最大 $t_i$ 值的任务。(这个练习受到了 [170] 的启发)。

接下来，让我们考虑一个并行的工作，其中各部分都有通信。在这种情况下，我们需要平衡标量工作负载和通信。

一个并行计算可以被表述为一个图（见附录 18 的图论介绍），其中处理器是顶点，如果两个顶点的处理器需要在某个点上进行通信，那么这两个顶点之间就有一条边。这样的图通常是由被解决的问题的基本图衍生出

来的。作为一个例子，考虑矩阵-向量乘积 $y = Ax$，其中 $A$ 是一个稀疏的矩阵，详细地看一下正在计算 $y_i$ 的处理器，对于一些 $i$。$y_i \leftarrow y_i + A_{ij}x_j$ 意味着这个处理器将需要 $x_j$ 的值，所以，如果这个变量在不同的处理器上，它需要被送过去。

我们可以将其标准化。让向量 $x$ 和 $y$ 不相连地分布在处理器上，并唯一地定义 $P(i)$ 为拥有索引 $i$ 的处理器。如果存在一个非零的元素 $a_{ij}$，且 $P = P(i)$，$Q = P(j)$，那么就有一条边 $(P, Q)$。在结构对称矩阵的情况下，这个图是无定向的，即 $a_{ij} \neq 0 \Leftrightarrow a_{ij} \neq 0$。

指数在处理器上的分布现在给了我们顶点和边的权重：一个处理器有一个顶点权重，即它所拥有的指数数量；一条边 $P, Q$ 有一个权重，即需要从 $Q$ 发送到 P 的向量成分的数量，如上所述。

现在可以将负载平衡问题表述如下。

找到一个分区 $\mathbb{P} = \cup_i \mathbb{P}_i$，这样顶点权重的变化最小，同时边缘权重也尽可能的低。

顶点权重的变化最小化意味着所有处理器的工作量大致相同。保持边缘权重低意味着通信量低。这两个目标不需要同时满足：可能会有一些折中。

**练习 2.45** 考虑极限情况，即处理器的速度是无限的，处理器之间的带宽也是无限的。剩下的决定运行时间的唯一因素是什么？你现在需要解决什么图形问题来找到最佳的负载均衡？稀疏矩阵的什么属性给出了最坏情况下的行为？

一个有趣的负载均衡方法来自谱图理论（第 18.6 节）：如果 $A_G$ 是无向图的邻接矩阵，$D_G - A_G$ 是**拉普拉斯矩阵**（graph Laplacian），那么通往最小特征值 0 的特征向量 $u_1$ 是正的，而通往下一个特征值的特征向量 $u_2$ 是与它正交。因此，$u_2$ 必须有交替符号的元素；进一步分析表明，有正符号的元素和负符号的元素是相连的。这就导致了图形的自然分割。

### 2.10.4 负载重分配

在某些应用中，最初的载荷分布是明确的，但后来需要调整。一个典型的例子是在有限元方法（FEM）代码中，载荷可以通过物理域的划分来分配。如果后来领域的离散化发生了变化，负载就必须重新或重新分配。在接下来的小节中，我们将看到旨在保持局部性的负载均衡和再均衡的技术。

#### 2.10.4.1 扩散负载均衡

在许多实际情况下，我们可以将处理器图与我们的问题联系起来：任何一对进程之间都有一个顶点，通过点对点通信直接互动。因此，在负载均衡中使用这个图似乎是一个很自然的想法，只把负载从一个处理器转移到图中的邻居那里。这就是扩散式负载均衡的想法 [37, 112]。虽然该图在本质上不是有向的，但为了负载均衡，我们在边上放上任意的方向。负载均衡的描述如下。设 $l$ 是进程 $i$ 的负载，$\tau(j)$ 是边 $j \to i$ 上负载的转移。那么

$$\ell_i \leftarrow \ell_i + \sum_{j \to i} \tau_i^{(j)} - \sum_{i \to j} \tau_j^{(i)} \tag{2.13}$$

虽然我们只是用了一个 $i, j$ 的边数，但在实践中，我们把边数线性化了。然后我们得到一个系统

$$AT = \bar{L} \tag{2.14}$$

其中，

- $A$ 是一个大小为 $|N| \times |E|$ 的矩阵，描述了连接到方节点的各条边，其元素值等于 $\pm 1$，取决于
- $T$ 是转移的向量，大小为 $|E|$；和
- $\bar{L}$ 是负荷偏差向量，表明每个节点比平均负荷高出/低出多少。

在线性处理器阵列的情况下，这个矩阵是欠确定的，边比处理器少，但在大多数情况下，系统将是超确定的，边

比进程多。因此，我们要解决

$$T = \left(A^t A\right)^{-1} A^t \bar{L} \quad \text{or} T = A^t \left(AA^t\right)^{-1} \bar{L} \tag{2.15}$$

由于 $A^t A$ 和 $AA^t$ 是非正定的，我们可以通过放松来解决大约，只需要局部知识。当然，这种松弛的收敛速度很慢，全局性的方法，如 Conjugate Gradients (CG)，会更快 [112].

### 2.10.4.2 用空间填充曲线实现负载均衡

在前面的章节中，我们考虑了负载均衡的两个方面：确保所有的处理器都有大致相等的工作量，以及让分布反映问题的结构，以便将通信控制在合理范围内。我们可以这样表述第二点，当分布在并行机器上时，试图保持问题的局部性：空间中靠近的点很可能会发生交互，所以它们应该在同一个处理器上，或者至少是一个不太远的处理器。

努力保持位置性显然不是正确的策略。在 BSP 中，有一个统计学上的论点，即随机放置将提供一个良好的负载均衡以及通信均衡。

**练习 2.46** 考虑将进程分配给处理器，问题的结构是每个进程只与最近的邻居通信，并让处理器在一个二维网格中排序。如果我们对进程网格进行明显的分配，就不会有争执。现在写一个程序，将进程分配给随机的处理器，并评估会有多少争用。

在上一节中，你看到了图划分技术是如何帮助实现第二点，即保持问题的局部性。在本节中，你将看到一种不同的技术，它对初始负载分配和后续的负载再均衡都有吸引力。在后一种情况下，一个处理器的工作可能会增加或减少，需要将一些负载转移到不同的处理器。

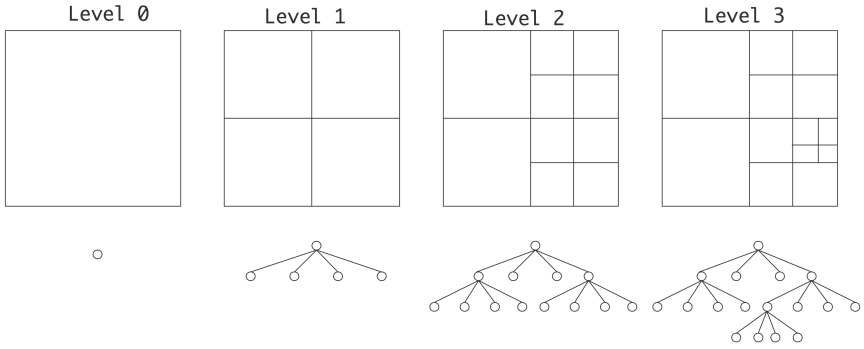



图 2.36: 在后续层次中对一个领域进行自适应细化

例如，有些问题是自适应细化的。这在图 2.37 中得到了说明。如果我们跟踪这些细化水平，问题就会得到一个树状结构，其中的叶子包含所有的工作。负载均衡变成了在处理器上划分树叶的问题；图 2.38。现在我们观察到，这个问题有一定的局部性：任何非叶子节点的子树在物理上都很接近，所以它们之间可能会有通信。

- 可能会有更多的子域出现在处理器上；为了尽量减少处理器之间的通信，我们希望每个处理器都包含一个简单连接的子域组。此外，我们希望每个处理器所覆盖的域的一部分是" 紧凑" 的，即它具有低长宽比和低表面体积比。
- 当一个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。这个负载重新分配的过程应该保持位置性。

为了满足这些要求，我们使用空间填充曲线（SFC）。负载均衡树的空间填充曲线（SFC）如图 2.39 所示。我们不会对 SFC 进行正式的讨论；相反，我们将让图 2.40 代表一个定义：SFC 是一个递归定义的曲线，每个子域都接触一次

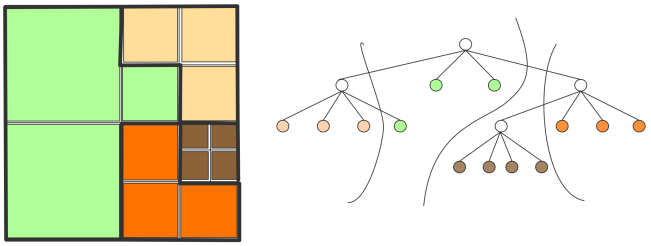


图 2.37: 自适应细化域的载荷分布

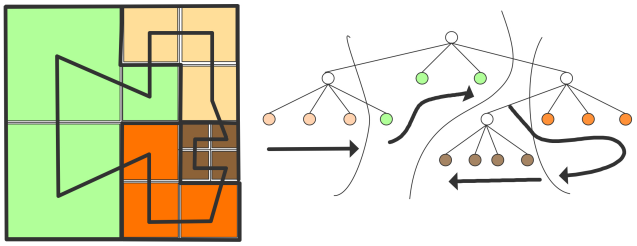


图 2.38: 负载平衡树的空间填充曲线

SFC 的特性是，在物理上相近的领域元素在曲线上也会相近，所以如果我们将 SFC 映射到处理器的线性排序上，我们将保留问题的局部性。

更重要的是，如果领域再细化一个层次，我们就可以相应地细化曲线。然后，负载可以被重新分配到曲线上的相邻处理器上，而我们仍然会保留位置性。

(空间填充曲线（SFCs）在 N 体问题中的使用在 [198] 和 [187] 中讨论过)

## 2.11 其他话题

### 2.11.1 分布式计算、网格计算、云计算

在本节中，我们将对云计算等术语以及早先的一个术语分布式计算进行简短的了解。这些都是与科学意义上的并行计算有关系的概念，但它们在某些基本方面是不同的。

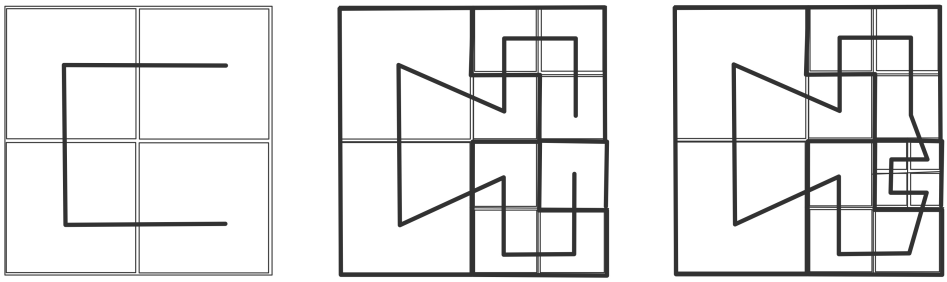


图 2.39: 空间填充曲线，定期和不定期的细化

**定义 2.7 (远程过程调用与存储区域网络)**

分布式计算可以追溯到来自大型数据库服务器，如航空公司的预订系统，它必须被许多旅行社同时访问。对于足够大的数据库访问量，单台服务器是不够的，因此发明了**远程过程调用**（remote procedure call）的机制，中央服务器将调用不同（远程）机器上的代码（有关的过程）。远程调用可能涉及数据的传输，数据可能已经在远程机器上，或者有一些机制使两台机器的数据保持同步。这就产生了**存储区域网络**（Storage Area Network，SAN）。比分布式数据库系统晚了一代，网络服务器不得不处理同样的问题，即许多人同时访问必须表现得像一个单一的服务器。

我们已经看到了分布式计算和高性能并行计算之间的一个巨大区别。科学计算需要并行性，因为单一的模拟对一台机器来说变得太大或者太慢；上面描述的商业应用涉及许多用户针对一个大数据集执行小程序（即数据库或网络查询）。对于科学需要，并行机器的处理器（集群中的节点）必须有一个非常快的连接；对于商业需要，只要中央数据集保持一致，就不需要这样的网络。

在高性能计算和商业计算中，服务器都必须保持可用和运行，但在分布式计算中，在如何实现这一点上有相当大的自由度。对于一个连接到数据库等服务的用户来说，由哪个实际的服务器来执行他们的请求并不重要。因此，分布式计算可以利用虚拟化：一个虚拟服务器可以在任何硬件上生成。

可以在远程服务器和电网之间做一个类比，前者在需要的地方提供计算能力，后者在需要的地方提供电力。这导致了网格计算或实用计算的出现，美国国家科学基金会拥有的 Teragrid 就是一个例子。网格计算最初是作为一种连接计算机的方式，通过**局域网**（Local Area Network，LAN）或**广域网**（Wide Area Network，WAN），通常是互联网连接起来。这些机器本身可以是平行的，而且通常由不同的机构拥有。最近，它被视为一种通过网络共享资源的方式，包括数据集、软件资源和科学仪器。

实用计算作为一种提供服务的方式的概念，我们从上述分布式计算的描述中认识到，随着谷歌的搜索引擎成为主流，它为整个互联网建立了索引。另一个例子是安卓手机的 GPS 功能，它结合了地理信息系统、GPS 和混搭数据。Google 的收集和处理数据的计算模型已经在 MapReduce[40] 中正式化。它结合了数据并行方面（" 地图" 部分）和中央积累部分（" 规约"）。两者都不涉及科学计算中常见的紧密耦合的邻居间通信。一个用于 MapReduce 计算的开源框架为 Hadoop[95]。亚马逊提供了一个商业的 Hadoop 服务。

即使不涉及大型数据集，由远程计算机为用户需求服务的概念也很有吸引力，因为它免除了用户在其本地机器上维护软件的需要。因此，Google Docs 提供了各种" 办公" 应用程序，用户无需实际安装任何软件。这种想法有时被称为软件即服务（SaS），用户连接到一个" 应用服务器"，并通过一个客户端（如网络浏览器）访问它。在谷歌文档的情况下，不再有一个大型的中央数据集，而是每个用户与他们自己的数据互动，这些数据在谷歌的服务器上维护。这当然有一个很大的好处，那就是用户可以从任何可以使用网络浏览器的地方获得数据。

SaS 的概念与早期技术有一些联系。例如，在大型机和工作站时代之后，所谓的瘦客户机想法曾短暂流行。在这里，用户将拥有一个工作站而不是一个终端，但却可以在一个中央服务器上存储的数据上工作。沿着这种思路的一个产品是 Sun 公司的 Sun Ray（大约在 1999 年），用户依靠一张智能卡在一个任意的、无状态的工作站上建立他们的本地环境。

#### 2.11.1.1 使用场景

按需提供服务的模式对企业很有吸引力，这些企业越来越多地使用云服务。它的优点是不需要最初的货币和时间投资，也不需要对设备的类型和大小做出决定。目前，云服务主要集中在数据库和办公应用上，但具有高性能互连的科学云正在开发中。

以下是对云资源使用场景的大致分类。

- **扩展**。在这里，云资源被用作一个平台，可以根据用户需求进行扩展。这可以被认为是平台即服务（PaS）：云提供软件和开发平台，免除了用户的管理和维护。我们可以区分两种情况：如果用户正在运行单个作业并积极等待。如果用户正在运行单个作业并积极等待输出，可以增加资源以减少这些作业的等待时间（能力测试）。另一方面，如果用户正在向一个队列提交作业，并且任何特定作业的完成时间并不重要（能力组合），资源可以随着队列的增长而增加。在 HPC 应用中，用户可以将云资源视为一个集群；这属于基础设施即服务（IaS）：云服务是一个计算平台，允许在操作系统层面进行定制。多租户。在这里，同一个软件被提供给多个用户，让每个人都有机会进行个性化定制。这属于软件即服务（SaS）：软件按需提供；客户不购买软件，只为其使用付费。
- **批量处理**。这是上述扩展方案之一的有限版本：用户有大量的数据需要以批处理模式进行处理。然后，云就成为一个批处理者。这种模式是 MapReduce 计算的良好候选者。
- **存储**。大多数云供应商都提供数据库服务，这些模式都是为了让用户不需要维护自己的数据库，就像缩放和批量处理模式让用户不需要担心维护集群硬件一样。

- **同步化**。这种模式在商业用户应用中很受欢迎。Netflix 和亚马逊的 Kindle 允许用户消费在线内容（分别是流媒体电影和电子书）；暂停内容后，他们可以从任何其他平台恢复。苹果公司最近的 iCloud 为办公应用中的数据提供了同步，但与 Google Docs 不同的是，这些应用不是" 在云中"，而是在用户机器上。

第一个可以公开访问的云是亚马逊的弹性计算云（EC2），于 2006 年推出。EC2 提供各种不同的计算平台和存储设施。现在有一百多家公司提供基于云的服务，远远超出了最初的计算机出租的概念。

从计算机科学的角度来看，云计算的基础设施可能是有趣的，涉及到分布式文件系统、调度、虚拟化和确保高可靠性的机制。

一个有趣的项目，结合了网格和云计算的各个方面，是加拿大天文研究高级网络 [179]。在这里，大量的中央数据集被提供给天文学家，就像在一个网格中一样，同时还有计算资源，以类似云的方式对其进行分析。有趣的是，云资源甚至采取了用户可配置的虚拟集群的形式。

### 2.11.1.2 角色定位

综上所述，我们有三种云计算服务模式。

**定义 2.8 (云计算服务模式)**

- **软件即服务**：消费者运行供应商的应用程序，通常通过浏览器等客户端；消费者不安装或管理软件。谷歌文档就是一个很好的例子。
- **平台即服务**：向消费者提供的服务是运行由消费者开发的应用程序的能力，消费者不管理所涉及的处理平台或数据存储。
- **基础设施即服务**：供应商向消费者提供运行软件的能力，并管理存储和网络。消费者可以负责操作系统的选择和网络组件，如防火墙。

♣

这些可以按以下方式部署。

- 私有云：云基础设施由一个组织管理，供其独家使用。
- 公共云：云基础设施由广大客户群管理使用。我们还可以定义混合模式，如社区云。

**定义 2.9 (云计算特点)**

- **按需和自我服务**：消费者可以快速请求服务和改变服务水平，而不需要与提供者进行人工互动。
- **快速弹性**：在消费者看来，存储或计算能力的数量是无限的，只受预算的限制。请求额外的设施是快速的，在某些情况下是自动的。
- **资源池**：虚拟化机制使云看起来像一个单一的实体，而不考虑其底层基础设施。在某些情况下，云会记住用户访问的" 状态"；例如，亚马逊的 Kindle 书籍允许人们在个人电脑和智能手机上阅读同一本书；云存储的书籍" 记住" 读者离开的地方，而不管平台如何。
- **网络访问**：云可以通过各种网络机制使用，从网络浏览器到专用门户。
- **测量服务**：云服务通常是" 计量" 的，消费者为计算时间、存储和带宽付费。

♣

## 2.11.2 能力与容量计算

大型并行计算机可以以两种不同的方式使用。在后面的章节中，你将看到科学问题是如何几乎可以任意扩大规模的。这意味着，随着对精度或规模的需求越来越大，需要越来越大的计算机。使用整台机器来解决一个问题，只以解决问题的时间作为衡量成功的标准，这被称为能力计算。

另一方面，许多问题需要比整台超级计算机更少的时间来解决，所以通常一个计算中心会设置一台机器，让它为连续的用户问题服务，每个问题都比整台机器小。在这种模式下，衡量成功的标准是单位成本的持续性能。这就是所谓的容量计算，它需要一个精细调整的作业调度策略。

一个流行的方案是公平分享调度，它试图在用户之间，而不是在进程之间平均分配资源。这意味着，如果一个用户最近有作业，它将降低该用户的优先级，它将给短的或小的作业以更高的优先级。这个原则的调度器的

例子是 SGE 和 Slurm。

作业可以有依赖性，这使得调度更加困难。事实上，在许多现实条件下，调度问题是 NP-complete 的，所以在实践中会使用启发式方法。这个话题虽然有趣，但在本书中没有进一步讨论。

### 2.11.3 MapReduce

MapReduce[40] 是一种用于某些并行操作的编程模型。它的一个显著特点是使用函数式编程来实现。MapReduce 模型处理以下形式的编译。

- 对于所有可用的数据，选择满足某种标准的项目。
- 并为它们发出一个键值对。这就是映射阶段。
- 可以选择有一个组合/排序阶段，将所有与相同键值有关的对归为一组。
- 然后对键进行全局还原，产生一个或多个相应的值。这
- 这是还原阶段。

现在我们将举几个使用 MapReduce 的例子，并介绍支撑 MapReduce 抽象的函数式编程模型。

#### 2.11.3.1 MapReduce 模型的表达能力

MapReduce 模型的减少部分使其成为计算数据集全局统计数据的主要候选者。一个例子是计算一组词在一些文档中出现的次数。被映射的函数知道这组词，并为每个文档输出一对文档名称和一个包含词的出现次数的列表。然后，减法对出现次数进行分量级的求和。

MapReduce 的组合阶段使得数据转换成为可能。一个例子是" 反向网络链接图"：map 函数为在名为" 源" 的页面中发现的每个目标 URL 链接输出目标-源对。reduce 函数将与一个给定的目标 URL 相关的所有源 URL 的列表连接起来，并排放出目标列表 (source) 对。

一个不太明显的例子是用 MapReduce 计算 PageRank（第 9.4 节）。在这里，我们利用 PageRank 的计算依赖于分布式稀疏矩阵-向量乘积的事实。每个网页对应于网页矩阵 W 的一列；给定一个在网页 $p$ 上的概率 $j$，然后该网页可以计算出图元 $< i, w_{ij} p_j >$。然后 MapReduce 的组合阶段将 $(Wp)_i = \sum_j w_{ij} p_j$。

数据库操作可以用 MapReduce 来实现，但由于它的延迟比较大，不太可能与独立的数据库竞争，因为独立的数据库是为快速处理单个查询而优化的，而不是批量统计。

第 8.5.1 节中考虑了用 MapReduce 进行排序。

其他应用见 http://horicky.blogspot.com/2010/08/designing-algorithmis-for-map-reduce.html。

#### 2.11.3.2 MapReduce 软件

谷歌对 MapReduce 的实现是以 Hadoop 的名义发布的。虽然它适合谷歌的单阶段读取和处理数据的模式，但对许多其他用户来说，它有相当大的缺点。

- Hadoop 会在每个 MapReduce 周期后将所有数据冲回磁盘，所以对于需要超过一个周期的操作来说，文件系统和带宽需求太大。
- 在计算中心环境中，用户的数据不是连续在线的，将数据加载到 Hadoop 文件系统（HDFS）所需要的时间很可能会压倒实际分析。

由于这些原因，进一步的项目，如 Apache Spark（https://spark.apache.org/）提供了数据的缓存。

#### 2.11.3.3 执行问题

在分布式系统上实现 MapReduce 有一个有趣的问题：键-值对中的键集是动态确定的。例如，在上面的" 字数" 类型的应用中，我们并不是先验地知道字数的集合。因此，我们并不清楚应该将键值对发送给哪个还原器进程。

例如，我们可以使用一个哈希函数来确定这一点。由于每个进程都使用相同的函数，所以不存在分歧。这就留下了一个问题，即一个进程不知道要接收多少个带有键值对的消息。这个问题的解决方案在第 6.5.6 节中描述过。

#### 2.11.3.4 函数式编程

映射和规约操作很容易在任何类型的并行架构上实现，使用线程和消息传递的组合。然而，在开发这个模型的谷歌公司，传统的并行性没有吸引力，原因有二。首先，处理器在计算过程中可能会出现故障，所以传统的并行模式必须要用容错机制来加强。其次，计算硬件可能已经有了负荷，所以部分计算可能需要迁移，而且一般来说，任务之间的任何类型的同步都会非常困难。

MapReduce 是一种从并行计算的这些细节中抽象出来的方法，即通过采用函数式编程模型。在这样的模型中，唯一的操作是对一个函数的评估，应用于一些参数，其中参数本身就是一个函数应用的结果，而计算的结果又被作为另一个函数应用的参数。特别是，在严格的函数模型中，没有变量，所以没有静态数据。

一个函数应用，用 Lisp 风格写成 $(fab)$（意思是函数 $f$ 被应用于参数 $a$ 和 $b$），然后通过收集输入，从它们所在的地方到评估函数 $f$ 的处理器，来执行。

```
(map f (some list of arguments))
```

而结果是一个将 f 应用于输入列表的函数结果列表。所有并行的细节和保证计算成功完成的所有细节都由 map 函数处理。现在我们只缺少还原阶段，它也同样简单。

```
(reduce g (map f (the list of inputs)))
```

reduce 函数接收一个输入列表并对其进行规约。

这种函数模型的吸引力在于函数不能有副作用：因为它们只能产生一个输出结果，不能改变环境，因此不存在多个任务访问相同数据的协调问题。

因此，对于处理大量数据的程序员来说，MapReduce 是一个有用的抽象。当然，在实现层面上，MapReduce 软件使用了熟悉的概念，如分解数据空间、保存工作列表、将任务分配给处理器、重试失败的操作等等。

### 2.11.4 TOP500 名单

有几种非正式的方法来衡量一台计算机“有多大”。最流行的是 TOP500 名单，在 http://www.top500.org/，它记录了计算机在 Linpack 基准上的性能。Linpack 是一个用于线性代数操作的软件包，现在已经不再使用了，因为它已经被用于共享内存的 Lapack 和用于分布式内存计算机的 Scalapack 取代了。基准操作是通过部分枢轴的 LU 因子化来解决一个（方形、非星形、密集）的线性系统，并随后进行前向和后向解决。

LU 因子化操作是一个有很大机会可以重用缓存的操作，因为它是基于矩阵-矩阵乘法内核。它还具有工作量大于通信量的特性：$O(n^3)$ 与 $O(n^2)$ 。因此，Linpack 基准的运行速度很可能是机器峰值速度的一小部分。另一种说法是，Linpack 基准是一种受 CPU 约束或计算约束的算法。

典型的效率数字在 60% 到 90% 之间。然而，应该注意的是，许多科学代码并不具有密集的线性求解内核，所以这个基准的性能并不代表典型代码的性能。例如，通过迭代方法的线性系统求解，从每秒钟运算次数的角度来看，效率要低得多，因为它受 CPU 和内存之间的带宽所支配（一种带宽约束算法）。

经常使用的 Linpack 基准的一个实现是" 高性能 LINPACK"（http: //www.netlib.org/benchmark/hpl/），它有几个参数，如块大小，可以选择来调整性能。

#### 2.11.4.1 超级计算近年来的 top500 名单

TOP500 名单提供了近 20 年的超级计算历史。在本节中，我们将简要介绍一下历史发展 15。首先，图 2.41 显示了体系结构类型的演变情况，图中显示了整个榜单的总峰值性能中有多少部分是由每一种类型引起的。

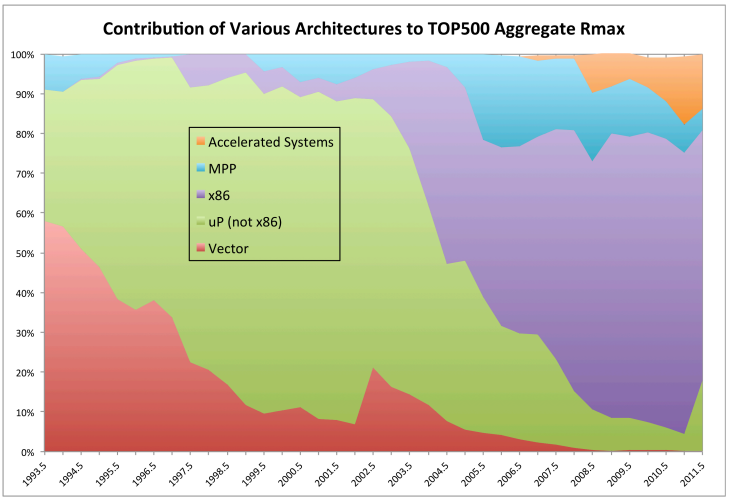



图 2.40: top500 名单上的架构类型的演变

- 向量机的特点是有相对少量的非常强大的向量流水线处理器。这种类型的架构已基本消失；这种类型的最后一台大型机器是日本的地球模拟器，它在 2002 年前后被视为图表中的尖峰，并在两年内一直处于榜首。
- 基于微处理器的架构从一台机器中的大量处理器中获得其动力。该图区分了 x86（英特尔和 AMD 处理器，英特尔 Itanium 除外）处理器和其他处理器；另见下图。
- 一些系统被设计成高度可扩展的架构：这些系统被表示为 MPP，即" 大规模并行处理器"。在时间轴的早期部分，这包括诸如 Connection Machine 这样的架构，后来几乎只包括 IBM 的 BlueGene。
- 近年来，" 加速系统" 是即将到来的趋势。在这里，一个处理单元（如 GPU）被连接到联网的主处理器上

接下来，图 2.42 显示了 x86 处理器类型相对于其他微处理器的主导地位。(由于我们将 IBM BlueGene 归为 MPP，其处理器在此不属于" 功率" 类别)。

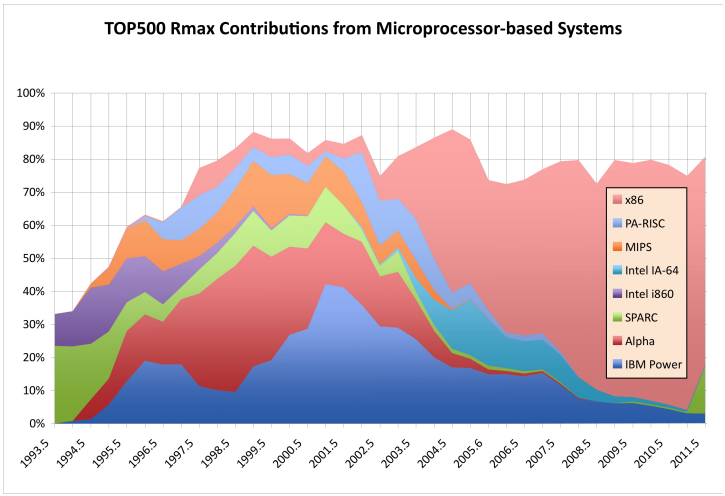



图 2.41: top500 名单上的架构类型的演变

- 在 1990 年代，男人的处理器是由多个芯片组成的。在某些情况下，一个插座实际上可以包含两个独立的芯片。
- 随着多核处理器的出现，图形中的部分非常接近垂直方向，这一点非常显著。这意味着新的处理器类型很快就会被采用，而较低的核心数量同样很快就会完全消失。
- 对于加速系统（主要是带有 GPU 的系统），" 核心数" 的概念更难定义；该图只是显示了这种类型的架构越来越重要。

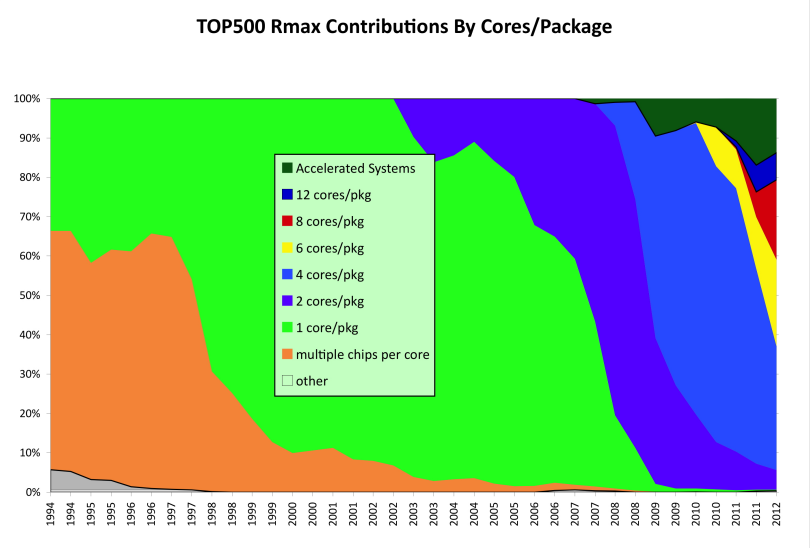



图 2.42: top500 名单上的架构类型的演变

### 2.11.5 异构计算

我们现在已经看到了几种计算模型：单核、共享内存多核、分布式内存集群、GPU。这些模型的共同点是，如果有一个以上的指令流处于活动状态，所有的指令流都是可以互换的。关于 GPU，我们需要细化这一说法：GPU 上的所有指令流都是可互换的。然而，GPU 并不是一个独立的设备，而是可以被认为是主机处理器的一个协处理器。

如果我们想让主机执行有用的工作，而协处理器处于活动状态，我们现在有两个不同的指令流或指令流类型。这种情况被称为异构计算。在 GPU 的情况下，这些指令流的编程机制甚至略有不同--为 GPU 使用 CUDA，但情况不必如此：英特尔许多集成核心（MIC）架构是用普通 C 语言编程的。

# 第 3 章 计算机中的运算

内容提要

- 位运算
- 整数
- 实数
- 舍入误差的例子
- 编程语言中的计算机运算
- 更多关于浮点运算的内容
- 结论

科学计算领域中常见的数字类型有：整数（或整型）$\cdots$,-2,-1,0,1,2,$\cdots$ 实数 0，1，-1.5，2/3，$\sqrt{2}$，log 10，$\cdots$，以及复数 $1+2i$，$\sqrt{3}-\sqrt{5}i,\cdots$，计算机硬件的组织方式是只给一定的空间来表示每个数字，是**字节**（bytes）的倍数，每个字节包含 8 **位**（bits）。典型的数值是整数为 4 字节，实数为 4 或 8 字节，复数为 8 或 16 字节。

由于内存空间有限，计算机并不能存储所有范围的数字。整数中计算机只能存储一个范围（Python 等语言有任意大的整数，但这没有硬件支持）；而在实数里，甚至不能存储一个范围，因为任意区间 [a,b] 都包含无限多的数字。任何实数的代表都会导致存储的数字之间存在间隔。计算机中的计算被称为**有限精度的运算**（finite precision arithmetic）。由于许多结果是无法表示的，任何导致这种数字的运算都必须通过发出错误或近似的结果来处理。在本章中，我们将研究这种对数值计算的" 真实" 结果的近似的影响。

关于详细的讨论，请参见 Overton 的书 [165]；在网上很容易找到 Goldberg 的文章 [80]。关于算法中舍入误差分析的广泛讨论，见 Higham [106] 和 Wilkinson [201] 的书。

## 3.1 位运算

计算机的最底层是以**位**（bits）来存储和表示的。比特，是" 二进制数字" 的简称，分为 0 和 1。使用比特我们就可以用二进制表达数字。

$$101_2 = 8_{10}$$

其中的下标表示数字的进制。

存储器的下一个组织层次是**字节**（bytes）：一个字节由 8 位组成，因此可以代表 0-255 的数值。

✍ **练习 3.1** 使用位操作来测试一个数字是奇数还是偶数。你能多想几种方法吗?

## 3.2 整数

科学计算绝大多数情况都是在实数上的运算。除了密码学等应用，对整数的计算很少增加到任意多的位数。也有一些应用，如" **粒子模型**（particle-in-cell）"，可以用位操作来实现。然而，整数在索引计算中仍有经验。

整数通常以 16、32 或 64 位存储，16 位越来越少，64 位则越来越多。这种增长的主要原因不是计算性质的变化，而是因为整数被用于数组索引。数据集的增长（特别是在并行计算中），需要更大的索引。例如，在 32 位中可以存储从 0 到 $2^{32}-1 \approx 410^9$ 的数字。换句话说，一个 32 位的索引可以解决 4GB 的内存。直到最近，这对大多数用途来说已经足够了；如今，对更大的数据集的需求使得 64 位索引成为必要。我们对数组进行索引时只需要正整数。当然，在一般的整数计算中，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。我们的初衷是，正负整数的算术应该和正整数一样简单：我们用于比较和操作比特串的电路应该可以用于（有符号）整数。

有几种实现负整数的方法。其中最简单的是保留一位作为符号位（sign bit），用剩下的 31 位（或 15 位或 63 位；从现在开始，我们将以 32 位为标准）位来存储绝对大小。通过比较，我们将把比特串的直接解释称为**无符**

**号整数**（unsigned integers）。

| bitstring | $00\cdots0$ | $\ldots$ | $01\cdots1$ | $10\cdots0$ | $\ldots$ | $11\cdots1$ |
|---|---|---|---|---|---|---|
| interpretation as unsigned int | 0 | $\ldots$ | $2^{31}-1$ | $2^{31}$ | $\ldots$ | $2^{32}-1$ |
| interpretation as signed integer | 0 | $\cdots$ | $2^{31}-1$ | $-0$ | $\ldots$ | $-(2^{31}-1)$ |

这种方案有一些缺点，其中之一是同时存在正数和负数 0。这意味着对平等的测试变得更加复杂，而不是简单地作为一个位串来测试平等。更重要的是，在比特串的后半部分，作为有符号的整数的解释减少了，向右走了。这意味着对大于的测试变得复杂；同时，将一个正数加到一个负数上，现在必须与将其加到一个正数上区别对待。

另一个解决方案是将无符号数 $n$ 解释为 $n-B$，其中 $B$ 是某个合理的基数，例如 $2^{31}$。

| bitstring | $00\cdots0$ | $\ldots$ | $01\cdots1$ | $10\cdots0$ | $\ldots$ | $11\cdots1$ |
|---|---|---|---|---|---|---|
| interpretation as unsigned int | 0 | $\ldots$ | $2^{31}-1$ | $2^{31}$ | $\ldots$ | $2^{32}-1$ |
| interpretation as shifted int | $-2^{31}$ | $\ldots$ | $-1$ | 0 | $\ldots$ | $2^{31}-1$ |

这种移位方案不存在 ±0 的问题，数字的排序也是一致的。然而，如果我们通过对代表 $n$ 的位串进行操作来计算 $n-n$，我们并没有得到零的位串。

为了保证这种理想的行为，我们改用正负数旋转数线，将零的模式放回零处。

由此产生的方案，也就是最常用的方案，被称为**二进制补码**（2's complement）。使用这种方案，整数的表示方法正式定义如下。

**定义 3.1**

设 $n$ 是一个整数，则其二进制 $\beta(n)$ 是个非负整数定义如下：

- 如果 $0 \leqslant n \leqslant 2^{31}-1$，则使用正常的 $n$ 比特模式，即
$$0 \leq n \leq 2^{31}-1 \Rightarrow \beta(n) = n$$
- 对于 $-2^{31} \leqslant n \leqslant -1$，$n$ 是由 $2^{32}-|n|$ 的比特模式表示。

♣

我们用 $\eta = \beta - 1$ 来表示接受一个比特模式并解释为整数的反函数。

下表显示了比特串与它们作为二进制整数的解释之间的对应关系。

| 比特串$n$ | $00\cdots0$ | $\cdots$ | $01\cdots1$ | $10\cdots0$ | $\ldots$ | $11\cdots1$ |
|---|---|---|---|---|---|---|
| 解释为无符号整数 | 0 | $\ldots$ | $2^{31}-1$ | $2^{31}$ | $\ldots$ | $2^{32}-1$ |
| interpretation$\beta(n)$as2's comp. integer | 0 | $\cdots$ | $2^{31}-1$ | $-2^{31}$ | $\ldots$ | $-1$ |

值得注意的是：

- 正整数和负整数的比特模式之间没有重叠，特别是，只有一个零的模式。
- 正数的前导位是零，而负数的前导位是 1。这使得前导位就像一个符号位；但是请注意上面的讨论
- 如果你有一个正数 $n$，你可以通过翻转所有的位，然后加 1 来得到 $-n$。

✍ **练习 3.2** 对于负数的原始方案和二进制补码方案，给出比较测试 $m < n$ 的伪码，其中 $m$ 和 $n$ 是整数。请注意区分 $mn$ 为正数、零数或负数的所有情况。

### 3.2.1 整数溢出

两个相同符号的数字相加，或两个任意符号的数字相乘，都可能导致结果过大或过小而无法表示。这就是所谓的**溢出**（overflow）。下面以一个例子进行讨论：

✍ **练习 3.3** 这样计算会发生什么？如果我们试图明确写下一个不可表示的数字，例如在一个赋值语句中，编译器会如何提示？

如果使用 C 语言，我们可能得到一个有意义的结果，这是因为在有符号数的情况下，C 标准下并没有定义溢出行为。

### 3.2.2 二进制加法

让我们考虑对二进制整数做一些简单的算术。我们首先假设我们拥有能处理无符号整数的硬件。我们的目标是看到我们可以用这个硬件对有符号的整数进行计算，就像用二进制表示一样。

我们考虑 $m+n$ 的计算，其中 $mn$ 是可表示的数字。

$$0 \leq |m|, |n| < 2^{31}$$

我们区分了不同的情况：

- 简单的情况是 $0 < m, n$。在这种情况下，我们进行正常的加法运算，只要结果保持在 $2^{31}$ 以下，我们就能得到正确的结果。如果结果是 $2^{31}$ 或更多，我们就会出现整数溢出，对此我们无能为力。

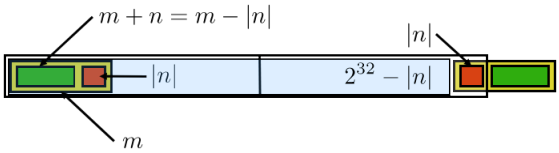



图 3.1: 一个正数与负数加法的二进制补码

- 当 $m > 0 n < 0$，并且 $m+n > 0$. 那么 $\beta(m) = m$ 和 $\beta(n) = 2^{32} - |n|$，所以无符号加法就变成

$$\beta(m) + \beta(n) = m + \left(2^{32} - |n|\right) = 2^{32} + m - |n|$$

由于 $m - |n| > 0$，这个结果 $> 2^{32}$。(见图 3.1) 然而，我们观察到这基本上是 $m+n$ 的第 33 位被设置。如果我们忽略这个溢出的位，我们就会得到正确的结果。

- \- 当 $m > 0, n < 0$ 但 $m + n < 0$，那么

$$\beta(m) + \beta(n) = m + \left(2^{32} - |n|\right) = 2^{32} - (|n| - m)$$

因为 $|n| - m > 0$，所以得到

$$\eta\left(2^{32} - (|n| - m)\right) = -|(|n| - m)| = m - |n| = m + n$$

### 3.2.3 二进制减法

在上面的练习 3.2 中，我们探索了两个整数的比较。现在让我们来探讨一下如何实现两个补数的减法。考虑 $0 \leqslant m \leqslant 2^{31} - 1$ 和 $1 \leqslant n \leqslant 2^{31}$，让我们看看在计算 $m - n$ 时会发生什么。

假设我们有一个无符号 32 位数加减法的算法。我们能不能用它来减去两个补码的整数？我们先观察一下，整数减法 $m - n$ 变成无符号加法 $m + (2^{32} - n)$。

- 当 $m < |n|$ 时，$m - n$ 为负数且 $1 \leqslant |m - n| \leqslant 2^{31}$，那么 $m - n$ 的比特形式为

$$\beta(m - n) = 2^{32} - (n - m)$$

现在，$2^{32} - n - m = m + 2^{32} - n$，所以我们可以通过将 $m$ 和 $-n$ 的位型相加作为无符号整数来计算 $m - n$ 的二进制码。

$$\eta(\beta(m) + \beta(-n)) = \eta\left(m + \left(2^{32} - |n|\right)\right) = \eta\left(2^{32} + (m - |n|)\right) = \eta\left(2^{32} - |m - |n||\right) = m - |n| = m + n$$

- 当 $m > n$，我们注意到 $m + (2^{32} - n) = 2^{32} + m - n$。由于 $m - n > 0$，这个数 >232，因此不是一个合法的负数表示。然而，如果我们将这个数字存储在 33 位，我们会发现它是正确的结果 $m - n$，加上 33 位的一个比特。因此，通过执行无符号加法，并忽略溢出位，我们再次得到正确的结果。

在这两种情况下，我们的结论是，我们可以通过将代表 $m$ 和 $-n$ 的无符号数相加，并忽略溢出的情况，来执行减法 $m - n$。

### 3.2.4 其他操作

有些操作在二进制中非常简单：乘以 2 相当于将所有位向左移动一个，而除以 2 相当于向右移动一位。至少，无符号整数是这样的。

✍ **练习 3.4** 当我们使用位移在二进制码中乘以或除以 2 时，是否有额外的复杂情况？

在 C 语言中，左移操作是 <<，右移是 >>，因此

```
i<<3
```

相当于乘以 8。

### 3.2.5 基于二进制的十进制编码

十进制在科学计算中并不重要，但在金融领域却十分有用，因为设计货币的计算绝对要精确。二进制并不擅长使用十进制转换，因为像 1/10 这样的数字在二进制中是重复的分数。由于尾数的位数有限，这意味着 1/10 这个数字不能用二进制精确表示。由于这个原因，二进制的十进制编码方案被用于老式的 IBM 主机，事实上，在 IEEE 754[113] 的修订中也被标准化了。

在 BCD 方案中，一个或多个十进制数字被编码为若干比特。最简单的方案是将数字 0...9 的四个比特。这样做的好处是，一个 BCD 数字中每个数字都很容易被识别；它的缺点是，大约有 1/3 的比特被浪费了，因为 4 个比特可以编码 0...15. 更有效的编码方法是将 0...999 的 10 个比特，原则上可以存储数字 $0...10^{23}$. 虽然这样做的效率很高，因为浪费的位数很少，但是识别这样一个数字中的各个位数需要一些解码。由于这个原因，BCD 算术需要处理器的硬件支持，现在很少有这种支持；一个例子是 IBM Power 架构，从 IBM Power6 开始。

### 3.2.6 用于计算机算术的其他数基

已经有一些关于三元运算的实验（见 http://en.wikipedia.org/wiki/Ternary_computer 和 http://www.computer-museum.ru/english/setun.htm），但是，没有实际的硬件存在。

## 3.3 实数

在这一节中，我们将研究实数如何在计算机中表示，以及各种方案的局限性。下一节将探讨这对涉及计算机数字的算术的影响。

### 3.3.1 它们不是真正的实数

在数学科学中，我们通常用实数工作，所以假装计算机也能这样做是很方便的。然而，由于计算机中的数字只有有限的比特数，大多数实数都不能被准确表示。事实上，甚至许多分数也不能准确表示，因为它们会重复；例如，1/3 = 0.333...，这在十进制或二进制中都不能表示。附录 37.6 中给出了这方面的一个说明。

✍ **练习 3.5** 一些编程语言允许我们在写循环时不仅使用整数，还可以使用实数作为" 计数器"。解释一下为什么这是个坏主意。提示：何时达到上界？

一个分数是否重复取决于数字系统。(在二进制计算机中，这意味着像 1/10 这样的分数是重复的，而在十进制算术中，这些分数的位数是有限的。由于小数运算在金融计算中很重要，所以有些人关心这种算术的准确性。

✍ **练习 3.6** 显示每个二进制分数，即形式为 $1.01010111001_2$ 的数字，都可以精确地表示为一个终止的十进制分数。不是每个十进制分数都能表示为二进制分数的原因是什么?

### 3.3.2 实数的表示

实数的存储方式类似于所谓的" 科学符号"，即一个数字用一个显数和一个指数表示，例如 $6.02210^{23}$，它的显数是 6022，第一个数字后有一个**小数点**（radix point），指数是 23。这个数字代表

$$6.022 \cdot 10^{23} = \left[6 \times 10^{0} + 0 \times 10^{-1} + 2 \times 10^{-2} + 2 \times 10^{-3}\right] \cdot 10^{23}$$

我们引入一个基数，一个小的整数，在前面的例子中是 10，在计算机数字中是 2，用它来写数字，作为 $t$ 项的和。

$$\begin{aligned} x &= \pm 1 \times \left[d_1\beta^0 + d_2\beta^{-1} + d_3\beta^{-2} + \cdots + d_t\beta^{-t+1}b\right] \times \beta^e \\ &= \pm \sum_{i=1}^{t} d_i\beta^{1-i} \times \beta^e \end{aligned}$$

其中的组成部分是

- **符号位**（sign bit）：存储数字是正数还是负数的一个位。
- $\beta$ 是数字系统的基数。
- $0 \leqslant d_i \leqslant \beta - 1$ 尾数或显数的位数 - 小数点的位置（小数的小数点）被隐含地假定为小数。小数点的位置被隐含地假定为紧随第一位的位置。
- $t$ 是尾数的长度。
- $e \in [L, U]$ 指数；通常 $L < 0 < U$ 和 $L \approx -U$。

注意，整数有一个明确的符号位；指数的符号处理方式不同。出于效率的考虑，$e$ 不是一个有符号的数字；相反，它被认为是一个超过某个最小值的无符号数字。例如，数字 0 的比特模式被解释为 $e = L$。

#### 3.3.2.1 案例

让我们看一下浮点表示法的一些具体例子。对于人类来说，基数 10 是最合理的选择，但计算机是二进制的，所以基数 2 占据主导地位。老式的 IBM 大型机将比特分组，使之成为基数 16 的表示法。

| | $\beta$ | $t$ | $L$ | $U$ |
|---|---|---|---|---|
| IEEE single precision (32 bit) | 2 | 24 | −126 | 127 |
| IEEE double precision (64 bit) | 2 | 53 | −1022 | 1023 |
| Old Cray 64 bit | 2 | 48 | −16383 | 16384 |
| IBM mainframe 32 bit | 16 | 6 | −64 | 63 |
| packed decimal | 10 | 50 | −999 | 999 |
| Setun | 3 | | | |

其中，单精度和双精度格式是迄今为止最常见的。

### 3.3.3 限制：溢出和下溢

由于我们只用有限的比特来存储浮点数，所以不是所有的数字都能被表示出来。那些不能被表示的数字分为两类：那些太大或太小（在某种意义上）的数字，以及那些落在空白处的数字。

第二类是计算结果必须经过四舍五入或截断才能表示，这是舍入误差分析领域的基础。我们将在下面的章节中详细研究这个问题。

数字过大或过小有以下几种情况。

溢出：我们可以存储的最大的数字，其每个数字都等于 $\beta$。

| | unit | fractional | | | exponent |
|---|---|---|---|---|---|
| position | 0 | 1 | $\cdots$ | $t-1$ | |
| digit | $\beta-1$ | $\beta-1$ | $\cdots$ | $\beta-1$ | |
| value | 1 | $\beta^{-1}$ | $\cdots$ | $\beta^{-(t-1)}$ | |

加起来就是

$$(\beta-1)\cdot 1+(\beta-1)\cdot\beta^{-1}+\cdots+(\beta-1)\cdot\beta^{-(t-1)}=\beta-\beta^{-(t-1)}$$

而最小的数字（即最负数）是 $-(\beta-\beta^{-(t-1)})$；任何大于前者或小于后者的情况都会导致溢出。大于前者或小于后者都会导致**溢出**（overflow）的情况发生。

下溢最接近零的数字是 $\beta-(t-1)L$。如果计算结果小于该值 (绝对值)，就会导致一种叫做**溢出**（underflow）的情况。(参见第 3.3.4 节的渐进式下溢)。

只有少数实数可以被精确表示，这一事实是舍入误差分析领域的基础。我们将在下面的章节中详细研究这个问题。

溢出或下溢的发生意味着你的计算将从这一点上" 出错"。溢出将使计算在本应是非零的地方以零进行；溢出被表示为 Inf，简称" 无限"。

✍ **练习 3.7** 对于实数 $x, y$，$g=\sqrt{(x^2+y^2/2)}$ 满足

$$g\leqslant \max\{|x|,|y|\}$$

所以，如果 x 和 y 是可表示的。如果你用上述公式计算 ，会出现什么问题？你能想到一个更好的方法吗？

用 Inf 计算在某种程度上是可能的：将这些数量中的两个相加又会得到 Inf。然而，减去它们会得到 NaN：" 不是一个数字"。

在这些情况下，计算都不会结束：处理器会继续，除非你告诉它不这样做。这个" 否则" 是指你告诉编译器产生一个**中断**（interrupt），用一个错误信息来停止计算。

### 3.3.4 归一化和非归一化的数字

浮点数的一般定义，方程给我们留下了一个问题，即数字有不止一种表示方法。例如，$.5\times 10^2=.05\times 10^3$ 。由于这将使计算机运算变得不必要的复杂，例如在测试数字是否相等时，我们使用规范化的浮点数。如果一个数字的第一个数字是非零的，那么这个数字就是归一化的。这意味着尾数部分是

$$\beta > x_m > 1$$

在二进制数的情况下，一个实际的含义是，第一个数字总是 1，所以我们不需要明确地存储它。在 IEEE 754 标准中，这意味着每个浮点数的形式为

$$1.d_1d_2...d_t\times 2^{exp}$$

这个方案的另一个含义是，我们必须修改下溢的定义：任何小于 $1\beta L$ 的数字现在都会导致下溢。试图计算一个绝对值小于该值的数，有时会通过使用非正常化的浮点数来处理，这个过程被称为**渐进式下溢**（gradual underflow）。在这种情况下，指数的一个特殊值表明该数字不再被规范化。在 IEEE 标准算术的情况下，这是通过一个零指数域来实现的。

然而，这通常比用普通的浮点数计算要慢几十或几百倍。在写这篇文章的时候，只有 IBM Power6 有硬件支持渐进式下溢。

### 3.3.5 表示性误差

让我们考虑一个在计算机的数字系统中无法表示的实数。

一个不可表示的数字可以通过普通四舍五入、向上或向下四舍五入或截断来近似表示。这意味着，一个机器数 $x$ 是它周围的所有 $x$ 的代表。在尾数为 $t$ 的情况下，这是与 $x$ 不同的数字的区间，在 $t+1$ 个数字中。对于尾数部分，我们得到。

$$\begin{cases} x \in \left[\tilde{x}, \tilde{x} + \beta^{-t+1}\right) & \text{truncation} \\ x \in \left[\tilde{x} - \frac{1}{2}\beta^{-t+1}, \tilde{x} + \frac{1}{2}\beta^{-t+1}\right) & \text{rounding} \end{cases}$$

如果 x 是一个数字，$\tilde{x}$ 它在计算机中的表示，我们称 $x - \tilde{x}$ 为**表示性误差**（representation error）或**绝对表示误差**（absolute representation error），$(x - \tilde{x})/x$ 为**相对表示误差**（relative representation error）。通常情况下，我们对误差的符号不感兴趣。所以我们可以将误差和相对误差分别应用于 $|x - \tilde{x}|$ 和 $|\frac{\tilde{x}-x}{x}|$。

通常，我们只对误差的界限感兴趣。如果 $\epsilon$ 是对误差的约束，我们将写成

$$\tilde{x} = x \pm \epsilon \underset{\text{D}}{\equiv} |x - \tilde{x}| \leq \epsilon \Leftrightarrow \tilde{x} \in [x - \epsilon, x + \epsilon]$$

对于相对误差，我们注意到

$$\tilde{x} = x(1 + \epsilon) \Leftrightarrow \left|\frac{\tilde{x} - x}{x}\right| \leq \epsilon$$

让我们考虑一个十进制算术的例子，即 $\beta = 10$，并且有一个 3 位数的尾数：$t = 3$。数字 $x = 1.256$，其表示方法取决于我们是四舍五入还是截断。$\tilde{x}_{round} = 1.26, \tilde{x}_{truncate} = 1.25$。误差在第四位：如果 $\epsilon = x - \tilde{x}$ 那么 $|\epsilon| < \beta^{-(t-1)}$。

✍ **练习 3.8** 本例中的数字没有指数部分。如果有的话，其误差和相对误差是多少？

✍ **练习 3.9** 如上所述，在二进制运算中，单位数总是 1。这对表示错误有什么影响？

### 3.3.6 机器精度

通常我们只对表示误差的数量级感兴趣，我们将写 $\tilde{x} = x1 + \epsilon$，其中 $|\epsilon| \leqslant \beta - t$。这个最大的相对误差被称为**机器精度**（machine precision）（有时也称为 machine epsilon），典型的数值是。

$$\begin{cases} \epsilon \approx 10^{-7} & \text{32-bit single precision} \\ \epsilon \approx 10^{-16} & \text{64-bit double precision} \end{cases}$$

机器精度可以用另一种方式定义：$\epsilon$ 是可以加到 1 上的最小的数字，这样 $1 + \epsilon$ 的表示方法与 1 不同。一个小例子表明，对齐指数可以转移一个太小的操作数，这样它在加法运算中就被有效地忽略。

$$\begin{array}{rr} & 1.0000 \times 10^{0} \\ + & 1.0000 \times 10^{-5} \end{array} \Rightarrow \begin{array}{rll} & 1.0000 & \times 10^{0} \\ + & 0.00001 & \times 10^{0} \\ \hline = & 1.0000 & \times 10^{0} \end{array}$$

另一种方法是，在加法 $x + y$ 中，如果 $x$ 和 $y$ 的比例过大，结果将与 $x$ 相同。

机器精度是计算可达到的最大精度：如果要求单精度超过 6 位或更多位数的精度，或者双精度超过 15 位，是没有意义的。

✍ **练习 3.10** 写一个小程序，计算机器的 $\epsilon$ 值。如果你把编译器的优化级别设置得低或高，有什么区别吗？

✍ **练习 3.11** 数字 $e \approx 2.72$，自然对数的基数，有多种定义。其中一个是

$$e = \lim_{n \to \infty} (1 + 1/n)^n.$$

写一个单精度程序，尝试用这种方式计算 $e$。评估上界 $n = 10^k$ 的表达式，$k = 1, ...., 10$. 解释一下大 $n$ 的输出。对误差的行为进行评论。

### 3.3.7 IEEE 754 的浮点数标准

几十年前，像尾数的长度和操作的四舍五入行为等问题在不同的计算机制造商之间，甚至在同一制造商的不同型号之间可能会有所不同。从代码的可移植性和结果的可重复性来看，这显然是一个坏情况。IEEE 754 标准对这一切进行了编纂，例如，规定单精度和双精度算术的尾数为 24 和 53 位，使用符号位、指数、尾数的存储序列，见下表。表中列出了单精度标准中所有可能的位模式的含义。

| sign | exponent | mantissa |
|---|---|---|
| $p$ | $e = e_1 \cdots e_8$ | $s = s_1 \cdots s_{23}$ |
| 31 | $30 \cdots 23$ | $22 \cdots 0$ |
| $\pm$ | $2^{e-127}$ (except $e = 0, 255$) | $2^{-s_1} + \cdots + 2^{-s_{23}}$ |

| sign | exponent | mantissa |
|---|---|---|
| $s$ | $e_1 \cdots e_{11}$ | $s_1 \ldots s_{52}$ |
| 63 | $62 \cdots 52$ | $51 \cdots 0$ |

注释 754 标准的全称是'IEEE 二进制浮点运算标准（AN- SI/IEEE Std 754-1985）'。它也与 IEC559：'微处理器系统的二进制浮点算术' 相同，被 ISO/IEC/IEEE 60559:2011 所取代。

IEEE 754 是二进制算术的标准；还有一个标准，IEEE 854，允许十进制算术。

注释 令人瞩目的是，在场的这么多硬件人士在知道 p754 有多难的情况下，都同意它应该对整个社区有益。如果它能鼓励浮点软件的生产，缓解可靠软件的开发，就能为大家的硬件创造一个更大的市场。这种利他主义的程度是如此惊人，以至于 MATLAB 的创建者 Cleve Moler 博士曾经建议外国游客不要错过该国最令人敬畏的两大景观：大峡谷和 IEEE p754 的会议。W. Kahan，http://www.cs.berkeley.edu/ wkahan/ieee754status/754story.html

该标准还宣布四舍五入行为是正确的四舍五入：一个操作的结果应该是精确结果的四舍五入版本。关于四舍五入（和截断）对数字计算的影响，下面会有更多的介绍。

| $(e_1 \cdots e_8)$ | numerical value | range |
|---|---|---|
| $(0 \cdots 0) = 0$ | $\pm 0.s_1 \cdots s_{23} \times 2^{-126}$ | $s = 0 \cdots 01 \Rightarrow 2^{-23} \cdot 2^{-126} = 2^{-149} \approx 10^{-50}$<br>$s = 1 \cdots 11 \Rightarrow (1 - 2^{-23}) \cdot 2^{-126}$ |
| $(0 \cdots 01) = 1$ | $\pm 1.s_1 \cdots s_{23} \times 2^{-126}$ | |
| $(0 \cdots 010) = 2$ | $\pm 1.s_1 \cdots s_{23} \times 2^{-125}$ | |
| $\cdots$ | | |
| $(01111111) = 127$ | $\pm 1.s_1 \cdots s_{23} \times 2^{0}$ | $s = 0 \cdots 00 \Rightarrow 1 \cdot 2^0 = 1$<br>$s = 0 \cdots 01 \Rightarrow 1 + 2^{-23} \cdot 2^0 = 1 + \epsilon$<br>$s = 1 \cdots 11 \Rightarrow (2 - 2^{-23}) \cdot 2^0 = 2 - \epsilon$ |
| $(10000000) = 128$ | $\pm 1.s_1 \cdots s_{23} \times 2^{1}$ | |
| $\cdots$ | | |
| $(11111110) = 254$ | $\pm 1.s_1 \cdots s_{23} \times 2^{127}$ | |
| $(11111111) = 255$ | $s_1 \cdots s_{23} = 0 \Rightarrow \pm\infty$<br>$s_1 \cdots s_{23} \neq 0 \Rightarrow$ NaN | |

图 3.2: 单精度数的解释取决于指数位模式

在上面，我们已经看到了溢出和下溢的现象，也就是导致不可表示的数字的操作。还有一种特殊情况需要处理：如果程序要求进行非法运算，如 $\sqrt{-4}$，应该返回什么结果？IEEE 754 标准对此有两个特殊量。Inf 和 NaN 代表" 无穷大" 和" 不是一个数字"。Inf 是指溢出或除以 0 的结果，note-a-number 是指，例如，从 infinity 中减去 infinity 的结果。如果 NaN 出现在一个表达式中，整个表达式将评估为该值。用 Inf 计算的规则要复杂一些 [80]。

图 3.2 给出了 IEEE 754 单精度中所有位模式的含义清单。从上面可以看出，对于归一化的数字，第一个非

零位是 1，它不被存储，所以位模式 $d_1 d_2 ... d_t$ 被解释为 $1.d_1 d_2 ... d_t$ 。

✍ **练习 3.12** 每个程序员都会犯这样的错误：将一个实数存储在一个整数中，或者反过来存储。例如，如果调用一个函数的方式与它的定义不同，就会发生这种情况。

```
void a(double x) {....}
   int main() {
      int i;
      .... a(i) ....
}
```

当在函数中打印 $x$ 时会发生什么？考虑一个小整数的比特模式，并使用图 3.3 中的表格将其解释为一个浮点数。解释一下，它将是一个未归一化的数字。

如今，几乎所有的处理器都遵守了 IEEE 754 标准。早期的 NVidia Tesla GPU 在单精度方面不符合标准。这样做的理由是，单精度更可能用于图形，在那里，准确的合规性不太重要。对于许多科学计算，双精度是必要的，因为计算的精度会随着问题大小或运行时间的增加而变差。这对于第四章中的那种计算来说是正确的，但对于其他的计算，如格子玻尔兹曼法（LBM），则不是这样。

### 3.3.8 浮点数异常

各种各样的操作可能会给出一个无法表示为浮点数的结果。这种情况被称为**异常**（exception），我们说提出了一个异常。结果取决于错误的类型，而计算则正常进行。

#### 3.3.8.1 Not-a-Number

以下情况处理器将表示为 NaN（' 不是一个数字'）的结果。

- 两个无穷大相减，注意两个无穷大相加仍为无穷大
- 0 乘以无穷大
- 0 除以 0 或无穷大除以无穷大
- $\sqrt{x}$ 当 $x < 0$ 时
- 比较 $x < y$ 或 $x > y$ 时，其中任意一个数为 $NaN$

由于处理器可以继续对这样的数字进行计算，所以它被称为**安静的 NaN**（quiet NaN）。相比之下，一些 NaN 数量可以导致处理器产生一个中断或异常。这被称为**信号型 NaN**（signalling NaN）。

信号 NaN 是有用途的。例如，你可以用这样一个值来填充分配的内存，以表明它在计算中是未初始化的。任何使用这样一个值的行为都是一个程序错误，并会引起一个异常。

2008 年修订的 IEEE 754 建议使用 NaN 的最有效位作为 is_quiet 位来区分安静和信号 NaN。

关于 GNU 编译器中对 Nan 的处理，请参见 https://www.gnu.org/software/libc/manual/html_node/Infinity-and-NaN.html。

#### 3.3.8.2 除以零

除以 0 的结果是 Inf。如果一个结果不能作为一个有限的数字来表示，就会引发这个异常。

#### 3.3.8.3 下溢

如果一个数字太小，不能被表示，就会出现这个异常。

#### 3.3.8.4 不精确

如果出现不精确的结果，例如平方根，就会引发这个异常，如果没有被困住，就会出现溢出。

### 3.3.9 舍入误差分析

过大或过小的数字无法表示，导致溢出和下溢，是不正常的：通常可以安排计算，使这种情况不会发生。相比之下，计算机数字之间的计算结果（甚至像一个简单的加法）无法表示的情况是非常普遍的。因此，看一个算法的实现，我们需要分析这种小错误在计算中传播的影响。这就是通常所说的**舍入误差分析**（round-off error analysis）。

### 3.3.10 正确的舍入

前面提到的 IEEE 754 标准，不仅声明了浮点数的存储方式，还给出了加、减、乘、除等运算的准确性标准。该标准中的算术模型是正确的四舍五入模型：一个操作的结果应该像遵循以下程序一样。

- 计算出运算的确切结果，无论这是否可以表示。
- 将这个结果四舍五入到最接近的计算机数字。

简而言之：一个操作的结果的表示就是该操作的四舍五入的准确结果。(当然，在两次操作之后，它不再需要坚持计算的结果是精确结果的四舍五入版本)。

如果这句话听起来微不足道或不言而喻，请考虑以减法为例。在尾数为两位的十进制数制中，计算结果为 $1.0 - 9.4 10^{-1} = 1.0 - 0.94 = 0.06 = 0.6 10^{-2}$。请注意，在一个中间步骤中，尾数.094 出现了，它比我们为我们的数字系统声明的两个数字多了一个数字。这个额外的数字被称为**警戒位**（guard digit）。

如果没有警戒位，这个运算将以 $1.0 - 9.4 10^{-1}$ 的形式进行，其中 $9.4 10^{-1}$ 将被四舍五入为 0.9，最终结果为 0.1，这几乎是正确结果的两倍。

✍ **练习 3.13** 考虑 $1.0 - 9.5 10^{-1}$ 的计算，并再次假设数字被四舍五入以适应两位数的尾数。为什么这个计算在某种程度上比刚才的例子要差很多?

一个警戒位不足以保证正确的舍入。一项我们在此不做转载的分析表明，需要额外的三个比特 [79]。

#### 3.3.10.1 多重添加操作

2008 年，IEEE 754 标准进行了修订，以包括融合乘加（FMA）操作的行为，也就是说，操作形式为

$$c \leftarrow a * b + c.$$

这种操作有两方面的考虑。

首先，FMA 有可能比单独的乘法和加法更精确，因为它可以对中间结果使用更高的精度，例如使用 80 位的扩展精度格式。

这里的标准定义了正确的四舍五入，即这种组合计算的结果应该是四舍五入后的正确结果。这种操作的原始实现将涉及两次舍入：一次在乘法之后，一次在加法之后 3。

✍ **练习 3.14** 你能想出一个例子，说明对 FMA 进行正确的舍入比对乘法和加法分别进行舍入更准确吗？提示：让 $c$ 项的符号与 $a * b$ 相反，并尝试在减法中强制取消。

其次，FMA 指令是一种获得更高性能的方法：通过流水线，我们可以在每个周期内获得两个操作。因此，一个 FMA 单元比单独的加法和乘法单元更便宜。幸运的是，FMA 在实际计算中经常出现。

✍ **练习 3.15** 你能想到一些以 FMA 运算为特征的线性代数运算吗?

### 3.3.11 加法

两个浮点数的加法是通过几个步骤完成的。首先，指数被对齐：两个数字中较小的数字被写成与较大的数字具有相同的指数。然后再加上尾数。最后，对结果进行调整，使其再次成为一个标准化的数字。

作为一个例子，考虑 $1.00+2.0010^{-2}$。对准指数，这就变成了 1.00+0.02=1.02，这个结果不需要最后调整。我们注意到这个计算是精确的，但是和 $1.00+2.5510^{-2}$ 有同样的结果，这里的计算显然是不精确的：精确的结果是 1.0255，它不能用三位数的尾数来表示。

在 $6.15\times10^1+3.98\times10^1=10.13\times101=1.013\times10^2\rightarrow1.01\times10^2$ 的例子中，我们看到在加上尾数后，需要对指数进行调整。误差又来自于对不适合尾数的结果的第一个数字的截断或四舍五入：如果 $x$ 是真实的和，$\tilde{x}$ 是计算的和，那么 $\tilde{x}=x(1+\epsilon)$ 其中，3 位尾数 $|\epsilon|<10^{-3}$。

形式上，让我们考虑计算 $s=x_1+x_2$，我们假设数字 $i$ 表示为 $\tilde{x}_i=x_i(1+\epsilon_i)$。那么和 $s$ 就表示为

$$\begin{aligned}\tilde{s}&=(\tilde{x}_1+\tilde{x}_2)(1+\epsilon_3)\\&=x_1(1+\epsilon_1)(1+\epsilon_3)+x_2(1+\epsilon_2)(1+\epsilon_3)\\&\approx x_1(1+\epsilon_1+\epsilon_3)+x_2(1+\epsilon_2+\epsilon_3)\\&\approx s(1+2\epsilon)\end{aligned}$$

在所有 $\epsilon_i$ 都很小且大小大致相等，并且 $x_i>0$ 的假设下，我们看到相对误差在加法下被加上了。

### 3.3.12 乘法

浮点乘法，就像加法一样，包括几个步骤。为了使两个数字 $m_1\times\beta^{e_1}$ 和 $m_2\times\beta^{e_2}$ 相乘，需要采取以下步骤。

- 指数相加：$e\leftarrow e_1+e_2$。
- 尾数相乘：$m\leftarrow m_1\times m_2$。
- 尾数被归一化，指数也相应调整。

例如：$1.2310^05.6710^1=0.6974110^1\rightarrow6.974110^0\rightarrow6.9710^0$。

✍ **练习 3.16** 分析乘法的相对误差。

### 3.3.13 减法

减法的表现与加法非常不同。在加法中，误差是相加的，只是逐步增加整体的舍入误差，而减法则有可能在一次操作中大大增加误差。

例如，考虑尾数为 3 位的减法：$1.24-1.23=0.01\rightarrow1.0010^{-2}$ 。虽然结果是准确的，但它只有一个有效数字 4 。为了了解这一点，可以考虑这样的情况：第一个操作数 1.24 实际上是一个四舍五入的计算结果，其结果应该是 1.235。在这种情况下，减法的结果应该是 $5.0010^{-3}$，也就是说，存在 100% 的误差，尽管输入的相对误差是可以预期的小。显然，涉及这一减法结果的后续操作也将是不准确的。我们的结论是，减去几乎相等的数字可能是造成数字四舍五入的原因。

这个例子有一些微妙之处。几乎相等的数字的减法是准确的，而且我们有 IEEE 算术的正确舍入行为。尽管如此，单一运算的正确性并不意味着包含它的运算序列会是准确的。虽然加法的例子只显示了数字精度的适度下降，但这个例子中的取消会产生灾难性的影响。

✍ **练习 3.17** 考虑迭代

$$x_{n+1}=f(x_n)=\begin{cases}2x_n & \text{if} 2x_n<1\\2x_n-1 & \text{if} 2x_n 1\end{cases}$$

这个函数是否有一个固定点，$x_0 \equiv f(x_0)$，或者是否有一个循环 $x_1 = f(x_0)x_0 \equiv x_2 = f(x_1)$ 等等？现在对这个函数进行编码。是否有可能重现固定点？不同的起始点 $x_0$ 会发生什么。你能解释一下吗？

### 3.3.14 关联性

处理浮点数的方式的另一个影响是对运算的**关联性**（associativity），如求和。虽然求和在数学上是关联性的，但在计算机运算中却不再是这样。

让我们考虑一个简单的例子，说明这如何由浮点数的舍入行为引起。让浮点数存储为尾数的一个数字，指数的一个数字，以及一个保护数字；现在考虑 4+6+7 的计算。从左到右的计算结果是：

$$\begin{aligned}
(4\cdot 10^0 + 6\cdot 10^0) + 7\cdot 10^0 &\Rightarrow 10\cdot 10^0 + 7\cdot 10^0 && \text{addition}\\
&\Rightarrow 1\cdot 10^1 + 7\cdot 10^0 && \text{rounding}\\
&\Rightarrow 1.0\cdot 10^1 + 0.7\cdot 10^1 && \text{using guard digit}\\
&\Rightarrow 1.7\cdot 10^1 &&\\
&\Rightarrow 2\cdot 10^1 && \text{rounding}
\end{aligned}$$

另一方面，从右到左的评估给出了。

$$\begin{aligned}
4\cdot 10^0 + \left(6\cdot 10^0 + 7\cdot 10^0\right) &\Rightarrow 4\cdot 10^0 + 13\cdot 10^0 && \text{addition}\\
&\Rightarrow 4\cdot 10^0 + 1\cdot 10^1 && \text{rounding}\\
&\Rightarrow 0.4\cdot 10^1 + 1.0\cdot 10^1 && \text{using guard digit}\\
&\Rightarrow 1.4\cdot 10^1 &&\\
&\Rightarrow 1\cdot 10^1 && \text{rounding}
\end{aligned}$$

结论是，对中间结果进行四舍五入和截断的顺序是有区别的。还可以观察到，从较小的数字开始会得到更准确的结果。

✍ **练习 3.18** 上面的例子使用了四舍五入。你能在算术系统中想出一个使用截断的类似例子吗？

通常情况下，表达式的求值顺序是由编程语言的定义决定的，或者至少是由编译器决定的。后面我们将看到在并行计算中，关联性不是那么唯一地确定。

## 3.4 舍入误差的例子

从上面的介绍中，读者可能会得到这样的印象：舍入误差只在特殊情况下才会导致严重的问题。在这一节中，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在计算结果中变得非常明显。这些将是相当简单的例子；更复杂的例子存在于本书的范围之外，例如矩阵反演的不稳定性。有兴趣的读者可以参考 [201，106]。

### 3.4.1 取消："abc 模式"。

作为一个实际的例子，考虑二次方程 $ax^2 + bx + c = 0$，其解 $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$。假设 $b > 0$ 且 $b^2 >> 4ac$，则 $\sqrt{b^2 - 4ac} \approx b$，'+' 解将是不准确的。在这种情况下，最好计算 $x_- = -b - \sqrt{b^2 - 4ac}$ 并使用 $x_+ - x_- = c/a$。

✍ **练习 3.19** 探索计算的根基

$$\epsilon x^2 - (1 + \epsilon^2)x + \epsilon$$

通过" 教科书" 的方法，并如上所述。

- 这些根是什么?
- 为什么" 教科书" 方法把一个小的根计算为零?
- 两种方法计算的函数值是多少? 相对误差?

✍ **练习 3.20** 写一个程序来计算一元二次方程的根，包括" 教科书" 上的方法和上面描述的方法。

- 让 $b=-1$，$a=-c$，$4ac\downarrow 0$，逐步取较小的 $a$ 和 $c$ 值。
- 打印出计算出的根，使用稳定计算的根，以及计算出的根中的 $fx=ax^2+bx+c$ 的值。

现在，假设我们不太关心根的实际值：你想确保在计算的根中，残差 $f(x)$ 很小。让 $x$ 是准确的根，那么

$$f(x^*+h)\approx f(x^*)+hf'(x^*)=hf'(x^*)$$

现在分别研究 $a\downarrow 0$,$c=-1$ 和 $a=-1$,$c\downarrow 0$ 的情况，你能解释其中的区别吗?

✍ **练习 3.21** 考虑函数

$$\begin{cases} f(x)=\sqrt{x+1}-\sqrt{x} \\ g(x)=1/(\sqrt{x+1}+\sqrt{x}) \end{cases}$$

- 证明它们在精确算术中是相同的；但是。
- 证明 $f$ 可以表现出取消，而 $g$ 则没有这个问题。
- 编写代码以显示 $f$ 和 $g$ 之间的差异。你可能需要使用较大的 $x$ 的值。
- 从 x 和机器精度的角度来分析取消的情况。当 $\sqrt{x+1}$ 和 $\sqrt{x}$ 的距离小于 $\epsilon$? 这时会发生什么? (为了更精确的分析，当它们之间相差 $\sqrt{\epsilon}$，又是如何表现出来的?)
- $y=f(x)$ 的反函数是

$$x=(y^2-1)^2/(4y^2)$$

把这个添加到你的代码中。这是否说明了计算的准确性?

请确保在单精度和双精度下测试代码。如果你会用 python，可以试试 bigfloat 包。

### 3.4.2 总结系列

前面的例子是关于防止一次操作中出现大的舍入误差。这个例子表明，即使是逐渐积累的舍入误差也可以用不同的方法来处理。

考虑总和 $\sum_{n=1}^{10000}\frac{1}{n^2}=1.644834$，假设我们使用的是单精度，这对大多数计算机上意味着机器精度为 $10^{-7}$. 这个例子的问题在于，无论是项之间的比率，还是项与部分和的比率，都在不断增加。在前面，我们注意到过大的比率会导致加法的一个操作数被忽略。

如果我们按照给出的序列对数列进行求和，我们会发现第一项是 1，所以所有的部分和（$\sum_{n=1}^{N}$，其中 $N<10000$）至少是 1。这意味着任何 $1/n^2<10^{-7}$ 的项都会被忽略，因为它小于机器精度。具体来说，最后 7000 个项被忽略，计算出的总和是 1.644725。前 4 位数字是正确的。

然而，如果我们以相反的顺序评估和，我们会得到单精度的精确结果。我们仍然是把小量加到大量上，但现在的比例永远不会像一比 $\epsilon$ 那样糟糕，所以小的数字永远不会被忽略。要看到这一点，请考虑两个项的比率随后的项。

$$\frac{n^2}{(n-1)^2}=\frac{n^2}{n^2-2n+1}=\frac{1}{1-2/n+1/n^2}\approx 1+\frac{2}{n}$$

由于我们只对 105 项求和，而且机器的精度是 $10^{-}7$，所以在加法 $1/n^2+1/(n-1)^2$ 中，第二项不会像我们从大到小求和时那样被完全忽略。

✍ **练习 3.22** 在我们的推理中还缺少一个步骤。我们已经表明，在加两个后续项时，较小的一项不会被忽略。然而，在计算过程中，我们对序列中的下一个项添加了部分和。说明这不会使情况恶化。

这里的教训是，单调（或接近单调）的数列应该从小到大相加，因为如果要加的量的大小比较接近，误差就最小。请注意，这与减法的情况相反，涉及类似数量的操作会导致较大的误差。这意味着，如果一个应用要求对数列进行加减运算，而我们预先知道哪些项是正数，哪些项是负数，那么相应地重新安排算法可能会有收获。

✍ **练习 3.23** 正弦函数定义为

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$
$$= \sum_{i \geq 0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

下面是两个计算这个和的代码片段（假设给定了 $x$ 和 $n$ 个项）。

```
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);
```

- 解释一下，如果你计算 $x > 1$ 的大量项会发生什么。
- 对于大量的术语，这两种代码是否有意义？
- 是否有可能从最小的项开始对其进行求和？
- 你能提出其他方案来改进 $sin(x)$ 的计算吗？

### 3.4.3 不稳定的算法

现在我们将考虑一个例子，在这个例子中，我们可以直接论证该算法无法应对因不准确表示的实数而引起的问题。

考虑递归 $y_n = \int_0^1 \frac{x^n}{x-5} dx = \frac{1}{n} - 5y_{n-1}$，它是单调递减的；第一个项可以计算为 $y_0 = ln6 - ln5$。

以小数点后 3 位数进行计算，我们得到。

| computation | | correct result |
|---|---|---|
| $y_0 = \ln 6 - \ln 5 = .182 \mid 322 \times 10^1 \ldots$ | | 1.82 |
| $y_1 = .900 \times 10^{-1}$ | | .884 |
| $y_2 = .500 \times 10^{-1}$ | | .0580 |
| $y_3 = .830 \times 10^{-1}$ | going up? | .0431 |
| $y_4 = -.165$ | negative? | .0343 |

我们看到，计算出来的结果很快就不只是不准确，而且实际上是毫无意义的。我们可以分析一下为什么会出现这种情况。

如果我们将 $n$ 在第 $n$ 步中的误差 $\epsilon_n$ 定义为：

$$\tilde{y}_n - y_n = \epsilon_n,$$

那么

$$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5n_{n-1} + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1}$$

于是 $\epsilon_n \geqslant 5\epsilon_{n-1}$. 这种计算所产生的误差呈现指数式增长。

### 3.4.4 线性系统求解

有时我们甚至可以在不指定使用何种算法的情况下对问题的数值精度做出说明。假设我们想解决一个线性系统，也就是说，我们有一个 $n \times n$ 矩阵和一个大小为 $n$ 的向量 $b$，我们想计算出使 $Ax = b$ 的向量。(由于向量 $b$ 将是某种计算或测量的结果，我们实际上是在处理一个向量 $\tilde{b}$，它是理想 $b$ 的某种扰动。

$$\tilde{b} = b + \Delta b.$$

扰动向量 $\Delta b$ 可以是机器精度的数量级，如果它仅仅来自于代表误差。

扰动向量 $\Delta b$ 可以是机器精度的数量级，如果它仅仅来自于代表误差，或者它可以更大，这取决于产生 $\tilde{b}$ 的计算。

我们现在要问的是 $x$ 的精确值与计算值之间的关系，前者是通过对 $A$ 和 $b$ 进行精确计算得到的，而后者是通过对 $A$ 和 $\tilde{b}$ 进行计算得到的。(在讨论中我们将假设 $A$ 本身是精确的，但这是一种简化)。

写作 $\tilde{x} = x + \Delta x$，我们的计算结果现在是

$$A\tilde{x} = \tilde{b}$$

或者

$$A(x + \Delta x) = b + \Delta b.$$

由于 $Ax = b$，我们得到 $A\Delta x = \Delta b$。由此，我们可以得到（详见附录 13)。

$$\left\{\begin{array}{l} \Delta x = A^{-1}\Delta b \\ Ax = b \end{array}\right\} \Rightarrow \left\{\begin{array}{ll} \|A\|\|x\| & \geqslant \|b\| \\ \|\Delta x\| & \leqslant \left\|A^{-1}\right\| \Delta b\| \end{array}\right. \Rightarrow \frac{\|\Delta x\|}{\|x\|} \leqslant \|A\| \left\|A^{-1}\right\| \frac{\|\Delta b\|}{\|b\|}$$

$\|A\|\|A^{-1}\|$ 的数量被称为矩阵的条件数。边界指的是，任何右手边的扰动都会导致解决方案的扰动，该扰动最多只能大于矩阵的条件数 $A$。请注意，这并不是说 x 的扰动必须接近这个大小，但我们不能排除这个可能性，而且在某些情况下，确实可以达到这个界限。

假设 $b$ 是精确到机器精度的，并且 $A$ 的条件数是 $10^4$。边界通常被解释为：$x$ 的最后 4 位数字是不可靠的，或者说，计算" 失去了 4 位数字的准确性"。

方程也可以解释为：当我们解决一个线性系统 $Ax = b$ 时，我们得到一个近似解 $x + \Delta x$，这是一个扰动系统 $Ax + \Delta x = b + \Delta b$ 的精确解。解中的扰动可以与系统中的扰动相关，这一事实可以通过说该算法表现出逆向稳定性来表达。

线性代数算法的精度分析本身就是一个研究领域；例如，见 Higham 的书 [106]。

### 3.4.5 并行计算中的舍入误差

计算机算术中的加法不是关联的。一个类似的事实也适用于乘法。这对并行计算来说有一个有趣的结论：计算在并行处理器上的分布方式会影响结果。

作为一个简单的例子，考虑计算总数 $a + b + c + d$。在单个处理器上，普通执行对应于以下关联性。

$$((a+b)+c)+d$$

另一方面，将这个计算分散到两个处理器上，其中处理器 0 有 a，b，处理器 1 有 c，d，相当于

$$((a+b)+(c+d))$$

推而广之，我们看到，在不同数量的处理器上，规约操作很可能会得到不同的结果。(MPI 标准规定，在同一组处理器上运行的两个程序应该得到相同的结果)。有可能规避这个问题，用对所有处理器的集合操作来代替还原操作，然后再进行局部规约。然而，这增加了处理器的内存需求。

对于并行求和问题，还有一个有趣的解决方案。如果我们用 4000 比特的尾数来存储浮点数，就不需要指数，这样存储的数字的所有计算都是精确的，因为它们是定点计算的一种形式 [129, 128]。虽然用这样的数字做整个应用是非常浪费的，但只为偶尔的内积计算保留这种方案可能是解决可重复性问题的办法。

## 3.5 编程语言中的计算机运算

不同的语言有不同的方法来声明整数和浮点数。这里我们研究一些问题。

### 3.5.1 Fortran

在 Fortran 中，变量声明可以采取各种形式。例如，一个类型标识符有可能声明存储一个变量所需的字节数 integer2, real8。这种方法的一个优点是容易与其他语言或 MPI 库互操作。

通常情况下，可以只用 INTEGER、REAL 来写代码，用编译器标志来表示整数和实数的字节数大小。

更复杂的、现代版本的 Fortran 可以指出一个浮点数需要有多少位的精度。

```
integer, parameter :: k9 = selected_real_kind(9)
real(kind=k9) :: r
r = 2._k9; print *, sqrt(r) ! prints 1.4142135623730
```

kind 值通常为 4,8,16，但这取决于编译器。

C99 和 Fortran2003 最近的 C 语言和 Fortran 语言的标准包含了 C/Fortran in-teroperability 标准，它可以用来声明一种语言的类型，使其与另一种语言的某种类型兼容。

### 3.5.2 C

在 C 语言中，常用的类型标识符并不对应于一个标准的长度。对于整数来说，有 short int、int、long int，而对于浮点 float 来说，有 double。sizeof() 操作符给出了用于存储一个数据类型的字节数。

C 整数的数值范围在 limit.h 中定义，通常给出一个上限或下限。例如，INT_MAX 被定义为 32767 或更大。

浮点类型在 float.h 中指定。

C 语言中存在指定的存储类型：常数如 int64_t 是由 stdint.h 中的 typedef 定义的。

常数 NAN 是在 math.h 中声明的。对于检查一个值是否为 NaN，可以使用 isan()。

### 3.5.3 Printing bit patterns

```
// 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);
  }
}
```

用作：

```
// bits.c
int five = 5;
printf("Five=%d, in bits: ",five);
printBits(sizeof(five),&five);
printf("\n");
```

### 3.5.4 C++

C++ 语言有以下浮点类型。

- float：这通常是作为 IEEE 754 32 位浮点数实现的。
- double：定义为至少和浮点数一样精确，通常实现为 IEEE 754 的 64 位浮点数。
- long double：这也被定义为至少和 double 一样精确。在一些架构上，它可以是 80 位的扩展精度，在其他架构上则是全 128 位的精度。处理器通常通过软件和硬件功能的结合来实现后者，所以性能会比前两种类型低很多。

#### 3.5.4.1 边界

你仍然可以使用 C 头的 limit.h 或 climits，但最好使用 std::numeric_limits，它在类型上是模板化的。比如说

```
std::numerical_limits<int>.max();
```

有以下几种功能。

- std::numeric_limits<T>::max() for the largest number.
- std::numeric_limits<T>::min() for the smallest normalized positive number.
- std::numeric_limits<T>::lowest() for the most negative number.
- std::numeric_limits<T>::epsilon() for machine epsilon.
- std::numeric_limits<T>::denorm_min() for smallest subnormal. (See also std::numeric_limits<T >::has_denorm.)
- std::nextafter(x,y)

#### 3.5.4.2 例外的情况

定义的例外情况。

- FE_DIVBYZERO pole error occurred in an earlier floating-point operation.
- FE_INEXACT inexact result: rounding was necessary to store the result of an earlier floating-point operation.
- FE_INVALID domain error occurred in an earlier floating-point operation.
- FE_OVERFLOW the result of the earlier floating-point operation was too large to be representable.
- FE_UNDERFLOW the result of the earlier floating-point operation was subnormal with a loss of precision.
- FE_ALL_EXCEPT bitwise OR of all supported floating-point exceptions .

用法：

```
std::feclearexcept(FE_ALL_EXCEPT);
if(std::fetestexcept(FE_UNDERFLOW)) {/* ... */}
```

在C++中，std::numeric_limits<double>::quiet_NaN() 是在 limit 中声明的，如果 std::numeric_limits::has_quiet_NaN 为真，这就是有意义的，如果 std::numeric_limits::is_iec559 为真。(ICE 559 本质上是 IEEE 754)。

同一模块还有 infinity() 和 signaling_NaN()。

对于检查一个值是否为 NaN，可以使用 C++ 中 cmath 的 std::isan()。请进一步参阅 http://en.cppreference.com/w/cpp/numeric/m

### 3.5.4.3 例外情况

IEEE 754 标准和 C++ 语言都定义了一个例外的概念，这两个概念是相互不同的。754 例外是指" 没有适合每个合理应用的结果" 的操作的发生。这不一定能转化为语言定义的异常。

### 3.5.4.4 打印位元模式

```
// bitprint.cxx
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());
    //codesnippet cppbitprint
    return 0;
}
```

## 3.5.5 程序设计中的舍入行为

从上面的讨论中可以看出，一些对数学实数成立的简单说法在浮点数上并不成立。例如，在浮点运算中

$$(a+b)+c \neq +(b+c)$$

这意味着编译器不能在不影响取舍行为的情况下进行某些优化。在一些代码中，这种轻微的差异是可以被容忍的，例如，因为方法有内置的保护措施。例如，静止迭代方法就能抑制任何引入的错误。

另一方面，如果程序员在编写代码时考虑到了舍入行为，那么编译器就没有这样的自由了。这在上面的练习 3.10 中有所暗示。我们用价值安全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，编译器是不允许做任何影响计算结果的改变的。

编译器通常有一个选项，控制是否允许优化，以改变数值行为。对于英特尔的编译器，它是-fp-model=....。另一方面，像-Ofast 这样的选项只是为了提高性能，可能会严重影响数值行为。对于 Gnu 编译器来说，完全符合 754 标准的选项是-frounding-math，而-ffast-math 则允许以性能为导向的编译器转换，这违反了 754 和/或语言标准。

如果你关心结果的可重复性，这些问题也很重要。如果一个代码被两个不同的编译器编译，用相同的输入运行时，应该有相同的输出？如果一个代码在两个不同的处理器配置上并行运行？这些问题是非常微妙的。在第一种情况下，人们有时会坚持位数的可重复性，而在第二种情况下，只要结果保持" 科学" 上的等价，一些差异是允许的。当然，这个概念是很难做到严格的。

下面是在考虑编译器对代码行为和重现性的影响时的一些相关问题。

左到右的评估，所以重新关联实际上是标准所不允许的。Fortran 语言标准没有这样的规定，但是编译器必须尊重小括号所暗示的评估顺序。

重新关联的一个常见来源是循环展开。在严格的值安全条件下，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，以及是否进行展开，都取决于编译器的优化水平、编译器的选择和目标平台。

重新关联的一个更微妙的来源是并行执行。这意味着代码的输出在不同的并行配置上的两次运行之间不需要严格的重现。

**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

```
floaat one = 1, ;
...
x = 2. + y + one;
```

编译器将赋值改为 $x = y + 3$。然而，这违反了上面的重新关联规则，而且它忽略了任何动态设置的四舍五入行为。

**表达式评估**在评估表达式 $a + (b + c)$ 时，处理器会产生一个中间结果为 $b + c$，这个结果没有分配给任何变量。许多处理器能够分配一个更高的中间结果精度。编译器可以有一个标志来决定是否使用这种设施。

**浮点单元的行为**舍入行为（截断与舍入）和渐进下溢的处理可由库函数或编译器选项控制。

**库函数** IEEE 754 标准只规定了简单的操作；目前还没有处理正弦或对数函数的标准。因此，它们的实现可能是一个变化的来源。更多的讨论，见 [144]。

### 3.5.6 改变舍入行为

IEEE 754 标准还声明，一个处理器应该能够在普通四舍五入、向上或向下四舍五入（有时分别表述为" 向正无穷大" 和" 向负无穷大"）或截断之间切换其四舍五入行为。在 C99 中，这个 API 包含在 fenv.h 中（或者对于 C++ cfenv）。

```
#include <fenv.h>
int roundings[] = {FE_TONEAREST, FE_UPWARD, FE_DOWNWARD, FE_TOWARDZERO};
rchoice = ....
int status = fesetround(roundings[rchoice]);
```

在 Fortran2003 中，函数 IEEE_SET_ROUNDING_MODE 在 IEEE_ARITHMETIC 模块中可用。设置四舍五入行为可以作为一种快速测试算法稳定性的方法：如果结果在两种不同的四舍五入策略之间有明显的变化，那么该算法很可能不稳定。

如果结果在两种不同的四舍五入策略之间有明显的变化，那么该算法可能是不稳定的。

### 3.5.7 捕捉异常情况

异常这个词有几种含义。

- 浮点异常是指" 无效数字" 的发生，比如通过溢出或除以零。
- 如果发生任何类型的意外事件，编程语言可以" 抛出异常"，也就是中断正常的程序控制流程。

溢出时的行为也可以被设置为产生一个异常。在 C 语言中，你可以用一个库调用来指定。

```
#include <fenv.h>
int main() {
   ...
   feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW);
```

#### 3.5.7.1 编译器特定行为

捕获异常有时可以由编译器指定。例如，gcc 编译器可以通过标志-ffpe-trap=list 来捕获异常；见 https://gcc.gnu.org/onlinedocs/Options.html。

## 3.6 更多关于浮点运算的内容

### 3.6.1 Kahan 的总结

我们看到了计算机算术的一些问题：四舍五入会导致结果相当错误，而且非常依赖于评估顺序。有一些算法试图弥补这些问题，特别是在加法的情况下。我们简单讨论一下以 William Kahan 命名的 Kahan 求和法 [117]，它是补偿求和算法的一个例子。

$$
\begin{aligned}
&\text{sum} \leftarrow 0\\
&\text{correction} \leftarrow 0\\
&\text{while there is another input do}\\
&\text{oldsum} \leftarrow \text{sum}\\
&\text{input} \leftarrow \text{input} - \text{correction}\\
&\text{sum} \leftarrow \text{oldsum} + \text{input}\\
&\text{correction} \leftarrow (\text{sum} - \text{oldsum}) - \text{input}
\end{aligned}
$$

✍ **练习 3.24** 通过 3.4.5 节中的例子，增加最后一项 3；即在该例子的条件下计算 4+6+7+3 和 6+7+4+3。表明当 17 被四舍五入为 20 时，修正值正好是 3 的下限，或者当 14 被四舍五入为 10 时，修正值是 4 的过限；在这两种情况下，都能计算出 20 的正确结果。

### 3.6.2 其他计算机运算系统

有人提出了其他系统来处理计算机上的不精确算术问题。一个解决方案是扩展精度算术，即以比平常更多的比特存储数字。这方面的一个常见用途是计算向量的内积：在内部以扩展精度进行累加，但以普通浮点数返回。另外，还有一些库，如 GMPlib [77]，允许以更高的精度进行任何计算。

另一个解决计算机算术不精确问题的方法是" 区间算术"[114]，即对每个计算都保持区间界限。虽然这种方法已经研究了相当长的时间，但除了通过专门的库 [21] 外，它并没有实际使用。

### 3.6.3 扩展精度

在制定 IEEE 754 标准时，人们设想处理器可以有一系列的精度。在实践中，只有单精度和双精度的定义被使用。然而，有一个扩展精度的例子仍然存在。英特尔处理器有 80 位寄存器用于存储中间结果。(这可以追溯到英特尔 80287 协处理器。) 这种策略在 FMA 指令和内积的积累中是有意义的。

这些 80 位寄存器有一个奇怪的结构，有一个显著的整数位，可以产生不是任何定义数字的有效表示的位模式 [171]。

### 3.6.4 降低精度

你可以问" 双精度是否总是比单精度有好处"，答案并不总是" 是"，而是：" 这取决于"。

#### 3.6.4.1 迭代细化中的低精度

在迭代线性系统求解中（第 5.5 节，精度是由计算残差的精度决定的，而不是由求解步骤的精度决定的。因此，人们可以在降低精度的情况下进行操作，如应用预处理程序（第 5.5.6 节）[1]。这是一种迭代细化的形式。

#### 3.6.4.2 深度学习中的低精度

IEEE 754-2008 有一个二进制 16 半精度格式的定义，它有一个 5 位指数和 11 位尾数。

在深度学习（DL）中，表达值的范围比精确的值更重要。(这与传统的科学应用相反，在传统的科学应用中，接近的数值需要重新解决)。这导致了 bfloat16 " 大脑浮点" 格式的定义 https://en.wikipedia.org/wiki/Bfloat16_floating-point_format 这是一种 16 位的浮点格式。它使用 8 位作为指数，7 位作为尾数。这意味着它与 IEEE 单精度格式共享相同的指数范围；见图 3.3。

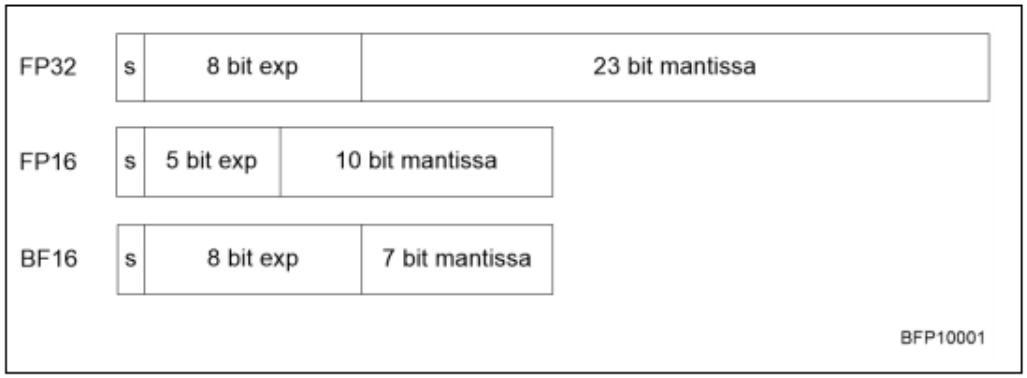



图 3.3: fp32、fp16 和 bfloat16 格式的比较。(插图来自 [35])

- 由于 bfloat16 和 fp32 的前两个字节结构相同，通过截断第三和第四字节，可以从 fp32 的数字中得出 bfloat16 的数字。然而，在实践中，四舍五入可能会得到更好的结果。
- 相反，将一个 bloat16 转换成 fp32 只需要在最后两个字节中填充 0。

bfloat16 的有限精度可能足以代表 DL 应用中的数量，但为了不失去更多的精度，我们设想 FMA 硬件在内部使用 32 位数：两个 bfloat16 数字的乘积是一个常规的 32 位数。为了计算内积（作为 DL 中矩阵-矩阵乘法的一部分），我们需要一个 FMA 单元，如图 3.4 所示。

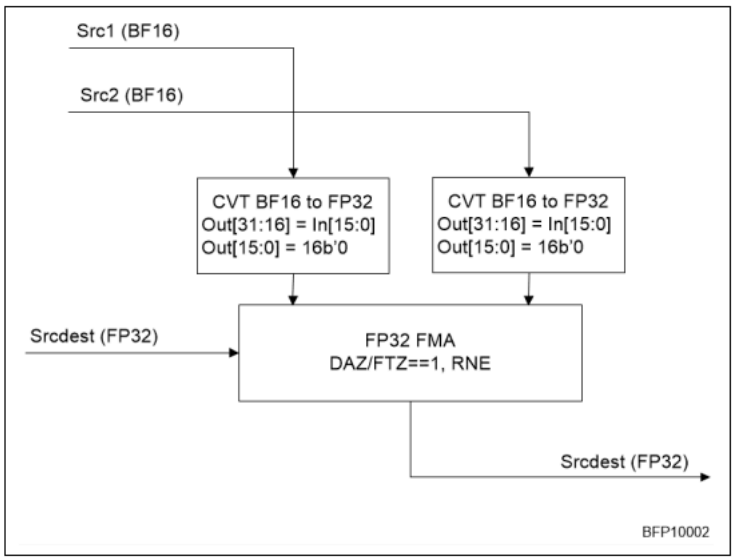



图 3.4: 一个 FMA 单元取两个 bloat16 和一个 fp32 数字。(插图来自 [35])

甚至在 [46] 中讨论了进一步减少到 8 位。

### 3.6.5 定点运算

一个定点数（比这里更深入的讨论，见 [206]）可以表示为 $< N, F >$ 其中 $N \geqslant \beta_0$ 是整数部分，$F < 1$ 是小数部分。另一种说法是，定点数字是以 $N + F$ 位数存储的整数，在第一个 $F$ 位数后隐含小数点。

固定点计算可能会溢出，没有可能调整指数。考虑乘法 $< N_1, F_1 > \times < N_2, F_2 >$，其中 $N_1 \geqslant \beta^{n_1}$，$N_2 \geqslant \beta^{n_2}$。如果 $n_1 + n_2$ 超过了整数部分的可用位置数，则溢出。(非正式地，乘积的位数是操作数的位数之和)。这意味着在使用定点运算的程序中，如果要进行乘法运算，需要有一定数量的前导零位，这就降低了数字的准确性。这也意味着程序员必须更努力地思考计算问题，以保证不会发生溢出，并在合理范围内保持数字的准确性。那么，人们为什么要使用定点数字呢？其中一个重要的应用是嵌入式低功耗设备，比如电池供电的数字温度计。由于定点计算与整数计算基本相同，因此不需要浮点运算单元，从而降低了芯片尺寸，减少了对功耗的需求。另外，许多早期的视频游戏系统的处理器要么没有浮点单元，要么整数单元比浮点单元快得多。在这两种情况下，使用整数单元将非整数计算变成定点计算，是实现高吞吐量的关键。

另一个仍然使用定点运算的领域是信号处理。在现代 CPU 中，整数和浮点运算的速度基本相同，但它们之间的转换相对较慢。现在，如果正弦函数是通过查表实现的，这意味着在 $sin(sinx)$ 中，一个函数的输出被用来索引下一个函数的应用。显然，以定点方式输出正弦函数就不需要在实数和整数之间进行转换，这就简化了所需的芯片逻辑，并加快了计算速度。

### 3.6.6 复数

有些编程语言将复数作为一种内置的数据类型，有些则不是，还有一些则介于两者之间。例如，在 Fortran 中可以声明

```
COMPLEX z1,z2, z(32)
COMPLEX*16 zz1, zz2, zz(36)
```

一个复数由一对实数组成，分为实部和虚部，在内存中相邻分配。第一个声明用 8 个字节来存储 REAL *4 的数字，第二个声明用 REAL*8 来存储实部和虚部。(另外，第二行使用 DOUBLE COMPLEX 或在 Fortran90 中使用 COMPLEX(KIND=2))。

相比之下，C 语言没有直接的复数，但是 C99 和 C++ 都有一个复数.h 头文件。

complex.h 头文件。这就像 Fortran 中定义复数一样，定义为两个实数。

像这样存储一个复数是很容易的，但有时在计算上并不是最好的解决方案。当我们研究复数的数组时，这就变得很明显了。如果一个计算经常完全依赖于对复数实部（或虚部）的访问，那么在复数数组中跨步，就会有一个跨步二，这是很不利的。在这种情况下，最好为实部分配一个数组，为虚部分配另一个数组。

✍ **练习 3.25** 假设复数数组是以 Fortran 方式存储的。分析对数组相乘的内存访问模式，即 $\forall_i c_i \leftarrow a_i b_i$，其中 a(), b(), c() 是复数的数组。

✍ **练习 3.26** 说明复数上的 $n \times n$ 线性系统 $Ax = b$ 可以写成实数上的 $2n \times 2n$ 系统。提示：将矩阵和向量分成实部和虚部。论证复数数组的实部和虚部分开存储的效率。

## 3.7 结论

在计算机上进行的计算无一例外地存在着数字错误。有些时候错误的原因是计算机运算的不完善：如果我们能用实际的实数进行计算，就不会有问题。(仍然会有数据的测量误差和数值方法中的近似问题；见下一章)。然而，如果我们接受四舍五入作为生活中的一个事实，那么各种观察就会成立。

- 从稳定性的角度来看，数学上的等价运算不需要表现得完全一样；见"abc 公式" 的例子。
- 即使是相同计算的重新排列也不会有相同的表现；请看求和的例子。

因此，必须分析计算机算法的舍入行为：舍入是否作为问题参数的一个缓慢增长的函数而增加，比如被演算的项的数量，或者有可能出现更糟糕的行为？我们不会在本书中进一步详细讨论这些问题。

## 3.8 复习题

✍ **练习 3.27** 判断真假

- 对于整数类型，" 最负" 的整数是" 最正" 的负数。
- 对于浮点类型，" 最负" 的数字是" 最正" 的数字的负数。
- 对于浮点类型，最小的正数是最大的正数的倒数。

# 第 4 章 微分方程的数值处理

内容提要

- 初值问题
- 初始边界值问题
- 边界值问题

在这一章中，我们将研究常微分方程（ODEs）和偏微分方程（PDEs）的数值解。这些方程在物理学中常用来描述一些现象，如飞机周围的空气流动，或桥梁在各种压力下的弯曲。虽然这些方程通常相当简单，但从它们中得到具体的数字（" 如果有一百辆汽车在上面，这座桥会下垂多少"）则比较复杂，往往需要大型计算机来产生所需的结果。这里我们将描述将常微分方程和偏微分方程转化为可计算问题的技术。

我们将首先介绍**初值问题**（IVPs），它描述了随着时间变换的过程。此处我们仅考虑常微分方程：只依赖于时间的标量函数。接下来，我们将研究描述空间过程的**边界值问题**（BVPs）。边界值问题通常涉及到多个空间变量，因此我们可以得到偏微分方程。

最后，我们将考虑" 热方程"，这是一个**初始边界值问题**（IBVP），它同时具有 IVP 和 BVP 的特点：它描述了热量在一个物理物体（如杆）上的传播。初始值描述了初始温度，而边界值给出了杆两端的规定温度。

我们在这一章的目的是展示一类重要的计算问题的起源。因此，我们不会去讨论解的存在性、唯一性或条件的理论问题。关于这一点，见 [99] 或任何专门讨论常微分方程或偏微分方程的书。为了便于分析，我们还将假设所有涉及的函数都存在任意高阶导数，并且处处光滑。

## 4.1 初值问题

许多物理现象随着时间的推移而变化，通常物理学定律给出了对变化的描述，而不是对感兴趣的数量本身的描述。例如，牛顿第二定律

$$F = ma$$

是一个关于点质量的位置变化的公式：表示为

$$a(t) = \frac{d^2}{dt^2}x(t) = F/m$$

它指出，加速度线性地取决于施加在质量上的力。对于质量的位置，有时可以通过分析得出一个封闭式的描述 $x(t) = \ldots$，但在许多情况下需要某种形式的近似或数值计算。这也被称为" **数值积分**"。

牛顿第二定律（4.1）是一个 ODE，因为它描述了一个变量的函数，即时间。它是一个 IVP，因为它描述了从一些初始条件开始的时间发展。作为一个 ODE，它是" 二阶" 的，因为它涉及到一个二阶导数，如果我们允许矢量，我们可以把它减少到一阶，只涉及一阶导数。我们引入一个双分量矢量 $u$，结合位置 $x$ 和速度 $x$。

$$u(t) = (x(t), x'(t))^t$$

以 $u$ 表示的牛顿方程就变成了。

$$u' = Au + B, \quad A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ F/a \end{pmatrix}$$

为了简单起见，在本课程中我们将只考虑标量方程；那么我们的参考方程是

$$u'(t) = f(t, u(t)), \quad u(0) = u_0, \quad t > 0$$

方程（4.2）允许过程有明确的时间依赖性，但一般来说，我们只考虑没有这种明确依赖性的方程，即所谓的" **自治**（autonomous）" 常微分方程，其形式为

$$u'(t) = f(u(t))$$

其中右手边并不明确地依赖于 $t$。

**注释** 非自治偏微分方程可以转化为自治偏微分方程，所以这并不是什么限制。如果 $u = u(t)$ 是一个标量函数，并且 $f = f(t,u)$，我们定义 $u_2(t) = t$，并考虑等同的自治系统 $\begin{pmatrix} u' \\ u_2' \end{pmatrix} = \begin{pmatrix} f(u_2,u) \\ 1 \end{pmatrix}$

通常情况下，在某个起点（通常选择 $t = 0$）的初始值是给定的：$u(0) = u_0$，对于某个值 $u_0$，我们对 $u$ 随着 $t \to \infty$ 的行为感兴趣。举例来说，$f(x) = x$ 给出的方程是：$u(u) = u(t)$。这是一个简单的人口增长模型：该方程指出，增长速度等于人口规模。对于某些 $f$ 的选择，方程（4.2）可以用分析法解决，但我们不会考虑这个问题。相反，我们只考虑数值解和这个过程的准确性。

在数值方法中，我们考虑用离散大小的时间步长来近似解决连续的时间依赖过程。由于这引入了一定量的误差，我们将分析在每个时间步长中引入的误差时相互收敛，以及这些误差是如何叠加成一个整体误差的。在某些情况下，限制全局误差的需要会对数值方案施加限制。

### 4.1.1 误差和稳定性

由于机器运算会产生不精确性，我们希望尽量避免因为初始值的小扰动而造成的干扰。因此，如果与不同初值 $u_0$ 相关的解在 $t \to \infty$，我们将称微分方程为 **稳定**的。

稳定性的一个充分标准是

$$\frac{\partial}{\partial u} f(u) = \begin{cases} > 0 & \text{unstable} \\ = 0 & \text{neutrally stable} \\ < 0 & \text{stable} \end{cases}$$

**证明** 如果 $u$ 是 $f$ 的零点，即 $f(u) = 0$，那么常数函数 $u(t) \equiv u$ 就是 $u = fu$ 的一个解，即所谓' 平衡' 解。现在我们将考虑从平衡状态出发的小扰动是如何表现的。让 $u$ 是偏微分方程的一个解，写成 $u(t) = u + \eta(t)$，那么我们有

$$\begin{aligned} \eta' = u' = f(u) = f(u^* + \eta) = f(u^*) + \eta f'(u^*) + O\left(\eta^2\right) \\ = \eta f'(u^*) + O\left(\eta^2\right) \end{aligned}$$

忽略二阶项，存在一个解：

$$\eta(t) = e^{f'(x^\star)t}$$

这意味着，如果 $f(x) < 0$，扰动将被阻尼，如果 $f(x) > 0$，扰动将被放大。

我们经常会提到简单的例子 $f(u) = -\lambda u$，其解 $u(t) = u_0 e^{-\lambda t}$。如果 $\lambda > 0$，这个问题是稳定的。

### 4.1.2 有限差分近似法：欧拉显式和隐式方法

为了数值解决这个问题，我们通过研究有限时间/空间步长，将连续问题变成离散问题。假设所有的函数都足够平滑，一个简单的泰勒级数展开（见附录 16）就可以得到。

$$u(t + \Delta t) = u(t) + u'(t)\Delta t + u''(t)\frac{\Delta t^2}{2!} + u'''(t)\frac{\Delta t^3}{3!} + \cdots$$

这就得到了 $u'$

$$\begin{aligned} u'(t) &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + \frac{1}{\Delta t}\left(u''(t)\frac{\Delta t^2}{2!} + u'''(t)\frac{\Delta t^3}{3!} + \cdots\right) \\ &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + \frac{1}{\Delta t} O\left(\Delta t^2\right) \\ &= \frac{u(t+\Delta t) - u(t)}{\Delta t} + O(\Delta t) \end{aligned}$$

如果所有的导数都是有界的，我们可以用一个 $O(\Delta t_2)$ 来近似高阶导数的无限之和。

或者，我们可以证明这个和等于 $\Delta t^2 u''(t+\alpha\Delta t)$ 当 $0<\alpha<1$

我们看到，我们可以通过有限差分来近似微分算子，其误差是已知的，其数量级是时间步长的函数。

将其代入 $u=f(t,u)$，得到

$$\frac{u(t+\Delta t)-u(t)}{\Delta t}=f(t,u(t))+O(\Delta t)$$

或者

$$u(t+\Delta t)=u(t)+\Delta t f(t,u(t))+O\left(\Delta t^2\right)$$

**注释** 前面的两个方程是数学上的等式，不应该被理解为对一个给定的函数 $u$ 进行计算的方法。回顾前面的讨论，你可以看到这样的公式对于小 $\Delta t$ 来说很快就会被取消。关于数值微分的进一步讨论超出了本书的范围，请参见任何标准的数值分析教科书。

我们现在使用上述方程来推导一个数值方案:: 在 $t_0=0t_{k+1}=t_k+\Delta t=\cdots=(k+1)\Delta t$ 的情况下，我们得到一个差分方程

$$u_{k+1}=u_k+\Delta t f(t_k,u_k)$$

为 $u_k$ 量，我们希望 $u_k$ 将是对 $ut_k$ 的良好近似。这就是所谓的" 显式欧拉" 或" 欧拉正向" 方法。

从微分方程到差分方程的过程通常被称为**离散化**（discretization），因为我们只在离散的点集中计算函数值。计算的数值本身仍然是实值的。另一种说法是：如果我们计算 $k$ 个时间步长，就可以在有限的二维空间 $\mathbb{R}k$ 中找到数值解。原问题的解是在 $\mathbb{R}\to\mathbb{R}$ 的函数空间中找到的。

在上面，我们用一个算子逼近另一个算子，这样做的时候，当 $\Delta t\downarrow 0$ 时，截断误差为 $O(\Delta t)$（见附录 14 对这个数量级的符号的更正式介绍）。这并不意味着差分方程计算出的解就接近于真解。为此，还需要进行一些分析。

我们从分析" 局部误差" 开始：如果我们假设计算出的解在 $k$ 步骤是精确的，即 $u_k=u(t_k)$，那么 $k+1$ 步时将会出现怎样的错误？我们有

$$\begin{aligned}u\left(t_{k+1}\right)&=u\left(t_k\right)+u'\left(t_k\right)\Delta t+u''\left(t_k\right)\frac{\Delta t^2}{2!}+\cdots\\&=u\left(t_k\right)+f\left(t_k,u\left(t_k\right)\right)\Delta t+u''\left(t_k\right)\frac{\Delta t^2}{2!}+\cdots\end{aligned}$$

和

$$u_{k+1}=u_k+f(t_k u_k)\Delta t$$

于是

$$\begin{aligned}L_{k+1}&=u_{k+1}-u\left(t_{k+1}\right)=u_k-u\left(t_k\right)+f\left(t_k,u_k\right)-f\left(t_k,u\left(t_k\right)\right)-u''\left(t_k\right)\frac{\Delta t^2}{2!}+\cdots\\&=-u''\left(t_k\right)\frac{\Delta t^2}{2!}+\cdots\end{aligned}$$

这表明，在每一步中，我们的误差为 $O(\Delta t^2)$。如果我们假设这些误差可以相加，我们发现全局误差为

$$E_k\approx\Sigma_k L_k=k\Delta t\frac{\Delta t^2}{2!}=O(\Delta t)$$

由于全局误差在 $\Delta t$ 中是一阶的，我们称之为" 一阶方法"。需要注意的是，这个误差（衡量真实解和计算解之间的距离）与截距误差（即算子的近似误差）是同阶的 $O(\Delta t)$。

#### 4.1.2.1 欧拉显式方法的稳定性

考虑 IVP $u=f(t,u)$ 对于 $t\geqslant 0$，其中 $f(t,u)=-\lambda u$，给出初始值 $u(0)=u_0$。存在一个精确的解，即 $u(t)=u_0e^{-\lambda t}$。从上面的讨论中可知这个问题是稳定的，也就是说，如果 $\lambda>0$，解的小扰动最终会被抑制。现在我们将研究数值解的表现是否与精确解相同，也就是说，数值解是否也收敛于零。

这个问题的欧拉正向或显式欧拉方案是

$$u_{k+1} = u_k - \Delta t \lambda u_k = (1 - \lambda \Delta t) u_k \Rightarrow u_k = (1 - \lambda \Delta t)^k u_0$$

为了稳定，我们要求 $u_k \to 0$，因为 $k \to \infty$。这就相当于

$$\begin{aligned} u_k \downarrow 0 &\Leftrightarrow |1 - \lambda \Delta t| < 1 \\ &\Leftrightarrow -1 < 1 - \lambda \Delta t < 1 \\ &\Leftrightarrow -2 < -\lambda \Delta t < 0 \\ &\Leftrightarrow 0 < \lambda \Delta t < 2 \\ &\Leftrightarrow \Delta t < 2/\lambda \end{aligned}$$

我们看到，数值求解方案的稳定性取决于 $\Delta t$ 的值：只有当 $\Delta t$ 足够小时，该方案才是稳定的。为此，我们称显式 Euler 方法为条件稳定的。请注意，微分方程的稳定性和数值方案的稳定性是两个不同的问题。如果 $\lambda > 0$，连续问题是稳定的；数值问题有一个额外的条件，取决于所用的离散化方案。

请注意，我们刚刚进行的稳定性分析是专门针对微分方程 $u = -\lambda u$ 的。如果你要处理的是不同的 IVP，你必须进行单独的分析。然而，你会发现，显式方法通常会给出条件稳定性。

#### 4.1.2.2 欧拉隐式方法

你刚才看到的显式方法很容易计算，但条件稳定性是一个潜在的问题。例如，它可能意味着时间步骤的数量将是一个限制性因素。有一个替代显式方法的方法，不会受到同样的反对。

与其扩展 $u(t + \Delta t)$，不如考虑以下对 $u(t - \Delta t)$ 的扩展。

$$u(t - \Delta t) = u(t) - u'(t)\Delta t + u''(t)\frac{\Delta t^2}{2!} + \cdots$$

这意味着

$$u'(t) = \frac{u(t) - u(t - \Delta t)}{\Delta t} + u''(t)\Delta t/2 + \cdots$$

如前所述，我们取方程 $u(t) = f(t, u(t))$，用差分公式近似计算 $u(t)$。

$$\frac{u(t) - u(t - \Delta t)}{\Delta t} = f(t, u(t)) + O(\Delta t) \Rightarrow u(t) = u(t - \Delta t) + \Delta t f(t, u(t)) + O\left(\Delta t^2\right)$$

我们再次定义固定点 $t_k = kt$，并定义一个数值方案：

$$u_{k+1} = u_k + \Delta t f(t_{k+1, u_{k+1}})$$

其中 $u_k$ 是 $u(t_k)$ 的近似值。

与显式方案的一个重要区别是，$u_{k+1}$ 现在也出现在方程的右侧。也就是说，$u_{k+1}$ 的计算现在是隐含的。例如，让 $f(t, u) = -u^3$，那么 $u_{k+1} = u_k - \Delta t u_{k+1}^3$。换句话说，$u_{k+1}$ 是方程 $\Delta t x = u_k$ 的解。这是一个非线性方程，通常可以用牛顿法求解。

#### 4.1.2.3 隐式欧拉方法的稳定性

让我们再看看这个例子 $f(t, u) = -\lambda u$。用隐式方法计算，可以得到

$$u_{k+1} = u_k - \lambda \Delta t u_{k+1} \Leftrightarrow (1 + \lambda \Delta t) u_{k+1} = u_k$$

那么

$$u_{k+1} = \left(\frac{1}{1 + \lambda \Delta t}\right) u_k \Rightarrow u_k = \left(\frac{1}{1 + \lambda \Delta t}\right)^k u_0$$

如果 $\lambda > 0$，这是一个稳定方程的条件，我们发现，对于所有的 $u_k$ 和 $\Delta t$ 的值，$\lambda \to 0$。这种方法被称为无条件稳定。与显式方法相比，隐式方法的一个优势显然是稳定性：可以采取较大的时间步长而不用担心非物理行为。当然，大的时间步长会使收敛到稳定状态（见附录 15.4）变慢，但至少不会出现发散。

另一方面，隐式方法更加复杂。正如你在上面看到的，它们可能涉及到在每个时间步长中需要解决的非线性系统。在 u 是矢量值的情况下，比如下面讨论的热方程，你会发现隐式方法需要解决一个方程组。

✍ **练习 4.1** 分析以下 IVP $u(x) = f(x)$ 方案的准确性和计算性。

$$u_{i+1} = u_i + h\left(f\left(x_i\right) + f\left(x_{i+1}\right)\right)/2$$

✍ **练习 4.2** 考虑初值问题 $y(t) = y(t)(1-y)$。请注意，$y \equiv 0$ 和 $y \equiv 1$ 是解决方案。这些被称为" 平衡解"。

1. 一个解决方案是稳定的，如果扰动" 收敛到解决方案"，意味着对于 $\epsilon$ 足够小。

$$\text{if} \quad y(t) = \epsilon \quad \text{for some} t, \text{then} \lim_{t\to\infty} y(t) = 0$$

并且

$$\text{if} \quad y(t) = 1 + \epsilon \quad \text{for some} t, \text{then} \lim_{t\to\infty} y(t) = 1$$

这就要求，例如

$$y(t) = \epsilon \Rightarrow y(t) < 0.$$

0 是一个稳定的解决方案吗？ 1 是吗？

2. 考虑显式方法

$$y_{k+1} = y_k + \Delta t y_k(1 - y_k)$$

用于计算微分方程的数值解。说明

$$y_k \in (0,1) \Rightarrow y_{k+1} > y_k, \quad y_k > 1 \Rightarrow y_{k+1} < y_k$$

3. 编写一个小程序来研究在不同的 $\Delta t$ 的选择下数值解的行为。在你提交的作业中包括程序清单和一些运行的情况。
4. 通过运行你的程序，你发现数值解会出现振荡。请对 $\Delta t$ 提出一个条件，使数值解呈单调性。只要说明 $y_k < 1 \Rightarrow y_{k+1} < 1$，以及 $y_k > 1 \Rightarrow y_{k+1} > 1$ 即可。
5. 现在考虑隐式方法

$$y_{k+1} - \Delta t y_{k+1}(1 - y_{k+1}) = y_k$$

并说明 $y_{k+1}$ 可以从 $y_k$ 计算出来。编写一个程序，并研究在不同的 $\Delta t$ 选择下的数值解的行为。
6. 显示出对所有 $\Delta t$ 的选择，隐式方案的数值解都是单调的。

## 4.2 边界值问题

在上一节中，我们看到了初值问题，它模拟的是随时间变化的现象。现在我们将转向边界值问题，一般来说，边界值问题在时间上是静止的，但它描述的是与位置有关的现象。例如，桥梁在负载下的形状，或窗玻璃中的热量分布，因为外面的温度与里面的不同。

二阶一维 BVP 的一般形式是

$$u''(x) = f\left(x, u, u'\right) \text{对于} x \in [a,b] \text{其中} u(a) = u_a, u(b) = u_b$$

但这里我们只考虑简单的形式

$$-u''(x) = f(x) \text{for} x \in [0,1] \text{with} u(0) = u_0, u(1) = u_1$$

在一个空间维度上，或

$$-u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) \text{for} \bar{x} \in \Omega = [0,1]^2 \text{with} u(\bar{x}) = u_0 \text{on} \delta\Omega$$

在两个空间维度上。这里，$\delta\Omega$ 是域 $\Omega$ 的边界。由于我们在边界上规定了 $u$ 的值，这样的问题被称为**边界值**

**问题**（Boundary Value Problem，BVP）。

注释 边界条件可以更普遍，涉及区间端点上的导数。这里我们只看 Dirichlet 边界条件，它规定了域的边界上的函数值。

### 4.2.1 一般性偏微分方程理论

有几种类型的偏微分方程，每种都有不同的数学特性。最重要的属性是影响区域：如果我们对问题进行修补，使解决方案在一个点上发生变化，那么还有哪些点会受到影响。

#### 4.2.1.1 双曲方程

偏微分方程的形式为

$$Au_{xx} + Bu_{yy} + \text{lower order terms} = 0$$

A，B 的符号相反。这样的方程描述的是波，或更一般的对流现象，是保守的，不倾向于稳定状态。

直观地说，在任何一点上改变波浪方程的解，只会改变未来的某些点，因为波有一个传播速度，使得一个点不可能影响到空间上太远的近期的点。这种类型的 PDE 将不会在本书中讨论。

#### 4.2.1.2 抛物线方程

偏微分方程的形式为

$$Au_x + Bu_{yy} + \text{no highter terms in} x = 0$$

并且它们描述了类似于扩散的现象；这些现象往往趋于稳定状态。描述它们的最好方法是考虑在空间和时间的每一点上的解决方案受到空间的每一点上的某个有限区域的影响。

注释 这导致了一个限制 IBVP 时间的条件，即所谓的 Courant-Friedrichs-Lewy 条件（http://en.wikipedia.org/wiki/Courant-Friedrichs-Lewy_condition）。它描述了这样一个概念：在精确问题中，$u(x,t)$ 取决于 $u(x,t-\Delta t)$ 的数值范围；数值方法的时间步长必须小到足以让数值解考虑到所有这些点。

### 4.2.2 一维空间的泊松方程

算子 $\Delta$

$$\Delta u = u_{xx} + u_{yy},$$

是二阶微分算子，方程是二阶 PDE。具体来说，问题是

$$-\Delta u = -u_{xx}(\bar{x}) - u_{yy}(\bar{x}) = f(\bar{x}) \text{for} \bar{x} \in \Omega = [0,1]^2 \text{with} u(\bar{x}) = u_0 \text{on} \delta\Omega$$

被称为**泊松方程**（Poisson equation），定义在单位平方上。二阶 PDEs 相当常见，它们描述了流体和热流以及结构力学中的许多现象。

首先，为了简单起见，我们考虑一维泊松方程

$$-u_{xx} = f(x).$$

我们先考虑二维情况，再进而推广到三维：为了找到一个数值方案，我们像以前一样使用泰勒级数，用 $u(x+h)$ 和 $u(x-h)$ 来表示。

$u$ 及其在 $x$ 的导数。设 $h > 0$，则

$$u(x+h) = u(x) + u'(x)h + u''(x)\frac{h^2}{2!} + u'''(x)\frac{h^3}{3!} + u^{(4)}(x)\frac{h^4}{4!} + u^{(5)}(x)\frac{h^5}{5!} + \cdots$$

以及

$$u(x-h)=u(x)-u'(x)h+u''(x)\frac{h^2}{2!}-u'''(x)\frac{h^3}{3!}+u^{(4)}(x)\frac{h^4}{4!}-u^{(5)}(x)\frac{h^5}{5!}+\cdots$$

我们现在的目标是对 $u''(x)$ 进行近似。我们看到，这些方程中的 $u$ 项在加法下会被抵消，剩下 $2u(x)$。

$$u(x+h)=u(x)+u'(x)h+u''(x)\frac{h^2}{2!}+u'''(x)\frac{h^3}{3!}+u^{(4)}(x)\frac{h^4}{4!}+u^{(5)}(x)\frac{h^5}{5!}+\cdots$$

于是

$$-u''(x)=\frac{2u(x)-u(x+h)-u(x-h)}{h^2}+u^{(4)}(x)\frac{h^2}{12}+\cdots$$

那么，上述数值方案的基础是观察

$$\frac{2u(x)-u(x+h)-u(x-h)}{h^2}=f\left(x,u(x),u'(x)\right)+O\left(h^2\right)$$

这表明我们可以用一个差分算子来近似微分算子，在 $h\downarrow 0$ 时有一个 $O(h^2)$。截断误差为 $h\downarrow 0$。

为了得出一种数值方法，我们将区间 $[0,1]$ 划分为等距的点。$x_k=k_h$ 其中 $h=1/(n+1)k=0...n+1$. 有了这些，有限差分（FD）公式（4.9）导致了一个形成方程组的数值方案。

$$-u_{k+1}+2u_k-u_{k-}=h^2f(x_k)\quad \text{for}k=1,...,n$$

这种使用 FD 公式近似解决 PDE 的过程被称为**有限差分法**（Finite Difference Method，FDM）。

对于大多数的 $k$ 值，这个方程将 $u_k$ 未知数与 $u_{k-1}$ 和 $u_{k+1}$ 的未知数联系起来。例外情况是 $k=1$ 和 $k=n$。在这种情况下，我们记得 $u_0$ 和 $u_{n+1}$ 是已知的边界条件，我们把左边是未知数，右边是已知量的方程写为

$$\begin{cases}-u_{i-1}+2u_i-u_{i+1}=h^2f\left(x_i\right)\\ 2u_1-u_2=h^2f\left(x_1\right)+u_0\\ 2u_n-u_{n-1}=h^2f\left(x_n\right)+u_{n+1}\end{cases}$$

我们现在可以将这些方程总结为 $u_kk=1...n-1$ 的矩阵方程。

$$\begin{pmatrix}2&-1&&\\-1&2&-1&\\&\ddots&\ddots&\ddots\end{pmatrix}\begin{pmatrix}u_1\\u_2\\\vdots\end{pmatrix}=\begin{pmatrix}h^2f_1+u_0\\h^2f_2\\\vdots\end{pmatrix}$$

其形式为 $Au=f$，$A$ 为完全已知矩阵，$f$ 为完全已知向量，$u$ 为未知向量。请注意，右边的向量在第一个和最后一个位置有问题的边界值。这意味着，如果你想用不同的边界条件解决同一个微分方程，只有向量 $f$ 会发生变化。

**练习 4.3** $u(0)=u_0$ 这种类型的条件被称为狄利克雷边界条件。在物理学上，这相当于，例如，知道一个棒端点的温度。其他边界条件也存在。如果我们对流体流动进行建模，并且已知 $x=0$ 时的流出率，那么为导数指定一个值，$u(0)=u_0$，而不是为函数值指定一个值，是合适的。这就是所谓的诺伊曼边界条件。诺伊曼边界条件 $u(0)=u_0$ 可以通过以下方式来模拟

$$\frac{u_0-u_1}{h}=u_0'.$$

证明与狄利克雷边界条件的情况不同，这影响了线性系统的矩阵。

证明在两端都有诺伊曼边界条件会产生一个奇异矩阵，因此线性系统没有唯一的解。(提示：猜测特征值为零的向量)。

在物理学上这是有意义的。例如，在一个弹性问题中，狄利克雷边界条件说明杆子被夹在一定的高度；诺伊曼边界条件只说明它在端点的角度，这使得它的高度无法确定。

让我们列举一些的属性，你会发现这些属性与解决此类方程组有关。

- 矩阵非常稀疏：非零元素的百分比很低，零元素不是随机分布的，而是位于主对角线周围的一个带状结构中。在一般情况下，我们称之为**带状矩阵**（banded matrix），而在这个特定情况下称之为**三对角矩阵**（tridiagonal matrix）。带状结构是典型的 PDEs，但不同应用中的稀疏矩阵可能不太规则。
- 矩阵非对称。这一特性并不涉及由 BVP 分解而来的所有矩阵，但如果没有奇数阶（指第一、第三、第五.....）

导数，例如 $u_x u_{xxx} u_{xy}$。

- 矩阵元素在每个对角线上是恒定的，也就是说，在每一组点 $\{(i,j)i-j=c\}$，对于某个 c。这只对非常简单的问题是正确的。如果微分方程 $\frac{d}{dx}\left(a(x)\frac{d}{dx}u(x)\right)$。如果我们在区间内设置 h 变量，它也不再成立，例如，因为我们想更详细地模拟左端点周围的行为。
- 矩阵元素符合以下符号模式：对角线元素是正的，而非对角线元素是非正的。这一属性取决于所使用的数字方案，但它通常是真实的。连同下面的确定性属性，这被称为 $M-$ 矩阵。关于这些矩阵有一整套数学理论 [12]。
- 矩阵是正定的：$x^t Ax > 0$，适用于所有非零向量 $x$。如果仔细选择数值方案，这一属性将从原始连续问题中继承下来。虽然这一点的用途目前看来并不明确，但以后你会看到取决于它的线性系统的求解方法。

严格地说，方程的解很简单：$u = A^{-1}f$。然而，计算 $A^{-1}$ 并不是找到 $u$ 的最好方法。正如刚才所观察到的，矩阵 $A$ 只有 $3N$ 个非零元素可以存储。另一方面，它的逆矩阵则没有一个非零元素。虽然我们不会证明这一点，但这种说法对大多数稀疏矩阵都是成立的。因此，我们希望以一种不需要储存 $O(n^2)$ 的方式来解决 $Au = f$。

- 你将如何解决这个三对角方程组？证明系数矩阵的 LU 因子化给出了双对角矩阵形式的因子：它们有一个非零对角线和正好一个非零子对角线或超对角线。解决三对角方程组的总操作数是多少？一个向量与这样的矩阵相乘的运算次数是多少？这种关系并不典型！

### 4.2.3 二维空间的泊松方程

上面的一维 BVP 在很多方面是不典型的，特别是与由此产生的线性代数问题有关。在本节中，我们将研究二维泊松问题。你会看到，它构成了一维问题的非微观概括。三维的情况与二维的情况非常相似，所以我们将不讨论它。

上面的一维问题有一个函数 $u = u(x)$，现在变成了二维的 $u = u(x,y)$。那么我们感兴趣的二维问题就是

$$-u_{xx} - u_{yy} = f, \quad (x,y) \in [0,1]^2,$$

其中边界上的数值是给定的。我们通过在 x 和 y 方向上应用方程（4.9）得到我们的离散方程。

$$-u_{xx}(x,y) = \frac{2u(x,y)-u(x+h,y)-u(x-h,y)}{h^2} + u^{(4)}(x,y)\frac{h^2}{12} + \cdots$$
$$-u_{yy}(x,y) = \frac{2u(x,y)-u(x,y+h)-u(x,y-h)}{h^2} + u^{(4)}(x,y)\frac{h^2}{12} + \cdots$$

或者说，合起来看。

$$4u(x,y) - u(x+h,y) - u(x-h,y) - u(x,y+h) - u(x,y-h) = 1/h^2 f(x,y) + O\left(h^2\right)$$

让 $h = 1/(n+1)$，定义 $x_i = ih$，$y_j = jh$；让 $u_{ij}$ 是对 $u(x_i, y_j)$ 的近似，那么我们的离散方程为

$$4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = h^{-2} f_{ij}$$

我们现在有 $n \times n$ 未知数 $u_{ij}$。为了像以前一样将其转化为一个线性系统，我们需要将它们放在一个线性排序中，我们通过定义 $I = I_{ij} = j + i \times n$ 来实现。这被称为 ** 字典序 **（lexicographic ordering），因为它将坐标 $(i,j)$ 当作字符串来排序。

使用这个排序，我们可以得到 $N = n^2$ 的方程

$$4u_I - u_{I+1} - u_{I-1} - u_{I+n} - u_{I-n} = h^{-2} f_I, \quad I = 1, \ldots, N$$

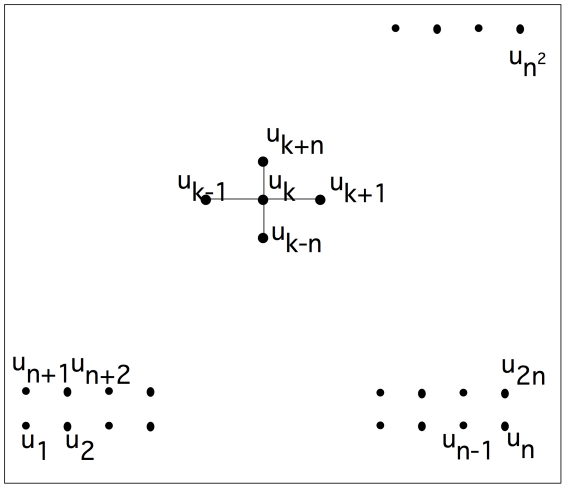



图 4.1: 应用于二维正方形域的 stencil 差分

而线性系统看起来像

$$
A = \left(\begin{array}{ccccc|ccccc|c}
4 & -1 & & & \varphi & -1 & & & & \varphi & \\
-1 & 4 & -1 & & & & -1 & & & & \\
 & \ddots & \ddots & \ddots & & & & \ddots & & & \\
 & & \ddots & \ddots & -1 & & & & \ddots & & \\
\varphi & & & -1 & 4 & \varphi & & & & -1 & \\
\hline
-1 & & & & \varphi & 4 & -1 & & & & -1 \\
 & -1 & & & & -1 & 4 & -1 & & & \\
 & \uparrow & \ddots & & & \uparrow & \uparrow & \uparrow & & & \uparrow \\
 & k-n & & & & k-1 & k & k+1 & & -1 & k+n \\
 & & & & -1 & & & & -1 & 4 & \\
\hline
 & & & & & \ddots & & & & & \ddots
\end{array}\right)
$$

矩阵的大小为 $N \times N$，其中 $N = n^2$。与一维的情况一样，我们看到 BVP 会产生一个稀疏的矩阵。

以矩阵形式考虑这个线性方程组似乎是很自然的事情。然而，以明确未知数的二维联系的方式呈现这些方程可能更有洞察力。为此，图 4.1 展示了领域中的变量，以及方程如何通过有限差分模板将它们联系起来。从现在开始，在制作这样的领域图片时，我们将只使用变量的索引，而省略"u" 标识符。

方程矩阵和一维的情况一样是带状的，但是和一维的情况不同，带状的内部有零点。因为该矩阵有五个非零对角线，所以被称为五对角线结构。

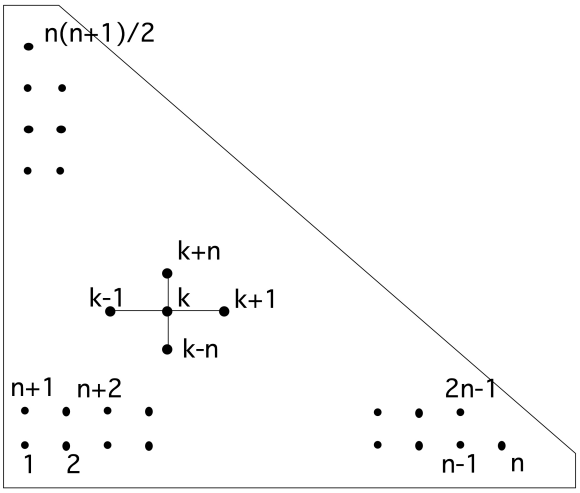



图 4.2: 泊松方程的三角定义域

你也可以在矩阵上放一个块状结构，把在域的一行中的未知数组合起来。这被称为**分块矩阵**（block matrix），在块的层面上，它有一个三对角的矩阵结构，所以我们称之为**分块三对角矩阵**（block tridiagonal matrix）。请注意，对角线块本身是三对角的；非对角线块是减去单位矩阵的。

这个矩阵和上面的一维例子一样，有恒定的对角线，但这又是由于问题的简单性质造成的。在实际问题中，这不会是真的。也就是说，这样的" 恒定系数" 问题是有的，当它们在矩形域上时，有非常有效的方法来解决线性系统，其时间复杂性为 $N \log N$。

**练习 4.4** 矩阵的块状结构，所有的对角线块都有相同的大小，这是因为我们在方形域上定义了我们的 BVP。画出方程 (4.12) 离散化所产生的矩阵结构，同样是中心差分，但这次是定义在一个三角形域上；见图 4.2。说明同样有一个块状三对角矩阵结构，但现在块的大小不一。提示：先画出一个小例子。对于 $n = 4$，你应该得到一个 $10 \times 10$ 的矩阵，其块结构为 $4 \times 4$。

对于更加不规则的域，矩阵结构也将是不规则的。

有规律的块状结构也是由我们决定将未知数按行和列排序造成的。这被称为自然排序或词法排序；其他各种排序也是可能的。一种常见的未知数排序方式是红黑排序或棋盘排序，这对并行计算有好处。

关于 BVP 的分析方面还有更多要说的（例如，解有多平滑，它是如何取决于边界条件的），但这些问题超出了本课程的范围。这里我们只关注矩阵的数值方面。在线性代数一章中，特别是第 5.4 和 5.5 节，我们将讨论从 BVP 中求解线性系统。

### 4.2.4 stencils 差分

stencils 差分

$$\begin{matrix} . & -1 & , \\ -1 & 4 & -1 \\ . & -1 & . \end{matrix}$$

到函数 $u$。给定一个物理域，我们将模版应用于该域中的每一个点，得出该点的方程。图 4.1 说明了一个由 $n \times n$ 点组成的正方形域的情况。将此图与上述方程联系起来，你会发现同一条线上的连接产生了主对角线和第一条上下对角线；与下一条线和上一条线的连接则成为非对角线块的非零点。

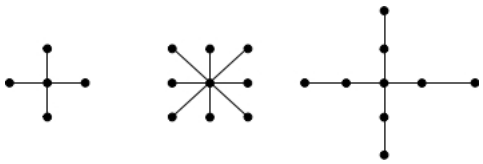


图 4.3: 二维上的 stencil 差分结构

这种特殊的 stencils 通常被称为" 五点星" 或五点模版。还有其他不同的网板；其中一些网板的结构在图 4.3 中得到描述。只有水平或垂直方向连接的网板被称为" 星形网板"，而有交叉连接的网板（如图 4.3 中的第二种）则被称为" 星形网板"。

✍ **练习 4.5** 考虑图 4.3 中的第三个模版，用于正方形域上的 BVP。如果我们再次将变量按行和列排序，所得到的矩阵的稀疏性结构是什么样子的?

除了 5 点星形之外，还可以使用其他模版来达到更高的精度，例如，给出一个 $O(h^4)$ 的截断误差。它们还可以用于上面讨论的微分方程以外的其他微分方程。例如，不难看出，对于方程 $u_x + u_{yyy} = f$，我们需要一个同时包含 $x$，$y \pm h$ 和 $x$，$y \pm 2h$ 连接的钢网，如图中的第三个钢网。相反，使用 5 点模版，没有系数值的情况下，四阶问题的离散化小于 $O(1)$ 截断误差。

虽然到目前为止讨论的是二维问题，但对于诸如 $-u_x - u_y - u_z = f$ 这样的方程，它可以被推广到更高维。例如，5 点模版的直接泛化，在三维空间中变成了 7 点模版。

### 4.2.5 其他离散化技术

在上面，我们用有限差分法来寻找微分方程的数值解。还有其他各种技术，事实上，在边界值问题的情况下，它们通常比有限差分更受欢迎。最流行的方法是有限元法和有限体积法。尤其是有限元方法很有吸引力，因为它比有限差分更容易处理不规则的形状，而且它更适合于近似误差分析。然而，在这里讨论的简单问题上，它给出的线性系统与 FD 方法相似甚至相同，所以我们将讨论限制在有限差分上，因为我们主要关注的是线性系统的计算方面。

## 4.3 初始边界值问题

现在我们将继续讨论初始边界值问题（IBVP），正如你可能从名字中推断的那样，它结合了 IVP 和 BVP 的各个方面。在这里，我们将把自己限制在一个空间维度。

我们考虑的问题是棒材中的热传导问题，其中 $T(x,t)$ 描述了在时间上 $x$ 的温度，对于 $x \in [a,b] t > 0$。所谓的热方程（见附录 15 对一般的 PDE，特别是热方程的快速介绍）是

$$\frac{\partial}{\partial t} T(x,t) - \alpha \frac{\partial^2}{\partial x^2} T(x,t) = q(x,t)$$

其中

- 初始条件 $T(x,0) = T_0(x)$ 描述初始温度分布。
- 边界条件 $T(a,t) = T_a(t)$，$T(b,t) = T_b(t)$ 描述棒的末端，例如可以固定在一个已知温度的物体上。
- 杆的材料由一个参数 $\alpha > 0$ 来模拟，即热扩散率，它描述了热量在材料中扩散的速度。
- 强迫函数 $q(x,t)$ 描述了外部应用的加热，作为时间和地点的函数。

IBVP 和 BVP 之间有一个简单的联系：如果边界函数 $T_a$ 和 $T_b$ 是常数，并且 $q$ 不依赖于时间，只依赖于位置，那么直观地说，$T$ 将收敛到一个稳定状态。这方面的方程式是 $-\alpha u''(x) = q$。

### 4.3.1 离散化

现在我们将空间和时间都离散化，即 $x_{j+1} = x_j + \Delta x_{k+1} = t_k + \Delta t$，边界条件 $x_0 = a$，$x_n = b$，并且 $t_0 = 0$。我们写出 $T_{jk}$ 的数值解，$x = x_j$，$t = t_k$；运气好的话，这将近似于精确解 $Tx_j t_k$。

对于空间离散化，我们使用中心差分公式。

$$\frac{\partial^2}{\partial x^2} T\left(x, t_k\right)\Big|_{x=x_j} \Rightarrow \frac{T_{j-1}^k - 2T_j^k + T_{j+1}^k}{\Delta x^2}$$

对于时间离散化，我们可以使用 4.1.2 节中的任何一种方案。我们将再次研究显式和隐式方案，对所产生的稳定性有类似的结论。

#### 4.3.1.1 显式方案

通过明确的时间步进，我们将时间导数近似为

$$\frac{\partial}{\partial t} T\left(x_j, t\right)\Big|_{t=t_k} \Rightarrow \frac{T_j^{k+1} - T_j^k}{\Delta t}$$

将此与空间的中心差异结合起来，我们现在有

$$\frac{T_j^{k+1} - T_j^k}{\Delta t} - \alpha \frac{T_{j-1}^k - 2T_j^k + T_{j+1}^k}{\Delta x^2} = q_j^k$$

我们将其改写为

$$T_j^{k+1} = T_j^k + \frac{\alpha \Delta t}{\Delta x^2}\left(T_{j-1}^k - 2T_j^k + T_{j+1}^k\right) + \Delta t q_j^k$$

在图 4.4 中，我们将其呈现为一个差分 stencils。这表示每个点的函数值是由前一个时间层次上的点的组合决定的。

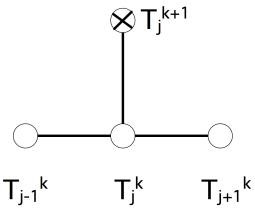



图 4.4: 热方程前向欧拉法差分图

将给定的 $k$ 和所有 $j$ 值的方程组用矢量形式概括为

$$T^{k+1} = \left(I - \frac{\alpha \Delta t}{\Delta x^2} K\right) T^k + \Delta t q^k$$

其中

$$K = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{pmatrix}, \quad T^k = \begin{pmatrix} T_1^k \\ \vdots \\ T_n^k \end{pmatrix}$$

这里的重要观察是，从 $T_{k+1}$ 得出向量 $T_k$ 的主要计算是一个简单的矩阵-向量乘法。

$$T^{k+1} \leftarrow AT^k + \Delta t q^k$$

其中 $A = I - \frac{\alpha \Delta t}{\Delta x^2} K$。这是第一个迹象，表明稀疏矩阵-向量乘积是一个重要的操作。使用显式方法的实际计算机程序通常不形成矩阵，而是评估方程。然而，为了分析的目的，线性代数公式是更有见地的。

在后面的章节中，我们将考虑操作的并行执行。现在，我们注意到显式方案是琐碎的并行的：每个点都可以只用周围几个点的信息进行更新。

### 4.3.1.2 隐式方案

在上述方程中，我们让 $T_{k+1}$ 从 $T_k$ 定义。我们可以通过从 $T_{k-1}$ 定义 $T_k$ 来扭转这一局面，正如我们在第 4.1.2.2 节中对 IVP 所做的那样。对于时间离散化，这就得到

$$\left.\frac{\partial}{\partial t}T(x,t)\right|_{t=t_k} \Rightarrow \frac{T_j^k - T_j^{k-1}}{\Delta t} \quad \text{or} \quad \left.\frac{\partial}{\partial t}T(x,t)\right|_{t=t_{k+1}} \Rightarrow \frac{T_j^{k+1} - T_j^k}{\Delta t}$$

整个热力方程的隐式时间步长离散化，在 $t_{k+1}$ 中进行评估，现在变成了。

$$\frac{T_j^{k+1} - T_j^k}{\Delta t} - \alpha \frac{T_{j-1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}}{\Delta x^2} = q_j^{k+1}$$

或者

$$T_j^{k+1} - \frac{\alpha \Delta t}{\Delta x^2}\left(T_{j-1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}\right) = T_j^k + \Delta t q_j^{k+1}$$

图 4.5 将其渲染成一个模版；这表达了当前时间层上的每一个点都会影响到下一层的点的组合。我们再一次用矢量的形式来写这个。

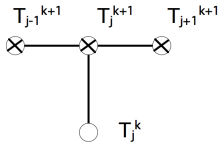



图 4.5: 热方程后向欧拉法的差分

$$\left(I + \frac{\alpha \Delta t}{\Delta x^2}K\right) T^{k+1} = T^k + \Delta t q^{k+1}$$

与显式方法相比，在显式方法中，矩阵-向量乘法就足够了，从 $T^{k+1}$ 推导出的向量 $T^k$ 现在涉及一个线性系统解决方案。

$$T^{k+1} \leftarrow A^{-1}\left(T^k + \Delta t q^{k+1}\right)$$

其中 $A = I + \frac{\alpha \Delta t}{\Delta x^2}K$ 一个比矩阵-向量乘法更难的操作。在这种情况下，不可能像上面那样，直接评估方程。使用隐式方法的代码实际上形成了系数矩阵，并以此来解决系统。解决线性系统将是第 5 章和第 6 章的重点。

与显式方案相比，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第 6.6 节及以后的章节中占据我们的位置。

✍ **练习 4.6** 证明隐式方法的一个时间步骤的 flop 数与显式方法的一个时间步骤的 flop 数是相同的。(这只适用于一个空间维度的问题)。至少给出一个论据，说明为什么我们认为隐式方法在计算上" 更难"。

我们在这里使用的数值方案是时间上的一阶和空间上的二阶：截断误差为 $O(\Delta t + \Delta x^2)$。也可以通过使用时间上的中心差分来使用一个时间上的二阶方案。另外，见练习 4.8。

### 4.3.2 稳定性分析

现在我们在一个简单的情况下分析显式和隐式方案的稳定性。让 $q \equiv 0$，并假设 $T_j^k = \beta^k e^{i\ell x_j}$，对于某些 $\ell$。这一假设在直觉上是站得住脚的：由于微分方程没有" 混合"$x$ 和 $t$ 的坐标，我们推测解决方案将是 $T(x,t) = v(x)\omega(t)$ 的单独解决方案的乘积。

$$\begin{cases} v_{xx} = c_1 v, & v(0) = 0, v(1) = 0 \\ w_t = c_2 w & w(0) = 1 \end{cases}$$

唯一有意义的解发生在 $c_1, c_2 < 0$，在这种情况下我们发现。

$$v_{xx} = -c^2 v \Rightarrow v(x) = e^{-icx} = e^{-i\ell\pi x}$$

其中我们用 $c = \ell\pi$ 代替，以考虑到边界条件。

$$w(t) = e^{-ct} = e^{-ck\Delta t} = \beta^k, \quad \beta = e^{-ck}$$

如果对这种形式的解决方案的假设成立，我们需要 $|\beta| < 1$ 的稳定性。将推测的 $T_j^k$ 形式代入显式方案，可以得到

$$\begin{aligned}
T_j^{k+1} &= T_j^k + \frac{\alpha\Delta t}{\Delta x^2}\left(T_{j-1}^k - 2T_j^k + T_{j+1}^k\right) \\
\Rightarrow \beta^{k+1} e^{i\ell x_j} &= \beta^k e^{i\ell x_j} + \frac{\alpha\Delta t}{\Delta x^2}\left(\beta^k e^{i\ell x_{j-1}} - 2\beta^k e^{i\ell x_j} + \beta^k e^{i\ell x_{j+1}}\right) \\
&= \beta^k e^{i\ell x_j}\left[1 + \frac{\alpha\Delta t}{\Delta x^2}\left[e^{-i\ell\Delta x} - 2 + e^{i\ell\Delta x}\right]\right] \\
\Rightarrow \beta &= 1 + 2\frac{\alpha\Delta t}{\Delta x^2}\left[\frac{1}{2}\left(e^{i\ell\Delta x} + e^{-\ell\Delta x}\right) - 1\right] \\
&= 1 + 2\frac{\alpha\Delta t}{\Delta x^2}(\cos(\ell\Delta x) - 1)
\end{aligned}$$

为了保持稳定，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 $\ell$ 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 $\ell$ only if $2\frac{\alpha\Delta t}{\Delta x^2} < 1$, that is $\Delta t < \frac{\Delta x^2}{2\alpha}$

后一个条件对允许的时间步长提出了很大的限制：时间步长必须足够小，方法才能稳定。这与 IVP 的显式方法的稳定性分析类似；然而，现在时间步长也与空间离散化有关。这意味着，如果我们决定需要更多的空间精度，并将空间离散化 $\Delta x$ 减半，时间步数将乘以 4。现在让我们考虑隐式方案的稳定性。将解的形式 $T_j^k = \beta^k e^{i\ell x_j}$ 替换到数值方案中，可以得到

$$\begin{aligned}
T_j^{k+1} - T_j^k &= \frac{\alpha\Delta t}{\Delta x^2}\left(T_{j_1}^{k+1} - 2T_j^{k+1} + T_{j+1}^{k+1}\right) \\
\Rightarrow \beta^{k+1} e^{i\ell\Delta x} - \beta^k e^{i\ell x_j} &= \frac{\alpha\Delta t}{\Delta x^2}\left(\beta^{k+1} e^{i\ell x_{j-1}} - 2\beta^{k+1} e^{i\ell x_j} + \beta^{k+1} e^{i\ell x_{j+1}}\right)
\end{aligned}$$

除去 $e^{i\ell x_j}\beta^{k+1}$，可得

$$\begin{aligned}
1 &= \beta^{-1} + 2\alpha\tfrac{\Delta t}{\Delta x^2}(\cos\ell\Delta x - 1) \\
\beta &= \tfrac{1}{1+2\alpha\frac{\Delta t}{\Delta x^2}(1-\cos\ell\Delta x)}
\end{aligned}$$

由于 $1 - \cos l\Delta x \in (0, 2)$，分母严格 >1。因此，无论 l 的值如何，也无论 $\Delta x$ 和 $\Delta t$ 的选择如何，条件 $|\beta| < 1$ 总是被满足的：该方法总是稳定的。

✍ **练习 4.7** 我们在这里考虑的方案是时间上的一阶和空间上的二阶：它们的离散化顺序是 $O(\Delta t) + O(\Delta x^2)$。推导出由显式和隐式方案平均化得到的 Crank-Nicolson 方法，说明它是不稳定的，并且在时间上是二阶的。

# 第 5 章 数值线性代数

内容提要

- 消除未知数
- 线性代数在计算机运算中的应用
- LU 分解
- 稀疏矩阵
- 迭代法
- 特征值法
- 延伸阅读

在第四章中，我们了解了偏微分方程的数值解法是如何产生线性代数的问题。在前向欧拉法的情况下是一个矩阵与向量乘法，较为简单；而在后向欧拉方法下是一个线性方程组的解，较为复杂。解决线性系统将是本章的重点；此处我们将不讨论需要解决特征值问题。

你可能已经学过一种解线性方程组的简单算法：消除未知数，也叫**高斯消元法**（Gaussian elimination）。这种方法仍然可以使用，但我们需要对其效率进行一些仔细的讨论。还有其他一些算法即所谓的迭代求解法，它们通过逐步逼近线性系统的解来进行，这也是我们要讨论的内容。

由于 PDE 的背景，我们只考虑方形和非星形的线性系统。矩形系统，特别是超定系统，在最优化理论的数值分析中也有重要的应用。然而，在本书中我们不会涉及这些。

关于数值线性代数的标准著作是 Golub 和 Van Loan 的《矩阵计算》[83]。它包括算法、误差分析和计算细节。Heath 的《科学计算》涵盖了科学计算中出现的最常见的计算类型；这本书有许多优秀的练习和实践项目。

## 5.1 消除未知数

本章的讨论主线是各种算法的效率。即便你学会了高斯消元法，也可能从未在大于 [公式] 的矩阵上使用这种方法。在偏微分方程求解中出现的线性系统可能要大几千倍，计算它们需要操作数以及内存是十分重要的。

注释 我们可以通过**高斯-若尔当** (Gauss Jordan) 方法，乘以逆矩阵 $A^{-1}x \leftarrow A^{-1}x$，来求解方程，但出于数字精度的考虑，本书不讨论这种方法。

**定义 5.1**

**克拉默法则**（Cramer's rule）指出：线性方程组的解可以用一个相当简单的公式表示，即行列式。尽管它在数学上很优雅，但对我们来说却并不实用。

如果给定了一个矩阵 $A$ 和一个向量 $b$，想要求解 $Ax = b$ 的解，则 $|A|$ 的行列式：

$$x_i = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1n} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2n} \\ \vdots & & & & \vdots & & & \vdots \\ a_{n1} & & \dots & & b_n & & \dots & a_{nn} \end{vmatrix} / |A|$$

对于任何矩阵 $M$，行列式被递归定义为

$$|M| = \sum_i (-1)^i m_{1i} \left| M^{[1,i]} \right|$$

其中 $M[1,i]$ 表示从 $M$ 中删除第 1 行和第 $i$ 列而得到的矩阵。计算一个 $n$ 阶行列式要计算 $n$ 次 $n-1$ 阶行列式。每一次都需要 $n-1$ 个大小为 $n-2$ 的行列式，所以行列式的计算所需操作是矩阵大小的阶乘。在本章的后面，我们会用其他合理的方法解决线性方程组。

现在让我们看一下用消除未知数的方法解线性方程的一个简单例子。考虑以下线性方程组

$$6x_1 - 2x_2 + 2x_3 = 16$$
$$12x_1 - 8x_2 + 6x_3 = 26$$
$$3x_1 - 13x_2 + 3x_3 = -19$$

我们从第二和第三个方程中消除 $x_1$，方法是

- 将第二个方程减去第一个方程 $\times 2$；
- 将第三个方程减去第一个方程 $\times 1/2$。

这样，线性方程组就变成了

$$6x_1 - 2x_2 + 2x_3 = 16$$
$$0x_1 - 4x_2 + 2x_3 = -6$$
$$0x_1 - 12x_2 + 2x_3 = -27$$

最后，我们通过将第二个方程乘以 3，在第三个方程中将其减去，消除三式中的 $x_2$。

$$6x_1 - 2x_2 + 2x_3 = 16$$
$$0x_1 - 4x_2 + 2x_3 = -6$$
$$0x_1 + 0x_2 - 4x_3 = -9$$

现在我们可以根据上一个方程求出 $x_3 = 9/4$。将其代入第二个方程，我们得到 $-4x_2 = -6 - 2x_2 = -21/2$，所以 $x_2 = 21/8$。最后，从第一个方程中，$6x_1 = 16 + 2x_2 - 2x_3 = 16 + 21/4 - 9/2 = 76/4$，所以 $x_1 = 19/6$。

我们可以通过省略 $x_i$ 系数来写得更紧凑。将

$$\begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 16 \\ 26 \\ -19 \end{pmatrix}$$

记为

$$\left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 12 & -8 & 6 & 26 \\ 3 & -13 & 3 & -19 \end{array}\right]$$

那么消元的过程为

$$\left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 12 & -8 & 6 & 26 \\ 3 & -13 & 3 & -19 \end{array}\right] \to \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & -12 & 2 & -27 \end{array}\right] \to \left[\begin{array}{ccc|c} 6 & -2 & 2 & 16 \\ 0 & -4 & 2 & -6 \\ 0 & 0 & -4 & -9 \end{array}\right]$$

在上面的例子中，矩阵系数可以是任何实数（或复数）系数。我们可以重复上面的过程求解任何线性方程组，但有一个例外。数字 6、-4、-4，最后处在矩阵的对角线上，这些非零数字被称为**主元**（pivots）。

✍ **练习 5.1** 线性方程组

$$6x_1 - 2x_2 + 2x_3 = 16$$
$$12x_1 - 4x_2 + 6x_3 = 26$$
$$3x_1 - 13x_2 + 3x_3 = -19$$

与我们刚才在公式中研究的相同，除了 $(2,2)$ 元素。确认你在第二步中得到一个零主元。

第一个主元是原矩阵的一个元素；正如你在前面的练习中所看到的，如果不消元，就无法找到其他主元。没有简单的方法可以预测零主元。

如果一个主元被证明是零，所有的计算都不会丢失：我们总是可以交换两个矩阵行；这就是所谓的主元。不难发现（你可以在任何一本初级线性代数教科书中找到），对于一个非奇异矩阵，总有一个行的交换可以将一个非零元素放在主元的位置。

✍ **练习 5.2** 假设我们想交换方程中方程组的矩阵第 2 行和第 3 行。还需要做哪些调整以确保仍然能计算出正确的解？通过交换第 2 行和第 3 行，继续求解上一个练习的解，并检查你得到的答案是否正确。

✍ **练习 5.3** 再看一下练习 5.1。不交换第 2 和第 3 行，而要交换第 2 和第 3 列。就线性方程组而言，这意味着什么？继续求解该系统；检查你是否得到与之前相同的解。

一般来说，在浮点数和四舍五入的情况下，在计算过程中，一个矩阵元素不太可能完全变成零。另外，在 PDE 背景下，对角线通常是不为零的。这是否意味着主元运算在实践中没有必要？答案是否定的：从数值稳定性的角度来看，主元是可取的。在下一节，你将看到一个例子来说明这个事实。

## 5.2 线性代数在计算机运算中的应用

本章的大部分内容都可以通过数学运算求解，然而由于计算机的精度有限，我们需要设计将舍入误差降到最小的算法。对数值线性代数的算法需要进行严格而全面的误差分析，然而，这超出了本课程的范围。计算机算术中的计算误差分析是威尔金森的经典《代数过程中的舍入误差》[201] 和海姆最近的《数值算法的准确性和稳定性》[106] 的重点。

本书将注重计算机运算过程中出现的经典例子：将说明为何在 LU 分解中主元方法不仅仅是理论手段，此外，我们将给出两个由于计算机算术的有限精度而导致的特征值计算中的问题的例子。

### 5.2.1 消除过程中的舍弃控制

上面我们看到，如果在消除该行和列的过程中，对角线上出现了一个零元素，那么行间交换（' 主元'）是必要的。现在我们讨论如果主元元素不是零，但接近零会发生什么。

考虑线性方程组

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1+\epsilon \\ 2 \end{pmatrix}$$

其中的解决方案是 $x = (1,1)^t$。使用 (1, 1) 元素清除第一列的剩余部分，可以得到。

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1+\epsilon \\ 2 \end{pmatrix}$$

现在我们可以解决 $x_2$，并从它得到 $x_1$。

$$\begin{cases} x_2 = \left(1-\epsilon^{-1}\right)/\left(1-\epsilon^{-1}\right) = 1 \\ x_1 = \epsilon^{-1}\left(1+\epsilon-x_2\right) = 1 \end{cases}$$

如果 $\epsilon$ 较小，如 $\epsilon < \epsilon_{mach}$，则右侧的 1+替换为 1：线性方程组改写为

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

但解 $(1,1)^t$ 仍将满足机器运算。接下来，$1/\epsilon$ 将非常大，所以消除后的右手边第二部分将是 $2-\frac{1}{\epsilon} = -1/\epsilon$，并且 (2,2) 矩阵中的元素是 $-1/\epsilon$ 而不是 $1-1/\epsilon$。

$$\begin{pmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{pmatrix} x = \begin{pmatrix} 1 \\ 2-\epsilon^{-1} \end{pmatrix} \Rightarrow \begin{pmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{pmatrix} x = \begin{pmatrix} 1 \\ -\epsilon^{-1} \end{pmatrix}$$

首先求解 $x_2$，然后求解 $x_1$，我们得到。

$$\begin{cases} x_2 = \epsilon^{-1}/\epsilon^{-1} = 1 \\ x_1 = \epsilon^{-1}\left(1-1\cdot x_2\right) = \epsilon^{-1}\cdot 0 = 0 \end{cases}$$

所以 $x_2$ 是正确的，但 $x_1$ 是完全错误的。

✍ **练习 5.4** 在这个例子中，计算机运算中的数字与精确运算中的数字偏差不大。然而，结果却可能是大错特错。对这种现象的分析见 [99] 的第一章。

如果我们按照上述方法进行透视，会发生什么？我们交换矩阵的行，得到

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} x = \begin{pmatrix} 2 \\ 1+\epsilon \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix} x = \begin{pmatrix} 2 \\ 1-\epsilon \end{pmatrix}$$

我们交换矩阵的行，得到

$$x_2 = \frac{1-\epsilon}{1-\epsilon} = 1, \quad x_1 = 2 - x_2 = 1$$

现在我们得到的是，无论 $\epsilon$ 的大小如何。

$$x_2 = \frac{1-\epsilon}{1-\epsilon} = 1 \quad x_1 = 2 - x_2 = 1$$

在这个例子中，我们使用了一个非常小的 $\epsilon$ 值；更精细的分析表明，即使 $\epsilon$ 值大于机器精度，主元交换仍然是有意义的。一般的经验法则是。始终进行换行，使当前列中最大的剩余元素进入主元位置。在第 4 章中，你看到了在某些实际应用中出现的矩阵；可以证明，对它们来说主元是没有必要的；见练习 5.13。

上面讨论的主元也被称为**部分主元**（partial pivoting），因为它只基于行的交换。另一个选择是**完全主元**（full pivoting），即结合行和列的交换，找到剩余子块中最大的元素，作为主元。最后，对角线主元将同样的交换应用于行和列。(这相当于对问题的未知数进行重新编号，我们将在第 6.8 节考虑这一策略，以增加问题的并行性)。这意味着主元只在对角线上被搜索到。从现在开始，我们将只考虑部分主元的问题。

### 5.2.2 舍入对特征值计算的影响

考虑矩阵

其中 $\epsilon_{mach} < |\epsilon| < \sqrt{\epsilon_{mach}}$，其特征值为 $1+\epsilon$ 和 $1-\epsilon$。如果我们用计算机运算来计算它的特征多项式

$$\begin{vmatrix} 1-\lambda & \epsilon \\ \epsilon & 1-\lambda \end{vmatrix} = \lambda^2 - 2\lambda + \left(1-\epsilon^2\right) = \lambda^2 - 2\lambda + 1$$

我们发现一个双重特征值 1。请注意，准确的特征值是可以用工作精度来表达的；是算法导致了错误。显然，使用特征多项式并不是计算特征值的正确方法，即使在行为良好的对称正定矩阵中也是如此。

一个非对称的例子：让 $A$ 为大小为 20 的矩阵

$$A = \begin{pmatrix} 20 & 20 & & & \varnothing \\ & 19 & 20 & & \\ & & \ddots & \ddots & \\ & & & 2 & 20 \\ \varnothing & & & & 1 \end{pmatrix}$$

由于这是一个三角矩阵，其特征值是对角线元素。如果我们通过设置 $A_{20,1} = 10^{-6}$ 来扰动这个矩阵，我们发现特征值的扰动要比元素的扰动大得多。

$$\lambda = 20.6 \pm 1.9i, 20.0 \pm 3.8i, 21.2, 16.6 \pm 5.4i, \ldots$$

另外，有几个计算出来的特征值有虚数的成分，而准确的特征值没有这个成分。

## 5.3 LU 分解

到目前为止，我们已经在解决单一线性方程组的背景下研究了消除未知数的问题。假设我们需要解决一个以上的具有相同矩阵的系统，但有不同的右手边。例如，如果在隐式欧拉方法中采取多个时间步骤，就会发生这种情况。我们能否利用第一个系统中所做的工作来使后面的内容更加容易解决?

答案是肯定的。我们可以将求解过程分为只涉及矩阵的部分和专门针对右手边的部分。如果有一系列的系统需要解决，我们只需要做一次第一部分，幸运的是，这甚至是我们需要做的主要工作。

让我们再看一个例子：

在消除的过程中，我们把第二行减去 2× 第一行，第三行减去 1/2× 第一行。说服你自己，这种合并行的做法可以通过从左边的 A 乘以

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix}$$

这是与对角线下第一列中的消除系数相同的。消除变量的第一步相当于将系统 $Ax = b$ 转换为 $L_1Ax = L_1b$。

在下一步，你从第三行减去 3× 第二行。让你自己相信，这相当于将当前的矩阵 $L_1A$ 左乘以

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

这是与对角线下第一列中的消除系数相同的。消除变量的第一步相当于将系统 $Ax = b$ 转换为 $L_2L_1Ax = L_2L_1b$。并且 $L_2L_1A$ 是" 上三角" 形式。如果我们定义 $U = L_2L_1A$, 那么 $A = L_1^{-1}L_2^{-1}U$。计算诸如 $L^{-1}$ 这样的矩阵有多难？很容易，事实证明是这样。

我们提出以下意见。

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1/2 & 0 & 1 \end{pmatrix} \quad L_1^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 0 & 1 \end{pmatrix}$$

同样地

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix} \quad L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{pmatrix}$$

甚至更显著的

$$L_1^{-1}L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 3 & 1 \end{pmatrix}$$

即 $L_1^{-1}L_2^{-1}$ 包含 $L^{-1}$ 的非对角线元素，$L^{-1}$ 不变，它们又包含消除系数。(这是 Householder 反射器的一个特殊情况，见 13.6)。

✍ **练习 5.5** 证明类似的声明是成立的，即使在对角线之上有元素存在。

如果我们定义 $L = L_1^{-1}L_2^{-1}$，我们现在有 $A = LU$；这被称为 $LU$ 因子化。我们看到，对角线以下的 $L$ 系数是消除过程中所用系数的负数。更好的是，$L$ 的第一列可以在消除 $A$ 的第一列时写出来，所以 $L$ 和 $U$ 的计算可以在没有额外存储的情况下完成，至少我们可以承受失去 $A$。

### 5.3.1 算法

我们把 LU 因式分解算法写成正式代码

```
<LU factorization>:
  for k = 1, n - 1:
    <eliminate values in column k>
<eliminate values in column k>:
  for i = k + 1 to n:
    <compute multiplier for row i>
    <update row i>
```

```
<compute multiplier for row i>
    a_{ik} <- a_{ik}/a_{kk}
<update row i>:
  for j = k + 1 to n:
    a_{ij} <- a_{ij} - a_{ik}  a_{kj}
```

或者说，把所有东西放在一起。

```
<LU factorization>:
  for k = 1, n - 1:
    for i = k + 1 to n:
      a_{ik} <- a_{ik}/a_{kk}
        for j = k + 1 to n:
          a_{ij} <- a_{ij} - a_{ik}  a_{kj]
```

这是呈现 LU 因子化的最常见方式。然而，也存在其他计算相同结果的方法。像 LU 因子化这样的算法可以用几种方式进行编码，这些方式在数学上是等价的，但它们有不同的计算行为。这个问题，在密集矩阵的背景下，是 van de Geijn 和 Quintana 的《矩阵计算编程的科学》[193] 的重点。

### 5.3.2 Cholesky 因式分解

一个对称矩阵的 $LU$ 因式分解并不能得到相互转置的 $L$ 和 $U$：$L$ 在对角线上有 1，而 $U$ 有主元。然而，我们可以对对称矩阵 $A$ 进行因式分解，其形式为 $A = LL^t$。这样做的好处是，因式分解所占用的空间与原始矩阵相同，即 $nn + 1/2$ 个元素。如果运气好的话，我们可以像在 $LU$ 情况下一样，用因子化覆盖矩阵。

我们通过归纳推理来推导出这个算法。让我们把 $ALL^t$ 写成块状。

$$A = \begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t = \begin{pmatrix} \ell_{11} & 0 \\ \ell_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \ell_{11} & \ell_{21}^t \\ 0 & L_{22}^t \end{pmatrix}$$

则 $\ell_{11}^2 = a_{11}$，由此得到 $\ell_{11}$。我们还发现 $\ell_{11}(L^t)_{1j} = \ell_{j1} = a_{1j}$，所以我们可以计算出整个 $L$ 的第一列。最后，$A_{22} = L_{22}L_{22}^t + \ell_{12}\ell_{12}^t$，所以

$$L_{22}L_{22}^t = A_{22} - \ell_{12}\ell_{12}^t$$

这表明 $L_{22}$ 是更新的 $A_{22}$ 块的 Cholesky 因子。递归后，现在定义了该算法。

### 5.3.3 唯一性

我们时常需要分析不同的计算方式是否会导致相同的结果，这被称为结果的“唯一性”：如果计算结果唯一，那么不论我们使用何种软件库，都不会改变计算的有效性。

下面我们分析 LU 因式分解的唯一性。LU 因式分解算法的定义是：给定一个非奇异矩阵 $A$，它将给出一个下三角矩阵 L 和上三角矩阵 $U$，使得 $A = LU$。上述计算 LU 因式分解的算法是确定的（它不包含" 取任何满足... 的行" 的指令），所以给定相同的输入，它将总是计算相同的输出。然而，其他的算法也是可能的，所以我们需要担心他们是否会得到相同的结果。

我们假设 $A = L_1U_1 = L_2U_2$，其中 $L1L2$ 为下三角，$U_1U_2$ 为上三角。那么，$L^{-1}L = UU^{-1}$。在这个等式中，左手边是下三角矩阵的乘积，而右手边只包含上三角矩阵。

✍ **练习 5.6** 证明下三角矩阵的乘积是下三角，而上三角矩阵的乘积是上三角。对于非奇异三角形矩阵的倒数，类似的说法是否成立?

积 $L^{-1}L$ 显然既是下三角又是上三角，所以它一定是对角线的。我们称它为 $D$，那么 $L_1 = L_2D$，$U_2 = D_1$。结论是，$LU$ 因式分解不是唯一的，但它是唯一的" 直到对角线缩放"。

✍ **练习 5.7** 第 5.3.1 节中的算法产生了一个下三角因子 L，其对角线上有 1。证明这个额外的条件使得因式分解是唯一的。

✍ **练习 5.8** 证明另一个条件，即 U 的对角线上有 1，也足以实现因式分解的唯一性。

由于我们可以要求 $L$ 或 $U$ 中的单位对角线，你可能想知道是否有可能两者都有。(我们可以这样做：假设 $A = LU$，其中 $L$ 和 $U$ 是非星形下三角和上三角，但没有以任何方式归一化。请写出

$$L = (I + L')D_L, \quad U = D_u(I + U')$$

经过一些重命名，我们现在有一个因式分解

$$A = (I + L)D(I + U)$$

其中 $D$ 是一个包含主元的对角线矩阵。

✍ **练习 5.9** 证明可以将因式分解的形式定为 $A = (D + L)D^{-1}(D + U)$ 。这个 $D$ 与前面的有什么关系?

✍ **练习 5.10** 这样考虑一个三对角矩阵的因式分解。$L$ 和 $U$ 与 $A$ 的三角部分有什么关系？推导出 $D$ 和 $D_A$ 之间的关系，并说明这是产生主元的方程。

### 5.3.4 主元

在上面的因式分解例子中，我们为了保证非零主元的存在，或者为了数值的稳定性，需要进行主元化，也就是交换行。现在我们将把透视纳入 $LU$ 因式分解。

首先，行交换可以用矩阵乘法来描述。令

那么 $P^{(i,j)}A$ 是交换了行 $i$ 和 $j$ 的矩阵 $A$。由于我们可能要在因式分解过程的每次迭代中进行透视，我们引入一个序列 $p()$，其中 $p(i)$ 是与第 i 行交换的第 j 行的值。简写为 $P^{(i)} \equiv P^{(i,p(i))}$。

✍ **练习 5.11** 证明 $P^{(i)}$ 是其自身的逆。

现在可以将部分主元的因式分解过程描述为:

- 让 $A^{(i)}$ 为矩阵，列 $1...i-1$ 被消除，并应用部分主元以获得 $(i,i)$ 位置上的所需元素。
- 让 $\ell^{(i)}$ 为第 $i$ 个消除步骤中的乘数向量。(也就是说，这一步的消除矩阵 $L_i$ 是身份加 $\ell(i)$ 的第 $i$ 列)。
- 让 $P^{(i+1)}\, j \geqslant i+1$ 是为下一个消除步骤做部分透视的矩阵，如上所述。
- 那么 $A^{(i+1)} = P^{(i+1)}L_iA^{(i)}$。

这样，我们就得到了一个因式分解的形式

$$L_{n-1}P^{(n-2)}L_{n-2}\cdots L_1P^{(1)}A = U$$

此时我们无法写出 $A = LU$：而是写出

$$A = P^{(1)}L_1^{-1}\cdots P^{(n-2)}L_{n-1}^{-1}U$$

✍ **练习 5.12** 回顾前面的内容，从性能角度看，分块算法通常是可取的。为什么方程 (5.5) 中的"$LU$ 因子化与交错主元矩阵'' 对性能来说是个坏消息?

幸运的是，方程（5.5）可以被简化：$P$ 和 $L$ 矩阵'' 几乎相通''。我们通过一个例子来证明这一点：$P^{(2)}L_1 = \tilde{L}_1P^{(2)}$，其中 $\tilde{L}_1$ 非常接近于 $L_1$。

$$\begin{pmatrix} 1 & & \\ & 0 & 1 \\ & I & \\ & 1 & 0 \\ & & I \end{pmatrix}\begin{pmatrix} 1 & & \varnothing \\ \vdots & 1 & \\ \ell^{(1)} & & \ddots \\ \vdots & & 1 \end{pmatrix} = \begin{pmatrix} 1 & & \varnothing \\ \vdots & 0 & 1 \\ \tilde{\ell}^{(1)} & & \\ \vdots & 1 & 0 \\ \vdots & & I \end{pmatrix} = \begin{pmatrix} 1 & & \varnothing \\ \vdots & 1 & \\ \tilde{\ell}^{(1)} & & \ddots \\ \vdots & & 1 \end{pmatrix}\begin{pmatrix} 1 & & \\ & 0 & 1 \\ & I & \\ & 1 & 0 \\ & & I \end{pmatrix}$$

✍ **练习 5.13** 如果我们把方程写成 $PA = LU$，得到 $A = P^{-1}LU$。是否能得出一个用 $P$ 表示的 $P^{-1}$？提示：每个 $P^{(i)}$ 都是对称的，而且是它自己的逆数；见上面的练习。

✍ **练习 5.14** 早些时候，我们看到二维 BVP 产生了某种矩阵。我们在没有证明的情况下指出，对于这些矩阵不需要主元。现在我们可以正式证明这一点，重点放在对角线支配的关键属性上。

$$\forall_i a_{ii} \geqslant \sum_{j \neq i} |a_{ij}|.$$

假设矩阵 $A$ 满足 $\forall j \neq i : a_{ij} \leqslant 0$，表明该矩阵是对角域的，即存在向量 $u, v \geqslant 0$（意味着每个分量都是非负的），使得 $Au = v$。

证明在消除一个变量后，对于剩余的矩阵 $\tilde{A}$ 又有向量 $\tilde{u}\tilde{v} \geqslant 0$，使得 $\tilde{A}\tilde{u} = \tilde{v}$。现在完成论证，如果 $A$ 是对称的、对角线主导的，（部分）主元是不必要的。（实际上可以证明，主元对于任何对称正定（SPD）矩阵都是不必要的，对角线主导是比 SPD-性更强的条件）。

### 5.3.5 解决系统问题

现有一个因式分解 $A = LU$，我们可以用它来解决线性系统 $Ax = LUx = b$。如果我们引入一个临时向量 $y = Ux$，那么我们可以看到这需要两个步骤。

$$Ly = b, \quad Ux = z$$

第一部分，$Ly = b$ 被称为'' 下三角解''，因为它涉及下三角矩阵 $L$。

$$\begin{pmatrix} 1 & & & & \varnothing \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & & & \ddots & \\ \ell_{n1} & l_{n2} & & \cdots & 1 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

在第一行，你看到 $y_1 = b_1$. 然后，在第二行 $\ell_{21}y_1 + y_2 = b_2$，所以 $y_2 = b_2 - \ell_{21}y_1$。你可以想象这如何继续：在每 $i$ 行，你可以从以前的 $y$ 值计算 $y_i$。

$$y_i = b_i - \sum_{y<i} \ell_{ij} y_j$$

由于我们以递增的方式计算 $y_i$，这也被称为正向替换、正向求解或正向扫瞄。

求解过程的后半部分，即上三角求解、后向替换或后向扫描，从 $Ux = y$ 中计算出 $x$。

$$\begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & \ddots & \vdots \\ \varnothing & & & u_{nn} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

现在我们看一下最后一行，它告诉我们 $x_n = u_{nn}^{-1}y_n$。由此可见，最后一行说明了 $u_{n-1n-1}x_{n-1} + u_{n-1n}x_n = y_{n-1}$，从而得出 $x_{n-1} = u_{n-1n-1}^{-1}(y_{n-1}u_{n-1n}x_n)$。更一般地，我们可以计算

$$x_i = u_{ii}^{-1}\left(y_i - \sum_{j>i} u_{ij}y_j\right)$$

对 $i$ 的数值递减。

✍ **练习 5.15** 在回扫中，你必须用数字 $u_i$ 进行除法，此时任何元素都不为零。把这个问题与上面的讨论联系起来。

### 5.3.6 复杂度

在本章的开头，我们指出，并不是每一种解决线性系统的方法都需要相同数量的操作。因此，让我们仔细研究一下复杂度 1，也就是在解决线性系统时使用 LU 因子化的操作数与问题大小的函数关系。

首先，我们看一下从 $LUx = b$（'' 解决线性系统''）计算 $x$，因为我们已经有了因子化 $A = LU$。把下三角和上三角部分放在一起看，会发现与所有对角线外的元素（即元素 $\ell_{ij}$ 或 $u_{ij}$ 与 $i \neq j$）进行了乘法。此外，上三角解涉及到对 $u_i$ 元素的除法。现在，除法运算通常比乘法运算昂贵得多，所以在这种情况下，我们会计算出 $1/u_i$ 的值，并将其存储起来。

✍ **练习 5.16** 请看一下因式分解算法，并论证存储主元的倒数并不增加计算的复杂性。

总结一下会发现，在一个大小为 $n \times n$ 的系统中，我们要进行 $n^2$ 的乘法运算和大致相同数量的加法运算。这表明，给定一个因式分解，解决一个线性系统的复杂度与简单的矩阵-向量乘法相同，也就是说，给定 $A$ 和 $x$ 的计算复杂度。

构建 $LU$ 因式分解的复杂度，计算起来就比较麻烦了。你可以看到，在 $k$ 第三步发生了两件事：乘数的计算和行的更新。有 $n-k$ 个乘数需要计算，每个乘数都涉及一个除数。除法之后，更新需要 $(n-k)^2$ 次加法和乘法。如果我们暂时不考虑除法的问题。因为它们的数量较少，我们发现 $LU$ 因子化需要 $\sum_{k=1}^{n-1} 2(n-k)^2$ 次运算。如果我们顺序给这个和中的项编号，我们会发现

$$\#\text{ops} = \sum_{k=1}^{n-1} 2k^2$$

由于可以通过积分来近似求和，我们发现这是 $2/3n^3$ 加上一些低阶项。这比解决线性系统要高一阶：随着系统规模的增长，构建 $LU$ 因子化的成本完全占据了主导地位。

当然，算法分析还有比运算计数更重要的内容。虽然解线性系统的复杂度与矩阵-向量乘法相同，但这两种操作的性质非常不同。一个很大的区别是，因式分解和前向/后向求解都涉及递归，所以它们不容易并行化。关于这一点，我们将在后面详细说明。

### 5.3.7 分块算法

通常，矩阵有一个自然的块状结构，比如二维 BVP 的情况。许多线性代数操作可以用这些块来表述。与传统的矩阵标量观点相比，这可能有几个好处。例如，它可以改善缓存阻塞；它也有利于在多核架构上调度线性代数算法。

对于分块算法，我们把矩阵写成

$$A = \begin{pmatrix} A_{11} & \dots & A_{1N} \\ \vdots & & \vdots \\ A_{M1} & \dots & A_{MN} \end{pmatrix}$$

其中 $M, N$ 是块的大小，即用子块表示的大小。通常情况下。

我们选择的块是：$M = N$，对角线块为正方形。作为一个简单的例子，考虑矩阵-向量乘积 $y = Ax$，以块的

形式表示。

$$\begin{pmatrix} Y_1 \\ \vdots \\ Y_M \end{pmatrix} = \begin{pmatrix} A_{11} & \dots & A_{1M} \\ \vdots & & \vdots \\ A_{M1} & \dots & A_{MM} \end{pmatrix} \begin{pmatrix} X_1 \\ \vdots \\ X_M \end{pmatrix}$$

为了说明区块算法的计算结果与旧的标量算法相同，我们看一下一个分支 $Y_{i_k}$，也就是第 $i$ 块的 $k$ 个标量分量。首先。

$$Y_i = \sum_j A_{ij} X_j$$

于是

$$Y_{i_k} = \left( \sum_j A_{ij} X_j \right)_k = \sum_j \left( A_{ij} X_j \right)_k = \sum_j \sum_\ell A_{ij_{kl}} X_{j_\ell}$$

是 $a$ 的第 $i$ 个区块行的第 $k$ 行与整个 $X$ 的积。

一个更有趣的算法是 $LU$ 因子化的分块版。算法就变成了

**Algorithm 1** <$LU$ factorization>

**for** $k = 1, n-1$:
for$i = k+1$ **to** $n$
$A_{ij} \leftarrow A_{ik} A^{-1}$
**for** $j = k+1$ **to** n:
$A_{ij} \leftarrow A_{ij} - A_{ik} \cdot A_{kj}$

该算法与之前的算法有很大的不同，即除以 $a_{kk}$ 的方法被一个乘以 $A_{kk}^{-1}$。另外，$U$ 因子现在在对角线上有支点块，而不是支点元素。所以 $U$ 只是" 块状上三角形"，而不是严格的上三角形。

✍ **练习 5.17** 我们想表明，这里的块状算法再次计算出与标量算法相同的结果。通过明确查看计算的元素来做到这一点是很麻烦的，所以我们采取另一种方法。首先，回顾一下第 5.3.3 节，$LU$ 面化是唯一的：如果 $A = L_1U_1 = L_2U_2$ 且 $L_1L_2$ 有单位斜线，那么 $L_1 = L_2$。$U_1 = U_2$

接下来，考虑计算 $A_{kk}^{-1}$ 的问题。证明这可以通过以下方式完成：首先计算 $A_{kk}$ 的 $LU$ 因子化。现在用它来证明，块状 $LU$ 因子化可以给出严格意义上的三角形的 $L$ 和 $U$ 因子。$LU$ 因式分解的唯一性就证明了块算法可以计算出标量结果。

## 5.4 稀疏矩阵

BVP（和 IBVP）的离散化可能会引起稀疏矩阵的出现。由于这样的矩阵有 $N^2$ 个元素，但只有 $O(N)$ 个非零元素，将其存储为二维数组将是对空间的极大浪费。此外，我们希望避免对零元素进行操作。

在本节中，我们将探讨稀疏矩阵的有效存储方案，以及使用稀疏存储时熟悉的线性代数操作的形式。

### 5.4.1 稀疏矩阵的存储

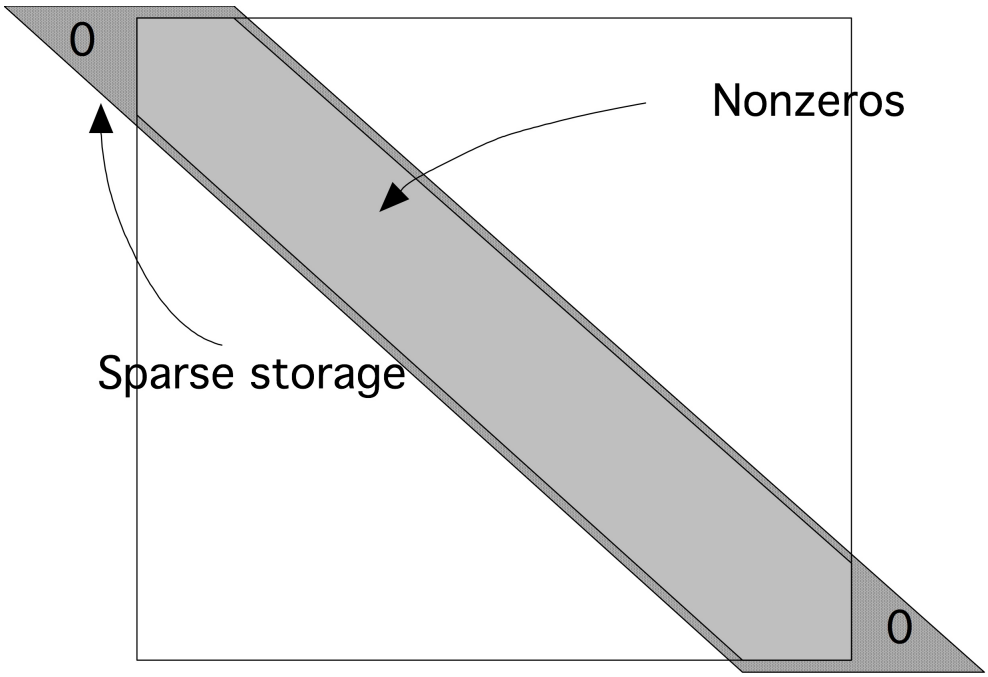
稀疏矩阵没有明确定义，只要当矩阵中的零元素多到需要专门存储时，该矩阵即为**稀疏矩阵**（sparse matrix）。

我们将在此简要讨论最流行的稀疏矩阵的存储方案。由于矩阵不再是作为一个简单的 2 维数组来存储，使用这种存储方案的算法也需要重写。

#### 5.4.1.1 带状存储和对角线存储

带状稀疏矩阵的非零元素正好位于一些子对角线上。对于这样的矩阵，可以采用专门的存储方案。

以一维 BVP 的矩阵为例。它的元素位于三个子对角线上：主对角线和第一个超对角线和子对角线。在带状存储中，我们只将包含非零点的带状存储在内存中。对于这样一个矩阵，最经济的存储方案是连续存储 $2n-2$ 的元素。然而，由于各种原因，浪费一些存储位置会更方便，如图 5.1 所示。




因此，对于一个大小为 $n\times n$，矩阵带宽为 $p$ 的矩阵，我们需要一个大小为 $n\times p$ 的矩形数组来存储矩阵。那么，矩阵将被存储为

$$\begin{matrix} \star & 2 & -1 \\ -1 & 2 & -1 \\ \vdots & \vdots & \vdots \\ -1 & 2 & \star \end{matrix}$$

其中星星表示不对应矩阵元素的数组元素：它们是图 5.1 中左上和右下的三角形。

当然，现在我们要想知道阵列元素 $A(i,j)$ 和矩阵元素 $A_{ij}$ 之间的转换。这在 Fortran 语言中是最容易做到的。如果我们将数组分配为

```
dimension A(n,-1:1)
```

那么主对角线 $A_i$ 就被储存在 $A(*,0)$ 中。例如，$A(1,0)\sim A_{11}$。在矩阵 $A$ 的同一行的下一个位置，$A(1,1)\sim A_{12}$。很容易看出，我们一起有这样的转换

$$A(i,j)\sim A_{i,i+j}.$$

✍ **练习 5.18** 什么是反向转换，也就是说，矩阵元素 $A_{ij}$ 对应于哪个数组位置 $A(?,?)$?

✍ **练习 5.19** 如果你是一个 C 语言程序员，请推导出矩阵元素 $A_{ij}$ 与数组元素 $A[i][j]$ 之间的转换。

如果我们将这种将矩阵存储为 $N\times p$ 数组的方案应用于二维 BVP 的矩阵，就会变得很浪费，因为我们会存储许多存在于带内的零。因此，在对角线存储或对角线存储中，我们通过只存储非零对角线来完善这一方案：如

果矩阵有 $p$ 个非零对角线，我们需要一个 $n \times p$ 数组。对于方程的矩阵，这意味着。

$$\begin{array}{ccccc} \star & \star & 4 & -1 & -1 \\ \vdots & \vdots & 4 & -1 & -1 \\ \vdots & -1 & 4 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & -1 & 4 & \star & \star \end{array}$$

当然，我们需要一个额外的整数阵来告诉我们这些非零对角线的位置。

✍ **练习 5.20** 对于 $d = 1, 2, 3$ 空间维度的中心差分矩阵，作为 $N$ 阶的带宽是多少？离散化参数 $h$ 的阶数是多少？

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。在一般情况下，这不一定是真的。为此，我们引入了以下概念

- **左半带宽度**（left halfbandwidth）：如果 $A$ 的左半带宽度为 $p$，那么 $A_{ij} = 0$，因为 $i > j + p$，和
- **右半带宽度**（right halfbandwidth）：如果 $A$ 的右半带宽度为 $p$，那么 $A_{ij} = 0$，因为 $j > i + p$。如果左边和右边的半带宽度相同，我们就直接指半带宽度。

### 5.4.1.2 对角线存储的操作

对稀疏矩阵最重要的操作是矩阵-向量乘积。对于一个按对角线存储的矩阵，如上所述，仍然可以使用转换公式（5.9）进行普通的 ROWISE 或 Lumnwise 乘积；事实上，这就是 Lapack 带状程序的工作方式。然而，在带宽较小的情况下，这样做的向量长度较短，循环开销相对较高，所以效率不高。有可能做得比这好得多。

如果我们看一下矩阵元素在矩阵-向量乘积中是如何使用的，我们会发现主对角线被用作

$$y_i \leftarrow y_i + A_{ii} x_i$$

第一个超对角线被用作

$$y_i \leftarrow y_i + A_{ii+1} x_{i+1}, \quad \text{for} i < n$$

第一条对角线为

$$y_i \leftarrow y_i + A_{ii-1} x_{i-1}, \quad \text{for} i > 1.$$

换句话说，整个矩阵-向量乘积只需执行三个长度为 $n$（或 $n-1$）的向量运算，而不是长度为 3（或 2）的 $n$ 内积。

```
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)
      end
end
```

✍ **练习 5.21** 写一个程序，通过对角线计算 $y \leftarrow A^t x$。用你喜欢的语言来实现它，并在一个随机矩阵上测试它。

✍ **练习 5.22** 如果矩阵在带内是密集的，上述代码片段是有效的。例如，二维 BVP 的矩阵就不是这种情况。写出只使用非零对角线的矩阵-向量乘积的代码。

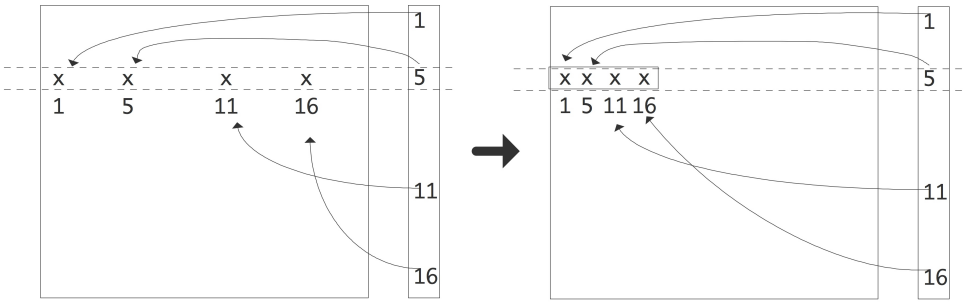
✍ **练习 5.23** 矩阵相乘比矩阵与向量相乘更难。如果矩阵 $A$ 的左半带宽度为 $p_A, q_Q$，而矩阵 $B$ 的左半带宽度为 $p_B, q_B$，$C = AB$ 的左半带宽度为多少？假设已经为 $C$ 分配了一个足够大的数组，请写一个程序来计算 $C \leftarrow AB$。

### 5.4.1.3 压缩行存储

如果我们有一个不具有简单带状结构的稀疏矩阵，或者非零对角线的数量变得不切实际，我们可以使用更通用的**压缩行存储**（Compressed Row Storage，CRS）方案。顾名思义，这个方案是基于压缩所有的行，消除零值；见图 5.2。由于这失去了非零最初来自哪一列的信息，我们必须明确地存储这些信息。考虑一个稀疏矩阵的例子。

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}$$

在压缩了所有的行之后，我们将所有的非零点存储在一个单一的实数数组中。列的索引也同样存储在一个整数数组中，我们存储指向列开始位置的指针。使用基于 0 的索引，这就得到了。




| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 的一个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 [55]。你可能会遇到的另一种存储方案是坐标存储，其中矩阵被存储为三联体列表 $< i, j, a_{ij} >$. 流行的矩阵市场网站 [161] 使用这种方案的一个变体。

### 5.4.1.4 关于压缩行存储的算法

在这一节中，我们将看一下一些算法在 CRS 中的形式。首先我们考虑稀疏矩阵-向量乘积的实现。

```
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;
}
```

标准的矩阵-向量乘积算法 $y = Ax$，其中每一行 $A_i$ 与输入向量 $x$ 进行内积。然而，请注意，内循环不再以列号为索引，而是以要找到该列号的位置为索引。这个额外的步骤被称为间接寻址。

✍ **练习 5.24** 比较密集矩阵-向量乘积（按行执行）和刚才给出的稀疏乘积的数据位置性。说明对于一般的稀疏矩阵来说，在处理输入向量 $x$ 时的空间局部性已经消失了。是否有一些矩阵结构，我们仍然可以期待一些空间局部性？

现在，如果我们想计算乘积 $y = A^t x$ 呢？在这种情况下，我们需要 $A^t$ 的行，或者说，相当于 $A$ 的列。找到 $A$ 的任意列是很难的，需要大量的搜索，所以可能认为这个算法也相应地难以计算。幸运的是，这并不是真的。

如果我们将标准算法中的 $i$ 和 $j$ 循环交换为 $y = Ax$，我们可以得到

| Original: | Indices reversed: |
|---|---|
| $y \leftarrow 0$ | $y \leftarrow 0$ |
| for $i$ : | for $j$ : |
| for $j$ : | for $i$: |
| $y_i \leftarrow y_i + a_{ij}x_j$ | $y_i \leftarrow y_i + a_{ji}x_j$ |

使用第二种算法来计算 $A^t x$ 的乘积。

✍ **练习 5.25** 写出转置积 $y = A^t x$ 的代码，其中 $A$ 是以 CRS 格式存储。写一个简单的测试程序，确认你的代码计算正确。

✍ **练习 5.26** 如果需要同时访问行和列呢？实现一种算法，测试以 CRS 格式存储的矩阵是否对称。提示：保留一个指针数组，每行一个，用来记录在该行的进展情况。

✍ **练习 5.27** 到目前为止所描述的操作是相当简单的，因为它们从未改变过矩阵的稀疏结构。如上所述，CRS 格式并不允许在矩阵中添加新的非零点，但做一个允许添加的扩展并不难。

令数字 $p_i, i = 1...n$，描述第 $i$ 行中非零点的数量。设计一个对 CRS 的扩展，使每一行都有 $q$ 额外元素的空间。实施这个方案并进行测试：构造一个第 $i$ 行有 $p_i$ 个非零点的矩阵，在添加新元素之前和之后检查矩阵-向量乘积的正确性，每行最多有 $q$ 个元素。

现在假设矩阵中的非零点总数不会超过 $q^n$。修改你的代码，使其能够处理从空矩阵开始，并逐渐在随机位置添加非零点。再次，检查正确性。

我们将在第 6.5.5 节在共享内存并行的背景下重新审视转置积算法。

### 5.4.2 稀疏矩阵和图论

许多关于稀疏矩阵的论点都可以用图论来表述。为了了解为什么可以这样做，考虑一个大小为 $n$ 的矩阵 $A$，并观察到我们可以通过 $V = 1, ..., n$ 定义一个图 $< E, V >$，$E = \{(i, j)a_{ij} \neq 0\}$。这被称为矩阵的邻接图。为简单起见，我们假设 $A$ 有一个非零的对角线。如果有必要，我们可以给这个图附加权重，定义为 $w_{ij} = a_{ij}$。然后，该图被表示为 $< E, V, W >$。(如果你不熟悉图论的基本知识，请看附录 18)。图的属性现在与矩阵属性相对应；例如，图的度数是每行的最大非零数，不包括对角线元素。再比如，如果矩阵的图形是一个无向图，这意味着 $a_{ij} \neq 0 \Leftrightarrow a_{ij} \neq 0$。我们称这样的矩阵为结构对称：它不是真正意义上的对称，即 $\forall_{ij} a_{ij} = a_{ij}$ ，但上三角的每个非零都对应于下三角的一个，反之亦然。

#### 5.4.2.1 变换下的图的特性

考虑矩阵图的一个好处是，图的性质并不取决于我们如何对节点进行排序，也就是说，它们在矩阵的变化下是不变的。

✍ **练习 5.28** 让我们来看看，当矩阵的图形 $G =< V, E, W >$ 的节点被重新编号后会发生什么。作为一个简单的例子，我们对节点进行反向编号；也就是说，以 $n$ 为节点数，我们将节点 $i$ 映射为 $n + 1 - i$。相应地，我们发现一个新

的图 $G' =< V, E', W' >$ 其中

$$(i, j) \in E' \Leftrightarrow (n+1-i, n+1-j) \in E$$

这种重新编号对与 $G$ 相对应的矩阵 $A'$ 意味着什么？如果你交换两个节点上的标签 $i, j$，对矩阵 $A$ 有什么影响？

注释 有些属性在变异中保持不变。说服自己变异不会改变矩阵的特征值。

有些图的特性很难从矩阵的稀疏模式中看出来，但从图中更容易推导出来。

练习 5.29 让 $A$ 是大小为 $n$ 的一维 BVP 的三对角矩阵，$n$ 为奇数。$A$ 的图形是什么样子的？请考虑将节点按以下顺序排列所产生的变化。

$$1, 3, 5, ..., n, 2, 4, 6, ...n-1.$$

矩阵的疏散模式是什么样子的？

现在取这个矩阵，将最接近矩阵" 中间" 的对角线元素归零：让

$$a_{(n+1)/2,(n+1)/2+1} = a_{(n+1)/2+1,(n+1)/2} = 0$$

描述一下这对 $A$ 的图形有什么影响。这样的图被称为**可约图**（reducible）。现在应用前面练习中的置换法，画出所产生的稀疏模式。请注意，现在图形的可还原性更难从稀疏模式中读出。

### 5.4.3 稀疏矩阵的 LU 因子化

一维 BVP 导致了一个具有三对角系数矩阵的线性系统。如果我们做一步高斯消除，唯一需要消除的元素是在第二行。

$$\begin{pmatrix} 2 & -1 & 0 & \dots & \\ -1 & 2 & -1 & & \\ 0 & -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots & \ddots \end{pmatrix} \Rightarrow \left(\begin{array}{c|ccc} 2 & -1 & 0 & \dots \\ \hline 0 & 2-\frac{1}{2} & -1 & \\ 0 & -1 & 2 & -1 \\ & \ddots & \ddots & \ddots & \ddots \end{array}\right)$$

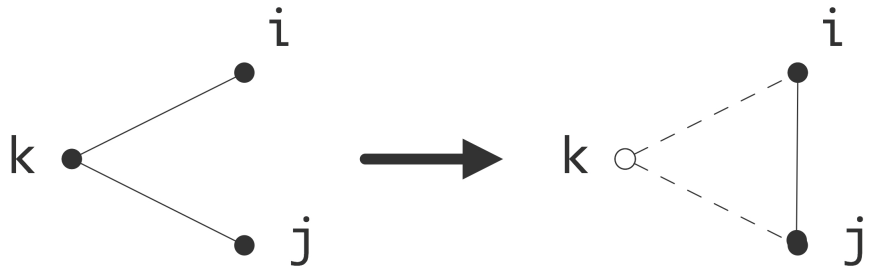
有两个重要的观察结果：一个是这个消除步骤并没有将任何零元素变为非零。另一个观察结果是，剩下要消除的那部分矩阵又是三对角的。归纳起来，在消除过程中，没有零元素变为非零：$L+U$ 的稀疏模式与 $A$ 相同，因此因式分解所需的存储空间与矩阵的相同。

不幸的是，三对角矩阵的情况并不典型，我们很快就会看到二维问题的情况。但首先我们将把第 5.4.2 节关于图论的讨论扩展到因子化。

#### 5.4.3.1 稀疏 LU 因子化的图论

在讨论稀疏矩阵的 LU 因子化时，图论往往是有用的。让我们研究一下消除第一个未知数（或扫除第一列）在图论方面意味着什么。我们假设是一个结构对称的矩阵。

我们认为消除一个未知数是一个过程，它将一个图 $G =< V, E >$ 变成一个图 $G' =< V', E' >$。这些图形之间的关系首先是一个顶点，比如说 $k$，已经从顶点中被移除。$k \notin V', V' \cup k = V$ 。




$E$ 和 $E$ 之间的关系更为复杂。在高斯消除算法中，消除变量 $k$ 的结果是：声明

$$a_{ij} \leftarrow a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}$$

对所有 $i, j \neq k$ 执行。如果 $a_{ij}$ 最初 $\neq 0$，即 $(ij) \in E$，那么 $a_{ij}$ 的值只是被改变。如果原矩阵中的 $a_{ij} = 0$，即 $(i, j) \notin E$，在消除 $k$ 未知数后，会有一个非零元素，称为补入元素。

$$(i, j) \notin E \quad \text{but} \quad (i, j) \in E'$$

这在图 5.3 中有所说明。

综上所述，消除一个未知数可以得到一个少了一个顶点的图，并且对所有 $ij$ 都有边，这样在 $i$ 或 $j$ 与被消除的变量 $k$ 之间有边。图 5.4 给出了一个小矩阵上的完整说明。

✍ **练习 5.30** 回到练习 5.28。使用图形参数来确定奇数变量被消除后的稀疏模式。

✍ **练习 5.31** 证明上述关于消除单一变量的论证的一般化。设 $I \subset V$ 是任意一个顶点集合，设 $J$ 是连接到 $I$ 的顶点。

$$J \cap I = \phi, \quad \forall_{i \in I} \exists_{j \in J} : (i, j) \in E.$$

现在表明，消除 $I$ 中的变量会导致一个图 $< VE >$，其中 $J$ 中的所有节点在剩余的图中是相连的，如果它们之间有一条路径通过 $I$。

$$\forall_{j_1, j_2 \in J} : \quad \text{there is a path} j_1 \rightarrow j_2 \text{through} I \text{in} E \Rightarrow (j_1, j_2) \in E'$$

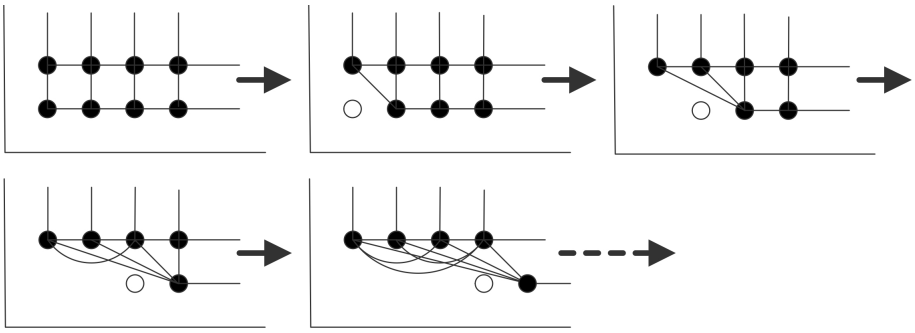
### 5.4.3.2 填充

现在我们回到二维问题的矩阵因式分解上。我们把这种大小为 $N \times N$ 的矩阵写成块维度为 $n$ 的块矩阵，每个块的大小为 $n$。(复习问题：这些块从何而来？) 现在，在第一个消除步骤中，我们需要将两个元素归零，即 $a_{21}$ 和 $_{n+1,1}$。

$$\left(\begin{array}{ccccc:ccc} 4 & -1 & 0 & \dots & & -1 & & \\ -1 & 4 & -1 & 0 & \dots & 0 & -1 & \\ & \ddots & \ddots & \ddots & & & \ddots & \\ \hdashline -1 & 0 & \dots & & & 4 & -1 & \\ 0 & -1 & 0 & \dots & & -1 & 4 & -1 \end{array}\right) \Rightarrow \left(\begin{array}{c:ccc:ccc} 4 & -1 & 0 & \dots & -1 & & \\ \hdashline & 4-\frac{1}{4} & -1 & 0 & \dots & -1/4 & -1 & \\ & \ddots & \ddots & \ddots & & & \ddots & \ddots \\ \hdashline & -1/4 & & & & 4-\frac{1}{4} & -1 & \\ & -1 & 0 & & & -1 & 4 & -1 \end{array}\right)$$

你看，消除 $a_{21}$ 和 $a_{n+1,1}$ 会导致两个填充元素的出现：在原矩阵中，$a_{2,n+1}$ 和 $a_{n+1,2}$ 为零，但在修改后的矩阵中，这些位置是非零的。我们将填充位置定义为 $(i, j)$，其中 $a_{ij} = 0$，但是 $(L+U)_{ij} \neq 0$。很明显，在因式分解过程中，矩阵被填满了。只要有一点想象力，你也可以看到，在第一个对角线块之外的带子中的每个元素都会填满。然而，使用第 5.4.3.1 节的图形方法，可以很容易地将所产生的填充连接可视化。在图 5.5 中，对 2d BVP 例子的图进行了说明。(第一行的每个变量被消除后，都会在下一个变量和第二行之间以及第二行的变量之间建立联系。归纳起来，你会发现在第一行被消除后，第二行是完全连接的。(将此与练习 5.30 联系起来)

✍ **练习 5.32** 完成这个论证。第二行的变量是完全相连的，这对矩阵结构意味着什么？在图中画出第二行的第一个变量被消除后的情况。



✍ **练习 5.33** 用于密集线性代数的 LAPACK 软件有一个 *LU* 因子化程序，可以用因子覆盖输入矩阵。上面你看到了这是可能的，因为 $L$ 的列正是随着 $A$ 的列被消除而产生的。如果矩阵是以稀疏格式存储的，为什么这样的算法是不可能的？

### 5.4.3.3 填充估计

在上面的例子中，你看到稀疏矩阵的因式分解所占用的空间可能比矩阵本身要大得多，但仍比存储整个矩阵维度大小的正方形数组要小。现在我们将给出因式分解的空间复杂度的一些界限，也就是执行因式分解算法所需的空间量。

✍ **练习 5.34** 证明以下陈述：

1. 假设矩阵 $A$ 有一个半带宽 $p$，也就是说，如果 $|i-j|>p$，则 $a_{ij}=0$。表明在无主元的因式分解后，$L+U$ 具有相同的半带宽度。
2. 证明在经过部分主元的因式分解后，$L$ 的左半带宽为 $p$，而 $U$ 的右半带宽为 $2p$。
3. 假设没有透视，表明填空可以有如下特征：考虑行 $i$。让 $j_{min}$ 是第 $i$ 行中最左边的非零点，即在 $j<j_{min}$ 时，$a_{ij}=0$。那么，在 $j_{min}$ 列左边的第 $i$ 行中就不会有填充物。同样地，如果 $i_{min}$ 是第 $j$ 列中最上面的非零值，那么在第 $j$ 列中就不会有高于 $i_{min}$ 行的填充。给定一个稀疏矩阵，现在很容易分配足够的存储来适应一个没有主元的因式分解：这被称为天际线存储。

✍ **练习 5.35** 考虑矩阵

$$A=\begin{pmatrix} a_{11} & 0 & a_{13} & & & & \phi \\ 0 & a_{22} & 0 & a_{24} & & & \\ a_{31} & 0 & a_{33} & 0 & a_{35} & & \\ & a_{42} & 0 & a_{44} & 0 & a_{46} & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots \\ \varnothing & & & & a_{n,n-1} & 0 & a_{nn} \end{pmatrix}$$

前面已经证明，执行 LU 因式分解的任何填充都只限于包含原始矩阵元素的带。在这种情况下，不存在填充。归纳地证明这一点。

看一下邻接图。（这种图有个名字，叫什么?）你能根据这个图给出一个没有填充的证明吗？练习 5.33 表明，我们可以为带状矩阵的因式分解分配足够的存储空间。

- 对于带宽为 p 的矩阵的因式分解，有一个大小为 $N\times p$ 的阵列就可以了。
- 矩阵的一半带宽 $p$ 和一半带宽 $q$ 的部分主元的因素化。可以用 $N\times(p+2q+1)$ 来描述。
- 可以根据具体的矩阵来构建足以存储因子化的 Askylin 配置文件矩阵。

我们可以将这个估计应用于二维 BVP 的矩阵，第 4.2.3 节。

✍ **练习 5.36** 证明在方程（4.16）中，原始矩阵有 $O(N) = O(n^2)$ 非零元素，$O(N^2) = O(n^4)$ 元素，而因子化有 $O(nN) = O(n^3) = O(N^{3/2})$ 非零。

这些估计表明，一个 LU 因式分解所需的存储量可能比 A 所需的更多，而且这个差别不是一个常数，而是与矩阵大小有关。在没有证明的情况下，我们指出，到目前为止你所看到的那种稀疏矩阵的求逆数是完全密集的，所以存储它们需要更多的时间。这是解决线性系统 $Ax = y$ 的一个重要原因，在实践中不是通过计算 $A^{-1}$ 和随后乘以 $x = A^{-1}y$ 来完成。(数值稳定性是不这样做的另一个原因)。事实上，即使是因式分解也会占用很多空间，这是考虑迭代方法的一个原因，我们将在第 5.5 节中进行。

上面，你看到一个大小为 $n \times n$ 的密集矩阵的因式分解需要 $O(n^3)$ 操作。对于稀疏矩阵来说，这又是怎么回事呢？让我们考虑一个半带宽度为 $p$ 的矩阵的情况，并假设原始矩阵在该带内是密集的。主元元素 $a_{11}$ 用于将第一列中的 $p$ 元素归零，对于每一个第一行的元素都要加到该行，涉及到 $p$ 乘法和加法。总而言之，我们发现操作的数量大致为

$$\sum_{i=1}^{n} p^2 = p^2 n$$

加上或减去低阶项。

✍ **练习 5.37** 对于二维 BVP 的矩阵来说，初始密集带的假设并不成立。为什么上述估计仍然成立，直到一些低阶项？

在上面的练习 5.33 中，你得出了一个易于应用的填充量的估计。然而，这可能是一个相当高的估计。我们希望能以比实际因式分解更少的工作量来计算或估计填充的数量。现在我们将勾勒出一种算法，用于寻找 $L+U$ 中非零点的确切数量，其成本与这个数字呈线性关系。我们将在（结构上）对称的情况下这样做。关键的观察是以下几点。假设列 i 在对角线下有一个以上的非零。

$$\begin{pmatrix} \ddots & & & & \\ & a_{ii} & & a_{ij} & & a_{ik} \\ & & \ddots & & & \\ & a_{ji} & & a_{jj} & & \\ & & & & \ddots & \\ & a_{ki} & & ?a_{kj}? & & a_{kk} \end{pmatrix}$$

在第 $i$ 步中消除 $a_{ki}$ 会导致更新 $a_{kj}$，如果最初 $a_{kj} = 0$，则会有一个填充元素。然而，我们可以推断出这个非零值的存在：消除 $a_{ji}$ 会导致位置 $(jk)$ 的填充元素，而且我们知道结构对称性被保留了。换句话说，如果我们只计算非零点，那么只需看看消除 $(j,i)$ 位置的影响，或者一般来说，消除主元下方的第一个非零点的影响。按照这个论点，我们只需要在每一个主元的一行中记录非零点，整个过程的复杂性与因式分解中的非零点数量成线性关系。

#### 5.4.3.4 减少填充

矩阵的图形属性，如度数和直径，在对变量重新编号时是不变的。其他属性，如因子化过程中的填充，会受到重新编号的影响。事实上，值得研究的是，是否有可能通过对矩阵图的节点重新编号来减少填充量，或者说，通过对线性系统进行置换来减少填充量。

✍ **练习 5.38** 考虑" 箭头" 矩阵，只在第一行和第二列以及对角线上有非零点。

$$\begin{pmatrix} \star & \star & \cdots & * \\ \star & \star & & \varnothing \\ \vdots & & \ddots & \\ \star & \varnothing & & * \end{pmatrix}$$

假设任何加法都不为零，那么矩阵中和因式分解中的非零数是多少？你能找到一个问题的变量的对称排列，使新的矩阵没有填充物吗？

这个例子并不典型，但是通过对矩阵进行巧妙的置换，有时确实可以改善填充估计（例如见第 6.8.1 节）。即使这样，作为一项规则，声明也是成立的，稀疏矩阵的 LU 因式分解比矩阵本身需要的空间大得多。这也是下一节中迭代方法的激励因素之一。

#### 5.4.3.5 Fill-in reducing orderings

一些矩阵的特性在对称排列下是不变的。

✍ **练习 5.39** 在线性代数课上，你通常会研究矩阵的特性，以及它们在基数变化下是否不变，特别是在单元基的变换下。

$$B = VAV^t, \quad \text{其中} VV^t = I.$$

证明对称互换是一种特殊的基础变化。说出一些矩阵属性在单数变换下不会改变。

其他属性则不然：在上一节中，你看到填空量就是其中之一。因此，你可能会想知道什么是最好的排序，以减少给定矩阵的因式分解的填充量。这个问题在实践中是难以解决的，但存在各种启发式方法。其中一些启发式方法也可以从并行的角度进行论证；事实上，嵌套剖分的排序将只在关于并行的 6.8.1 节中讨论。在这里，我们简要地展示一下其他两个启发式方法，它们比并行性的需求更早。

首先，我们看一下 Cuthill-McKee 排序，它直接将填充矩阵的带宽降到最低。由于填充量可以用带宽来约束，我们希望这样一个减少带宽的排序也能减少填充量。

其次，我们将考虑最小度数排序，其目的是更直接地减少填充量。

**Cuthill-McKee 排序**：Cuthill-McKee 排序 [36] 是一种减少带宽的排序，它通过对变量进行水平集排序来实现。它考虑了矩阵的邻接图，并按以下步骤进行

1. 取一个任意的节点，称其为" 零级"。
2. 给定一个级别 $n$，将所有连接到级别 n 的节点，以及尚未进入级别的节点，分配到级别 $n+1$。
3. 对于所谓的" 反向 Cuthill-McKee 排序"，请将各层的编号倒过来。

✍ **练习 5.40** 证明根据 Cuthill-McKee 排序对矩阵进行置换有一个块状三对角结构。

我们将在第 6.10.1 节中考虑并行性时重新审视这个算法。当然，我们可以想知道带宽可以减少到什么程度。

✍ **练习 5.41** 一个图的直径被定义为两个节点之间的最大最短距离。

1. 论证在二维椭圆问题的图中，这个直径是 $O(N^{1/2})$。
2. 用带宽来表示节点 1 和 $N$ 之间的路径长度。
3. 认为这给出了直径的下限，并使用指针对带宽进行了下限边界。

**最小程度排序**另一个排序的动机是观察到填充量与节点的程度有关。

✍ **练习 5.42** 证明消除一个度数为 d 的节点会导致最多 $2^d$ 的填充元素所谓最小度数排序的过程如下：

- 找到具有最低度数的节点。
- 剔除该节点，并更新其余节点的学位信息。
- 从第一步开始，用更新的矩阵图重复。

✍ **练习 5.43** 指出上述两种方法的区别。这两种方法都是基于对矩阵图的检查；然而，最小度数方法在使用的数据结构上需要更大的灵活性。解释一下原因并详细讨论两个方面。

## 5.5 迭代法

高斯消除法，即使用 LU 因式分解法，是一种寻找线性系统解的简单方法，但正如我们在上面看到的，在那种来自离散化 PDE 的问题中，它可以产生大量的填充。在这一节中，我们将研究一种完全不同的方法，即迭代求解，即通过一连串的逼近来找到系统的解。

这个计算方案看起来，非常粗略，就像。

$$\begin{cases} \text{Choose any starting vector} x_0 \text{and repeat for} i \geq 0: \\ x_{i+1} = Bx_i + c \\ \text{until some stopping test is satisfied.} \end{cases}$$

里的重要特征是，没有任何系统是用原始系数矩阵解决的；相反，每一次迭代都涉及到矩阵-向量乘法或一个更简单系统的解决。因此，我们用一个重复的更简单、更便宜的操作取代了一个复杂的操作，即构造一个 LU 因式分解并用它来解决一个系统。这使得迭代方法更容易编码，并有可能更有效率。

让我们考虑一个简单的例子来激励迭代方法的精确定义。假设我们想解决系统

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

其解为（2，1，1）。假设你知道（例如，从物理学的角度考虑），这么做的分量是大致相同的大小。观察对角线的主导大小，然后，决定

$$\begin{pmatrix} 10 & & \\ & 7 & \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

可能是一个很好的近似值。这有一个解决方案（2.1，9/7，8/6）。显然，解决一个只涉及原系统对角线的系统既容易做到，而且，至少在这种情况下，相当准确。

$$\begin{pmatrix} 10 & & \\ 1/2 & 7 & \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

有解（2.1，7.95/7，5.9/6）。解三角形系统比对角线系统更费事一些，但仍比计算 LU 因式分解容易得多。另外，我们在寻找这个近似解的过程中没有产生任何填充物。

因此我们看到，有一些容易计算的方法可以合理地接近解决方案。我们能否以某种方式重复这个技巧呢？

更抽象地表述一下，我们所做的是不求解 $Ax = b$，而是求解 $L\tilde{x} = b$。现在将 $\Delta x$ 定义为与真解的距离：$\tilde{x} = x + \Delta x$。这样，$A\Delta x = A\tilde{x} - b \equiv r$，其中 $r$ 是残差。接下来我们再次求解 $L\widetilde{\Delta x} = r$，并更新 $x = x - \widetilde{\Delta x}$。

| iteration | 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 |

在这种情况下，我们每次迭代得到两个小数，这并不典型。

现在很清楚为什么迭代方法会有吸引力。如上图所示，通过高斯消除法解决一个系统需要进行 $O(n^3)$ 运算。如果矩阵是密集的，则上述方案中的单次迭代需要 $O(n^2)$ 操作，对于稀疏矩阵，可能需要低至 $O(n)$ 操作。如果迭代次数少，这就使得迭代方法具有竞争力。

✍ **练习 5.44** 当比较迭代法和直接法时，运算次数不是唯一相关的指标。概述一些与两种情况下的代码效率有关的问题。同时比较解决一个线性系统和解决多个线性系统的情况。

### 5.5.1 抽象介绍

现在是时候对上述例子的迭代方案做一个正式介绍了。假设我们想解决 $Ax = b$，而直接求解的成本太高，但乘以 A 是可行的。再假设我们有一个矩阵 $K \approx A$，这样就可以廉价地求出 $Ax = b$ 。

我们不求解 $Ax = b$，而是求解 $Kx = b$，并定义 $x_0$ 为解决方案：$Kx_0 = b$。这就给我们留下了一个误差 $e_0 = x_0 - x$，对此我们有一个公式 $A(x_0 - e_0) = b$ 或 $Ae_0 = A_0 - b$。我们把 $r_0 \equiv Ax_0 - b$ 残差；然后误差满足 $Ae_0 = r_0$。

如果我们能从 $Ae_0 = r_0$ 的方程中解出误差，我们就完成了：然后找到真正的解决方案。然而，由于上次用 $A$ 求解的成本太高，我们这次也不能这样做。

所以我们近似地确定误差修正。我们求解 $K\tilde{e}_0 = r_0$，并设定 $x_1 = x_0 - \tilde{e}_0$；故事可以继续进行，$e_1 = x_1 - x$，$r_1 = Ax_1 - b$，$K\tilde{e}_1 = r_1$，$x_2 = x_1 - \tilde{e}_1$，等等。

那么，迭代方案就是

```
Let x0 be given
For i  0:
    let ri = Axi - b
    compute ei from Kei = ri update xi+1 = xi - ei
```

我们把基本方案称为

$$x_{i+1} = x_i - K^{-1}r_i$$

一个固定的迭代。它是静止的，因为每次更新都是以相同的方式进行的，不依赖于迭代数。这种方案的分析很简单，但不幸的是适用性有限。

关于迭代方案，我们有几个问题需要回答

- 这个方案是否总能带我们找到解决方案？- 如果这个方案收敛了，有多快?
- 我们何时停止迭代?
- 我们如何选择 $K$?

现在我们将对这些问题给予一些关注，尽管全面的讨论超出了本书的范围。

### 5.5.2 收敛性和误差分析

我们从迭代方案是否收敛，以及收敛速度如何的问题开始。考虑一个迭代步骤。

$$\begin{aligned} r_1 &= Ax_1 - b = A\left(x_0 - \tilde{e}_0\right) - b \\ &= r_0 - AK^{-1}r_0 \\ &= \left(I - AK^{-1}\right)r_0 \end{aligned}$$

归纳起来，我们发现 $r_n = (I - AK^{-1})^n r_0$，所以如果所有的特征值都满足 $|\lambda(I - AK^{-1})| < 1$[2]，那么 $r_n \downarrow 0$。最后这句话给了我们一个收敛的条件，即把 $K$ 与 $A$ 联系起来，以及一个几何收敛率。

如果 $K$ 足够接近的话，我们可以得到一个收敛的条件。

✍ **练习 5.45** 为 $e_n$ 推导一个类似的归纳关系。

通过计算实际的特征值，很难确定是否满足条件 $|\lambda(I - AK^{-1})| < 1$。然而，有时格什哥林定理（附录 13.5）给了我们足够的信息。

✍ **练习 5.46** 考虑我们从二维 BVP 离散化中得到的方程 (4.16) 的矩阵 $A$。设 $K$ 为包含 $A$ 的对角线的矩阵，即 $k_i = a_i$ 和 $k_{ij} = 0$，对于 $i \neq j$。使用格什哥林定理证明 $|\lambda(I - AK^{-1})| < 1$。这个练习中的论证很难推广到更复杂的 $K$ 的选择，例如你将在下面看到。这里我们只说对于某些矩阵 $A$，这些 $K$ 的选择将总是导致收敛，其速度随着矩阵大小的增加而减少。除了说明对于 $M$ 矩阵（见第 4.2.2 节），这些迭代方法会收敛外，我们将不做详细说明。关于静止迭代方法收敛理论的更多细节，见 [195]。

### 5.5.3 计算形式

在上面第 5.5.1 节中，我们将静止迭代推导为一个涉及到乘以 $A$ 和解 $K$ 的过程。然而，在某些情况下，更简单的实现是可能的。考虑一下这样的情况：$A = K - N$ ，并且我们知道 $K$ 和 $N$ 。然后，我们将 $Ax = b$ 写为

$$Kx = Nx + b$$

我们观察到，满足 (5.13) 的 x 是迭代的一个固定点。

$$Kx^{(n+1)} = Nx^{(i)} + b$$

很容易看出，这是一个静止的迭代。

这就是方程（5.11）的基本形式。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，让 $K = DA$，即包含 $A$ 的对角线部分的矩阵：$k_i = a_i$ 和 $k_{ij} = 0$，对于所有 $i \neq j$。同样地，$n_i = 0$ 和 $n_{ij} = -a_{ij}$ 对于所有 $i \neq j$。

这就是所谓的雅可比方法。迭代方案 $Kx^{(n+1)} = Nx^{(n)} + b$ 现在变为

```
for t = 1, ... until convergence, do:
    for i = 1 ... n:
        //ax(t+1)=\sum ax(t)+b becomes:
ii i j\neqslanti ij j i
x(t+1)=a-1(\sum ax(t)+b) i ii j\neqslant iijj i
```

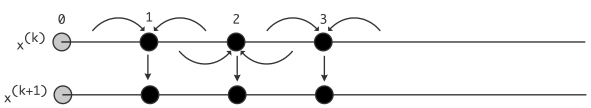
(考虑到除法的成本相对较高，第 1.2 节，我们实际上会明确地存储 $a^{-1}$ 的数量，并以乘法代替除法)。

这就要求我们为当前迭代 $x^{(t)}$ 准备一个向量，为下一个向量 $x^{(t+1)}$ 准备一个临时 $u$。最简单的写法可能是。

```
for t = 1, ... until convergence, do: for i = 1 ... n:
u=a-1(-\sum ax+b) i ii j\neqslantiijj i
copy x <- u
```

对于一个简单的一维问题，如图 5.6 所示：在每个 $x_i$ 点上，两个相邻点的值与当前值相结合，产生一个新值。由于所有 $x_i$ 点的计算都是独立的，这可以在并行计算机上并行完成。

但是，你可能会想，在总和 $\sum_{j \neq i} a_{ij} x_j$ 为什么不使用已经计算过的 $x^{(t+1)}$ 的值？就向量 $x^(t)$ 而言，这意味着



```
for k = 1, ... until convergence, do:
    for i = 1 ... n:
        x(t+1)=a-1(-\sum ax(t+1)-\sum ax(t)+b)
```

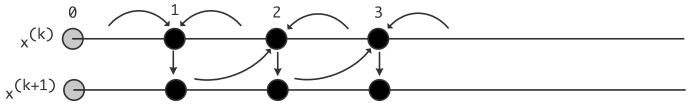
令人惊讶的是，该方法的实现比雅可比方法更简单。

```
for t = 1, ... until convergence, do:
    for i = 1 ... n:
         x=a-1(-\sum ax+b)
```

如果你把这写成一个矩阵方程，你会发现新计算的元素 $x^(t+1)$ 与 $D_A + L_A$ 的元素相乘，而旧的元素 $x_j$ 被 $U_A$ 所取代，得到

$$(D_A + L_A)x^{(k+1)} = -U_A x^{(k)} + b$$

这被称为 Gauss-Seidel 方法。




对于一维的情况，高斯-塞德尔方法如图 5.7 所示；每一个 $x_i$ 点仍然结合其邻居的值，但现在左边的值实际上是来自下一个外迭代。

最后，我们可以在 Gauss-Seidel 方案中插入一个阻尼参数，从而得到 Successive Over- Relaxation（SOR）方法。

```
for t = 1, ... until convergence, do:
    for i = 1 ... n:
         x(t+1)=wa-1(-\sum a x(t+1)-\sum a x(t)+b)+(1-w)x(t)
```

令人惊讶的是，对于看起来像插值的东西，该方法实际上在 $\omega \in (02)$ 的范围内对 $\omega$ 起作用，最佳值大于 1[96]。计算最佳的 $\omega$ 并不简单。

### 5.5.4 该方法的收敛性

我们对两个问题感兴趣：首先，迭代方法是否完全收敛，如果是的话，速度如何。这些问题背后的理论远远超出了本书的范围。上面我们说过，对于 $M$ 矩阵来说，收敛通常是可以保证的；至于收敛速度，通常只有在模型情况下才能进行全面分析。对于来自 BVP 的矩阵，如第 4.2.3 节所述，我们不需要证明，就说系数矩阵的最小特征值是 $O(h^2)$。那么，上面得出的几何收敛比 $|\lambda(I - AK^{-1})|$ 可以证明如下：

- 对于雅可比方法，该比率为 $1 - O(h^2)$。
- 对于高斯-赛德尔迭代，它也是 $1 - O(h^2)$，但该方法的收敛速度是两倍。
- 对于 SOR 方法，最佳 $\Omega$ 可以将收敛系数提高到 $1 - O(h)$。

### 5.5.5 雅可比与高斯-塞德尔和并行性的对比

以上，我们主要是从数学的角度看雅可比、高斯-赛德尔和 SOR 方法。然而，这种考虑在很大程度上被现代计算机上的并行化问题所取代。

首先，我们观察到雅可比方法的一次迭代中的所有计算都是完全独立的，所以它们可以简单地被矢量化或并行完成。高斯-赛德尔法则不同（从现在开始我们忽略 SOR，因为它与高斯-赛德尔法的区别仅在于阻尼参数）：由于一次迭代的 $x_i$ 点的计算现在是独立的，这种类型的迭代不是简单的矢量化或在并行计算机上实现。

在许多情况下，这两种方法被认为是被 CG 或广义最小残差（GMRES）方法所取代的（第 5.5.11 和 5.5.13 节）。雅可比方法有时被用作这些方法的预处理。高斯-赛德尔方法仍然很流行的一个地方是作为一个多网格平滑器。在这种情况下，经常通过使用变量的红黑排序来找到并行性。

关于这些问题的进一步讨论可以在第 6.7 节找到。

### 5.5.6 K 的选择

上面的收敛和误差分析表明，$K$ 越接近 $A$，收敛就越快。在最初的例子中，我们已经看到 $K$ 的对角线和下三角的选择。我们可以通过让 $A = D_A + L_A + U_A$ 来正式描述这些，即把 $A$ 分成对角线、下三角、上三角部分。下面是一些方法及其传统名称：

- 理查德森迭代法：$K = \alpha I$。
- 雅可比方法：$K = D_A$（对角线部分）。
- 高斯-塞德尔方法：$K = D_A + L_A$（下三角，包括对角线）
- SOR 法：$K = \omega^{-1} D_A + L_A$
- 对称 SOR（SSOR）法：$K \neq D_A + L_A) D_A^{-1} (D_A + U_A)$。
- 在迭代细化中，我们让 $K = LU$ 是 $A$ 的真正因子化。在精确算术中，求解系统 $LUx = y$ 可以得到精确解，所以在迭代方法中使用 $K = LU$ 会在一步之后得到收敛。在实践中，舍入误差会使解不精确，所以人们有时会迭代几步以获得更高的精度。

✍ **练习 5.47** 假设是密集系统，几步迭代细化的额外成本是多少?

✍ **练习 5.48** 线性系统 $Ax = b$ 的雅可比迭代定义为

$$x_{i+1} = x_i - K^{-1}(Ax_i - b)$$

其中 K 是 A 的对角线。证明你可以转换这个线性系统（也就是说，找到一个不同的系数矩阵和右手边的向量，仍然有相同的解），这样你就可以计算相同的 $x_i$ 向量，但是 K = I，身份矩阵。

这种策略在存储和操作数方面有什么影响？如果 A 是一个稀疏的矩阵，是否有特殊的影响?

假设 A 是对称的。请举一个简单的例子，说明 K-1A 不一定是对称的。你能想出一个不同的系统变换，使系数矩阵的对称性得到保留，并具有与上述变换相同的优点吗？你可以假设该矩阵有正对角线元素。

✍ **练习 5.49** 证明上一练习的变换也可以用于高斯-赛德尔方法。给出几个理由，为什么这不是一个好主意。

✍ **练习 5.50** 静态迭代可以看作是不精确的牛顿方法的一种形式，每次迭代都使用相同的导数逆的近似值。标准的函数分析结果 [120] 说明了这种近似值可以偏离精确的逆值多远。

一个特殊的情况是迭代细化，牛顿方法应该在一个步骤内收敛，但实际上由于计算机运算中的舍入，需要多个步骤。只要函数（或残差）的计算足够精确，牛顿方法就会收敛，这一事实可以通过以较低的精度进行 $LU$ 解来加以利用，从而获得更高的性能 [1]。

选择预处理矩阵 $K$ 有许多不同的方法。其中有一些是按常规定义的，比如下面讨论的不完全因式分解。其他的选择则是受到差分方程的启发。例如，如果算子是

$$\frac{\delta}{\delta x}\left(a(x,y)\frac{\delta}{\delta x}u(x,y)\right) + \frac{\delta}{\delta y}\left(b(x,y)\frac{\delta}{\delta y}u(x,y)\right) = f(x,y)$$

那么矩阵 $K$ 可由算子导出

$$\frac{\delta}{\delta x}\left(\tilde{a}(x)\frac{\delta}{\delta x}u(x,y)\right) + \frac{\delta}{\delta y}\left(\tilde{b}(y)\frac{\delta}{\delta y}u(x,y)\right) = f(x,y)$$

对于一些选择的 $\tilde{a}\tilde{b}$。第二组方程被称为可分离问题，并且有快速解算器，意思是它们具有 $O(N \log N)$ 的时间复杂性；见 [200]。

#### 5.5.6.1 构建 $K$ 的不完全 $LU$ 因子化

我们简单提一下 $K$ 的另一个选择，这是受高斯消除法的启发。和高斯消除法一样，我们让 $K = LU$，但现在我们使用不完全 $LU$（$ILU$）分解法。记住，常规的 $LU$ 因式分解是昂贵的，因为有填充现象。在不完全因式分解中，我们人为地限制了填入的情况。

如果我们把高斯消除法写成

```
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整的变体

```
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]
```

- 得到的因式分解不再是精确的：$LU \approx A$，所以它被称为不完全 $LUILU$ 因式分解。
- $ILU$ 因式分解比完全因式分解占用的空间要小得多：$L+U$ 的稀疏程度与 $A$ 相同。

上述算法被称为"ILU(0)"，其中的"0 " 指的是在不完全因式分解过程中绝对不允许填入。其他允许有限数量的填充的方案也存在。关于这个方法可以说得更多；我们只想说，对于 $M$ 矩阵，这个方案通常会给出一个收敛的方法 [154]。

✍ **练习 5.51** 矩阵-向量乘积的运算次数与用 ILU 因式分解解系统的运算次数如何比较?

你已经看到，稀疏矩阵的完全因式分解可能需要更高的存储量（因式分解需要 3/2，而矩阵需要 N），但是不完全因式分解需要 $O(N)$，就像矩阵一样。因此，我们可能会惊讶地发现，误差矩阵 $R = A - LU$ 不是密集的，而是本身是稀疏的。

✍ **练习 5.52** 设 A 是泊松方程的矩阵，LU 是不完全因式分解，R = A - LU 。证明 R 是一个双对角矩阵

- 考虑到 R 是由那些在分解过程中被忽略的元素组成的。它们在矩阵中的位置是什么?
- 或者，写出乘积 LU 的稀疏性模式，并与 A 的稀疏性模式进行比较。

#### 5.5.6.2 构建预处理程序的成本

在热方程的例子中（第 4.3 节），你看到每个时间步骤都涉及到解决一个线性系统。作为一个重要的实际结果，解决线性系统的任何设置成本，如构建预处理程序，将在要解决的系统序列中摊销。类似的论点在非线性方程的背景下也是成立的，我们将不讨论这个问题。非线性方程是通过迭代过程（如牛顿方法）来解决的，它的多维形式导致了一连串的线性系统。尽管这些系统有不同的系数矩阵，但通过重复使用牛顿步骤的预处理程序，又可以摊销设置成本。

#### 5.5.6.3 并行调节器

构建和使用一个预调节器是许多考虑因素的平衡行为：一个更准确的预调节器会导致在更少的迭代中收敛，但这些迭代可能更昂贵；此外，一个更准确的预调节器的构建成本可能更高。在并行情况下，这种情况甚至更

加复杂，因为某些预处理程序一开始就不是非常并行的。因此，我们可能会接受一个并行的预处理程序，但它的迭代次数比串行预处理程序要少。更多的讨论请见第 6.7 节。

### 5.5.7 停止测试

我们需要解决的下一个问题是何时停止迭代。上面我们看到，误差以几何级数递减，所以很明显，我们永远不会精确地达到解决方案，即使这在计算机运算中是可能的。既然我们只有这种相对收敛的行为，那么我们怎么知道什么时候已经足够接近了呢?

我们希望误差 $e_i = x - x_i$ 很小，但是测量这个误差是不可能的。上面我们观察到，$Ae_i = r_i$，所以

$$\|e_i\| \leq \left\|A^{-1}\right\| \|r_i\| \leq \lambda_{\max}\left(A^{-1}\right)\|r_i\|$$

如果我们对 $A$ 的特征值有所了解，这就给了我们一个误差的约束。($A$ 的准则是只有对称性 $A$ 的最大特征值。一般来说，我们在这里需要奇异值)。另一种可能性是监测计算出的解决方案的变化。如果这些变化很小。

$$\|x_{n+1} - x_n\| / \|x_n\| < \varepsilon$$

我们也可以得出结论，我们已经接近解决了。

✍ **练习 5.53** 证明迭代数之间的距离和到真解的距离之间的分析关系。如果你的方程中含有常数，它们可以在理论上或实践中确定吗?

✍ **练习 5.54** 编写一个简单的程序来试验线性系统的解法。从一维 BVP 中获取矩阵（使用有效的存储方案），并使用选择 $K = D_A$ 的迭代方法编程。对残差和迭代间的距离进行停止测试的实验。迭代次数如何取决于矩阵的大小?

改变矩阵结构，使某一数量加入对角线，即在原矩阵中加入 $\alpha I$。当 $\alpha > 0$ 时会发生什么? 当 $\alpha < 0$ 时会发生什么? 你能找到行为改变的数值吗? 该值是否取决于矩阵的大小?

### 5.5.8 一般多项式迭代法的理论

上面，你看到了 $x_{i+1} = x_i - K^{-1}r_i$ 的迭代方法，现在我们将看到更一般形式的迭代方法。

$$x_{i+1} = x_i + \sum_{j \leqslant i} K^{-1} r_j \alpha_{ji},$$

也就是说，使用所有以前的残差来更新迭代。有人可能会问，" 为什么不引入一个额外的参数而写成 $x_{i+1} = \alpha_i x_i + \ldots$" 这里我们给出一个简短的论证，即前一种方案描述了一大类方法。事实上，目前作者并不知道有哪些方法不属于这个方案。

在给定近似解 $\tilde{x}$ 的情况下，我们将残差定义为 $\tilde{r} = A\tilde{x} - b$。在这个一般性讨论中，我们将系统的预设条件定为 $K^{-1}AxK^{-1}b$。(见第 5.5.6 节，我们讨论了线性系统的转换。) 初始猜测 $x$ 的相应残差为

$$\tilde{r} = K^{-1}A\tilde{x} - K^{-1}b.$$

我们发现，

$$x = A^{-1}b = \tilde{x} - A^{-1}K\tilde{r} = \tilde{x} - (K^{-1}A)^{-1}\tilde{r}.$$

现在，Cayley-Hamilton 定理指出，对于每一个 $A$，都存在一个多项式 $\phi(x)$（特征多项式），以便

$$\phi(A) = 0$$

我们观察到，我们可以把这个多项式 $\phi$ 写为

$$\phi(x) = 1 + x\pi(x)$$

其中 $\pi$ 是另一个多项式。将此应用于 $K^{-1}A$，我们有

$$0 = \phi(K^{-1}A) = I + K^{-1}A\pi(K^{-1}A) \Rightarrow (K^{-1}A)^{-1} = -\pi(K^{-1}A)$$

所以 $x = \tilde{x} + \pi(K^{-1}A)\tilde{r}$。现在，如果我们让 $x_0 = \tilde{x}$，那么 $\tilde{r} = K^{-1}r_0$，得到的公式是

$$x = x_0 + \pi(K^{-1}A)K^{-1}r_0$$

这个方程提出了一个迭代方案：如果我们能找到一系列度数为 i 的多项式 $\pi(i)$ 来近似 $\pi$，它将给我们一串迭代的结果

$$x_{i+1} = x_0 + \pi^{(i)}(K^{-1}A)K^{-1}r_0 = x_0 + K^{-1}\pi^{(i)}(AK^{-1})r_0$$

最终达到真解。基于这种在迭代过程中对多项式的使用，这种方法被称为多项式迭代法。

✍ **练习 5.55** 静止的迭代方法是多项式方法吗？你能与霍纳法则建立联系吗？

将方程（5.15）乘以 $A$，两边减去 $b$，得到

$$r_{i+1} = r_0 + \tilde{\pi}^{(i)}(AK^{-1})r_0$$

其中 $\tilde{\pi}^{(i)}(x) = x\pi^{(i)}(x)$。这就立即得到了

$$r_i = \hat{\pi}^{(i)}(AK^{-1})r_0$$

其中 $\hat{\pi}(i)$ 是一个度数为 i 的多项式，$\hat{\pi}(i)(0) = 1$。这个声明可以作为迭代方法收敛理论的基础。然而，这已经超出了本书的范围。

让我们看一下方程（5.16）的几个实例。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 $\alpha_i, \beta_i$。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 $\alpha_i$。但我们已经确定 $AK_0^{-1}$ 是 $r_1, r_0$ 的组合，所以现在我们可以得出

$$\left(AK^{-1}\right)^2 r_0 \in [\![r_2, r_1, r_0]\!]$$

而且很清楚如何归纳证明

$$\left(AK^{-1}\right)^i r_0 \in [\![r_i, \cdots, r_0]\!]$$

将其代入（5.15），最后得到

$$x_{i+1} = x_0 + \sum_{j \leqslant i} K^{-1}r_j\alpha_{ji}$$

很容易看出，方案（5.14）的形式是（5.18），反过来的含义也是成立的。综上所述，迭代方法的基础是一个方案，其中迭代数被迄今为止计算的所有残差所更新。

$$x_{i+1} = x_i + \sum_{j \leqslant i} K^{-1}r_j\alpha_{ji}$$

与静止迭代（第 5.5.1 节）相比，在静止迭代中，迭代数只从最后的残差中更新，而且系数保持不变。

我们可以对 $\alpha_{ij}$ 系数说得更多。如果我们用方程（5.19）乘以 $A$，再从两边减去 $b$，我们会发现

$$r_{i+1} = r_i + \sum_{j \leqslant i} AK^{-1}r_j\alpha_{ji}$$

让我们暂时考虑一下这个方程。如果我们有一个起始残差 $r_0$，下一个残差的计算方法是

$$r_1 = r_0 + AK^{-1}r_0\alpha_{00}$$

由此我们得到 $AK^{-1}r_0 = \alpha_{00}^{-1}(r_1 - r_0)$，所以对于下一个残差。

$$\begin{aligned} r_2 &= r_1 + AK^{-1}r_1\alpha_{11} + AK^{-1}r_0\alpha_{01} \\ &= r_1 + AK^{-1}r_1\alpha_{11} + \alpha_{00}^{-1}\alpha_{01}\,(r_1 - r_0) \\ \Rightarrow AK^{-1}r_1 &= \alpha_{11}^{-1}\left(r_2 - \left(1 + \alpha_{00}^{-1}\alpha_{01}\right)r_1 + \alpha_{00}^{-1}\alpha_{01}r_0\right) \end{aligned}$$

我们看到，我们可以把 $AK^{-1}r_1$ 表示为总和 $r_2\beta_2 + r_1\beta_1 + r_0\beta_0$，并且 $\sum_i \beta_i = 0$。推而广之，我们发现（与上述不同的 $\alpha_{ij}$）：

$$\begin{aligned} r_{i+1} &= r_i + AK^{-1}r_i\delta_i + \textstyle\sum_{j\le i+1} r_j\alpha_{ji} \\ r_{i+1}\left(1 - \alpha_{i+1,i}\right) &= AK^{-1}r_i\delta_i + r_i\,(1 + \alpha_{ii}) + \textstyle\sum_{j<i} r_j\alpha_{ji} \\ r_{i+1}\alpha_{i+1,i} &= AK^{-1}r_i\delta_i + \textstyle\sum_{j\le i} r_j\alpha_{ji} && \text{substituting} \alpha_{ii} := 1 + \alpha_{ii} \\ & && \alpha_{i+1,i} := 1 - \alpha_{i+1,i} \\ r_{i+1}\alpha_{i+1,i}\delta_i^{-1} &= AK^{-1}r_i + \textstyle\sum_{j\le i} r_j\alpha_{ji}\delta_i^{-1} && \text{note that} \alpha_{i+1,i} = \textstyle\sum_{j\le i}\alpha_{ji} \\ r_{i+1}\alpha_{i+1,i}\delta_i^{-1} &= AK^{-1}r_i + \textstyle\sum_{j\le i} r_j\alpha_{ji}\delta_i^{-1} \\ r_{i+1}\gamma_{i+1,i} & AK^{-1}r_i + \textstyle\sum_{j\le i} r_j\gamma_{ji} && \text{substituting} \gamma_{ij} = \alpha_{ij}\delta_j^{-1} \end{aligned}$$

而我们有 $\gamma_{i+1,i} = \sum_{j\leqslant i}\gamma_{ji}$。

我们可以把这最后一个方程写成 $AK^{-1}R = RH$ 其中

$$H = \begin{pmatrix} -\gamma_{11} & -\gamma_{12} & \cdots & & \\ \gamma_{21} & -\gamma_{22} & -\gamma_{23} & \cdots & \\ 0 & \gamma_{32} & -\gamma_{33} & -\gamma_{34} & \\ \varnothing & \ddots & \ddots & \ddots & \ddots \end{pmatrix}$$

其中，$H$ 是一个所谓的海森堡矩阵：它是上三角加一个下对角线。我们还注意到，$H$ 每一列中的元素之和为零。

由于同一性 $\gamma_{i+1,i} = \sum_{j\leqslant i}\gamma_{ji}$，我们可以从 $b$ 方程的两边减去 $r_{i+1}$ 并" 除以 $A$"，得到

$$x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j\le i} x_j\gamma_{ji}$$

这为我们提供了迭代方法的一般形式。

$$\begin{cases} r_i = Ax_i - b \\ x_{i+1}\gamma_{i+1,i} = K^{-1}r_i + \sum_{j\le i} x_j\gamma_{ji} & \text{where} \gamma_{i+1,i} = \displaystyle\sum_{j\le i}\gamma_{ji} \\ r_{i+1}\gamma_{i+1,i} = AK^{-1}r_i + \sum_{j\le i} r_j\gamma_{ji} \end{cases}$$

ij 这种形式适用于许多迭代方法，包括你在上面看到的静止迭代方法。在接下来的章节中，你将看到 $\gamma_{ij}$ 系数是如何从残差的正交条件中得出的。

### 5.5.9 通过正交进行迭代

上面描述的静止方法（第 5.5.1 节）已经以某种形式存在了很长时间了。高斯在给一个学生的信中描述了一些变体。它们在 1950 年杨的论文 [208] 中得到完善；最后的参考资料可能是瓦尔加 [195] 的书。这些方法现在很少被使用，除了在多网格平滑器的专门背景下，这个主题在本课程中没有讨论。

几乎在同一时间，基于正交化的方法领域由两个人 [136, 104] 拉开了序幕，尽管他们花了几十年时间才找到广泛的适用性。(进一步的历史，见 [81])。

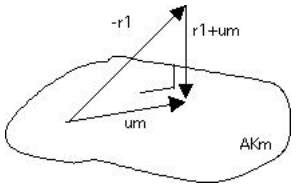
其基本思想如下。

如果你能使所有的残差相互正交，并且矩阵的尺寸为 $n$，那么经过 $n$ 的迭代，你就已经收敛了：不可能有一个 $n+1$ 的残差与之前所有的残差正交并且不为零。由于零残差意味着相应的迭代就是解，所以我们得出结论，经过 $n$ 的迭代，我们已经掌握了真正的解。

随着当代应用所产生的矩阵的大小，这种推理已经不再适用：在计算上，迭代 n 次是不现实的。此外，四舍

五入可能会破坏解决方案的任何准确性。然而，后来人们意识到 [175]，在对称正定（SPD）矩阵的情况下，这种方法是一种现实的选择。当时的推理是。

残差序列跨越了一系列维度增加的子空间，通过正交，新的残差被投射到这些空间上。这意味着它们的尺寸会越来越小。




这在图 5.8 中有所说明。在本节中，你将看到正交迭代的基本思想。这里介绍的方法只具有理论意义；接下来你将看到共轭梯度（CG）和广义最小残差（GMRES）方法，它们是许多现实生活应用的基础。

现在让我们采取基本方案（5.21）并对残差进行正交。我们使用 $K^{-1}$ 内积来代替正常内积。

$$(x, y)_{K^{-1}} = x^t K^{-1} y$$

并且我们将强制残差为 $K^{-1}$-正交。

$$\forall_{i \neq j} : r_i \perp_{K^{-1}} r_j \Leftrightarrow \forall_{i \neq j} : r_i K^{-1} r_j = 0$$

这被称为全正交法（FOM）方案。

你可能认识到其中的 Gram-Schmidt 正交化（见附录 13.2 的解释）：在每次迭代中，$r_{i+1}$ 最初被设置为 $AK^{-1}r_i$，并对 $r_j$ 进行正交，$j \leqslant i$。

我们可以使用修改后的 Gram-Schmidt，将算法改写为。

```
Let r0 be given For i  0:
let s <- K-1ri let t <- AK-1ri for j \leqslant i:
let $\gamma$j be the coefficient so that t - $\gamma$jrj  rj form s <- s - $\gamma$j xj
and t<-t-$\gamma$jrj
let xi+1 = (\sumj $\gamma$j)-1s, ri+1 = (\sumj $\gamma$j)-1t.
```

这两个版本的 FOM 算法在精确算术中是等价的，但在实际情况中却有两个不同。

状况下有两个不同之处

- 修改后的 Gram-Schmidt 方法在数值上更加稳定。
- 未修改的方法允许你同时计算所有内积。我们在下面的 6.6 节中讨论

我们将在下文第 6.6 节讨论这个问题。即使 FOM 算法在实践中没有被使用，这些计算上的考虑也会延续到下面的 GMRES 方法中。

### 5.5.10 迭代方法的耦合递归形式

上面，你看到了生成迭代和搜索方向的一般方程（5.21）。这个方程通常被分割为

- $x_i$ 迭代的更新，来自单一搜索方向。

$$x_{i+1} = x_i - \delta_i p_i$$

且

- 从目前已知的残差中构建搜索方向。

$$p_i = K^{-1} r_i + \sum_{j<i} \beta_{ij} K^{-1} r_j$$

不难看出，我们也可以通过归纳法来定义

$$p_i = K^{-1} r_i + \sum_{j<i} \gamma_{ji} p_j$$

而这最后一种形式是在实践中使用的一种。

迭代依赖系数的选择通常是为了让残差满足各种正交性条件。例如，可以选择让方法通过让残差正交来定义 $(r_i^t r_j = 0 \text{if} i \neq j)$，或者 $A$ 正交 $(r_i^t A r_j = 0 \text{if} i \neq j)$ 这种方法的收敛速度比静止迭代快得多，或者对更多的矩阵和预处理程序类型都能收敛。下面我们将看到两种这样的方法；然而，它们的分析超出了本课程的范围。

### 5.5.11 共轭梯度的方法

在本节中，我们将推导出共轭梯度（CG）方法，它是 FOM 算法的一个具体实现。特别是，在 SPD 矩阵 A 的情况下，它具有令人愉快的计算特性。CG 方法的基本形式是上述的耦合递归公式，系数的定义是要求残差序列 $r_0, r_1, r_2, \ldots$ 满足

$$r_i^t K^{-1} r_j = 0 \quad \text{if} i \neq j$$

我们首先推导出非对称系统的 CG 方法，然后说明它在对称情况下是如何简化的。(这里的方法取自 [60])。基本方程是

$$\begin{cases} x_{i+1} = x_i - \delta_i p_i \\ r_{i+1} = r_i - \delta_i A p_i \\ p_{i+1} = K^{-1} r_{i+1} + \sum_{j \leq i} \gamma_{ji+1} p_j \end{cases}$$

其中第一个和第三个方程在上面已经介绍过了，第二个方程可以通过第一个方程乘以 A 找到（检查一下!）。

现在我们将通过归纳法推导出这个方法中的系数。实质上，我们假设我们有当前的残差 $r_{cur}$，一个待计算的残差 $r_{new}$，和一个已知的残差 $R_{old}$ 的集合。我们不使用下标"old, cur, new"，而使用以下惯例

- $x_1, r_1, p_1$ 是当前迭代数、残差和搜索方向。请注意，这里的下标 1 并不表示迭代数。
- $x_2, r_2, p_2$ 是我们将要计算的迭代数、残差和搜索方向。同样，下标不等于迭代数。
- $X_0, R_0, P_0$ 是所有以前的迭代、残差和搜索方向捆绑在一起的一个向量块。

就这些数量而言，更新方程为

$$\begin{cases} x_2 = x_1 - \delta_1 p_1 \\ r_2 = r_1 - \delta_i A p_1 \\ p_2 = K^{-1} r_2 + v_{12} p_1 + P_0 u_{02} \end{cases}$$

其中 $\delta_1 v_{12}$ 是标量，$u_{02}$ 是一个向量，长度为当前迭代前的次数。现在我们从残差的正交性推导出 $\delta_1 v_{12} u_{02}$。具体来说，残差在 $K^{-1}$ 内积下必须是正交的：我们希望有

$$r_2^t K^{-1} r_1 = 0, \quad r_2^t K^{-1} R_0 = 0$$

结合这些关系，我们可以得到，比如说。

$$(R_0, r_1, r_2) \left(\begin{array}{ccc|c|c} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ \hline & & -1 & 1 & \\ \hline & & & -1 & 1 \end{array}\right) = A\,(P_0, p_1, p_2)\,\mathrm{diag}\,(D_0, d_1, d_2)$$

$$(P_0, p_1, p_2) \begin{pmatrix} I - U_{00} & -u_{01} & -u_{02} \\ & 1 & -v_{12} \\ & & 1 \end{pmatrix} = K^{-1}\,(R_0, r_1, r_2)$$

或缩写为 $RJ = APD$，$P(I-U) = R$ 其中 $J$ 是具有同一对角线和减去同一对角线的矩阵。然后我们观察到

- $R^t K^{-1} R$ 是对角线，表达了残差的正交性。
- 结合 $R-1R$ 是对角线和 $P(I-U) = R$，可以得出 $R^t P = R^t K^{-1} R (I-U)^{-1}$。我们现在我们可以推断，$(I-U)^{-1}$ 是上对角线，所以 $R^t P$ 是上三角形。这告诉我们诸如 $r_2^t p_1$ 等量是零。

- 结合 $R$ 和 $P$ 的关系，我们首先得到的是

$$R^t K^{-t} AP = R^t K^{-t} RJD^{-1}$$

这告诉我们，$R^t K^{-t} AP$ 是下对角线。在此方程中展开 $R$，可以得到

$$P^t AP = (I - U)^{-t} R^t RJD^{-1}$$

这里 $D$ 和 $R^t K^{-1} R$ 是对角线，$(I - U)^{-t}$ 和 $J$ 是下三角形，所以 $P^t AP$ 是下三角形。

- 这告诉我们，$P_0^t Ap_2 = 0$，$p_1^t Ap_2 = 0$。
- 将 P0t A, p1t A 与方程（5.23）中的 p2 定义相乘，得到

$$u_{02} = -(P_0^t AP_0)^{-1} P_0^t AK^{-1} r_2, \quad v_{12} = -(p_1^t Ap_1)^{-1} p_1^t AK^{-1} r_2.$$

- 如果 $A$ 是对称的，$P^t AP$ 是下三角（见上文）和对称的，所以它实际上是对角的。另外，$R^t K^{-}tAP$ 是下对角线，所以，利用 $A = A^t$，$P^t AK^{-1} R$ 是上对角线。由于 $P^t AK^{-1} R = P^t AP(I - U)$，我们得出结论：$I - U$ 是上二角的，所以，只有在对称情况下，$u_{02} = 0$。

关于这个推导的一些看法：

- 严格地说，我们在这里只证明了必要的关系。可以证明，这些也是充分的。
- 有一些不同的公式可以计算出相同的向量，在精确的算术中。例如，很容易推导出 $p_1^t r_1 = r_1^t r_1$，所以可以将其代入刚才的公式中。图 5.9 给出了 CG 方法的典型实现方式。
- 在第 3 次迭代中，计算 $P_0^t Ar_2$（需要用于 $u_{02}$）需要用到 k 内积。首先，内积在并行情况下是很不利的。其次，这要求我们无限期地存储所有搜索方向。这第二点意味着工作和存储都随着迭代次数的增加而上升。与此相反，在静止迭代方案中，存储仅限于矩阵和几个向量，而每次迭代的工作量是一样的。
- 刚才提出的反对意见在对称情况下消失了。由于 $u_{02}$ 为零，对 $P_0$ 的依赖消失了，只剩下对 $p_1$ 的依赖。因此，存储是恒定的，每次迭代的工作量也是恒定的。可以证明每次迭代的内积数只有两个。

✍ **练习 5.56** 对 CG 方法的一次迭代中的各种操作做一个翻转计数。假设 A 是一个五点模版的矩阵，预处理器 M 是 A 的不完全因式分解（5.5.6.1 节）。让 N 为矩阵大小。

### 5.5.12 从最小化推导出

上述 CG 方法的推导在文献中并不常见。典型的推导是从一个具有对称正定（SPD）矩阵 $A$ 的最小化问题开始。

$$\text{For which vector} x \text{with} \|x\| = 1 \text{is} f(x) = 1/2 x^t Ax - b^t x \text{minimal?}$$

如果我们接受函数 f 有一个最小值的事实，这是由正定性得出的，我们通过计算导数找到最小值

$$f'(x) = Ax - b$$

试问：f'(x) = 0 在哪？

✍ **练习 5.57** 推导上面的导数公式。(提示：将导数的定义写成 $\lim_{h \downarrow 0} \ldots$）注意，这要求 $A$ 是对称的。

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**

对于迭代方法的推导，我们指出，迭代的 x 以一定的步长 $\delta_i$ 更新。沿着一个搜索方向 $p_i$ 更新。

$$x_{i+1} = x_i + p_i \delta_i$$

最佳步长

$$\delta_i = \frac{r_i^t p_i}{p_1^t A p_i}$$

然后推导出沿着 $x_i + \delta\delta p_i$ 的函数 $f$ 的最小化。

$$\delta_i = \arg\min_{\delta} \| f(x_i + p_i \delta) \|$$

从残差中构建搜索方向是通过归纳证明的，即要求残差是正交的。一个典型的证明，见 [5]。

### 5.5.13 GMRES

在上面关于 CG 方法的讨论中，指出残差的正交性需要存储所有的残差，以及第 $k$ 次迭代中的 k 内部积。不幸的是，可以证明 CG 方法的工作节省，就所有的实际情况而言，在 SPD 矩阵之外无法找到 [62]。

GMRES 方法是这种完全正交化方案的一个流行实现。为了将计算成本控制在一定范围内，它通常被实现为一种重启的方法。也就是说，只保留一定数量（比如 $k$ =5 或 20）的残差，每迭代 $k$ 次，方法就会重新启动。

还有一些方法没有 GMRES 那样不断增长的存储需求，例如 QMR [69] 和 BiCGstab [194]。尽管根据上面的评论，这些方法不能对残差进行正交，但在实践中仍然具有吸引力。

### 5.5.14 复杂性

高斯消除法的效率相当容易评估：一个系统的因式分解和求解，确定性地需要 $\frac{1}{3}n^3$ 的操作。对于一个迭代方法来说，运算次数是每个迭代的运算次数与迭代次数的乘积。虽然每个单独的迭代都很容易分析，但没有好的理论来预测迭代次数。(事实上，一个迭代方法甚至可能一开始就不收敛。) 此外，高斯消除法的编码方式可以有相当多的缓存重用，使算法在计算机峰值速度的相当大比例下运行。另一方面，迭代方法在每秒钟的运算量上要慢得多。

所有这些考虑使得迭代方法在线性系统求解中的应用介于工艺和黑色艺术之间。在实践中，人们做了相当多的实验来决定迭代方法是否会有回报，如果有的话，哪种方法更可取。

## 5.6 特征值法

在本章中，我们到目前为止只限于线性系统的求解。特征值问题是线性代数应用的另一个重要类别，但它们的兴趣更多地在于数学而不是计算本身。我们对所涉及的计算类型做一个简要的概述。

### 5.6.1 幂法

**幂法**（power method）是一个简单的迭代过程：给定一个矩阵 A 和一个任意的起始矢量 ，反复计算

$$v \leftarrow Av, \quad v \leftarrow v/\|v\|$$

向量"$v$ " 很快就成为与绝对值最大的特征值相对应的特征向量大小，因此 $\|Av\|/\|v\|$ 成为该最大特征值的近似值。

对 $A^{-1}$ 的应用被称为**反迭代**（inverse iteration），它可以得到绝对值最小的特征值的反值。

功率法的另一个变种是移位逆向迭代，它可以用来寻找内部特征值。如果 $\sigma$ 接近一个内部特征值，那么对 $A - \sigma I$ 的反迭代将找到该内部特征值。

### 5.6.2 正交迭代法

不同特征值的特征向量是正交的，这一事实可以被利用。例如，在找到一个特征向量后，可以在与之正交的子空间中进行迭代。另一个选择是在一个向量块上进行迭代，并在每次幂法迭代后对这个块进行正交。这将产生与区块大小一样多的显性特征值。重新启动的 Arnoldi 方法 [141] 是这种方案的一个例子。

### 5.6.3 全光谱法

刚才讨论的迭代方案只产生局部的特征值。其他方法是计算矩阵的全谱。其中最流行的是 QR 方法。

### 5.6.4 并行执行

基于 Lanczos 的方案比 QR 方法更容易并行化；讨论见 [13]。

## 5.7 延伸阅读

迭代方法是一个非常深入的领域。作为对所涉及问题的实际介绍，你可以阅读" 模板书"[9]，在 http://netlib.org/templates/。对于更深入的理论处理，请看 Saad 的书 [176]，其第一版可以在 http://www-users.cs.umn. edu/ saad/books.html 下载。

# 第 6 章 高性能线性代数

内容提要

- 集合操作
- 并行密集型矩阵-向量乘积
- 并行 LU 因子化
- 矩阵-矩阵乘积
- 稀疏的矩阵-向量乘积
- 迭代方法的计算问题
- 并行预处理程序
- 排序策略和并行性
- 解除 PDE 线性系统的并行性
- 并行性和隐式操作
- 网格更新
- 多核架构上的块状算法

在本节中，我们将讨论与并行计算机上的线性代数有关的一些问题。假设处理器的数量是有限的，而且相对于处理器的数量，问题数据总是很大。我们将关注处理器之间通信网络的物理方面问题。

我们将分析各种线性代数操作，包括迭代方法，以及它们在具有有限带宽和有限连接的网络中的行为。本章最后将对由于并行执行而产生的算法中的复杂问题进行各种简短的评论。

## 6.1 集合

集合运算在线性代数运算中起着重要的作用。事实上，操作的可扩展性可以取决于这些集合运算的成本，你将在下面看到。在此，我们对其基本思想做一个简短的讨论；详情请见 [28]。

在计算集体操作的成本时，三个架构常数足以给出下限。$\alpha$，发送单个消息的时间；$\beta$，发送数据的时间的倒数（见 1.3.2 节）；以及 $\gamma$，执行算术运算的时间的倒数。因此，发送 $n$ 数据项需要时间 $\alpha+\beta n$。我们进一步假设，一个处理器一次只能发送一条信息。我们对处理器的连接性不做任何假设；因此，这里得出的下限将适用于广泛的架构。

上述架构模型的主要含义是，在算法的每一步中，活动处理器的数量只能增加一倍。例如，要做一个广播，首先处理器 0 向 1 发送，然后 0 和 1 可以向 2 和 3 发送，然后 0-3 向 4-7 发送，等等。这种信息的级联被称为处理器网络的**最小生成树**（minimum spanning tree），由此可见，任何集体算法都有至少 $\alpha \log_2 p$ 与累计延迟有关的成本。

### 6.1.1 广播

向 p 处理器广播至少需要 $\lceil \log_2 p \rceil$ 步，总延迟为 $\lceil \log_2 p \rceil \alpha$。由于要发送 $n$ 元素，这至少会增加所有元素离开发送处理器的时间 $n\beta$，所以总成本下限为

$$\lceil \log_2 p \rceil \alpha + n\beta$$

我们可以用下面的方法来说明生成树的方法。

| | $t=1$ | $t=2$ | $t=3$ |
|---|---|---|---|
| $p_0$ | $x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$ | $x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$ | $x_0, x_1, x_2, x_3$ |
| $p_1$ | | $x_0\downarrow, x_1\downarrow, x_2\downarrow, x_3\downarrow$ | $x_0, x_1, x_2, x_3$ |
| $p_2$ | | | $x_0, x_1, x_2, x_3$ |
| $p_3$ | | | $x_0, x_1, x_2, x_3$ |

(在 $t=1$ 时，$p_0$ 发送至 $p_1$；在 $t=2$ 时，$p_0, p_1$ 发送至 $p_2, p_3$) 这个算法的 $\log_2 \alpha$ 项正确，但是处理器 0 重复发送整个向量。所以带宽成本为 $\log_2 n\beta$。如果 $n$ 较小，则延迟成本占主导地位，因此我们可以将其描述为一个**短向量集合操作**（short vector collective operation）。下面的算法将广播实现为散点算法和桶状旅算法的结合。首

先是散射：

| | $t=0$ | $t=1$ | $t=2$ | $t=3$ |
|---|---|---|---|---|
| $p_0$ | $x_0\downarrow, x_1, x_2, x_3$ | $x_0, x_1\downarrow, x_2, x_3$ | $x_0, x_1, x_2\downarrow, x_3$ | $x_0, x_1, x_2, x_3\downarrow$ |
| $p_1$ | | $x_1$ | | |
| $p_2$ | | | $x_2$ | |
| $p_3$ | | | | $x_3$ |

需要 $p-1$ 条大小为 $N/p$ 的信息，总时间为

$$T_{scatter}(N,P) = (p-1)\alpha + (p-1)\frac{N}{P}beta$$

**桶队算法**（bucket brigade algorithm）让每个处理器在每一步都处于活动状态，接受部分信息（除了第一步），并将其传递给下一个处理器。

| | $t=0$ | $t=1$ | *etcetera* | |
|---|---|---|---|---|
| $p_0$ | $x_0\downarrow$ | $x_0$ | $x_3\downarrow$ | $x_0, \quad x_2, x_3$ |
| $p_1$ | $x_1\downarrow$ | $x_0\downarrow, x_1$ | | $x_0, x_1, \quad x_3$ |
| $p_2$ | $x_2\downarrow$ | $x_1\downarrow, x_2$ | | $x_0, x_1, x_2$ |
| $p_3$ | $x_3\downarrow$ | $x_2\downarrow, x_3$ | | $x_1, x_2, x_3$ |

每个部分信息被发送 $p-1$ 次，所以这个阶段的复杂度也是

$$T_{bucket}(N,P) = (p-1)\alpha + (p-1)\frac{N}{P}beta$$

现在的复杂度变成了

$$2(p-1)\alpha + 2\beta n(p-1)/p$$

在延迟方面并不理想，但如果 $n$ 较大，则是一种较好的算法，使之成为一种**长向量集合操作**（long vector collective operation）。

### 6.1.2 规约

在规约操作中，每个处理器都有 $n$ 数据元素，一个处理器需要将它们进行元素组合，例如计算 $n$ 的和或积。

通过在时间上向后运行广播，我们看到规约操作的通信下限同样为 $\lceil \log_2 p \rceil \alpha + n\beta$。缩减操作也涉及到计算，按顺序计算总时间为 $(p-1)\gamma n$ 中的每个项目都在 $p$ 处理器上被缩减。由于这些操作有可能被并行化，因此计算的下限 $\frac{(p-1)}{p}\gamma n$，总共有

$$\lceil \log_2 p \rceil \alpha + n\beta + \frac{p-1}{p}\gamma n.$$

我们举例说明生成树算法，使用符号 $x_i^{(j)}$ 表示最初在处理器 $j$ 上的数据项 $i$，而 $x_i$ 表示处理器 $j\ldots k$ 的项目之和。

| | $t=1$ | $t=2$ | $t=3$ |
|---|---|---|---|
| $p_0$ | $x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}$ | $x_0^{(0:1)}, x_1^{(0:1)}, x_2^{(0:1)}, x_3^{(0:1)}$ | $x_0^{(0:3)}, x_1^{(0:3)}, x_2^{(0:3)}, x_3^{(0:3)}$ |
| $p_1$ | $x_0^{(1)}\uparrow, x_1^{(1)}\uparrow, x_2^{(1)}\uparrow, x_3^{(1)}\uparrow$ | | |
| $p_2$ | $x_0^{(2)}, x_1^{(2)}, x_2^{(2)}, x_3^{(2)}$ | $x_0^{(2:3)}\uparrow, x_1^{(2:3)}\uparrow, x_2^{(2:3)}\uparrow, x_3^{(2:3)}\uparrow$ | |
| $p_3$ | $x_0^{(3)}\uparrow, x_1^{(3)}\uparrow, x_2^{(3)}\uparrow, x_3^{(3)}\uparrow$ | | |

在时间 $t=1$ 时，处理器 $p_0$、$p_2$ 从 $p_1$、$p_3$ 接收，在时间 $t=2$ 时，$p_0$ 从 $p_2$ 接收。和上面的广播一样，这个算法没有达到下限；相反，它的复杂度为

$$\lceil \log_2 p \rceil (\alpha + n\beta + \frac{p-1}{p}\gamma n)$$

对于短向量，$\alpha$ 项占主导地位，所以这个算法就足够了。对于长向量，可以如上所述，使用其他算法 [28]。

长向量的减少可以用一个桶状旅，然后用一个聚集来完成。复杂度同上，只是桶队执行部分规约，时间为

$\gamma(p-1)N/p$。聚会不进行任何进一步的操作。

### 6.1.3 Allreduce

Allreduce 操作在每个处理器上对 n 元素进行同样的元素规约计算，但是将结果留在每个处理器上，而不是仅仅留在生成树的根部。这可以实现为先规约后广播的方式，但是存在更聪明的算法。

值得注意的是，Allreduce 的成本下限与简单规约的成本下限几乎相同：因为在规约过程中，并不是所有的处理器都在同一时间活动，我们假设额外的工作可以被完美地分散。这意味着，延迟和计算量的下限保持不变。对于带宽，我们的理由如下：为了使通信完全并行化，$\frac{p-1}{p}n$ 项目必须到达和离开每个处理器。因此，我们有一个总时间为

$$\lceil \log_2 p \rceil \alpha + 2\frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma.$$

### 6.1.4 Allgather

在对 n 元素的聚集操作中，每个处理器都有 $n/p$ 元素，一个处理器将它们全部收集起来，而不像规约操作那样将它们合并。Allgather 计算的是同样的集合，但将结果留在所有处理器上。

我们再次假设有多个目标的集合是同时活动的。由于每个处理器都会生成一个最小生成树，因此我们有 $\log_2 p\alpha$ 延迟；由于每个处理器从 $p-1$ 个处理器接收 $n/p$ 元素，因此有 $(p-1)\times(n/p)\beta$ 带宽成本。那么，通过 allgather 构建一个长度为 $n$ 的向量的总成本是

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p}n\beta$$

我们对此进行说明。

| | $t=1$ | $t=2$ | $t=3$ |
|---|---|---|---|
| $p_0$ | $x_0\downarrow$ | $x_0x_1\downarrow$ | $x_0x_1x_2x_3$ |
| $p_1$ | $x_1\uparrow$ | $x_0x_1\downarrow$ | $x_0x_1x_2x_3$ |
| $p_2$ | $x_2\downarrow$ | $x_2x_3\uparrow$ | $x_0x_1x_2x_3$ |
| $p_3$ | $x_3\uparrow$ | $x_2x_3\uparrow$ | $x_0x_1x_2x_3$ |

在时间 $t=1$ 时，邻居 $p_0$,$p_1$ 和同样的 p2,p3 之间有一个交换；在 $t=2$ 时，$p_0$,$p_2$ 和同样的 $p_1,p_3$ 之间有两个距离的交换。

### 6.1.5 Reduce-scatter

在 reduce-scatter 操作中，每个处理器都有 $n$ 元素，并对它们进行 $n$ 方式的规约。与 reduce 或 allreduce 不同的是，结果会被分解，并像散点操作那样进行分配。

形式上，处理器 i 有一个项目 $x^{(i)}$，它需要 $\sum_j x_i^{(j)}$。我们可以通过做大小 $p$ 的缩减来实现，在一个处理器上收集向量 $(\sum_i x_0^{(i)}, \sum_i x_1^{(i)}, \ldots)$，并将结果散布出去。然而，有可能在所谓的双向交换算法中结合这些操作。

| | $t=1$ | $t=2$ | $t=3$ |
|---|---|---|---|
| $p_0$ | $x_0^{(0)},x_1^{(0)},x_2^{(0)},x_3^{(0)}$ | $x_0^{(0:2:2)},x_1^{(0:2:2)}\downarrow$ | $x_0^{(0:3)}$ |
| $p_1$ | $x_0^{(1)},x_1^{(1)},x_2^{(1)}\downarrow,x_3^{(1)}\downarrow$ | $x_0^{(1:3:2)}\uparrow,x_1^{(1:3:2)}$ | $x_1^{(0:3)}$ |
| $p_2$ | $x_0^{(2)}\uparrow,x_1^{(2)}\uparrow,x_2^{(2)},x_3^{(2)}$ | $x_2^{(0:2:2)},x_3^{(0:2:2)}\downarrow$ | $x_2^{(0:3)}$ |
| $p_3$ | $x_0^{(3)}\uparrow,x_1^{(3)}\uparrow,x_2^{(3)},x_3^{(3)}$ | $x_0^{(1:3:2)}\uparrow,x_1^{(1:3:2)}$ | $x_3^{(0:3)}$ |

reduce-scatter 可以被认为是一个反向的 allgather 运行，加上算术，所以成本是

$$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p}n(\beta+\gamma)$$

## 6.2 并行密集型矩阵-向量乘积

在本节中，我们将详细讨论并行密集矩阵-向量乘积的性能尤其是可扩展性。首先，我们将考虑一个简单的案例，并在一定程度上详细讨论并行性方面。

### 6.2.1 实现块状行的情况

在设计一个算法的并行版本时，人们常常通过对相关对象进行数据分解来进行。在矩阵-向量运算的情况下，例如乘积 $y = Ax$，我们可以选择从向量分解开始，并探索其对如何分解矩阵的影响，或者从矩阵开始，并从它推导出向量分解。在这种情况下，似乎很自然地从分解矩阵开始，而不是从分解矢量开始，因为它很可能具有更大的计算意义。我们现在有两个选择。

1. 我们对矩阵进行一维分解，将其分割成块状行或块状列，并将其中的每一个--或每一组--分配给一个处理器。
2. 或者，我们可以进行二维分解，将一个或多个一般子矩阵分配给每个处理器。

我们首先考虑以块行的方式进行分解。考虑一个处理器 $p$ 和它所拥有的行的索引集合 $I_p$，并让 $i \in I_p$ 是分配给这个处理器的行。行中的元素 $i$ 被用于操作

$$y_i = \sum_j a_{ij} x_j$$

现在推理：

- 如果处理器 $p$ 拥有所有 $x_j$ 的值，那么矩阵-向量乘积就可以简单地执行，完成后，处理器拥有正确的值 $y_j \in I_p$。
- 这意味着每个处理器都需要有一个 $x$ 的副本，这是很浪费的。同时，这也提出了数据完整性的问题：你需要确保每个处理器都有正确的 $x$ 的值。
- 在某些实际应用中（例如你之前看到的迭代方法），矩阵-向量乘积的输出直接或间接地成为下一个矩阵-向量操作的输入。对于计算 $xAxA^2x\ldots$ 的功率法来说，情况当然是这样的。由于我们的操作开始时每个处理器拥有整个 $x$，但结束时它只拥有 $Ax$ 的局部部分，所以我们有一个不匹配。
- 也许我们应该假设每个处理器在操作开始时只拥有 $x$ 的局部部分，也就是那些 $i \in I_p$ 的 $x$，这样，算法的开始状态和结束状态是一样的。这意味着我们必须改变算法，包括一些通信，使每个处理器能够获得那些 $x_i \notin I_p$ 的值。

**Input:** Processor number $p$; the elements $x_i$ with $i \in I_p$; matrix elements $A_{ij}$ with $i \in I_p$.

**Output:** The elements $y_i$ with $i \in I_p$

**for** $i \in I_p$ **do**
  $s \leftarrow 0$;
  **for** $j \in I_p$ **do**
    $s \leftarrow s + a_{ij} x_j$
  **for** $j \notin I_p$ **do**
    send $x_j$ from the processor that owns it to the current one, then;
    $s \leftarrow s + a_{ij} x_j$
  $y_i \leftarrow s$

**Procedure** Naive Parallel MVP($A, x_{local}, y_{local}, p$)

✍ **练习 6.1** 对矩阵被分解成块列的情况进行类似的推理。像上面一样，详细描述并行算法，但不要给出伪代码。

现在让我们来看看通信的细节：我们将考虑一个固定的处理器 $p$，并考虑它执行的操作和需要的通信。根据上述分析，在执行语句 $y_i = \sum_j a_{ij}x_j$ 时，我们必须知道 $j$ 值属于哪个处理器。为了确认这一点，我们写道

$$y_i = \sum_{j \in I_p} a_{ij}x_j + \sum_{j \notin I_p} a_{ij}x_j$$

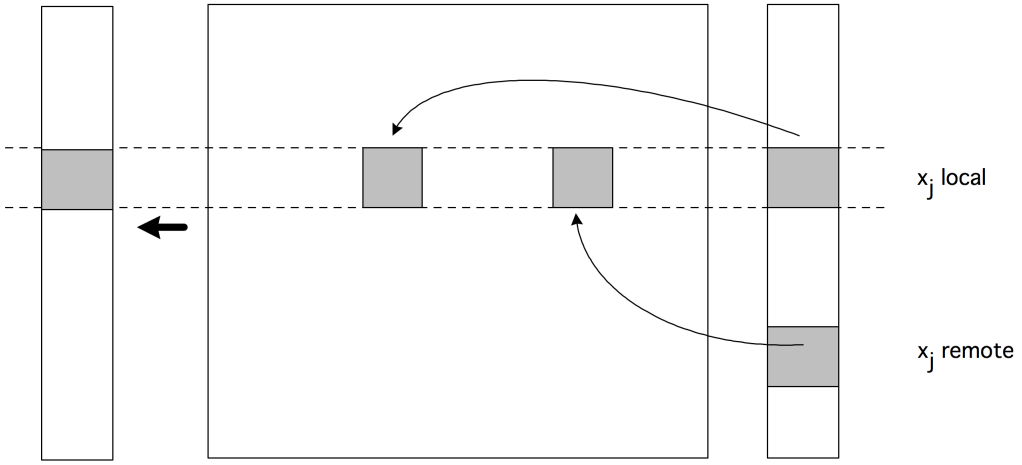
如果 $j \in I_p$，指令 $y_i \leftarrow y_i + a_{ij}x_j$ 只涉及已经属于处理器的量。因此，让我们集中讨论 $j \notin I_p$ 的情况。如果我们可以直接写出这样的语句就好了

```
y(i) = y(i) + a(i,j)*x(j)
```

而一些下层将自动把 $x^{(j)}$ 从它所存储的任何处理器转移到本地寄存器。(2.6.5 节的 PGAS 语言旨在做到这一点，但它们的效率远不能保证)。图 6.1 给出了一个基于这种乐观的并行性观点的实现。

这种" 局部" 方法的直接问题是会有太多的通信发生。

- 如果矩阵 $A$ 是密集的，那么每一行 $i \in I_p$ 都需要一次元素 $x_j$，因此每一行 $i \in I_p$ 都将被取走一次。
- 对于每个处理器 $q \neq p$，将有（很大）数量的元素 $x_j \in I_q$ 需要从处理器 $q$ 转移到 $p$。在不同的信息中做这些，而不是一次批量传输，是非常浪费的。




**Input:** Processor number $p$; the elements $x_i$ with $i \in I_p$; matrix elements $A_{ij}$ with $i \in I_p$.

**Output:** The elements $y_i$ with $i \in I_p$

**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($A, x_{local}, y_{local}, p$)

对于共享内存来说，这些问题并不是什么大问题，但在分布式内存的背景下，最好采取缓冲的方式。

我们不需要交流 x 的各个元素，而是为每个处理器 $B_{pq} \neq p$ 使用一个本地缓冲区，我们从 $q$ 收集需要在 $p$ 上执行乘法的元素。(见图 6.2 的说明。) 图 6.3 给出了并行算法。除了防止一个元素被取走超过一次之外，这也将许多小的消息合并成一个大的消息，这通常更有效率；回顾我们在 2.7.8 节中对带宽和延迟的讨论。

✍ **练习 6.2** 给出使用非阻塞操作的矩阵-向量乘积的伪码 (第 2.6.3.6 节) 上面我们说过，在每个处理器上都有一份整个 x 的副本，这是对空间的浪费。

这里隐含的论点是，一般来说，我们不希望本地存储是处理器数量的函数：理想情况下，它应该只是本地数据的函数。(这与弱缩放有关；第 2.2.5 节。) 你看，由于通信方面的考虑，我们实际上已经决定，每个处理器存储整个输入向量是不可避免的，或者至少是最好的。这种空间和时间效率之间的权衡在并行编程中相当普遍。对于密集的矩阵-向量乘积，我们实际上可以为这种开销辩护，因为向量存储的顺序比矩阵存储的顺序要低，所以我们的过度分配比例很小。下面（第 6.5 节），我们将看到，对于稀疏矩阵-向量乘积，开销会小很多。

我们很容易看到，如果允许我们忽略通信时间，那么上面描述的并行密集矩阵-向量乘积就有完美的加速。在接下来的几节中，你会发现如果我们考虑到通信，上面的块行实现并不是最佳的。对于可扩展性，我们需要一个二维分解。我们先来讨论一下集合体。

### 6.2.2 密集矩阵-向量乘积的可扩展性

在本节中，我们将对 $y \leftarrow Ax$ 的并行计算进行全面分析，其中 $x, y \in \mathbb{R}^n$ 和 $A \in \mathbb{R}^{n \times n}$。我们将假设使用 $p$ 节点，但我们对它们的连接性不做任何假设。我们将看到，矩阵的分布方式对算法的缩放有很大的影响；原始研究见 [101, 180, 188]，各种缩放形式的定义见 2.2.5 节。

#### 6.2.2.1 矩阵-向量乘积，按行划分

划分

$$A \rightarrow \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix} \quad x \rightarrow \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}, \quad \text{and} \quad y \rightarrow \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix}$$

其中，$A_i \in \mathbb{R}^{m_i \times n}$ 且 $x_i, y_i \in \mathbb{R}^{m_i}$，$\sum_{i=0}^{p-1} m_i = n$ 和 $m_i \approx n/p$。我们将首先假设 $A_i, x_i y_i$ 和最初被分配的 $P_i$。

计算的特点是，每个处理器需要整个向量 $x$，但只拥有其中的 $n/p$ 部分。因此，我们对 $x$ 进行了一次全集。之后，处理器可以执行本地产品 $y_i \leftarrow A_i x$；此后不需要进一步的通信。

那么，一个对 $y = Ax$ 进行成本计算的算法就可以通过以下方式得到

| Step | Cost (lower bound) |
|---|---|
| Allgather $x_i$ so that $x$ is available on all nodes | $\left\lceil \log_2(p) \right\rceil \alpha + \frac{p-1}{p} n\beta \approx \log_2(p)\alpha + n\beta$ |
| Locally compute $y_i = A_i x$ | $\approx 2\frac{n^2}{p}\gamma$ |

成本分析：算法的总成本约为。

$$T_p(n) = T_p^{\text{1D-row}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + n\beta}_{\text{Overhead}}$$

由于顺序成本为 $T_1(n) = 2n^2\gamma$，所以速度提升为

$$S_p^{\text{1D-row}}(n) = \frac{T_1(n)}{T_p^{\text{1D-row}}(n)} = \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \log_2(p)\alpha + n\beta} = \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}$$

和并行效率，由

$$E_p^{\text{1D-row}}(n) = \frac{S_p^{\text{1D-row}}(n)}{p} = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}}$$

乐观的观点：现在，如果我们固定 $p$，并让 $n$ 变得很大

$$\lim_{n\to\infty} E_p(n) = \lim_{n\to\infty} \left[ \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{p}{2n}\frac{\beta}{\gamma}} \right] = 1$$

因此，如果能使问题足够大，最终的并行效率几乎是完美的。然而，这是以无限内存为前提的，所以这种分析并

不实用。

悲观的观点：在强可扩展性分析中，我们固定 $n$，让 $p$ 变大，以得到

$$\lim_{p\to\infty} E_p(n) = \lim_{p\to\infty}\left[\frac{1}{1+\frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma}+\frac{p}{2n}\frac{\beta}{\gamma}}\right] = 0$$

因此，最终并行效率变得几乎不存在了。

现实主义者的观点在一个更现实的观点中，我们随着数据量的增加而增加处理器的数量。这被称为弱可扩展性，它使可用于存储问题的内存量与 $p$ 成线性比例。

让 $M$ 等于可以存储在单个节点的内存中的浮点数。那么总内存由 $Mp$ 给出。让 $n_{max}(p)$ 等于在 $p$ 节点的总内存中可以存储的最大问题大小。那么，如果所有内存都可以用于矩阵。

$$(n_{\max}(p))^2 = Mp \text{or} n_{\max}(p) = \sqrt{Mp}$$

现在的问题是，对于可以存储在 $p$ 节点上的最大问题，其并行效率如何。

$$\begin{aligned} E_p^{\text{1D-row}}(n_{\max}(p)) &= \frac{1}{1+\frac{p\log_2(p)}{2(n_{\max}(p))^2}\frac{\alpha}{\gamma}+\frac{p}{2n_{\max}(p)}\frac{\beta}{\gamma}} \\ &= \frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}} \end{aligned}$$

现在，如果分析一下当节点数量变得很大时发生了什么，就会发现

$$\lim_{p\to\infty} E_p\left(n_{\max}(p)\right) = \lim_{p\to\infty}\left[\frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2\sqrt{M}}\frac{\beta}{\gamma}}\right] = 0$$

因此，这种矩阵-向量乘法的并行算法并没有规模。

如果你仔细看一下这个效率表达式，你会发现主要问题是表达式中的 $1/\sqrt{p}$ 部分。这个条款涉及到一个系数 $\beta$，如果你跟着推导往后看，你会发现它来自处理器之间发送数据的时间。非正式地说，这可以被描述为消息大小太大，使问题可扩展。事实上，无论处理器的数量如何，消息大小都是恒定的 $n$。

另外，一个现实主义者意识到，完成一个计算的时间是有限的，$T_{max}$。在最好的情况下，即通信开销为零的情况下，我们能在 $T_{max}$ 时间内解决的最大问题为

$$T_p\left(n_{\max}(p)\right) = 2\frac{(n_{\max}(p))^2}{p}\gamma = T_{\max}$$

因此

$$(n_{\max}(p))^2 = \frac{T_{\max}p}{2\gamma} \text{or} n_{\max}(p) = \frac{\sqrt{T_{\max}}\sqrt{p}}{\sqrt{2\gamma}}$$

那么，对于可以在 $T_max$ 时间内解决的最大问题，该算法所达到的并行效率为

$$E_{p,n_{\max}} = \frac{1}{1+\frac{\log_2 p}{T}\alpha+\sqrt{\frac{p}{T}\frac{\beta}{\gamma}}}$$

而当节点数变多时，其并行效率接近

$$\lim_{p\to\infty} E_p = \sqrt{\frac{T\gamma}{p\beta}}$$

同样，随着处理器数量的增加和执行时间的上限，效率也无法维持。

我们也可以计算出这个操作的等效率曲线，即 $n$，$p$ 之间的关系，对于这个关系，效率保持不变（见 2.2.5.1 节）。如果我们将上述效率简化为 $E(n,p) = \frac{2\gamma}{\beta}\frac{n}{p}$，则 $E \equiv c$ 相当于 $n = O(p)$，因此

$$M = O(\frac{n^2}{p}) = O(p)$$

因此，为了保持效率，我们需要相当快地增加每个处理器的内存。这是有道理的，因为这淡化了通信的重要性。

#### 6.2.2.2 矩阵-向量乘积，按列划分

划分

$$A \to \left(A_0, A_1, \ldots, A_{p-1}\right) \quad x \to \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}, \quad \text{and} \quad y \to \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix}$$

其中 $A_j \in \mathbb{R}^{n \times n_j}$，$x_j, y_j \in \mathbb{R}^{n_j}$ 在 $\sum_{j=0}^{p-1} n_j = n$，$n_j \approx n/p$.

我们将首先假设 $A_j$、$x_j$ 和 $y_j$ 最初被分配给 $P_j$（但现在 $A_i$ 是一个列块）。在这种按列计算的算法中，处理器 i 可以计算长度 $n$ 向量 $A_i x_i$，而不需要事先沟通。然后，这些部分结果必须被加在一起

$$y \leftarrow \sum_i A_i x_i$$

每个处理器 $i$ 将其结果的一部分 $(A_i x_i)_j$ 分散到处理器 $j$ 中。然后，接收的处理器进行规约，将所有这些片段相加。

$$y_j = \sum_i (A_i x_i)_j.$$

然后，有成本的算法由以下方式给出。

| Step | Cost (lower bound) |
|---|---|
| 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$P_i$ | $\lceil \log_2(p) \rceil \alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$<br>$\approx \log_2(p)\alpha + n(\beta + \gamma)$ |

**成本分析**算法的总成本约为。

$$T_p^{\text{1D-col}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + n(\beta + \gamma)}_{\text{Overhead}}$$

请注意，这与成本 $T1D - row(n)$ 是相同的，只是将 $\beta$ 替换为 $(\beta + \gamma)$。不难看出关于可扩展性的结论是相同的。

#### 6.2.2.3 二维划分

下面，划分

$$A \to \begin{pmatrix} A_{00} & A_{01} & \ldots & A_{0,p-1} \\ A_{10} & A_{11} & \ldots & A_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p-1,0} & A_{p-1,0} & \ldots & A_{p-1,p-1} \end{pmatrix} \quad x \to \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{p-1} \end{pmatrix}, \quad \text{and} \quad y \to \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{p-1} \end{pmatrix}$$

其中，$A_{ij} \in \mathbb{R}^{n_i \times n_j}$，$x_i, y_i \in \mathbb{R}^{n_i}$，$\sum_{i=0}^{p-1} n_i = N$，$n_i \approx N/\sqrt{P}$.

我们将以 $r \times c$ 网格的形式来看待节点，其中 $P = rc$，并以 $p_{ij}$ 的形式来看待，其中 $i = 0, ..., r-1$ 和 $j = 0, ..., c-1$. 图 6.4，对于 3Œ4 处理器网格上的 12Œ12 矩阵，说明了数据对节点的分配，其中 $i, j$ " 单元" 显示了 $p_{ij}$ 拥有的矩阵和向量元素。

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_0$ | | | | $x_3$ | | | | $x_6$ | | | | $x_9$ | | | |
| $a_{00}$ | $a_{01}$ | $a_{02}$ | $y_0$ | $a_{03}$ | $a_{04}$ | $a_{05}$ | | $a_{06}$ | $a_{07}$ | $a_{08}$ | | $a_{09}$ | $a_{0,10}$ | $a_{0,11}$ | |
| $a_{10}$ | $a_{11}$ | $a_{12}$ | | $a_{13}$ | $a_{14}$ | $a_{15}$ | $y_1$ | $a_{16}$ | $a_{17}$ | $a_{18}$ | | $a_{19}$ | $a_{1,10}$ | $a_{1,11}$ | |
| $a_{20}$ | $a_{21}$ | $a_{22}$ | | $a_{23}$ | $a_{24}$ | $a_{25}$ | | $a_{26}$ | $a_{27}$ | $a_{28}$ | $y_2$ | $a_{29}$ | $a_{2,10}$ | $a_{2,11}$ | |
| $a_{30}$ | $a_{31}$ | $a_{32}$ | | $a_{33}$ | $a_{34}$ | $a_{35}$ | | $a_{37}$ | $a_{37}$ | $a_{38}$ | | $a_{39}$ | $a_{3,10}$ | $a_{3,11}$ | $y_3$ |
| | $x_1$ | | | | $x_4$ | | | | $x_7$ | | | | $x_{10}$ | | |
| $a_{40}$ | $a_{41}$ | $a_{42}$ | $y_4$ | $a_{43}$ | $a_{44}$ | $a_{45}$ | | $a_{46}$ | $a_{47}$ | $a_{48}$ | | $a_{49}$ | $a_{4,10}$ | $a_{4,11}$ | |
| $a_{50}$ | $a_{51}$ | $a_{52}$ | | $a_{53}$ | $a_{54}$ | $a_{55}$ | $y_5$ | $a_{56}$ | $a_{57}$ | $a_{58}$ | | $a_{59}$ | $a_{5,10}$ | $a_{5,11}$ | |
| $a_{60}$ | $a_{61}$ | $a_{62}$ | | $a_{63}$ | $a_{64}$ | $a_{65}$ | | $a_{66}$ | $a_{67}$ | $a_{68}$ | $y_6$ | $a_{69}$ | $a_{6,10}$ | $a_{6,11}$ | |
| $a_{70}$ | $a_{71}$ | $a_{72}$ | | $a_{73}$ | $a_{74}$ | $a_{75}$ | | $a_{77}$ | $a_{77}$ | $a_{78}$ | | $a_{79}$ | $a_{7,10}$ | $a_{7,11}$ | $y_7$ |
| | | $x_2$ | | | | $x_5$ | | | | $x_8$ | | | | $x_{11}$ | |
| $a_{80}$ | $a_{81}$ | $a_{82}$ | $y_8$ | $a_{83}$ | $a_{84}$ | $a_{85}$ | | $a_{86}$ | $a_{87}$ | $a_{88}$ | | $a_{89}$ | $a_{8,10}$ | $a_{8,11}$ | |
| $a_{90}$ | $a_{91}$ | $a_{92}$ | | $a_{93}$ | $a_{94}$ | $a_{95}$ | $y_9$ | $a_{96}$ | $a_{97}$ | $a_{98}$ | | $a_{99}$ | $a_{9,10}$ | $a_{9,11}$ | |
| $a_{10,0}$ | $a_{10,1}$ | $a_{10,2}$ | | $a_{10,3}$ | $a_{10,4}$ | $a_{10,5}$ | | $a_{10,6}$ | $a_{10,7}$ | $a_{10,8}$ | $y_{10}$ | $a_{10,9}$ | $a_{10,10}$ | $a_{10,11}$ | |
| $a_{11,0}$ | $a_{11,1}$ | $a_{11,2}$ | | $a_{11,3}$ | $a_{11,4}$ | $a_{11,5}$ | | $a_{11,7}$ | $a_{11,7}$ | $a_{11,8}$ | | $a_{11,9}$ | $a_{11,10}$ | $a_{11,11}$ | $y_{11}$ |

换句话说，$p_{ij}$ 拥有矩阵块 $A_j$ 以及 $x$ 和 $y$ 的一部分。这使得以下算法成为可能

算法：

- 由于 $x_j$ 分布在第 j 列，该算法首先在每个处理器 $p_{ij}$ 上通过处理器列内的 allgather 收集 $x_j$。
- 每个处理器 $p_{ij}$ 然后计算 $y_{ij} = A_{ij}x_j$ 。这不涉及进一步的沟通。
- 然后，通过将每个处理器行中的碎片 $y_i$ 收集起来，形成 $y_i$，然后将其分布在处理器行中，从而得到结果 $y_i$。这两个操作实际上是结合在一起的形成一个规约散射。
- 如果 $r = c$，我们可以在处理器上对 $y$ 数据进行转置，这样它就可以作为后续矩阵向量乘法的输入随后的矩阵-向量乘积。另一方面，如果我们正在计算 $AA^tx$，那么 $y$ 现在可以正确地分配给 $A^t$ 乘积。

**算法**带有成本分析的算法是

| Step | Cost (lower bound) |
|---|---|
| Allgather$x_i$'s within columns | $\lceil \log_2(r) \rceil \alpha + \frac{r-1}{p} n\beta$ |
| | $\approx \log_2(r)\alpha + \frac{n}{c}\beta$ |
| Perform local matrix-vector multiply | $\approx 2\frac{n^2}{p}\gamma$ |
| Reduce-scatter$y_i$'s within rows | $\lceil \log_2(c) \rceil \alpha + \frac{c-1}{p} n\beta + \frac{c-1}{p} n\gamma$ |
| | $\approx \log_2(c)\alpha + \frac{n}{c}\beta + \frac{n}{c}\gamma$ |

**成本分析**算法的总成本约为。

$$T_p^{r\times c}(n) = T_p^{r\times c}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + \left(\frac{n}{c} + \frac{n}{r}\right)\beta + \frac{n}{r}\gamma}_{\text{Overhead}}$$

现在我们将进行简化，即 $r = c = \sqrt{p}$，所以

$$T_p^{\sqrt{p}\times\sqrt{p}}(n) = T_p^{\sqrt{p}\times\sqrt{p}}(n) = 2\frac{n^2}{p}\gamma + \underbrace{\log_2(p)\alpha + \frac{n}{\sqrt{p}}(2\beta + \gamma)}_{\text{Overhead}}$$

由于顺序成本为 $T_1(n) = 2n^2\gamma$，所以速度提升为

$$S_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{T_1(n)}{T_p^{\sqrt{p}\times\sqrt{p}}(n)} = \frac{2n^2\gamma}{2\frac{n^2}{p}\gamma + \frac{n}{\sqrt{p}}(2\beta+\gamma)} = \frac{p}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

和并行效率，由

$$E_p^{\sqrt{p}\times\sqrt{p}}(n) = \frac{1}{1 + \frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}}$$

我们再次提出一个问题，对于可以存储在 $p$ 的最大问题，其并行效率是多少？节点的最大问题的并行效率是多

少。

$$E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) = \frac{1}{1+\frac{p\log_2(p)}{2n^2}\frac{\alpha}{\gamma}+\frac{\sqrt{p}}{2n}\frac{(2\beta+\gamma)}{\gamma}} = \frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}}$$

以至于

$$\lim_{p\to\infty} E_p^{\sqrt{p}\times\sqrt{p}}(n_{\max}(p)) = \lim_{p\to\infty}\frac{1}{1+\frac{\log_2(p)}{2M}\frac{\alpha}{\gamma}+\frac{1}{2\sqrt{M}}\frac{(2\beta+\gamma)}{\gamma}} = 0$$

然而，$\log_2 p$ 随 $p$ 的增长非常缓慢，因此被认为很像一个常数。在这种情况下，$E_p^{\sqrt{p}\times\sqrt{p}}(n_{max}(p))$ 下降的速度非常慢，而且该算法被认为是可扩展的，以满足实际需要。

请注意，当 $r = p$ 时，二维算法成为" 按行划分" 的算法，而当 $c = p$ 时，则成为" 按列划分" 的算法。不难看出，只要 $r/c$ 保持不变，当 $r = c$ 时，2D 算法在上述分析的意义上是可扩展的。

✍ **练习 6.3** 计算这个操作的等效率曲线。

## 6.3 并行 LU 因子化

矩阵-向量和矩阵-乘积在某种意义上是很容易并行化的。输出的元素都可以独立地以任何顺序计算，所以我们在算法的并行化方面有很多自由度。这对于计算 LU 因子化或用因子化的矩阵求解线性系统来说并不成立。

### 6.3.1 解决一个三角形系统

三角系统 $y = L^{-1}x$ （其中 $L$ 为下三角）的解是一个矩阵-向量运算，因此它与矩阵-向量乘积有其 $O(N^2)$ 的共同复杂性。然而，与乘积运算不同的是，这个求解过程包含了输出元素之间的递推关系。

$$y_i = \ell_{ii}^{-1}\left(x_i - \sum_{j<i}\ell_{ij}x_j\right)$$

这意味着并行化并非易事。在稀疏矩阵的情况下，可能会有特殊的策略；见 6.10 节。在这里，我们将对一般的稠密情况做一些评论。

为了简单起见，我们假设通信不需要时间，而且所有的算术运算都需要相同的单位时间。首先，我们考虑按行分布的矩阵，也就是说，处理器 $p$ 存储的元素 $\ell_{p*}$。有了这个，我们可以把三角解法实现为：

- 处理器 1 解决了 $t_1 = \ell_{11}^{-1}x_1$，并将其值发送给下一个处理器。
- 一般来说，处理器 $p$ 得到的值是 $y_1, ..., y_{p-1}$ 来自处理器 $p-1$，并计算出 $y_p$。
- 然后每个处理器 $p$ 将 $y_1, ..., y_p$ 发送给 $p+1$。

✍ **练习 6.4** 证明这个算法需要时间 $2N^2$，就像顺序算法一样。

这个算法中，每个处理器以流水线的方式将所有计算过的 yi 值传递给它的后继者。然而，这意味着处理器 $p$ 在最后一刻才收到 $y_1$，而这个值在第一步已经被计算出来了。我们可以以这样的方式来制定解决算法，使计算的元素尽快被提供出来。

- 处理器 1 解决 $y_1$，并将其发送给所有后来的处理器。
- 一般来说，处理器 $p$ 会等待值为 $y_q$ 的个别消息，因为 $q < p$。
- 然后，处理器 $p$ 计算 $y_p$，并将其发送给处理器 $q$，而 $q > p$。

在通信时间可以忽略不计的假设下，这个算法可以快很多。对于在 $p > 1$ 的所有处理器都同时收到 $y_1$，并能同时计算 $\ell_{p1}y_1$。

✍ **练习 6.5** 说明如果我们不考虑通信，这个算法变体需要的时间是 $O(N)$。如果我们将通信纳入成本，那么成本是多少?

✍ **练习 6.6** 现在考虑按列分布的矩阵：处理器 $p$p 储存 $\ell_{*p}$ 。概述这种分布下的三角解算法，并说明并行解的时间是 $O(N)$。

### 6.3.2 因式分解，密集型情况

对并行密集 $LU$ 分解的可扩展性的全面分析是相当复杂的，所以我们将声明，无需进一步证明，正如在矩阵-向量的情况下，需要一个二维分布。然而，我们可以发现一个进一步的复杂情况。由于任何类型的因式分解 3 都是通过矩阵进行的，处理器在部分时间内是不活动的。

✍ **练习 6.7** 考虑有规律的右看高斯消除法

```
for k=1..n
  p = 1/a(k,k)
  for i=k+1,n
    for j=k+1,n
      a(i,j) = a(i,j)-a(i,k)*p*a(k,j)
```

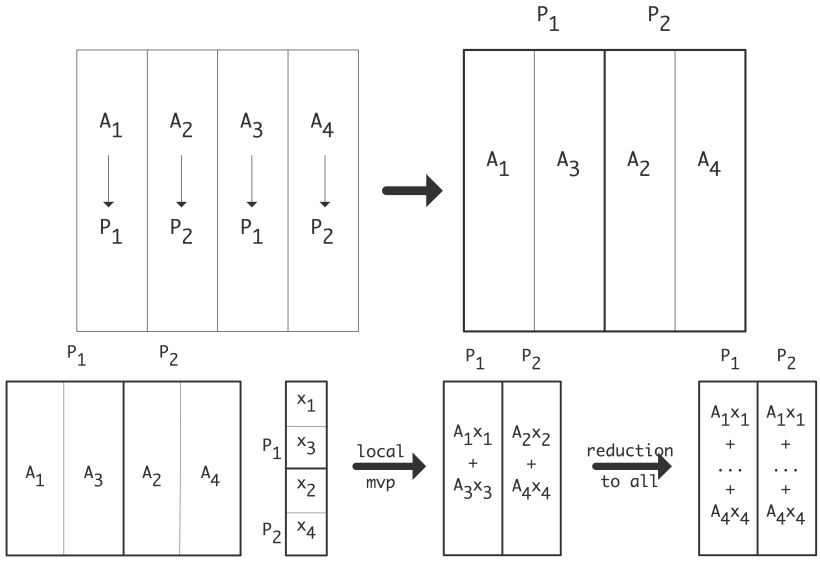
分析运行时间、加速和效率与 $N$ 的函数关系，如果我们假设有一个一维分布，并且有足够的处理器为每个处理器存储一列。表明提速是有限的。也对二维分解进行这种分析，每个处理器存储一个元素。

由于这个原因，使用了过度分解，矩阵被分成比处理器数量更多的块，每个处理器存储几个不相邻的子矩阵。我们在图 6.5 中说明了这一点，我们将一个矩阵的四个块列划分给两个处理器：每个处理器在一个连续的内存块中存储两个非连续的矩阵列。

接下来，我们在图 6.6 中说明，在不知道处理器存储了矩阵的非连续部分的情况下，可以对这样的矩阵进行矩阵-向量乘积。只需要输入的向量也是循环分布的。

✍ **练习 6.8** 现在考虑一个 4Œ4 的矩阵和一个 2Œ2 的处理器网格。矩阵的行和列都是循环分布的。说明只要输入分布正确，如何使用连续的矩阵存储来进行矩阵-向量乘积。输出是如何分布的？说明比起一维例子的规约，需要更多的通信。

具体来说，在 $P < N$ 处理器的情况下，为简单起见，假设 $N = cP$，我们让处理器 0 存储行 0，c，2c，3c，...；处理器 1 存储行 1，c+1，2c+1，... 等。这个方案可以概括为二维分布，如果 $N = c_1P_1 = c_2P_2$ 和 $P = P_1P_2$。这被称为二维循环分布。这个方案可以进一步扩展，考虑块的行和列（块的大小较小），将块的行 0，c，2c，....，分配给处理器 0。




✍ **练习 6.9** 考虑一个正方形 $n \times n$ 矩阵，以及一个正方形 $p \times p$ 处理器网格，其中 $p$ 除以 $n$ 无余。考虑上述的过度分解，并对 $n = 6$，$p = 2$ 的特定情况下的矩阵元素分配做一个草图。也就是说，画一个 $n \times n$ 表，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表格，描述局部到全局的映射，即给出局部矩阵中元素的全局 $i, j$ 坐标。(你会发现使用零基编号可以方便地完成这项任务。现在写出 $i, j$ 的函数 $P, Q, I, J$，描述全局到局部的映射，即矩阵元素 $a_{ij}$ 被存储在处理器 $(P(i, j), Q(i, j))$ 的 $(I(i, j), J(i, j))$ 位置上。

### 6.3.3 因式分解，稀疏情况

稀疏矩阵 $LU$ 因式分解是一个众所周知的难题。任何类型的因式分解都涉及到顺序部分，这使得它从一开始就不是简单的事情。为了了解稀疏情况下的问题，假设你按顺序处理矩阵的行。密集情况下，每行有足够多的元素，可以在那里衍生出并行性，但是在稀疏情况下，这个数字可能非常低。

解决这个问题的方法是认识到我们对因式分解本身不感兴趣，而是我们可以用它来解决一个线性系统。由于对矩阵进行置换可以得到相同的解，也许本身就是置换过的，所以我们可以探索具有更高平行度的置换。

矩阵排序的话题已经在第 5.4.3.5 节中出现了，其动机是填空式规约。我们将在下面考虑具有有利的并行性的排序：第 6.8.1 节中的嵌套剖析，以及第 6.8.2 节中的多色排序。

## 6.4 矩阵-矩阵乘积

矩阵三乘法 $C \leftarrow AB$(或 $C \leftarrow AB + \gamma C$，如基本线性代数子程序 (BLAS) 中使用的那样) 有一个简单的并行结构。假设所有大小为 $N \times N$ 的正方形矩阵

- $N^3$ 内积 $a_{ik}b_{kj}$ 可以分别被独立计算，之后
- $N^2$ 个元素 $c_{ij}$ 是通过独立的求和规约形成的，每个求和规约的时间为对数 $\log_2 N$。

然而，更有趣的是数据移动的问题。

- 在 $N^3$ 个操作的 $O(N^2)$ 元素上，有相当多的机会进行数据重用（第 1.6.1 节）。我们将在第 6.4.1 节探讨"Goto 算法"。
- 根据矩阵的遍历方式，TLB 的重用也需要被考虑（第 1.3.8.2 节）。
- 矩阵-矩阵乘积的分布式内存版本尤其棘手。假设 $A$、$B$、$C$ 都分布在一个处理器网格上，$A$ 的元素必须通过一个处理器行，而 $B$ 的元素必须通过一个处理器列。我们在下面讨论" 坎农算法" 和外积法。

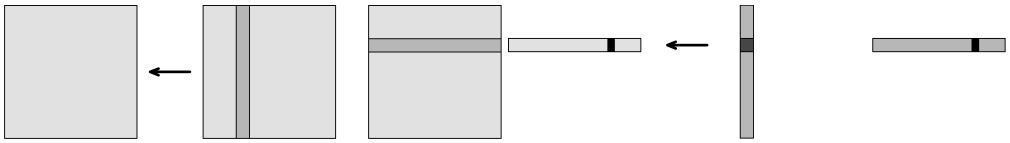
### 6.4.1 转到矩阵-矩阵乘积

在第 1.6.1 节中，我们认为矩阵的矩阵乘积（或 BLAS 术语中的 dgemm）有大量可能的数据重用：在 $O(n^3)$ 数据上有 $O(n^2)$ 操作。现在我们将考虑一个实现，由于 Kazushige Goto [85]，它确实达到了接近峰值的性能。

矩阵-矩阵算法有三个循环，每个循环我们都可以屏蔽，从而得到一个六路的嵌套循环。由于输出元素上没有递归，所有产生的循环交换都是合法的。结合这个事实，循环阻塞引入了三个阻塞参数，你会发现潜在实现的数量是巨大的。这里我们介绍了支撑 Goto 实现的全局推理；详细讨论见所引用的论文。

我们首先将乘积 $C \leftarrow A \cdot B$ （或者根据 Blas 标准，$C \leftarrow C + A \cdot B$）写成一个低秩更新序列。

$$C_{**} = \sum_k A_{*k} B_{k*}$$

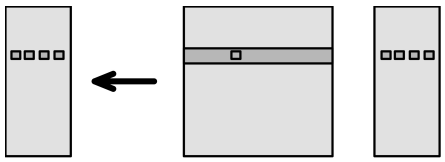


为 4，通过积累。

```
// compute C[i,*] :
  for k:
    C[i,*] = A[i,k] * B[k,*]
```

见图 6.9。现在调整该算法

- 我们需要足够的寄存器来处理 $C[i,*]$、$A[i,k]$ 和 $B[k,*]$。在目前的处理器上，这意味着我们要积累四个 $C$ 元素。
- $C$ 的这些元素被积累起来，所以它们停留在寄存器中，唯一的数据转移是对 $A$ 和 $B$ 元素的加载；没有存储！这就是为什么我们要把这些元素放在寄存器中。
- 元素 $A[i,k]$ 和 $B[k,*]$ 来自 L1。
- 由于相同的 $A$ 块被用于许多连续的 $B$ 片断，我们希望它保持常驻；我们选择 $A$ 的块大小，让它保持在二级缓存中。
- 为了防止 TLB 问题，$A$ 被按行存储。如果我们一开始就用（Fortran）列为主的方式存储矩阵，这意味着我们必须进行复制。由于复制的复杂程度较低，这个成本是可以摊销的。



这种算法可以在专门的硬件中实现，如谷歌的 TPU[84]，具有很大的能源效率。

### 6.4.2 坎农的分布式内存矩阵-矩阵乘积算法

在第 6.4.1 节中，我们考虑了单处理器矩阵-矩阵乘法的高性能实现。现在我们将简要地考虑这一操作的分布式内存版本。(值得注意的是，这并不是基于第 6.2 节的矩阵-向量乘积算法的泛化)。

这种操作的一种算法被称为坎农算法。它是一个正方形的处理器网格，处理器 $ij$ 逐渐积累 $ij$ 块 $C_{i,j} = \sum_k A_{i,k} B_{k,j}$；见图 6.10。

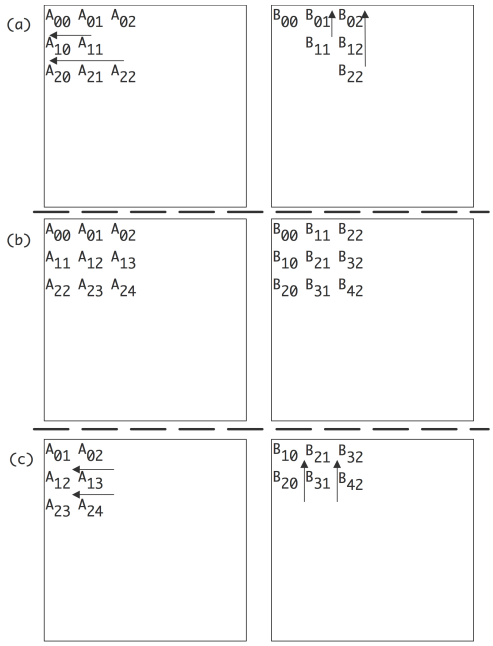
如果你从左上角开始，你会看到处理器（0，0）同时拥有 $A_{00}$ 和 $B_{00}$，所以它可以立即开始做一个局部乘法。处理器（0，1）有 $A_{01}$ 和 $B_{01}$，它们不需要在一起，但是如果我们把 $B$ 的第二列向上旋转一个位置，处理器（0，

1）将有 $A_{01}$，$B_{11}$，这两个确实需要被乘以。同样地，我们将 $B$ 的第三列向上旋转两个位置，这样（0，2）就有 $A_{02}$，$B_{22}$。

这个故事在第二行是如何进行的呢？处理器（1，0）有 $A_{10}$，$B_{10}$，它们不需要在一起。如果我们将第二行的 $A$ 向左旋转一个位置，它就包含了 $A_{11}, B_{10}$，这些是部分内积所需要的。而现在处理器（1,1）有 $A_{11}, B_{11}$。

如果我们继续这个故事，我们从一个矩阵 $A$ 开始，其中的行已经被向左旋转，而 B 的列已经被向上旋转。在这个设置中，处理器 $(i, j)$ 包含 $A_{i,i+j}$ 和 $B_{i+j,j}$，其中加法是按矩阵大小调制的。这意味着每个处理器都可以从做局部积开始。

现在我们观察到，$C_{ij} = \sum_k A_{ik} B_{kj}$ 意味着下一个局部积项来自于将 k 增加 1。$A$，$B$ 的相应元素可以通过旋转行和列的另一个位置移入处理器。




### 6.4.3 分布式矩阵-矩阵乘积的外积法

坎农的算法因需要一个正方形处理器网格的要求而受到影响。外积法更为普遍。它是基于矩阵-矩阵乘积计算的以下安排。

```
for ( k )
    for ( i )
      for ( j )
        c[i,j] += a[i,k] * b[k,j]
```

也就是说，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" 外积"）$A_{*,k} \cdot B_{k,*}$。然后我们对 $k$ 求和。

看一下这个算法的结构，我们注意到在步骤 $k$ 中，每一列 $j$ 收到 $A_{*,k}$，每一行 $i$ 收到 $B_{*,k}$。换句话说，$A_{*,k}$ 的元素通过他们的行广播，而 $B_{*,k}$ 的元素则通过他们的行广播。

使用 MPI 库，这些同步广播是通过为每一行和每一列设置一个子通信器来实现的。

## 6.5 稀疏的矩阵-向量乘积

在通过迭代方法求解线性系统时（见第 5.5 节），矩阵-向量乘积在计算上是一个重要的内核，因为它在每次可能的数百次迭代中都要执行。在本节中，我们先看一下矩阵-向量乘积在单处理器上的性能；多处理器的情况将在第 6.5 节中得到关注。

### 6.5.1 单处理器的稀疏矩阵-向量乘积

在迭代方法的背景下，我们并不太担心密集矩阵-向量乘积的问题，因为人们通常不会对密集矩阵进行迭代。在我们处理块状矩阵的情况下，请参考第 1.7.11 节对密集积的分析。稀疏乘积就比较麻烦了，因为大部分的分析并不适用。

稀疏矩阵-向量乘积中的数据重用按行执行的密集矩阵-向量乘积和 CRS 稀疏乘积（第 5.4.1.4 节）之间有一些相似之处。在这两种情况下，所有的矩阵元素都是按顺序使用的，所以任何加载的缓存线都被充分利用了。然而，CRS 的乘积至少在以下方面要差一些：

- 间接寻址需要加载一个整数向量的元素。这意味着在相同数量的操作中，稀疏产品有更多的内存流量。
- 源向量的元素不是按顺序加载的，实际上它们可以按随机的顺序加载。这意味着包含源元素的缓存线可能不会被完全利用。另外，内存子系统的预取逻辑（1.3.5 节）在这里也不能提供帮助。

由于这些原因，一个由稀疏矩阵-向量乘积主导计算的应用很可能以处理器峰值性能的 ≈5% 运行。

如果矩阵的结构在某种意义上是有规律的，那么就有可能提高这一性能。其中一种情况是，我们处理的是一个完全由小型密集块组成的块状矩阵。这至少会导致索引信息量的减少：如果矩阵由 2Œ2 块组成，我们会得到整数数据传输量的 4 倍减少。

✍ **练习 6.10** 再给出两个理由，说明这个策略有可能提高性能。提示：缓存线，和复用。

这样的矩阵细分可能会带来 2 倍的性能提升。假设有这样的改进，即使矩阵不是一个完美的块状矩阵，我们也可以采用这种策略：如果每个 2Œ2 的块都包含一个零元素，我们仍然可以得到 1.5 倍的性能改进 [197, 26]。

稀疏乘积中的矢量化在其他情况下，带宽和重用并不是最主要的问题。

- 在旧的矢量计算机上，如旧的 Craymachine，内存对处理器来说是足够快的，但矢量化是最重要的。这对稀疏矩阵来说是个问题，因为矩阵行中的零数，以及因此的矢量长度，通常都很低。
- 在 GPU 上，内存带宽相当高，但有必要找到大量相同的操作。矩阵可以被独立处理，但由于行可能是不等长的，这不是一个合适的并行性来源。

由于这些原因，稀疏矩阵的对角线存储方案的一个变种最近又出现了复兴。这里的观察是，如果你按行数对矩阵的行进行排序，你会得到少量的行块；每个行块都会相当大，而且每个行块中的元素数量都是一样的。

具有这种结构的矩阵对矢量架构来说是很好的 [38]。在这种情况下，乘积是通过对角线计算的。

✍ **练习 6.11** 写出这种情况的伪代码。行的排序是如何改善情况的？

这种排序的存储方案也解决了我们在 GPU 上注意到的问题 [20]。在这种情况下，我们传统的 CRS 产品算法，我们的并行量等于一个块中的行数。

当然，还有一个复杂的问题，就是我们已经对矩阵进行了置换：输入和输出的向量也需要进行相应的置换。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可能会否定任何性能上的提高。相反，我们可以对整个线性系统进行置换，并对置换后的系统进行迭代。

✍ **练习 6.12** 你能想出这样做的原因吗？为什么不可行呢？

### 6.5.2 并行稀疏矩阵-向量乘积

在第 5.4 节中，你看到了关于稀疏矩阵的第一次讨论，仅限于在单个处理器上使用。现在我们将讨论并行性问题。

密集矩阵-向量乘积，正如你在上面看到的，需要每个处理器与其他处理器通信，并有一个基本与全局向量大小相同的本地缓冲区。在稀疏的情况下，需要的缓冲区空间要少得多，通信也要少。让我们分析一下这种情

况。我们将假设矩阵是按块行分布的，其中处理器 $p$ 拥有索引在某个集合 $I_p$ 的矩阵行。

行 $y_i = y_i + a_{ij}x_j$ 现在必须考虑到，$a_{ij}$ 可以是零。特别是，我们需要考虑，对于某些对 $i \in I_p$，$j \notin I_p$ 不需要通信。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

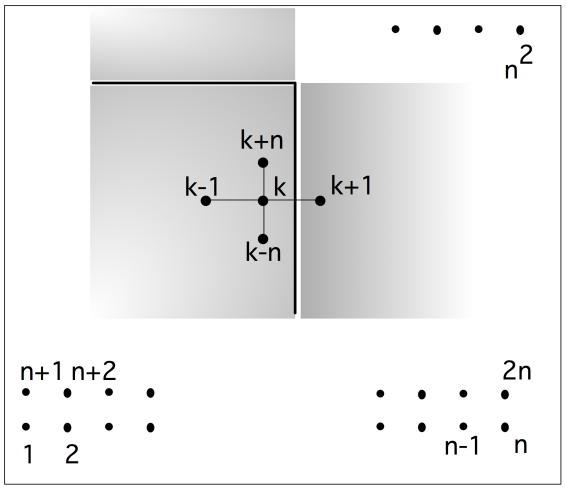
$$y_i += a_{ij}x_j \quad \text{if} j \in S_{p;i}$$

如果我们想避免如上所述的小消息的泛滥，我们把所有的通信合并成每个处理器的一个消息。定义

$$S_p = u_{i \in I_p} S_{p;i}$$

现在的算法是：

- 将所有必要的非处理器元素 $x_j \in S_p$ 收集到一个缓冲区。
- 执行矩阵-向量乘积，从本地存储中读取 $x$ 的所有元素。




这整个分析当然也适用于密集型矩阵。如果我们考虑到稀疏矩阵的来源。让我们从一个简单的案例开始。

回顾一下图 4.1，它说明了一个最简单的域（一个正方形）上的离散边界值问题，现在让我们来并行化它。我们通过分割域来做到这一点；每个处理器得到与其子域对应的矩阵行。图 6.11 显示了处理器之间的联系：元素 $a_{ij}$ 与 $i \in I_p, j \notin I_p$ 现在是模版的" 腿"，伸向处理器边界之外。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域；见图 6.12。

✍ **练习 6.13** 证明域的一维划分会导致矩阵被划分为块行，但域的二维划分则不会。你可以抽象地做到这一点，也可以举例说明：取一个 4Œ4 的域（给出一个大小为 16 的矩阵），并将其分割到 4 个处理器上。一维域的划分相当于给每个处理器一个行的域，而二维的划分则给每个处理器一个 2Œ2 的子域。画出这两种情况的矩阵。

✍ **练习 6.14** 图 6.13 描述了一个大小为 $N$ 的稀疏矩阵，半带宽度 $n = \sqrt{N}$。也就是说

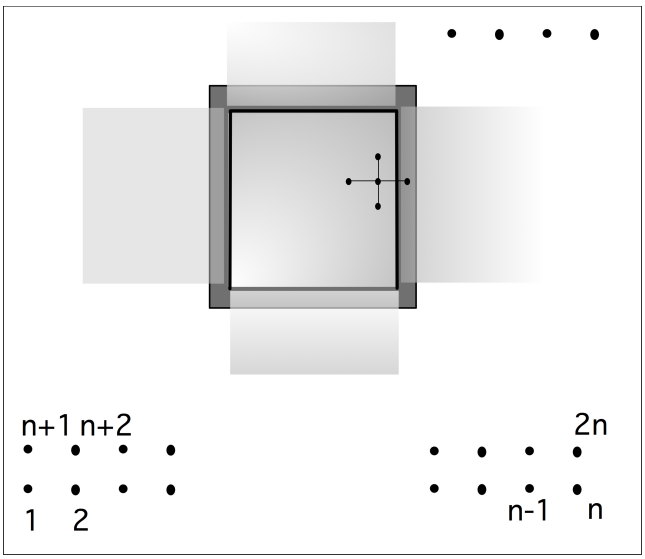
$$|i - j| > n \Rightarrow a_{ij} = 0$$

我们在 $p$ 处理器上对该矩阵进行一维分布，其中 $p = n = \sqrt{N}$ 。通过计算效率与处理器数量的关系，表明使用该方案的矩阵-向量乘积是弱可扩展的

$$E_p = \frac{T_1}{pT_p}$$

为什么这个方案不是真正的弱缩放，就像它通常定义的那样？

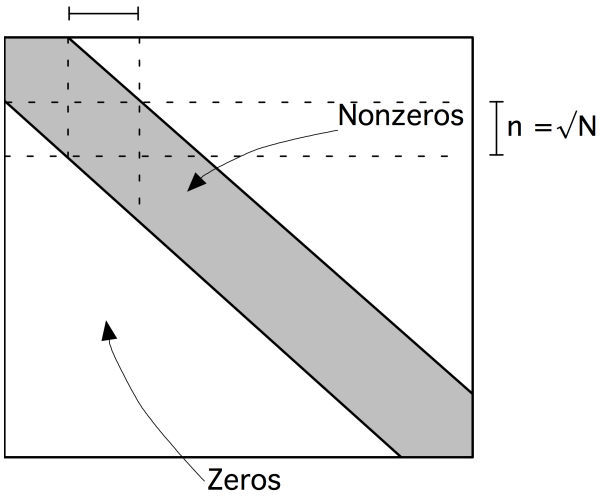
关于并行稀疏矩阵-向量乘积的一个关键观察是，对于每个处理器，它所涉及的其他处理器的数量是严格限制的。这对操作的效率有影响。




### 6.5.3 稀疏矩阵-向量乘积的并行效率

在密集矩阵-向量乘积的情况下（第 6.2.2 节），将矩阵按（块）行划分到处理器上并没有导致一个可扩展的算法。部分原因是每个处理器需要与之通信的邻居数量增加。图 6.11 显示，对于 5-5 点模版的矩阵，这个数量限制为 4 个。

练习 6.15 取一个正方形域和图 6.11 中的处理器的变量分区。对于图 4.3 中的盒子模版，一个处理器需要与之通信的最大邻居数是多少？在三个空间维度上，如果使用 7 点中心差分模版，那么邻居的数量是多少？




如果我们超越方形域而进入更复杂的物理对象，那么每个处理器只与几个邻居进行通信的观察就保持不变。如果一个处理器收到一个或多或少连续的子域，其邻居的数量将是有限的。这意味着即使在复杂的问题中，每个处理器也只能与少量的其他处理器通信。与密集情况相比，每个处理器不得不从其他每个处理器那里接收数据。很明显，稀疏的情况对互连网络要友好得多。(对于大型系统来说，这也是比较常见的事实，如果你准备购买一台新的并行计算机，可能会影响到对安装网络的选择)。

对于方形域，我们可以使这个论点正式化。让单位域 $[0,1]^2$ 在 $\sqrt{P}$ 网格中的 $P$ 处理器上进行划分，$\sqrt{P}\times\sqrt{P}$。从图 6.11 可以看出，每个处理器最多只能与四个邻居进行通信。让每个处理器的工作量为 $w$，与每个邻居的通

信时间为 $c$。那么在单个处理器上执行总工作的时间是 $T_1 = Pw$，而并行时间是 $T_p = w + 4c$，给出的速度为

$$S_P = Pw/(w + 4c) = P/(1 + 4c/w) \approx P(1 - 4c/w)$$

✍ **练习 6.16** 将 $c$ 和 $w$ 表示为 $N$ 和 $P$ 的函数，并说明在问题的弱标度下，速度的提高是渐近最优的。

✍ **练习 6.17** 在这个练习中，你将分析一个假设的、但现实的并行机器的并行稀疏矩阵-向量乘积。让机器参数具有以下特点 (见第 1.3.2 节)。

- 网络延迟：$\alpha = 1\mu s = 10^{-6}s$。
- 网络带宽：1Gb/s 对应于 $\beta = 10^{-9}$。
- 计算率。每核运算率为 1Gflops 意味着 $\gamma = 10^9$。这个数字看起来很低，但请注意，矩阵-向量乘积的重用率低于矩阵-矩阵乘积，后者可以达到接近峰值的性能，而且稀疏矩阵-向量乘积的带宽约束更大。我们进行了渐进式分析和推导具体数字的结合。

我们假设一个有 $10^4$ 个单核处理器的集群 4。我们将其应用于大小为 $N = 25 \cdot 10^{10}$ 的五点模版矩阵。这意味着每个处理器存储 $5 \cdot 8 \cdot N/p = 10^9$ 字节。如果矩阵来自一个正方形领域的问题，这意味着该领域的大小为 $n \times n$，$n = \sqrt{N} = 5 \cdot 10^5$。

情况 1：我们不是划分矩阵，而是划分域，我们首先通过大小为 $n \times n/p$ 的水平板块进行划分。论证通信复杂度为 $2(\alpha + n\beta)$，计算复杂度为 $10 \cdot n \cdot (n/p)$ 。证明所产生的计算量比通信量多出 50 倍。

情况 2：我们将领域划分为大小为 $(n/\sqrt{p}) \times (n/\sqrt{p})$ 的斑块。内存和计算时间与之前相同。推导出通信时间，并表明它比之前的时间要好 50 倍。

论证第一种情况不具有弱扩展性：在假设 $N/p$ 是恒定的情况下，效率会下降。(证明速度仍然随着 $\sqrt{p}$ 的增加而渐进地上升。论证第二种情况确实是弱扩展的。

一个处理器只与几个邻居连接的论点是基于科学计算的性质。这对 FDM 和 FEM 方法来说是真实的。在边界元素法（BEM）的情况下，任何子域都需要与它周围半径为 r 的所有东西进行通信。随着处理器数量的增加，每个处理器的邻居数量也会增加。

✍ **练习 6.18** 请对 BEM 算法的速度和效率进行正式分析。再假设每个处理器的单位工作量为 $w$，每个邻居的通信时间为 $c$。由于邻居的概念现在是基于物理距离，而不是基于图形属性，所以邻居的数量会增加。对于这种情况，给出 $T_1$、$T_p$、$S_p$、$E_p$。还有一些情况，稀疏矩阵需要与密集矩阵类似地处理。例如，谷歌的 PageRank 算法（见第 9.4 节）的核心是重复操作 $x \leftarrow A$，$A$ 是一个稀疏矩阵，如果网页 $j$ 链接到页面 $i$，$A \neq 0$；见第 9.4 节。这使得 $A$ 是一个非常稀疏的矩阵，没有明显的结构，所以每个处理器都很可能与其他处理器进行交流。

### 6.5.4 稀疏矩阵-向量乘积的记忆行为

在 1.7.11 节中，你看到了在单个处理器上对密集情况下的稀疏矩阵-向量乘积的分析。有些分析可以立即延续到稀疏情况下，比如每个矩阵元素只使用一次，性能受处理器和内存之间的带宽限制。

关于输入和输出向量的重用，如果矩阵是按行存储的，比如 CRS 格式（第 5.4.1.3 节），对输出向量的访问将被限制为每行写一次。另一方面，用于获得输入向量的重用的循环解卷技巧不能在这里应用。结合两个迭代的代码如下。

```
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;
}
```

这里的问题是，如果 $a_{ij}$ 为非零，就不能保证 $a_{i+1,j}$ 为非零。稀疏模式的不规则性使得优化矩阵-向量乘积变得困难。通过识别矩阵中的小密块部分，可以实现适度的改进 [26, 43, 196]。

在 GPU 上，稀疏的矩阵-向量乘积也受到内存带宽的限制。现在编程更难了，因为 GPU 必须在数据并行模式下工作，有许多活动线程。

如果我们考虑到稀疏矩阵-向量乘积通常出现的背景，一个有趣的优化就成为可能。这个操作最常见的用途是在线性系统的迭代求解方法中（第 5.5 节），在那里它被应用于同一个矩阵，可能有数百次迭代。因此，我们可以考虑将矩阵存储在 GPU 上，只为每个乘积操作复制输入和输出向量。

### 6.5.5 转置积

在第 5.4.1.3 节中，你看到正则和转置矩阵-向量乘积的代码都被限制在循环顺序上，其中矩阵的行被遍历。(在第 1.6.2 节中，你看到了关于循环顺序变化的计算效果的讨论；在这种情况下，我们被存储格式限制在行的遍历上)。

在这一节中，我们将简要地看一下并行转置积。与按行划分矩阵并进行转置乘积相当，我们看一下按列存储和划分的矩阵并进行常规乘积。

列间乘积的算法可以给出：。

$$y \leftarrow 0 \quad \text{for} j: \quad \text{for} i: \quad y_i \leftarrow y_i + a_{ij}x_j$$

在共享内存和分布式内存中，我们都将外迭代分布在处理器上。那么问题来了，每个外部迭代都会更新整个输出向量。这是一个问题：在共享内存中，它导致了对输出位置的多次写入，而在分布式内存中，它需要通信，目前还不清楚。

解决这个问题的一个方法是为每个进程分配一个私有的输出向量 $y^{(p)}$。

$$y^{(p)} \leftarrow 0 \qquad \text{for} j \in \text{the rows of processor} p \qquad \text{for all} i: \qquad y_i^{(p)} \leftarrow y_i^{(p)} + a_{ji}x_j$$

之后，我们对 $y \leftarrow \sum_p y^{(p)}$ 进行求和。

### 6.5.6 稀疏矩阵-向量乘积的设置

密集的矩阵-向量乘积依赖于集合体（见第 6.2 节），而稀疏的情况则使用点对点的通信，也就是说，每个处理器只向几个邻居发送信息，并从几个邻居那里接收信息。这对于来自 PDE 的稀疏矩阵类型是有意义的，正如你在第 4.2.3 节中看到的，PDE 有明确的结构。然而，有些稀疏矩阵是如此的随机，以至于你基本上不得不使用密集技术；见第 9.5 节。

然而，发送和接收之间存在着不对称性。对于一个处理器来说，要找出它将从哪些其他处理器那里接收信息是相当容易的。

✍ **练习 6.19** 假设矩阵是按块行划分给处理器的；见图 6.2 的说明。还假设每个处理器都知道其他处理器存储了哪些行。(你将如何实现这一知识?）勾勒出一个算法，通过这个算法，处理器可以发现它将从谁那里接收；这个算法本身不应该涉及任何通信。发现向谁发送则更难。

✍ **练习 6.20** 论证一下，在结构对称矩阵的情况下，这很容易。$a_{ij} \neq 0 \Leftrightarrow a_{ji} \neq 0$。在一般情况下，原则上可以要求一个处理器向任何其他处理器发送，所以简单的算法是如下所示。

每个处理器都会列出它所需要的非本地指数的清单。根据上述假设，它知道其他每个处理器拥有的指数范围，然后决定从哪些邻居那里获得哪些指数。

- 每个处理器向它的每个邻居发送一个指数列表；这个列表对大多数邻居来说都是空的，但我们不能省略发送它。
- 然后，每个处理器从所有其他处理器那里收到这些列表，并制定出要发送的指数列表。你会注意到，尽管矩阵-向量乘积过程中的通信只涉及每个处理器的几个邻居，给出的成本是处理器数量的 O(1)，但设置涉及所有对所有的通信，其时间复杂性为 $O(\alpha P)$
- 如果一个处理器只有几个邻居，上述算法是浪费的。理想情况下，你希望空间和运行时间与邻居的数量成正比。如果我们知道预计有多少消息，我们可以在设置中带来接收时间。这个数字是可以找到的。
- 每个进程都会产生一个长度为 $P$ 的数组，如果进程需要来自处理器 i 的任何数据，则 ereneed[i] 为 1，否则为 0。
- 在这个数组上的一个 reduce-scatter 集体，用一个 sum 运算符，然后在每个处理器上留下一个数字，表示有多少处理器需要它的数据。
- 该处理器可以执行那么多的接收调用。reduce-scatter 调用的时间复杂度为 $\alpha \log P + \beta P$，与之前的算法相同，但可能比例常数较低。通过一些复杂的技巧，设置所需的时间和空间可以减少到 $O(\log P)$[63,110]。

## 6.6 迭代方法的计算问题

所有的迭代方法都有以下操作。

- 矩阵-向量乘积；这在第 5.4 节讨论了顺序情况，在第 6.5 节讨论了平行情况。在并行情况下，FEM 矩阵的构造有一个复杂的问题，我们将在第 6.6.2 节讨论。
- 预处理矩阵 $K$ 的构造 $\approx A$，以及系统的解 $Kx = y$。这在第 5.5.6 节的顺序情况下讨论过。下面我们将在第 6.7 节中讨论并行性问题。
- 一些向量操作（包括内积，一般来说）。这些将在接下来讨论。

### 6.6.1 向量操作

在一个典型的迭代方法中，有两种类型的向量运算：向量加法和内积。

✍ **练习 6.21** 考虑第 5.5.11 节的 CG 方法，图 5.9，应用于二维 BVP 的矩阵；公式 (4.16)，首先考虑无条件的情况 $M = I$。证明在矩阵-向量乘积和向量运算中进行的浮点运算数量大致相等。用矩阵大小 $N$ 表示一切，忽略低阶项。如果矩阵每行有 20 个非零点，这种平衡会是怎样的?

接下来，研究一下 5.5.9 节中 FOM 方案的矢量和矩阵操作之间的这种平衡。由于向量操作的数量取决于迭代，考虑前 50 次迭代，计算在向量更新和内积与矩阵-向量乘积中进行了多少浮点操作。矩阵需要有多少个非零点才能使这些数量相等?

✍ **练习 6.22** 翻牌计数并不是全部的事实。对于在单个处理器上执行的迭代方法中的向量和矩阵操作的效率，你能说什么?

#### 6.6.1.1 向量加法

向量加法的典型形式是 $x \leftarrow \alpha y$ 或 $x \leftarrow \alpha + y$。如果我们假设所有的向量都以相同的方式分布，这个操作就是完全平行的。

#### 6.6.1.2 内积

内积是向量操作，但它们在计算上比更新更有趣，因为它们涉及通信。

当我们计算一个内积时，很可能每个处理器都需要接收计算的值。我们使用以下算法。

**for** *processor p* **do**
compute $a_p \leftarrow x_p^t y_p$ where $x_p, y_p$ are the part of $x, y$ stored on processor $p$
do a global reduction to compute $a = \sum_p a_p$
broadcast the result

**Algorithm 1:** Compute $a \leftarrow x^t y$ where $x, y$ are distributed vectors

规约和广播（可以加入到 Allreduce 中）结合了所有处理器的数据，所以它们的通信时间随着处理器的数量而增加。这使得内积有可能成为一个昂贵的操作，人们提出了一些方法来减少它们对迭代方法性能的影响。

✍ **练习 6.23** 迭代方法通常用于稀疏矩阵。在这种情况下，你可以认为内积中涉及的通信对整体性能的影响可能比矩阵-向量乘积中的通信更大。作为处理器数量的函数，矩阵-向量乘和内积的复杂性是什么？

下面是一些已经采取的方法：

- CG 方法每次迭代都有两个相互依赖的内部产品。有可能重写该方法，使其计算相同的迭代结果（至少在精确算术中），但使每次迭代的两个内积可以合并。见 [32, 39, 156, 205]。
- 也许可以将内积计算与其他并行计算重叠 [41]。
- 在 GMRES 方法中，使用经典的 Gram-Schmidt(GS) 方法需要的独立内积要比修改后的 GS 方法少得多，但它的稳定性较差。人们已经研究了决定何时允许使用经典 GS 方法的策略 [138]。

由于计算机算术不是关联性的，当同一计算在两个不同的处理器配置下执行时，内积是导致结果不同的主要原因。在第 3.5.5 节中，我们勾画了一个解决方案。

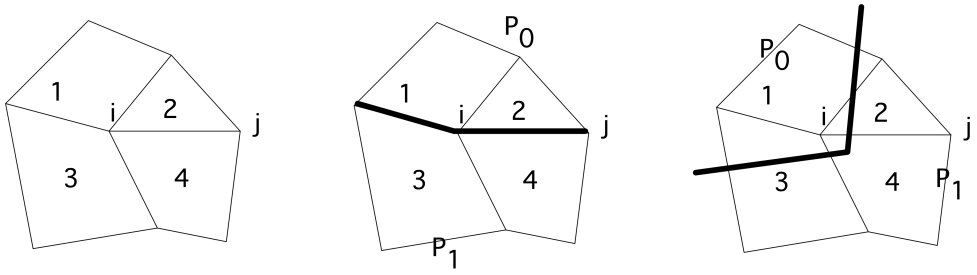
### 6.6.2 有限元矩阵构造

有限元导致了并行计算中一个有趣的问题。为此，我们需要勾勒出这种方法工作的基本轮廓。有限元的名称来自于这样一个事实：建模的物理对象被划分为小的二维或三维形状，即元素，如二维的三角形和正方形，或三维的金字塔和砖块。在每一个元素上，我们要建模的函数被假定为多项式，通常是低度的，如线性或双线性。

关键的事实是，一个矩阵元素 $a_{ij}$ 是所有包含变量 i 和 j 的元素的计算之和，特别是某些积分。

$$a_{ij} = \sum_{e:i,j\in e} a_{ij}^{(e)}$$

每个元素的计算都有许多共同的部分，所以很自然地把每个元素 e 唯一地分配给一个处理器 $P$，然后由它来计算所有的贡献 a(e)。在图 6.14 中，元素 2 被分配给处理器 0，元素 4 被分配给处理器 1。




现在考虑变量 $i$ 和 $j$ 以及矩阵元素 $a_{ij}$。它被构建为域元素 2 和 4 的计算之和，这些元素被分配给不同的处理器。因此，无论什么处理器的行 $i$ 被分配到，至少有一个处理器必须传达它对矩阵元素 $a_{ij}$ 的贡献。

显然，不可能对元素 $P_e$ 和变量 $P_i$ 进行分配，从而使 $P_e$ 对所有 $i \in e$ 都能完全计算系数 $a_{ij}$。换句话说，如果我们在本地计算贡献，需要有一定量的通信来集合某些矩阵元素。由于这个原因，现代线性代数库如 PETSc 允许任何处理器设置任何矩阵元素。

### 6.6.3 迭代方法性能的一个简单模型

上面我们已经说过，迭代方法很少有机会进行数据重用，因此它们被描述为带宽约束型算法。这使得我们可以对迭代方法的浮点性能做出简单的预测。由于迭代方法的迭代次数很难预测，这里的性能是指单次迭代的性能。与线性系统的直接方法不同（例如见第 6.8.1 节），解的次数很难事先确定。

首先，我们认为我们可以把自己限制在稀疏矩阵向量乘积的性能上：这个操作所花费的时间比向量操作多得多。此外，大多数预调节器的计算结构与矩阵-向量乘积相当相似。

然后考虑 CRS 矩阵-向量乘积的性能

- 首先我们观察到，矩阵元素和列索引的数组没有重复使用，所以加载它们的性能完全由可用带宽决定。缓存和完美流只是隐藏了延迟，但并没有改善带宽。对于矩阵元素与输入矢量元素的每一次乘法，我们加载一个浮点数字和一个整数。根据指数是 32 位还是 64 位，这意味着每次乘法都要加载 12 或 16 个字节。
- 对结果向量的存储要求并不重要：每个矩阵行的输出向量只写一次。
- 输入向量在带宽计算中可以忽略不计。乍一看，你会认为对输入向量的间接索引或多或少是随机的，因此很昂贵。然而，让我们从矩阵-向量乘积的算子角度出发，考虑矩阵所来自的 PDE 的空间域；见 4.2.3 节。我们现在看到，看似随机的索引实际上是被紧密地组合在一起的向量元素。这意味着这些向量元素很可能存在于 L3 缓存中，因此可以用比主存数据更高的带宽（例如，至少 5 倍的带宽）来访问。

对于稀疏矩阵-向量乘积的并行性能，我们认为在 PDE 背景下，每个处理器只与几个邻居进行通信。此外，从表面到体积的论证表明，信息量比节点上的计算量要低一阶。

总之，我们得出结论，稀疏矩阵向量乘积的一个非常简单的模型，从而也是整个迭代求解器的模型，包括测量有效带宽和计算每 12 或 16 字节加载一次加法和一次乘法操作的性能。

## 6.7 并行预处理程序

上面（第 5.5.6 节，特别是 5.5.6.1 节）我们看到了 $K$ 的几种不同选择。在本节中，我们将开始讨论并行化策略。讨论将在接下来的章节中继续详细进行。

### 6.7.1 雅克比预处理

雅可比方法（5.5.3 节）使用 A 的对角线作为预处理。应用这个方法是尽可能的平行：声明 $y \leftarrow K^{-1}x$ 独立地缩放输入向量的每个元素。不幸的是，使用雅可比预处理器对迭代次数的改进是相当有限的。因此，我们需要考虑更复杂的方法，如 ILU。与雅可比预处理程序不同的是，并行化并不简单。

### 6.7.2 与 ILU 并行的麻烦

上面我们看到，从计算的角度来看，应用 ILU 预处理（第 5.5.6.1 节）与做矩阵-向量乘积一样昂贵。如果我们在并行计算机上运行我们的迭代方法，这就不再是真的了。

乍一看，这些操作很相似。矩阵-向量乘积 $y = Ax$ 看起来像

```
for i=1..n
    y[i] = sum over j=1..n a[i,j]*x[j]
```

并行时，这看起来像

```
for i=myfirstrow..mylastrow
    y[i] = sum over j=1..n a[i,j]*x[j]
```

假设一个处理器拥有它所需要的 A 和 x 的所有元素的本地副本，那么这个操作就是完全并行的：每个处理器都可以立即开始工作，如果工作负荷大致相等，它们都会在同一时间完成。然后，矩阵-向量乘积的总时间除以处理器的数量，使速度或多或少得到了提高。

现在考虑正向解 $Lx = y$，例如在 ILU 预处理程序的背景下。

```
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]
```

但现在出现了一个问题。我们不能再说" 假设一个处理器拥有右手边所有东西的本地拷贝"，因为向量 x 同时出现在左手边和右手边。虽然矩阵-向量乘积原则上在矩阵行上是完全并行的，但这个三角解代码是递归的，因此是顺序的。

在并行计算的背景下，这意味着，第二个处理器要启动，需要等待第一个处理器计算的 x 的某些组件。显然，在第一个处理器完成之前，第二个处理器不能启动，第三个处理器必须等待第二个处理器，以此类推。令人失望的结论是，在并行中，任何时候都只有一个处理器处于活动状态，总时间与顺序算法相同。在密集矩阵的情况下，这实际上不是一个大问题，因为在处理单行的操作中可以找到并行性（见第 6.12 节），但是在稀疏的情况下，这意味着我们不能使用不完整的因式分解，而需要重新设计。

在接下来的几个小节中，我们将看到不同的策略来寻找能有效地并行执行的前置条件器。

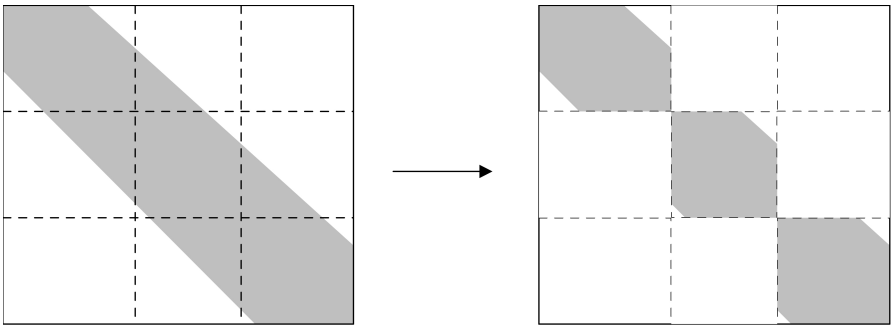
### 6.7.3 块状雅可比方法

人们提出了各种方法来弥补三角形解法的顺序性。例如，我们可以简单地让处理器忽略那些应该来自其他处理器的 x 的成分。

```
for i=myfirstrow..mylastrow
    x[i] = (y[i] - sum over j=myfirstrow..i-1 ell[i,j]*x[j])
           / a[i,i]
```

这在数学上并不等同于顺序算法（从技术上讲，它被称为以 ILU 为局部解的块状雅可比方法），但由于我们只是在寻找一个近似值 $K \approx A$，这只是一个稍微粗糙的近似。

✍ **练习 6.24** 以你上面写的高斯-赛德尔代码为例，模拟一次并行运行。增加（模拟的）处理器数量的效果如何？



块方法背后的想法可以通过图片来理解，见图 6.15。实际上，我们通过忽略处理器之间的所有连接，得到了一个 ILU 的矩阵。由于在 BVP 中，所有的点都是相互影响的（见第 4.2.1 节），如果在顺序计算机上执行，使用连接较少的预调节器将增加迭代的次数。然而，块状方法是并行的，正如我们上面所观察到的，顺序预处理器在并行情况下效率很低，所以我们容忍了这种迭代次数的增加。

### 6.7.4 并行 ILU

块状雅可比预处理通过解耦领域部分进行操作。虽然这可能会产生一种高度并行的方法，但它可能会比真正的 ILU 预处理程序产生更多的迭代次数。(可以从理论上论证这种解耦降低了迭代方法的效率；见第 4.2.1 节)。幸运的是，有可能出现一个并行的 ILU 方法。由于我们需要即将到来的关于变量重新排序的材料，我们把这个讨论推迟到 6.8.2.3 节。

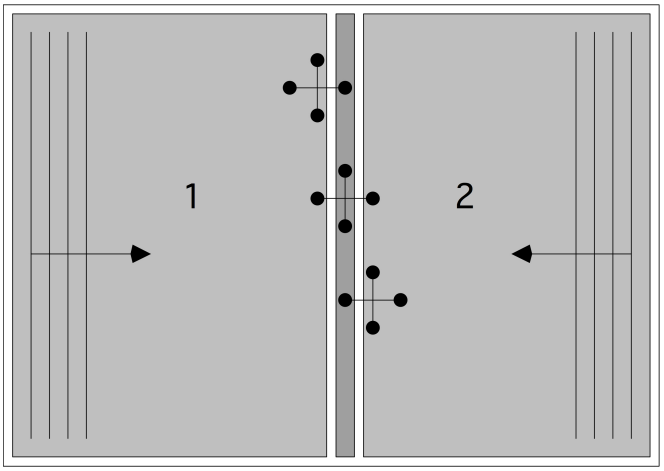
## 6.8 排序策略和并行性

在前文中，我们已经提到了一个事实，即解决线性方程组本身就是一个递归活动。对于密集系统来说，与递归长度相比，操作的数量足够多，因此寻找并行性是相当直接的。另一方面，稀疏系统则需要更多的复杂性。在这一节中，我们将研究一些重新排列方程的策略（或者说，等同于排列矩阵），这将增加可用的并行性。

这些策略都可以被认为是高斯消除法的变种。通过对它们进行不完全变体（见第 5.5.6.1 节），所有这些策略也适用于构建迭代求解方法的前置条件。

### 6.8.1 嵌套剖析

上面，你已经看到了几个在域中对变量进行排序的例子，而不是用列举式排序。在本节中，你将看到嵌套剖析排序，它最初被设计为一种减少填充的方法。然而，它在并行计算的背景下也是有利的。




3

嵌套剖析是一个递归的过程，用于确定在一个工作中的未知数的非线性排序。在第一步中，计算域被分割成两部分，在它们之间有一个分隔带；见图 6.16。准确地说，这个分隔带足够宽，以至于左右子域之间没有任何联系。由此产生的矩阵 ADD 具有 3Œ3 的结构，对应于域的三个分界。由于子域 Ω 和 Ω 不相连，子矩阵 ADD

和 ADD 为零。

$$A^{\mathrm{DD}} = \begin{pmatrix} A_{11} & \varnothing & A_{13} \\ \varnothing & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \left(\begin{array}{ccccc|ccccc|c} \star & \star & & & & & & & & & 0 \\ \star & \star & \star & & & & & & & & \vdots \\ & \ddots & \ddots & \ddots & & & & \varnothing & & & \vdots \\ & & \star & \star & \star & & & & & & 0 \\ & & & \star & \star & & & & & & \star \\ \hline & & & & & \star & \star & & & & 0 \\ & & & & & \star & \star & \star & & & \vdots \\ & & \varnothing & & & & \ddots & \ddots & \ddots & & \vdots \\ & & & & & & & \star & \star & \star & 0 \\ & & & & & & & & \star & \star & \star \\ \hline 0 & \cdots & \cdots & 0 & \star & 0 & \cdots & \cdots & 0 & \star & \star \end{array}\right) \left(n^2 - n\right)/2$$

这种用分离器划分域的过程也被称为域分解或子结构，尽管这个名字也与所产生的矩阵的数学分析有关 [14]。在这个矩形域的例子中，找到一个分隔符当然是微不足道的。然而，对于我们从 BVP 中得到的方程类型，通常可以有效地找到任何域的分离器 [146]；另见 18.6.2 节。

现在让我们考虑这个矩阵的 LU 因式分解。如果我们用 3Œ3 块状结构来分解它，我们可以得到

$$A^{\mathrm{DD}} = LU = \begin{pmatrix} I & & \\ \varnothing & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & \varnothing & A_{13} \\ & A_{22} & A_{23} \\ & & S_{33} \end{pmatrix}$$

其中

$$S_{33} = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

这里的重要事实是

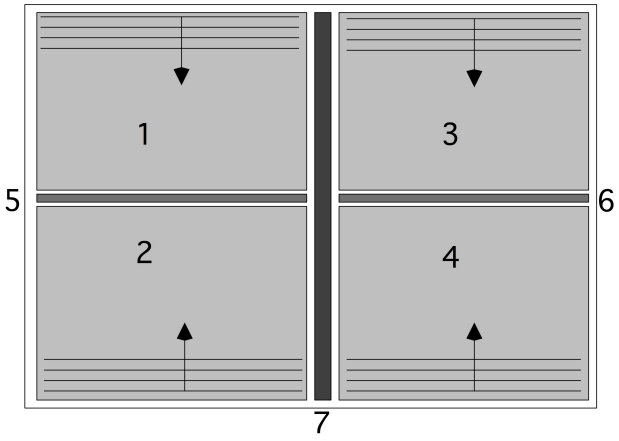
- 贡献 $A_{31}A_{11}^{-1}A_{13}$ 和 $A_{32}A_{22}^{-1}A_{23}$ 可以同时计算，所以因式分解在很大程度上是并行的；并且
- 在前向和后向求解中，解的第 1 和第 2 部分都可以同时计算，所以求解过程也基本上是并行的。第三块不能以并行方式处理，所以这在算法中引入了一个顺序的部分。算法。我们还需要仔细研究一下 $S_{33}$ 的结构。

**练习 6.25** 在第 5.4.3.1 节中，你看到了 LU 因子化和图论之间的联系：消除一个节点会导致图中的该节点被删除，但会增加某些新的连接。证明在消除前两组变量后，分离器上剩余矩阵的图将是完全连接的。结果是，在消除了第 1 和第 2 块中的所有变量后，我们剩下的矩阵 $S_{33}$ 是完全密集的，大小为 $n \times n$。

引入分离器后，我们得到了一个双向平行的因式分解。现在我们重复这个过程：我们在第 1 和第 2 块内放置一个分隔符（见图 6.17），这样就得到了以下矩阵结构。

$$A^{\mathrm{DD}} = \left(\begin{array}{cccc|cc|c} A_{11} & & & & A_{15} & & A_{17} \\ & A_{22} & & & A_{25} & & A_{27} \\ & & A_{33} & & & A_{36} & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ \hline A_{51} & A_{52} & & & A_{55} & & A_{57} \\ & & A_{63} & A_{64} & & A_{66} & A_{67} \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right)$$

(注意与第 5.4.3.4 节中的" 箭头" 矩阵的相似性，并回顾一下这导致较低填充的论点)。这个的 LU 因子化是。




$$\left(\begin{array}{cccc|cc|c}
I & & & & & & \\
 & I & & & & & \\
 & & I & & & & \\
 & & & I & & & \\
\hline
A_{51}A_{11}^{-1} & A_{52}A_{22}^{-1} & & & I & & \\
 & & A_{63}A_{33}^{-1} & A_{64}A_{44}^{-1} & & I & \\
\hline
A_{71}A_{11}^{-1} & A_{72}A_{22}^{-1} & A_{73}A_{33}^{-1} & A_{74}A_{44}^{-1} & A_{75}S_5^{-1} & A_{76}S_6^{-1} & I
\end{array}\right).$$

$$\left(\begin{array}{cccc|cc|c}
A_{11} & & & & A_{15} & & A_{17} \\
 & A_{22} & & & A_{25} & & A_{27} \\
 & & A_{33} & & & A_{36} & A_{37} \\
 & & & A_{44} & & A_{46} & A_{47} \\
\hline
 & & & & S_5 & & A_{57} \\
 & & & & & S_6 & A_{67} \\
\hline
 & & & & & & S_7
\end{array}\right)$$

其中

$$S_5 = A_{55} - A_{51}A_{11}^{-1}A_{15} - A_{52}A_{22}^{-1}A_{25}, \quad S_6 = A_{66} - A_{63}A_{33}^{-1}A_{36} - A_{64}A_{44}^{-1}A_{46}$$
$$S_7 = A_{77} - \sum_{i=1,2,3,4} A_{7i}A_{ii}^{-1}A_{i7} - \sum_{i=5,6} A_{7i}S_i^{-1}A_{17}$$

现在构建因式分解的过程如下。

- 对于 $i = 1,2,3,4$，块 $A_{ii}$ 被并行因式化；同样，对于 $i = 1,2,3,4$，$A_{5i}A_{ii}^{-1}A_{i5}$，对于 $i = 3,4$，$A_{6i}A_{ii}^{-1}A_{i6}$，$i = 1,2,3,4$，$A_{7i}A_{ii}^{-1}A_{i7}$ 可以被并行构建。
- 形成舒尔补码 $S_5, S_6$，并随后进行并行派生，对 $i = 5,6$ 的贡献 $A_{7i}S_i^{-1}A_{17}$ 也是并行构建。
- 舒尔补码 $S_7$ 被形成并被分解。

与上述推理类似，我们得出结论：在消除了第 1,2,3,4 块后，更新的矩阵 $S_5, S_6$ 是大小为 $n/2$ 的密集，在消除了第 5,6 块后，舒尔补码 $S_7$ 是大小为 $n$ 的密集。

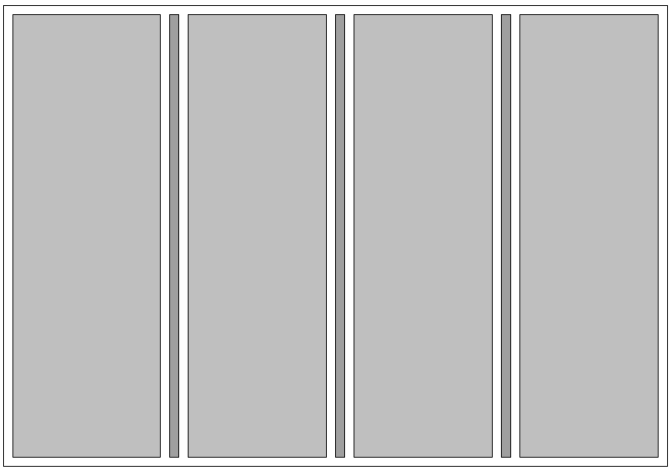
✍ **练习 6.26** 证明用 ADD 求解一个系统与构建上述因式分解有类似的并行性。

为了便于以后参考，我们将把集合 1 和 2 称为彼此的兄弟姐妹，同样，3 和 4 也称为兄弟姐妹。集合 5 是 1 和 2 的父，6 是 3 和 4 的父；5 和 6 是兄弟姐妹，7 是 5 和 6 的父。

#### 6.8.1.1 域的分解

在图 6.17 中，我们通过一个递归过程将领域分成四种方式。这就引出了我们对嵌套剖分的讨论。也可以立即将一个域分割成任意数量的条状，或者分割成子域的网格。只要分离器足够宽，这将给出一个具有许多独立子域的矩阵结构。在上面的讨论中，LU 因子化的特点是

- 在因式分解和 L, U 求解中，对子域进行并行处理，并且
- 要在分离器结构上求解的系统。



**练习 6.27** 二维 BVP 的矩阵有一个块状三对角结构。将该域分为四条，即使用三个分离器（见图 6.18）。注意这些分离器在原始矩阵中是不耦合的。现在勾勒出所得到的系统的稀疏性结构，即分离器是消除子域的。说明该系统是块状三对角的。

在我们到目前为止讨论的所有领域分割方案中，我们使用的领域都是矩形的，或者说是" 砖" 形的，在两个维度以上。所有这些论点都适用于二维或三维的更一般的域，但是像寻找分离器这样的事情变得更加困难 [145]，而这对于平行情况来说更是如此。参见第 18.6.2 节对这个话题的一些介绍。

#### 6.8.1.2 复杂度

嵌套剖析法重复上述过程，直到子域变得非常小。对于理论分析来说，我们一直在分割，直到我们有大小为 1Œ1 的子域，但在实践中，我们可以停止在 32 这样的大小，并使用一个有效的密集求解器来分解和反转块。

为了推导出算法的复杂度，我们再看一下图 6.17，可以看到复杂度论证，一个完整的递归嵌套分解所需要的总空间是以下的总和

- 在一个大小为 $n$ 的分离器上的一个密集矩阵，加上
- 大小为 $n/2$ 的分离器上的两个密集矩阵之和。
- 需要 $3/2n^2$ 的空间和 $5/12n^3$ 的时间。
- 上述两项在四个大小为 $(n/2) \times (n/2)$ 的子域上重复。

观察到 $n = \sqrt{N}$，这就意味着

$$\begin{aligned} \text{space} &= 3/2n^2 + 4 \cdot 3/2(n/2)^2 + \cdots \\ &= N(3/2 + 3/2 + \cdots) \quad \log n\text{terms} \\ &= O(N \log N) \end{aligned}$$

$$\begin{aligned} \text{space} &= 3/2n^2 + 4 \cdot 3/2(n/2)^2 + \cdots \\ &= N(3/2 + 3/2 + \cdots) \quad \log n\text{terms} \\ &= O(N \log N) \end{aligned}$$

显然，我们现在的因式分解在很大程度上是平行的，而且是在 $O(N \log N)$ 空间中完成的，而不是在 $O(N^{3/2})$(见 5.4.3.3 节)。因式分解时间也从 $O(N^2)$ 下降到 $O(N^{3/2})$。

不幸的是，这种空间节省只发生在二维空间：在三维空间中，我们需要

- 一个 $n$ 大小的分离器，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。
- 两个大小为 $n \times n/2$ 的分离器，占用 $N^{3/2}/2$ 空间和 $1/3 \cdot N^2/4$ 时间。
- 四个大小为 $n/2 \times n/2$ 的分离器，需要 $N^{3/2}/4$ 的空间和 $1/3 \cdot N^2/16$ 的时间。
- 加起来就是 $7/4N^{3/2}$ 空间和 $21/16N^2/3$ 时间。
- 在下一级，有 8 个子域贡献这些条款，$n \to n/2$，因此 $N \to N/8$。

这使得总空间

$$\frac{7}{4}N^{3/2}(1 + (1/8)^{4/3} + ...) = O(N^{3/2})$$

以及总的时间

$$\frac{21}{16}N^2(1 + 1/16 + ...)/3 = O(N^2)$$

我们不再有二维情况下的巨大节省。一个更复杂的分析表明，对于二维的一般问题，阶次的改进是成立的，而三维一般有更高的复杂度 [145]。

#### 6.8.1.3 并行性

嵌套剖析法显然引入了大量的并行性，我们可以将其描述为任务并行性（第 2.5.3 节）：与每个分离器相关的是对其矩阵进行因式分解的任务，随后是对其变量进行线性系统求解的任务。然而，这些任务并不是独立的：在图 6.17 中，域 7 上的因式分解必须等待 5 和 6，而且它们必须等待 1,2,3,4。因此，我们的任务以树的形式存在依赖关系：每个分离器矩阵只有在其子矩阵被因子化后才能被因子化。

将这些任务映射到处理器上并非易事。首先，如果我们处理的是共享内存，我们可以使用一个简单的任务队列。

```
Queue <- {}
for all bottom level subdomains d do
  add d to the Queue
while Queue is not empty do
  if a processor is idle then
     assign a queued task to it
  if a task is finished AND its sibling is finished then
      add its parent to the queue
```

这里的主要问题是，在某些时候，我们的处理器会多于任务，从而导致负载不平衡。这个问题由于最后的任务也是最重要的，而变得更加严重，因为分离器的大小从一层到另一层是双倍的。(回顾一下，密集矩阵的因式分解工作随着大小的三次方而增加！) 因此，对于较大的分离器，我们必须从任务并行转为中粒度并行，即处理器合作对一个块进行因子化。

有了分布式内存，我们现在可以用一个简单的任务队列来解决并行问题，因为这将涉及移动大量的数据。(但请记住，工作是矩阵大小的高次方，这一次对我们有利，使通信相对便宜)。那么解决方案就是使用某种形式的域分解。在图 6.17 中，我们可以有四个处理器，与块 1、2、3、4 相关联。然后，处理器 1 和 2 将协商哪一个因素是第 5 块（类似地，处理器 3 和 4 和第 6 块），或者它们都可以冗余地做这个。

#### 6.8.1.4 预处理

与所有的因式分解一样，通过使因式分解不完整，可以将嵌套剖分法变成一个预处理程序。(关于不完全因式分解的基本思想，见 5.5.6.1 节)。然而，这里的因式分解完全是用块矩阵来表述的，除以枢轴元素就变成了一个反转或与枢轴块矩阵的系统解。我们将不进一步讨论这个问题，详情见文献 [6, 57, 157]。



### 6.8.2 变量的重新排序和着色：独立集

在稀疏矩阵中可以通过使用图形着色（第 18.3 节）实现并行化。由于" 颜色" 被定义为只与其他颜色相连的点，根据定义它们是相互独立的，因此可以被并行处理。这导致我们采取了以下策略

1. 将问题的邻接图分解成少量独立的集合，称为" 颜色"。
2. 用与颜色数量相等的顺序步骤来解决这个问题；在每个步骤中，都会有大量的可独立处理的点。

#### 6.8.2.1 红黑相间的颜色

我们从一个简单的例子开始，我们考虑一个三对角矩阵 $A$。方程 $Ax = b$ 看起来像

$$\begin{pmatrix} a_{11} & a_{12} & & & \varnothing \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & a_{34} & \\ \varnothing & & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix}$$

我们观察到，$x_i$ 直接取决于 $x_{i-1}$ 和 $x_{i+1}$，但不取决于 $x_{i-2}$ 或 $x_{i+1}$。因此，让我们看看，如果我们把指数排列起来，把每一个其他的组成部分放在一起会发生什么。

图形上，我们把 $1, ...n$，并将它们涂成红色和黑色（图 6.19），然后我们将它们进行置换，先取所有红色的点，然后取所有黑色的点。相应地，经过置换的矩阵看起来如下。

$$\begin{pmatrix} a_{11} & & & a_{12} & & \\ & a_{33} & & a_{32} & a_{34} & \\ & & a_{55} & & \ddots & \ddots \\ a_{21} & a_{23} & & a_{22} & & \\ & a_{43} & a_{45} & & a_{44} & \\ & & \ddots & \ddots & & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_5 \\ \vdots \\ y_2 \\ y_4 \\ \vdots \end{pmatrix}$$

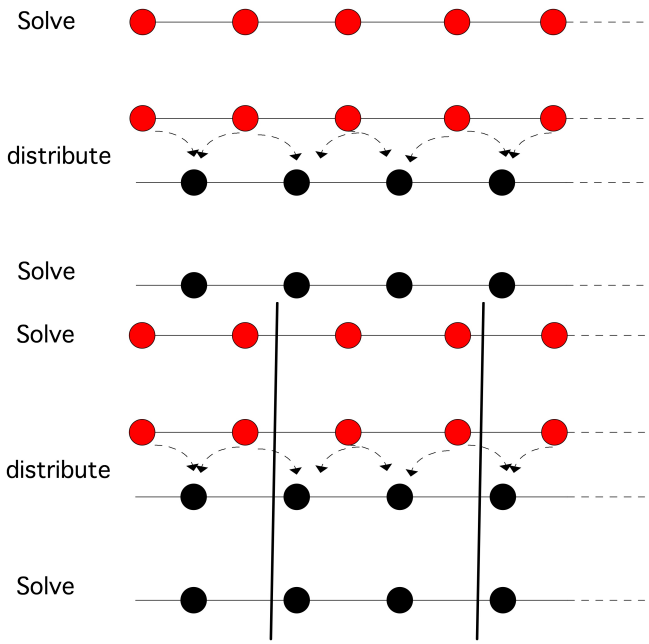
有了这个经过处理的 $A$，高斯-塞德尔矩阵 $D_A + L_A$ 看起来像是

$$\begin{pmatrix} a_{11} & & & & \varnothing & \\ & a_{33} & & & & \\ & & a_{55} & & & \\ & & & \ddots & & \\ a_{21} & a_{23} & & & a_{22} & \\ & a_{43} & a_{45} & & & a_{44} \\ & & \ddots & \ddots & & & \ddots \end{pmatrix}$$

这能给我们带来什么？好吧，让我们拼出一个系统的解决方案 $Lx = y$。

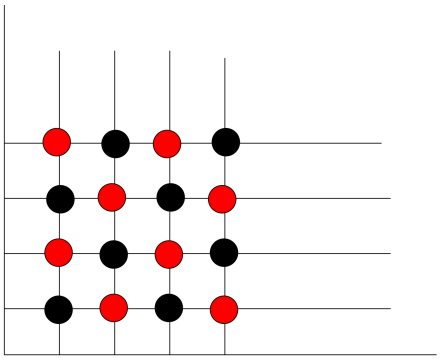
for$i = 1, 3, 5, \ldots, do$ solve$x_i \leftarrow y_i / a_{ii}$ for$i = 2, 4, 6, \ldots do$ compute$t = a_{ii-1}x_{i-1} + a_{ii+1}x_{i+1}$

显然，该算法有三个阶段，每个阶段在一半的域点上是平行的。这在图 6.20 中得到了说明。理论上，我们可以容纳的处理器数量是域点数量的一半，但实际上每个处理器都会有一个子域。现在你可以在图 6.21 中看到这如何导致一个非常适度的通信量：每个处理器最多发送两个红点的数据给它的邻居。




✍ **练习 6.28** 论证这里的邻接图是一个二边形图。我们看到，这样的图（以及一般的彩色图）与并行性有关。你还能指出非并行处理器的性能优势吗?

红黑排序也可以应用于二维问题。让我们对点 $i, j$ 应用红黑排序，其中 $1 \leqslant i, j \leqslant n$。在这里，我们首先对第一行的奇数点（1，1），（3，1），（5，1），.....，然后对第二行的偶数点（2，2），（4，2），（6，2），.....，第三行的奇数点进行连续编号，依此类推。这样对域中一半的点进行编号后，我们继续对第一行的偶数点、第二行的奇数点进行编号，依此类推。正如你在图 6.22 中看到的，现在红色的点只与黑色的点相连，反之亦然。用图论的术语来说，你已经找到了一个有两种颜色的矩阵图的着色（这个概念的定义见附录 18）。



✍ **练习 6.29** 对二维 BVP（4.12）应用红黑排序。画出所产生的矩阵结构。

红黑排序是图形着色（有时称为多重着色）的一个简单例子。在简单的情况下，如我们在第 4.2.3 节中考虑的单位正方形域或其扩展到三维，邻接图的色数很容易确定；在不太规则的情况下，则比较困难。

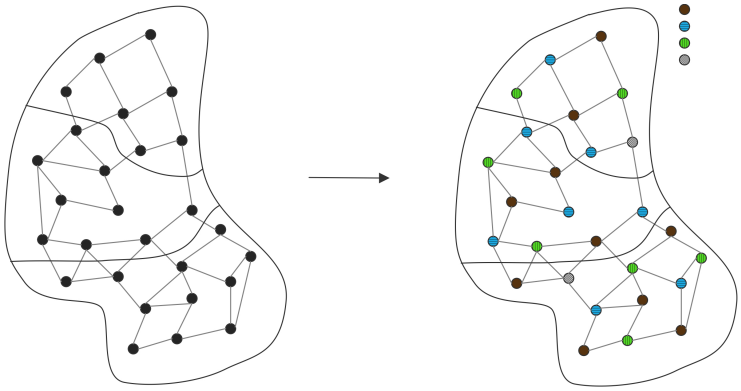
✍ **练习 6.30** 你看到了未知数的红黑排序加上有规律的五点星形模版给出了两个变量子集，它们之间没有联系，也就是说，它们形成了矩阵图的双着色。如果节点由图 4.3 中的第二个模版连接，你能找到一个着色吗?

### 6.8.2.2 一般性着色

对于稀疏矩阵的图形所需的颜色数量有一个简单的约束：颜色的数量最多为 $d+1$，其中 $d$ 是图形的度数。为了说明我们可以用 $d+1$ 种颜色来给学位为 $d$ 的图形着色，考虑一个学位为 $d$ 的节点。无论它的邻居是如何着

色的，在 $d+1$ 种可用的颜色中总有一种未使用的颜色。

✍ **练习 6.31** 考虑一个稀疏矩阵，该图可以用 $d$ 种颜色来着色。首先列举第一种颜色的未知数，然后列举第二种颜色的未知数，依此类推，对矩阵进行排列。你能说说所产生的排列矩阵的稀疏性模式是什么?



如果你在寻找线性系统的直接解，你可以在消除一种颜色后剩下的矩阵上重复着色和置换的过程。在三对角矩阵的情况下，你看到这个剩余的矩阵又是三对角的，所以很清楚如何继续这个过程。这就是所谓的递归翻倍法。如果矩阵不是三对角的，而是块状三对角的，这个操作可以在块上进行。

### 6.8.2.3 多色并行 ILU

在第 6.8.2 节中，你看到了图形着色和包络的结合。让 $P$ 是将相似颜色的变量组合在一起的置换，那么 $\tilde{A} = P^t AP$ 是一个具有如下结构的矩阵。

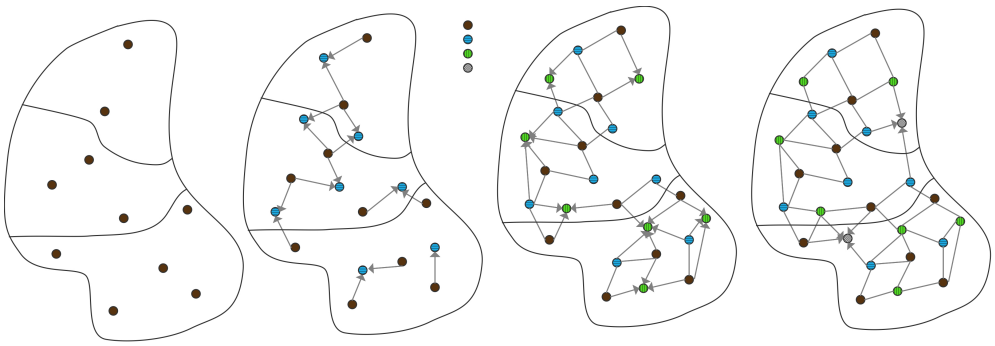
- $\tilde{A}$ 具有块状结构，其块数与 $A$ 的头顶相接图中的颜色数相等；并且
- 每个对角线块是一个对角线矩阵。

现在，如果你正在进行迭代系统求解，并且你正在寻找一个并行的预处理程序，你可以使用这个对角的矩阵。考虑到用包络系统来解决 $Ly = x$。我们将通常的算法（第 5.3.5 节）写为

$$\text{for} c \text{ in the set of colors:} \qquad \text{for} i \text{ in the variables of color} c : y_i \leftarrow x_i - \sum_{j<i} \ell_{ij} y_j$$

✍ **练习 6.32** 证明当从自然排序的 ILU 因式分解到颜色稀释排序的 ILU 因式分解时，解决系统 $LUx = y$ 的翻转数保持不变（最高阶项）。

这些着色对我们有什么好处？求解仍然是顺序的..... 嗯，的确，颜色的外循环是顺序的，但是一种颜色的所有点都是相互独立的，所以它们可以在同一时间被求解。因此，如果我们使用普通的域划分，并结合多色彩（见图 6.23），处理器在所有的色彩阶段都是活跃的；见图 6.24。好吧，如果你仔细看一下这个图，你会发现在最后一个颜色中，有一个处理器没有活动。在每个处理器有大量节点的情况下，这不太可能发生，但可能有一些负载不平衡。



剩下的一个问题是如何并行地生成多色。寻找最佳颜色数是 NP-hard。拯救我们的是，我们不一定需要最佳数量，因为我们反正是在做不完全因式分解。(甚至有一种观点认为，使用稍大的颜色数量可以减少迭代的次数)。

[116, 149] 发现了一种优雅的并行寻找多色的算法：

1. 给每个变量分配一个随机值。
2. 找到比所有相邻变量的随机值更高的变量；也就是颜色 1。
3. 然后找到比所有非 1 色的邻居有更高随机值的变量。这就是颜色 2。
4. 反复进行，直到所有的点都被染上颜色。

### 6.8.3 不规则迭代空间

在显式时间步进的背景下或作为稀疏矩阵-向量乘积，应用计算模版是并行的。然而，在实践中，分割迭代空间可能并非易事。如果迭代空间是直角坐标砖，这很容易，即使是嵌套式的并行。然而，在对称的情况下，要做到均匀的负载分布就比较困难。一个典型的迭代空间看起来像。

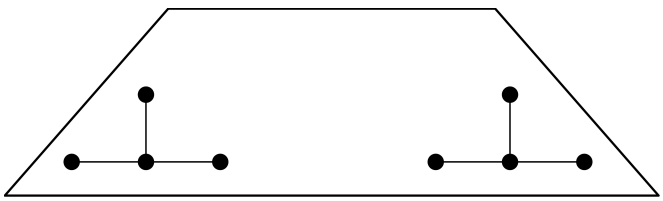
```
for (i=0; i<N; i++)
  for (j=i; j<N; j++)
    for (k=j; k<N; k++)
```

在某些情况下（见 [22]），界限可能更加复杂，如 $j = i + i\%2$ 或 $k < max(i + j, N)$。在这种情况下，可以做以下工作

1. 循环被遍历，以计算内迭代的总数；这将被分成许多部分，因为有进程。
2. 循环被遍历以找到每个过程的开始和结束的 $i$、$j$、$k$ 值。
3. 然后重写循环代码，使其能够在这样一个 $i, jk$ 子范围内运行。

### 6.8.4 缓存效率的排序

模板操作的性能通常相当低。这些操作没有明显的缓存重用；通常它们类似于流操作，从内存中获取长的数据流并只使用一次。如果只做一次模版评估，那就完了。然而，通常我们会做很多这样的更新，我们可以应用类似于 1.7.8 节中描述的循环叠加的技术。



如果我们考虑到模版的形状，我们可以做得比普通平铺更好。图 6.25 显示了（左）我们如何首先计算一个包含缓存的时空梯形。然后（右）我们计算另一个建立在第一个梯形上的包含缓存的梯形 [71]。

## 6.9 解除 PDE 线性系统的并行性

PDEs 的数值求解是一项重要的活动，所要求的精度往往使其成为并行处理的主要候选者。如果我们想知道我们能做到多大程度的并行，特别是可以达到什么样的速度，我们需要区分问题的各个方面。

首先，我们可以问问题中是否存在任何内在的并行性。在全局层面上，通常不会有这种情况（如果问题的一部分完全不耦合，那么它们将是独立的问题，对吗?）但在较小的层面上，可能存在并行性。

例如，看一下时间相关的问题，参考第 4.2.1 节，我们可以说每一个下一个时间步骤当然都依赖于上一个时间步骤，但不是下一个时间步骤的每一个单独的点都依赖于上一个步骤的每一个点：有一个影响区域。因此，有可能对问题域进行分区并获得并行性。

### 6.9.1 操作员拆分

在某些情况下，有必要通过一个二维或三维阵列的所有方向进行隐式计算。例如，在第 4.3 节中，你看到了热方程的隐式解是如何产生重复系统的

$$\left(\alpha I + \frac{d^2}{dx^2} + \frac{d^2}{dy^2}\right) u^{(t+1)} = u^{(t)}$$

在没有证明的情况下，我们指出，与时间有关的问题也可以通过以下方式解决

$$\left(\beta I + \frac{d^2}{dx^2}\right)\left(\beta I + \frac{d^2}{dy^2}\right) u^{(t+1)} = u^{(t)}$$

为合适的 $\beta$。这个方案不会在每个单独的时间步长上计算出相同的值，但它会收敛到相同的稳定状态。该方案也可以作为 BVP 情况下的一个预处理程序。

这种方法有相当大的优势，主要体现在运算次数上：原始系统的求解要么是对矩阵进行因式分解，产生填充，要么是通过迭代求解。

✍ **练习 6.33** 分析这些方法的相对优点，给出大致的运算次数。考虑 $\alpha$ 对 $t$ 有依赖性和没有依赖性的情况。同时讨论各种操作的预期速度。

当我们考虑 (6.3) 的并行解时，会出现进一步的优势。注意我们有一个二维的变量集 $u$，但是算子 $I + d^2u/dx^2$ 只连接 $u_{ij}, u_{ij-1}, u_{ij+1}$。也就是说，每一行对应的 i 值都可以被独立处理。因此，这两个算子都可以用一个一维分割域来完全并行地解决。另一方面，(6.2) 中系统的求解具有有限的并行性。

不幸的是，有一个严重的问题：$x$ 方向的算子需要在一个方向上对域进行分割，而 $y$ 方向的算子则需要在另一个方向上进行分割。通常采取的解决方案是在两个解之间对 $u_{ij}$ 值矩阵进行转置，以便同一处理器的分解可以处理这两个问题。这种转置可能会占用每个时间步骤的大量处理时间。

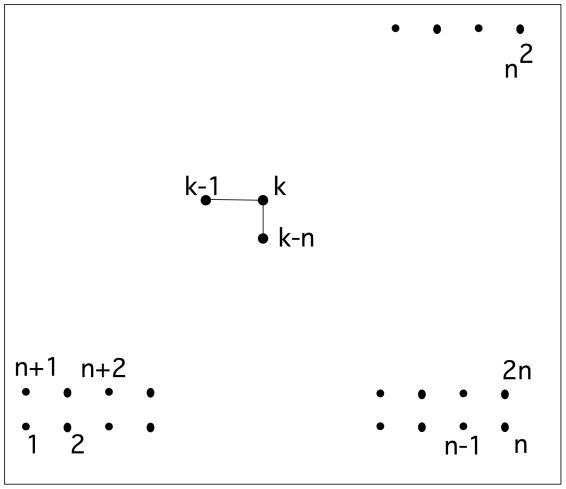
✍ **练习 6.34** 讨论使用 $P = p \times p$ 处理器的网格对域进行二维分解的优点和问题。你能提出一个方法来改善这些问题吗？加快这些计算的一个方法是用显式操作代替隐式求解；见 6.10.3 节。

## 6.10 并行性和隐式操作

在关于 IBVP 的讨论中（第 4.1.2.2 节），你看到从数值稳定性的角度来看，隐式运算有很大的优势。然而，你也看到，它们使基于简单操作（如矩阵-向量乘积）的方法与基于更复杂的线性系统求解的方法之间的区别。当你开始并行计算时，隐式方法会有更多的问题。

✍ **练习 6.35** 设 A 为矩阵

$$A = \begin{pmatrix} a_{11} & & \varnothing \\ a_{21} & a_{22} & \\ & \ddots & \ddots \\ \varnothing & & a_{n,n-1} \quad a_{nn} \end{pmatrix}$$




证明矩阵向量乘积 $y \leftarrow Ax$ 和系统解决方案 $x \leftarrow A^{-1}y$，通过解决三角系统 $Ax = y$，而不是通过反转 $A$ 得到，具有相同的操作数。现在考虑将乘积 $y \leftarrow Ax$ 并行化。假设我们有 $n$ 处理器，每个处理器 $i$ 存储 $x$ 和 $A$ 的第 $i$ 行。证明除了第一个处理器外，任何一个处理器都可以在没有闲置时间的情况下计算出 Ax 的结果。

三角形系统 $Ax = y$ 的解也可以这样做吗？显示出直接的实现方式是每个处理器在计算过程中都有 $(n-1)/n$ 的空闲时间。现在我们将看到一些处理这个固有的顺序部分的方法。

### 6.10.1 波峰

上面，你看到解决一个大小为 $N$ 的下三角系统，其顺序时间复杂度为 $N$ 步。在实践中，事情往往没有那么糟糕。像解三角系统这样的隐式算法本身就是顺序的，但步骤数可能比一开始看到的少。

✍ **练习 6.36** 再看一下单位面积上的二维 BVP 的矩阵，用中心差分法进行离散。如果我们按对角线排列未知数，请推导出矩阵结构。对于区块的大小和区块本身的结构，你能说什么？

让我们再看一下图 4.1，它描述了二维 BVP 的有限差分模版。图 6.26 是下三角因子的模版的相应图片。这描述了下三角解过程的顺序性 $x \leftarrow L^{-1}y$。

$$x_k = y_k - \ell_{k,k-1}x_{k-1} - \ell_{k,k-n}x_{k-n}$$

换句话说，如果点 $k$ 的左边（即变量 $k-1$）和下面（变量 $k-n$）的邻居是已知的，就可以找到它的值。

下方（变量 $k-n$）是已知的。

反过来，我们可以看到，如果我们知道 $x_1$，我们不仅可以找到 $x_2$，还可以找到 $x_{n+1}$。在下一步，我们可以确定 $x_3$、$x_{n+2}$ 和 $x_{2n+1}$。继续这样下去，我们可以通过波阵来解决 $x$：每个波阵上的 $x$ 的值都是独立的，所以它们可以在同一个顺序步骤中被并行解决。

✍ **练习 6.37** 完成这个论证。我们可以使用的最大处理器数量是多少，顺序步骤的数量是多少？最终的效率是多少？

当然，你不需要使用实际的并行处理来利用这种并行性。相反，你可以使用一个矢量处理器、矢量指令或GPU[148]。

在第 5.4.3.5 节中，你看到了用于减少矩阵填充的 Cuthill-McKee 排序。我们可以对这个算法进行如下修改，以得到波阵

1. 取一个任意的节点，并称其为" 零级"。
2. 对于第 $n+1$ 级，找到与第 $n$ 相连的点，这些点本身并不相连。
3. 对于所谓的" 反向 Cuthill-McKee 排序"，将层次的编号倒过来。

✍ **练习 6.38** 这个算法并不完全正确。问题出在哪里，你如何纠正？证明所产生的变换矩阵不再是三对角的，但可能仍有一个带状结构。

### 6.10.2 递推式翻倍

递归 $y_{i+1} = a_i y_i + b_i$, 例如在解双线性方程组时出现的情况 (见练习 4.4), 似乎是内在的顺序性。然而，你已经在练习 1.4 中看到，以一些初步操作为代价，计算是可以并行化的。

首先，从 (6.4) 中获取一般的双线性矩阵，并将其扩展为规范化的形式。

$$\begin{pmatrix} 1 & & \varnothing \\ b_{21} & 1 & \\ & \ddots & \ddots \\ \varnothing & & b_{n,n-1} \quad 1 \end{pmatrix}$$

我们把它写成 $A = I + B$。

✍ **练习 6.39** 说明可以通过与对角线矩阵相乘来实现对归一化形式的缩放。解决系统 $(I+B)x = y$ 对解决 $Ax = y$ 有什么帮助？以两种不同的方式求解该系统的运算量是多少?

现在我们做一些看起来像高斯消除的事情，只是我们不从第一行开始，而是从第二行开始。(如果你对矩阵 $I+B$ 进行高斯消除或 LU 分解，会发生什么?）我们用第二行来消除 $b_{32}$。

$$\begin{pmatrix} 1 & & \varnothing \\ & 1 & \\ & -b_{32} \quad 1 & \\ & & \ddots \\ \varnothing & & 1 \end{pmatrix} \times \begin{pmatrix} 1 & & \varnothing \\ b_{21} & 1 & \\ & b_{32} \quad 1 & \\ & & \ddots \quad \ddots \\ \varnothing & & b_{n,n-1} \quad 1 \end{pmatrix} = \begin{pmatrix} 1 & & & \varnothing \\ b_{21} & 1 & & \\ -b_{32}b_{21} & 0 & 1 & \\ \varnothing & & b_{n,n-1} & 1 \end{pmatrix}$$

我们把它写成 $L^{(2)}A = A^{(2)}$。我们还计算了 $L^{(2)}y = y^{(2)}$，因此 $A^{(2)}x = y^{(2)}$ 与 $Ax = y$ 有相同的解。解决转换后的系统让我们得到了一点好处：在我们计算了 $x_1$ 之后，$x_2$ 和 $x_3$ 可以被并行计算。

现在我们重复这个消除过程，用第四行来消除 $b_{54}$，第六行来消除 $b_{76}$，等等。最后的结果是，总结所有 $L(i)$) 矩阵。

$$\begin{pmatrix} 1 & & & & & & \\ 0 & 1 & & & & & \\ & -b_{32} \quad 1 & & & & & \\ & & 0 \quad 1 & & & & \\ & & & -b_{54} \quad 1 & & & \\ & & & & 0 \quad 1 & & \\ & & & & & -b_{76} \quad 1 & \\ & & & & & & \ddots \quad \ddots \end{pmatrix} \times (I+B) = \begin{pmatrix} 1 & & & & & & \varnothing \\ b_{21} & 1 & & & & & \\ -b_{32}b_{21} & 0 & 1 & & & & \\ & & b_{43} & 1 & & & \\ & & -b_{54}b_{43} & 0 & 1 & & \\ & & & & b_{65} & 1 & \\ & & & & -b_{76}b_{65} & 0 & 1 \\ & & & & & \ddots & \ddots \quad \ddots \end{pmatrix}$$

我们把它写成 $L(I+B) = C$，而解决 $(I+B)x = y$ 现在变成了 $C = L^{-1}y$。

这个最终结果需要仔细研究。

- 首先，计算 $y = L^{-1}y$ 很简单。(弄清楚细节，有多少并行性可用？)

- 解决 $Cx = y'$ 仍然是顺序的，但它不再需要 $n$ 步骤：从 $x_1$ 我们可以得到 $x_3$，从那里我们得到 $x_5$，等等。换句话说，$x$ 的奇数部分之间只存在顺序关系。
- $x$ 的偶数部分并不相互依赖，而只依赖奇数部分。$x_2$ 来自 $x_1$，$x_4$ 来自 $x_3$，依此类推。一旦奇数部分被计算出来，这一步就是完全并行的。

我们可以自行描述奇数部分的顺序解法。

$$\begin{pmatrix} 1 & & \varnothing & \\ c_{21} & 1 & & \\ & \ddots & \ddots & \\ \varnothing & & c_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y'_1 \\ y'_3 \\ \vdots \\ y'_n \end{pmatrix}$$

其中 $c_{i+1i} = -b_{2n+1,2n} b_{2n.2n-1}$。换句话说，我们已经将一个大小为 $n$ 的顺序问题简化为一个大小为同类的顺序问题和一个大小为 $n/2$ 的并行问题。现在我们可以递归地重复这个过程，将原来的问题还原为一连串的并行操作，每一个都是前者的一半大小。

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，但在抽象的情况下，它可以应用于任何关联运算符。

### 6.10.3 通过显性操作逼近隐性操作，系列扩展

如上所述，隐式运算在实践中是有问题的，有各种原因允许用另一种实际上更有利的方法来代替隐式运算：

- 只要我们遵守显式方法的步长限制，用显式方法代替隐式方法（第 4.3 节）同样是合法的。
- 在迭代方法中修补预处理程序（第 5.5.8 节）是允许的，因为它只会影响收敛速度，而不会影响方法收敛到的解。你已经在块状雅可比方法中看到了这个一般想法的一个例子；6.7.3 节。在本节的其余部分，你将看到预处理程序中的递归，即隐式操作，如何被显式操作所取代，从而带来各种计算优势。

求解线性系统是隐式运算的一个很好的例子，由于这归结为求解两个三角形系统，让我们来看看如何找到替代求解下三角系统的计算方法。如果 U 是上三角且非 Singular，我们让 D 是 U 的对角线，我们写成 U = D(I - B) 其中 B 是一个对角线为零的上三角矩阵，也称为严格上三角矩阵；我们说 I - B 是一个单位上三角矩阵。

✍ **练习 6.40** 设 $A = LU$ 是一个 LU 分解，其中 $L$ 的对角线上有 1。说明如何解决 $Ax = b$ 的问题，只涉及单位上下三角系统的解决。证明在系统求解过程中不需要除法。

我们现在感兴趣的操作是解决系统 $(I - B)x = y$。我们观察到

$$(I - B)^{-1} = I + B + B^2 + \cdots$$

和 $B^n = 0$，其中 $n$ 是矩阵的大小（检查这个!），所以我们可以通过以下方法精确求解 $(I - B)x = y$。

$$x = \sum_{k=0}^{n-1} B^k y$$

当然，我们希望避免明确计算幂 $B^k$，所以我们观察到

$$\sum_{k=0}^{1} B^k y = (I + B)y, \quad \sum_{k=0}^{2} B^k y = (I + B(I + B))y, \quad \sum_{k=0}^{3} B^k y = (I + B(I + B((I + B))))y$$

诸如此类。由此产生的评估 $\sum_{k=0}^{n-1} B^k y$ 的算法被称为霍纳规则，你看它避免了计算矩阵幂 $B^k$。

✍ **练习 6.41** 假设 $I - B$ 是二对角线。证明上述计算需要 $n(n + 1)$ 操作。通过三角解法计算 $(I - B)x = y$ 的操作数是多少?

我们现在已经把隐式运算变成了显式运算，但不幸的是，这种运算的次数很高。然而，在实际情况下，我们可以截断矩阵的功率之和。

✍ **练习 6.42** 设 $A$ 为三对角矩阵

$$A = \begin{pmatrix} 2 & -1 & & & \varnothing \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ \varnothing & & & -1 & 2 \end{pmatrix}$$

4.2.2 节中的一维 BVP。

1. 回顾 5.3.4 节中对角线支配的定义。这个矩阵是对角线主导的吗?
2. 证明该矩阵的 LU 因子化中的枢轴（无枢轴）满足递归。提示：说明经过 $n$ 消除步骤（$n \geqslant 0$）后，剩余的矩阵看起来像

$$A^{(n)} = \begin{pmatrix} d_n & -1 & & & \varnothing \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ \varnothing & & & -1 & \end{pmatrix}$$

并说明 $d_{n+1}$ 与 $d_n$ 之间的关系。

3. 证明序列 $n \mapsto d_n$ 是递减的，并推导出其极限值。
4. 写出以 $L$ 和 $U$ 为单位的 $d_n$ 枢轴的系数。
5. $L$ 和 $U$ 因子是对角线主导的吗?

上述练习意味着（注意，我们实际上并没有证明!），对于来自 BVP 的矩阵，我们发现 $B^k \downarrow 0$，在元素大小和规范方面都是如此。这意味着我们可以用诸如 $(I - B)x = y$ 的方法来近似地计算 $(I + B)y$ 或者 $x = (I + B + B^2)y$。这样做仍然比直接三角解法有更多的操作数，但至少在两个方面有计算上的优势。

- 显式算法有更好的流水线行为。
- 正如你所看到的，隐式算法在并行时有问题；显式算法更容易并行化。

当然，这种近似可能对整个数值算法的稳定性有进一步的影响。

✍ **练习 6.43** 描述霍纳法则的并行性方面；方程 (6.6)。

## 6.11 网格更新

第四章的结论之一是，时间相关问题的显式方法在计算上比隐式方法容易。例如，它们通常涉及矩阵-向量乘积而不是系统解，而且显式操作的并行化相当简单：矩阵-向量乘积的每个结果值都可以独立计算。这并不意味着还有其他值得一提的计算方面。

由于我们处理的是稀疏矩阵，源于一些计算模版，我们从操作者的角度出发。在图 6.11 和 6.12 中，你看到了在域的每一点上应用模版是如何引起处理器之间的某些关系的：为了在一个处理器上评估矩阵-向量乘积 $y \leftarrow A$，该处理器需要获得其幽灵区域的 $x$ 的值。在合理的假设下，在处理器上划分领域，涉及的信息数量将相当少。

✍ **练习 6.44** 推理一下，在有限元或有限元分析的背景下，当 $h \downarrow 0$ 时，信息的数量是 $O(1)$。

在第 1.6.1 节中，你看到矩阵-向量乘积几乎没有数据重用，尽管计算有一定的位置性；在第 5.4.1.4 节中指出，稀疏矩阵-向量乘积的位置性更差，因为稀疏性必须有索引方案。这意味着稀疏乘积在很大程度上是一种受带宽限制的算法。

只看一个单一的乘积，我们对此没有什么办法。然而，我们经常连续做一些这样的乘积，例如作为一个随时间变化的过程的步骤。在这种情况下，可能会对操作进行重新安排，以减少对带宽的需求。作为一个简单的例

子，可以考虑

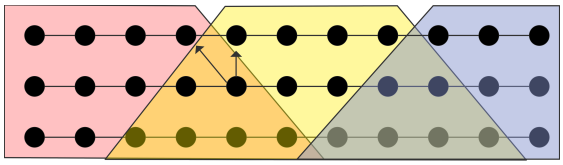
$$\forall_i : x_i^{(n+1)} = f\left(x_i^{(n)}, x_{i-1}^{(n)}, x_{i+1}^{(n)}\right)$$

并假设集合 $\{x(n)\}$ 太大，无法装入缓存。这是一个模型，例如，在一个空间维度上热方程的显式 $i$ 方案；4.3.1.1 节。从原理上讲。

$$\begin{array}{ccc} x_0^{(n)} & x_1^{(n)} & x_2^{(n)} \\ \downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\ x_0^{(n+1)} & x_1^{(n+1)} & x_2^{(n+1)} \\ \downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\ x_0^{(n+2)} & x_1^{(n+2)} & x_2^{(n+2)} \end{array}$$

在普通计算中，我们先计算所有 $x(n+1)$，然后再计算所有 $x(n+2)$，在 $n+1$ 级别的中间值产生后会从缓存中刷掉，然后再作为 $n+2$ 级别数量的输入带回缓存中。

然而，如果我们计算的不是一个，而是两个迭代，中间值可能会留在缓存中。考虑到 $x(n+2)$：它需要 $x(n+1)$、$x(n+1)$，而后者又需要 $x(n)$、..., $x(n)$。



现在假设我们对中间结果不感兴趣，而只对最后的迭代感兴趣。图 6.27 是一个简单的例子。第一个处理器计算了 $n+2$ 层的 4 个点。为此，它需要从 $n+1$ 层计算 5 个点，而这些点也需要从 $n$ 层的 6 个点中计算出来。我们看到，一个进程显然需要收集一个宽度为 2 的重影区域，而常规的单步更新只需要一个。第一个处理器计算的一个点是 $x(n+2)$，它需要 $x(n+1)$。这个点也需要用于计算 $x(n+2)$，属于第二个处理器。

最简单的解决方法是让中间层的这种点冗余计算，在需要它的两个区块的计算中，在两个不同的处理器上进行。

✍ **练习 6.45** 你能想到一个点会被两个以上的处理器冗余计算的情况吗？

我们可以对这种按块计算多个更新步骤的方案给出几种解释：

- 首先，如上所述，用一个处理器进行计算可以增加 locality：如果一个彩色块中的所有点（见图）都适合于缓存，我们就可以重复使用中间的点。
- 其次，如果我们把它看作是分布式内存计算的化学反应，它减少了信息流量。通常，对于每一个更新步骤，处理器都需要交换他们的边界数据。如果我们接受一些多余的重复工作，我们现在可以消除中间层的数据交换。通信的减少通常会超过工作的增加。

✍ **练习 6.46** 讨论一下在多核计算中使用这种策略的情况。有哪些节省？有哪些潜在的隐患？

### 6.11.1 分析

让我们分析一下我们刚刚勾画的算法。如同方程（6.7），我们把自己限制在一个一维的点集和一个三点的函数。描述这个问题的参数是这样的。

- $N$ 是要更新的点的数量，$M$ 表示更新步骤的数量。因此，我们进行 $MN$ 函数评估。
- $\alpha, \beta, \gamma$ 是描述延迟、单点传输时间和操作时间（这里认为是 $f$ 评价）的通常参数。
- $b$ 是我们挡在一起的步骤数。

每个光环通信由 $b$ 点组成，我们这样做 $\sqrt{N}/b$ 多次。所做的工作包括 $MN/p$ 局部更新，再加上由于晕轮而产生的冗余工作。后者包括 $b^2/2$ 个操作，在处理器域的左边和右边都进行。

将所有这些条款加在一起，我们发现成本为

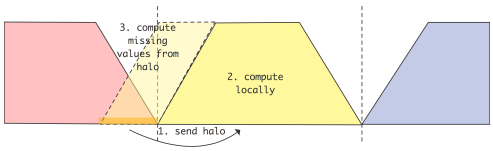
$$\frac{M}{b}\alpha + M\beta + \left(\frac{MN}{p} + Mb\right)\gamma$$

我们观察到，$\alpha M/b + \gamma Mb$ 的开销是与 $p$ 无关的。

✍ **练习 6.47** 计算 $b$ 的最优值，并指出它只取决于结构参数 $\alpha, \beta, \gamma$ 而不取决于问题参数。

### 6.11.2 沟通和工作最小化战略

我们可以通过将计算与通信重叠来使这个算法更有效率。如图 6.28 所示，每个处理器从通信其光环开始，并将此通信与可在本地完成的通信部分重叠。然后，依赖于光环的值将被最后计算。

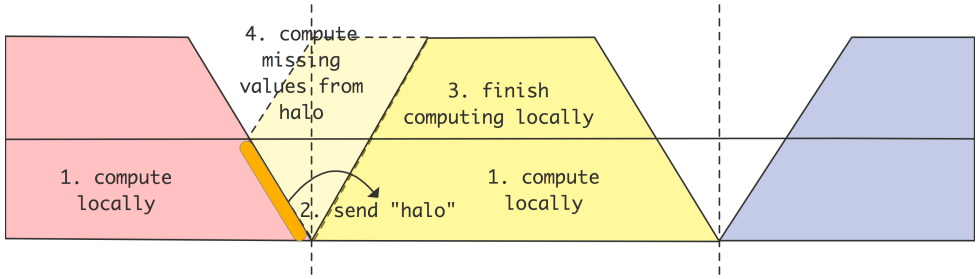



✍ **练习 6.48** 这样组织你的代码（重点是' 代码'!）有什么大的实际问题?

如果每个处理器的点数足够大，那么相对于计算来说，通信量就很低，你可以把 $b$ 拿得相当大。然而，这些网格更新大多用于迭代方法，如 CG 方法（第 5.5.11 节），在这种情况下，对舍入的考虑使你不能把 $b$ 拿得太大 [32]。

✍ **练习 6.49** 在点被组织成二维网格的情况下，通过对非重叠算法的复杂性分析。假设每个点的更新涉及四个邻居，每个坐标方向上有两个邻居。

上述算法的进一步细化是可能的。图 6.29 说明有可能使用一个使用不同时间步骤的不同点的晕区。这种算法（见 [42]）减少了冗余的计算量。然而，现在需要先计算交流的光环值，所以这需要将本地交流分成两个阶段。




## 6.12 多核架构上的块状算法

在第 5.3.7 节中，你看到某些线性代数算法可以用子矩阵来表述。这个观点对于在共享内存架构（如目前的多核处理器）上高效执行线性代数操作是有益的。

作为一个例子，让我们考虑 Cholesky 因式分解，它计算 $A = LL^t$ 为一个对称的正定矩阵。

正定矩阵 $A$；另见 5.3.2 节。递归地，我们可以将该算法描述如下。

$$\mathrm{Chol}\begin{pmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{pmatrix} = LL^t \quad \text{where} \quad L = \begin{pmatrix} L_{11} & 0 \\ \tilde{A}_{21} & \mathrm{Chol}\left(A_{22} - \tilde{A}_{21}\tilde{A}_{21}^t\right) \end{pmatrix}$$

其中 $\tilde{A}_{21} = A_{21}L_{11}^{-t}, A_{11} = L_{11}L_{11}^t$

在实践中，区块实现被应用于一个分区

$$\left(\begin{array}{c|cc} & \multicolumn{2}{c}{\text{finished}} \\ \hline & A_{kk} & A_{k,>k} \\ \hline & A_{>k,k} & A_{>k,>k} \end{array}\right)$$

其中 k 是当前块行的索引，对于所有索引 $< k$，因式分解已经完成。因式分解的写法如下，用 Blas 的名字表示操作。

for$k = 1$, nblocks : Chol: factor$L_k L_k^t \leftarrow A_{kk}$Trsm: solve$\tilde{A}_{>k,k} \leftarrow A_{>k,k} L_k^{-t}$Gemm: form the product$\tilde{A}_{>k,k} \tilde{A}_{>k,k}^t$Syrk: symmmetric ra

并行性能的关键是对指数 $> k$ 进行分区，并以这些块为单位编写算法。

$$\left(\begin{array}{c|cc} & \multicolumn{2}{c}{\text{finished}} \\ \hline A_{kk} & A_{k,k+1} & A_{k,k+2}\cdots \\ \hline A_{k+1,k} & A_{k+1,k+1} & A_{k+1,k+2}\cdots \\ A_{k+2,k} & A_{k+2,k+2} & \\ \vdots & \vdots & \end{array}\right)$$

该算法现在得到了一个额外的内循环级别。

for$k = 1$, 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$\tilde{A}_{\ell_1,k} \tilde{A}_{\ell_2,k}^t$f

很明显，该算法具有很好的并行性：每个 l 环的迭代都可以独立处理。然而，这些循环在外层 $k$-循环的每一次迭代中都会变短，所以我们能容纳多少个处理器并不直接。此外，没有必要保留上述算法的操作顺序。例如，在

$$L_1 L_1^t = A_{11}, \quad A_{21} \leftarrow A_{21} L_1^{-t}, \quad A_{22} \leftarrow A_{22} - A_{21} A_{21}^t$$

因式分解 $L_2 L_2^t = A_{22}$ 可以开始，即使剩下的 $k = 1$ 迭代仍未完成。因此，比起我们仅仅对内循环进行并行化，可能存在着更多的并行性。

在这种情况下，处理并行性的最好方法是将算法的控制流观点（其中操作顺序是规定的）转变为数据流观点。在后者中，只有数据的依赖性被指出，而且任何服从这些依赖性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，代之以部分排序 5) 。表示算法的数据流的最好方法是构建一个任务的有向无环图（DAG）（见第 18 节关于图的简要教程）。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 6.50** 在 2.6.1.6 节中，你学到了顺序一致性的概念：一个线程化的并行代码程序在并行执行时应该给出与顺序执行时相同的结果。我们刚刚说过，基于 DAG 的算法可以自由地以任何服从图节点的部分顺序来执行任务。讨论一下在这种情况下，顺序一致性是否是一个问题。

在我们的例子中，我们通过为每个内部迭代制定一个顶点任务来构造一个 DAG。图 6.30 显示了 4Œ4 块的矩阵的所有任务的 DAG。这个图是通过模拟上面的 Cholesky 算法构建的。

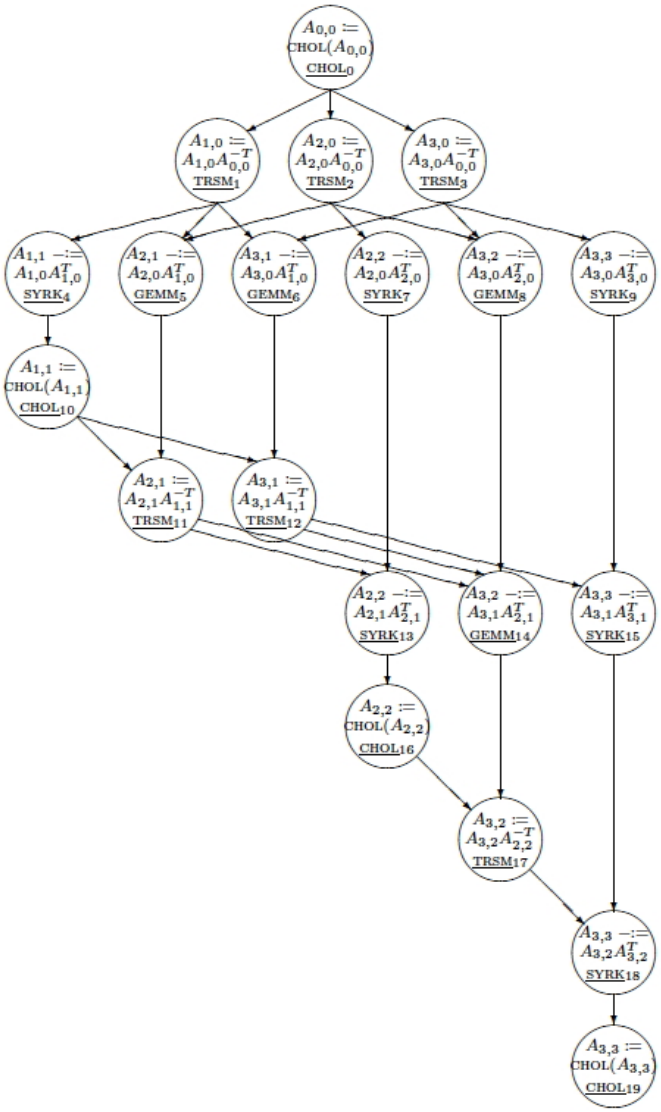
✍ **练习 6.51** 这个图的直径是多少？识别出位于决定直径的路径上的任务。这些任务在该算法中的意义是什么？这条路径被称为关键路径。它的长度决定了并行计算的执行时间，即使有无限多的处理器可用。

✍ **练习 6.52** 假设有 $T$ 个任务，都需要一个单位时间来执行，并假设我们有 $p$ 个处理器。理论上执行该算法的最小时间是多少？现在修改这个公式以考虑到关键路径；称其长度为 $C$。

在执行任务的过程中，一个 DAG 可以有几个观察点：

- 如果有一个以上的更新被加载到锁上，那么让这些更新由同一个进程来计算，可能会更有优势。这样可以简化维护缓存一致性的工作。

- 如果数据被使用并随后被修改，那么在修改开始之前必须完成使用。如果这两个动作是在不同的处理器上进行的，这甚至可能是真的，因为内存子系统通常会保持缓存一致性，所以修改会影响正在读取数据的进程。这种情况可以通过在主内存中有一个数据的拷贝来补救，给读取数据的进程保留一个数据（见 1.4.1 节）。



# 第二部分

# 应用（Applications）

# 第 7 章 分子动力学

内容提要

- 受力计算
- 并行分解
- 并行快速傅立叶变换
- 分子动力学整合

分子动力学是模拟分子的逐个原子行为并从这些原子运动中推导出宏观属性的技术。它适用于生物分子，如蛋白质和核酸，以及材料科学和纳米技术的天然和合成分子。分子动力学属于粒子方法的范畴，其中包括天体力学和天体物理学中的 N 体问题，这里介绍的许多观点将延续到这些其他领域。此外，还有一些分子动力学的特殊情况，包括从头开始的分子动力学，其中电子被量子力学地处理，因此可以对化学反应进行建模。我们将不讨论这些特殊情况，而是集中讨论经典的分子动力学。

分子动力学的想法非常简单：一组粒子根据牛顿运动定律相互作用，$F = ma$。考虑到初始粒子的位置和速度、粒子的质量和其他参数，以及粒子之间的作用力模型，牛顿运动定律可以通过数值集成来给出每个粒子在未来（和过去）所有时间的运动轨迹。通常情况下，粒子居住在一个具有周期性边界条件的计算箱中。

因此，一个分子动力学的时间步骤由两部分组成：

- 计算各粒子的力量；
- 更新位置（整合）。

力的计算是开销最大的部分。最先进的分子动力学模拟是在并行计算机上进行的，因为力的计算十分复杂，而且合理的模拟长度需要大量的时间步骤。在许多情况下，分子动力学也被应用于具有非常多原子的分子的模拟，例如，对于生物分子和长时间尺度来说，可以达到一百万，对于其他分子和短时间尺度来说，可以达到数十亿。

数值整合技术在分子动力学中也很有意义。对于需要大量时间步长的模拟来说，能量等量的保存比准确度更重要，必须使用的求解器与第四章中介绍的传统 ODE 求解器不同。

在下文中，我们将介绍用于生物分子模拟的力场，并讨论计算这些力的快速方法。然后，我们将专门讨论短程力的分子动力学并行化和用于长程力快速计算的三维 FFT 的并行化。

最后，我们用一个章节来介绍适合于分子动力学模拟的集成技术类别。我们在本章中对分子动力学主题的处理是为了介绍和实用；如果想了解更多的信息，建议阅读文本 [68]。

## 7.1 受力计算

### 7.1.1 力场分析

在经典的分子动力学中，势能和原子间作用力的模型被称为力场。力场是一个可操作但近似的量子力学效应模型，对于大分子来说，计算成本太高，无法确定。不同的力场用于不同类型的分子，也有不同的研究人员用于同一分子，而且没有一个是理想的。

在生化系统中，常用的力场将势能函数建模为键合能、范德瓦尔斯能和静电（库仑）能之和。

$$E = E_{\text{bonded}} + E_{\text{Coul}} + E_{\text{vdW}}$$

势是模拟中所有原子位置的函数。原子上的力是原子位置上的这个势的负梯度。粘合能是由于分子中的共价键造成的。

$$E_{\text{bonded}} = \sum_{\text{bonds}} k_i \left(r_i - r_{i,0}\right)^2 + \sum_{\text{angles}} k_i \left(\theta_i - \theta_{i,0}\right)^2 + \sum_{\text{torsions}} V_n 1 + \cos(n\omega - \gamma))$$

其中，这三个项分别是所有共价键的总和、两个键形成的所有角度的总和以及三个键形成的所有二面体角度的总和。固定参数 $k_i, r_{i,0}$ 等取决于所涉及的原子类型，对于不同的力场可能有所不同。额外的项或具有不同

函数形式的项也是常用的。

势能的其余两个项 $E$ 统称为非键合项。理论上，它们构成了力计算的主体。静电能是由原子电荷引起的，其模型是我们熟悉的

$$E_{\text{Coul}} = \sum_i \sum_{j>i} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

其中，总和为所有原子对，$q_i$ 和 $q_j$ 是原子 $i$ 和 $j$ 的电荷，$r_{ij}$ 是原子 $i$ 和 $j$ 之间的间距。最后，范德瓦尔斯能近似于剩余的吸引和排斥效应，通常用伦纳德-琼斯函数来模拟

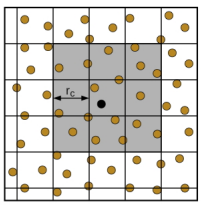
$$E_{\text{vdW}} = \sum_i \sum_{j>i} 4\epsilon_{ij} \left[ \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right]$$

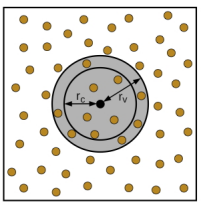
其中 $\epsilon_{ij}$ 和 $\sigma_{ij}$ 是力场参数，取决于原子类型。在短距离上，排斥性的（$r^{12}$）项起作用，而在长距离上，分散（吸引，$-r^6$）项起作用。

分子动力学力计算的并行化取决于这些不同类型的力计算的并行化。键合力是局部计算，即对于一个给定的原子，只需要附近的原子位置和数据。范德瓦耳斯力也是局部的，被称为短程的，因为它们对于大的原子分离来说是可以忽略的。静电力是长程的，各种技术已经被开发出来以加快这些计算。在接下来的两个小节中，我们分别讨论短程和长程非键合力的计算。

### 7.1.2 计算短程非结合力

对一个粒子的短程非粘合力的计算可以在超过该粒子的截止半径 $r_c$ 时被截断。对某一粒子 $i$ 进行这种计算的天真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 $n$ 粒子来说，这种方法的复杂性是 $On^2$，相当于计算所有粒子对之间的力。有两种数据结构，即单元格列表和 Verlet 邻居列表，可以独立用于加速这种计算，还有一种方法是将两者结合起来。




#### 7.1.2.1 元胞列表

元胞列表的想法经常出现在寻求给定点附近的一组点的问题中。参照图 7.1（a），我们用一个二维的例子来说明这个想法，在粒子集合上铺设一个网格。如果网格间距不小于 $r_c$，那么为了计算对粒子 i 的作用力，只需要考虑包含 $i$ 的单元和 8 个相邻单元的粒子。对所有的粒子进行一次扫描就可以为每个单元构建一个粒子列表。这些元胞列表被用来计算所有粒子的力。在下一个时间步骤中，由于粒子已经移动，必须重新生成或更新元胞列表。这种方法的复杂性是计算数据结构的 $O(n)$ 和计算力的 $O(n \times n_c)$，其中 $n_c$ 是 9 个单元的平均粒子数（三维的 27 个单元）。元胞列表数据结构所需的存储量为 $O(n)$。

#### 7.1.2.2 Verlet 近邻表

元胞列表结构有些低效，因为对于每个粒子 $i$，$n_c$ 粒子被考虑，但这远远多于截止点 $r_c$ 内的粒子数量。一个 Verlet 邻居列表是一个粒子 i 的截止点内的粒子列表。每个粒子都有自己的列表，因此需要的存储量为 $O(n \times n_v)$，

其中 $n_v$ 是截止点内粒子的平均数量。一旦构建了这些列表，计算力就会非常快，需要最小的复杂度 $O(n \times n_v)$。构建列表的成本较高，需要检查每个粒子的所有粒子，即不低于原来的复杂性 $O(n^2)$。然而，其优点是，如果使用扩大的截止点 $r_v$，近邻表可以在许多时间步骤中重复使用。参考图 7.1（b）中的一个二维例子，只要没有来自两个圆圈外的粒子在内圈内移动，近邻表就可以被重复使用。如果粒子的最大速度可以被估计或限定，那么我们可以确定一个时间步数，在这个时间步数内重复使用邻居列表是安全的。（另外，也可以在任何粒子穿越到截止点内的位置时发出信号）。从技术上讲，Verlet 近邻表是在扩展的截止点内的粒子列表，$r_v$。

#### 7.1.2.3 同时使用元胞和近邻表

混合方法是简单地使用 Verlet 近邻表，但使用元胞列表来构建近邻表。这减少了需要重新生成近邻表时的高成本。这种混合方法非常有效，也是最先进的分子动力学软件中经常使用的方法。

单元列表和 Verlet 近邻表都可以被修改，以利用以下事实：由于粒子 $f_{ij}$ 的作用力等于 $-f_{ji}$（牛顿第三定律），只需要计算一次。例如，对于元胞列表，只需要考虑 8 个单元格中的 4 个（在二维）。

### 7.1.3 计算长程力

静电力的计算具有挑战性，因为它们是长程的：每个粒子都感受到来自模拟中所有其他粒子的不可忽略的静电力。有时使用的一种近似方法是在某个截止半径后截断粒子的力计算（就像对短程范德瓦尔斯力所做的那样）。然而，这通常会在结果中产生不可接受的假象。

有几种更精确的方法可以加快静电力的计算，避免对所有 $n$ 粒子的 $On^2$ 和。我们在此简要介绍其中一些方法。

#### 7.1.3.1 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非常流行，但对于生物分子模拟所要求的精度来说，通常成本太高。在 Barnes-Hut 方法中，空间被递归地划分为 8 个相等的单元（三维），直到每个单元包含零或一个粒子。附近的粒子之间的作用力是单独计算的，就像正常情况一样，但是对于远处的粒子，作用力是在一个单元内的一个粒子和一组远处的粒子之间计算的。准确度的测量被用来确定是否可以使用远处的单元来计算力，或者必须通过单独考虑其子单元来计算。Barnes-Hut 方法的复杂度为 $On \log n$。快速多极法的复杂度为 $On$；该方法计算势，不直接计算力。

#### 7.1.3.2 粒子网格法

在粒子网格方法中，我们利用泊松方程

$$\nabla^2 \phi = -\frac{1}{\epsilon}\rho$$

它将电势 $\phi$ 与电荷密度 $\rho$ 联系起来，其中 $1/\epsilon$ 是一个比例常数。为了利用这个方程，我们用一个网格来离散空间，给网格点分配电荷，解决网格上的泊松方程，得出网格上的势。力是电位的负梯度（对于保守力，如静电力）。许多技术已经被开发出来，用于将空间中的点电荷分布到一组网格点上，也用于数值插值点电荷上由于网格点的电势而产生的力。许多快速方法可用于解决泊松方程，包括多网格方法和快速傅里叶变换。在术语方面，粒子网格方法与原始的粒子-粒子方法相反，后者是在所有粒子对之间计算力。

事实证明，粒子网格法不是很准确，一个更准确的替代方法是将每个力分成短程、快速变化的部分和长程、缓慢变化的部分。

$$f_{ij} = f_{ij}^{sr} + f_{ij}^{lr}$$

实现这一目标的一个方法是用一个函数 $hr$ 来衡量 f，它强调短程部分（小 $r$ 用 $1 - hr$ 来强调长程部分（大 $r$）。短程部分是通过计算截止点内所有粒子对的相互作用来计算的（粒子-粒子方法），长程部分是用粒子-网格

方法计算的。由此产生的方法，称为粒子-粒子-网格（PPPM，或 P3），是由于霍克尼和伊斯特伍德在 1973 年开始的一系列论文中提出的。

#### 7.1.3.3 埃瓦尔德法

Ewald 方法是迄今为止描述的生物分子模拟中最流行的静电力的方法，是为周期性边界条件的情况而开发的。该方法的结构与 PPPM 相似，即力被分成短程和长程两部分。同样，短程部分用粒子-粒子方法计算，长程部分用傅里叶变换计算。Ewald 方法的变种与 PPPM 非常相似，因为长程部分使用网格，而快速傅里叶变换被用来解决网格上的泊松方程。更多的细节，请参见，例如 [68]。在第 7.3 节中，我们描述了三维 FFT 的并行化以解决三维 Poisson 方程。

## 7.2 并行分解

我们现在讨论力的并行计算。Plimpton[169] 创建了一个非常有用的分子动力学并行化方法的分类，确定了原子、力和空间分解的方法。在此，我们紧跟他对这些方法的描述。我们还增加了第四类，这类方法已被公认为与前面的类别不同，称为中性领土方法，这个名字是由 Shaw[182] 创造的。中立领地方法目前被许多最先进的分子动力学代码所使用。空间分解和中性领土方法对基于截止点的计算的并行化特别有利。

### 7.2.1 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，并在整个模拟中更新其位置。为了使计算大致平衡，每个处理器被分配到大致相同数量的粒子（随机分布效果好）。原子分解的一个重要观点是，每个处理器一般都需要与所有其他处理器进行通信，以共享更新的粒子位置。

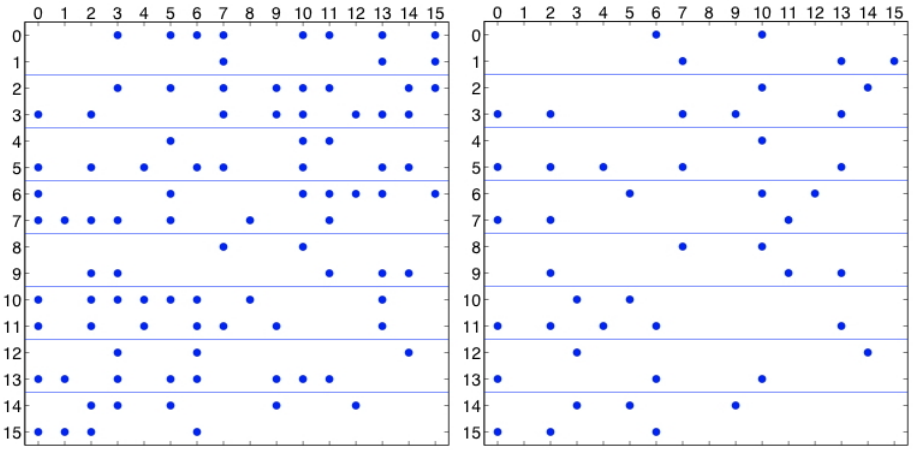



**图 7.1:** 原子分解，显示了分布在 8 个处理器中的 16 个粒子的力矩阵。一个点代表力矩阵中的一个非零条目。在左边，矩阵是对称的；在右边，为了利用牛顿第三定律，只计算一对偏斜对称元素中的一个元素。

图 7.2（a）中的力矩阵说明了一个原子分解。对于 $n$ 粒子，力矩阵是一个 $n \times n$ 的矩阵；行和列按粒子指数编号。矩阵中的非零条目 $f_{ij}$ 表示由于粒子 $j$ 而对粒子 $i$ 产生的非零力，必须计算出来。这个力可能是一个非粘结力和/或粘结力。当使用截断时，矩阵是稀疏的，如本例中。如果在所有粒子对之间计算力，矩阵是密集的。由于牛顿第三定律 $f_{ij} = -f_{ji}$，该矩阵是歪斜对称的。图 7.2（a）中的线条显示了粒子是如何被分割的。在图中，16 个粒子被划分到 8 个处理器中。

算法 1 从一个处理器的角度展示了一个时间步骤。在时间步骤开始时，每个处理器持有分配给它的粒子的位置。

一个优化是将计算量减半，这是有可能的，因为力矩阵是

```
Algorithm 1 Atom decomposition time step
1: send/receiveparticlepositionsto/fromallotherprocessors
2: （ifnonbondedcutoffsareused）determinewhichnonbondedforcesneedtobecomputed 3:
    computeforcesforparticlesassignedtothisprocessor
4: updatepositions（integration）forparticlesassignedtothisprocessor
```

歪斜对称的。为了做到这一点，我们为所有偏斜对称对准确选择 $f_i$ 或 $f_{ji}$ 中的一个，这样每个处理器负责计算的力的数量大致相同。选择力矩阵的上三角或下三角部分是一个不好的选择，因为计算负荷是不平衡的。更好的选择是，如果 $i + f_{ij}$ 在上三角中是偶数，或者如果 $i + j$ 在下三角中是奇数，就计算 $j$，如图 7.2（b）所示。还有许多其他选择。

当利用力矩阵中的歪斜对称性时，一个处理器所拥有的粒子上的所有力不再由该处理器来计算。例如，在图 7.2（b）中，粒子 1 上的力不再只由第一个处理器计算。为了完成力的计算，处理器必须通过通信来发送其他处理器需要的力，并接收其他处理器计算的力。现在必须修改上述算法，增加一个通信步骤（步骤 4），如算法 2 所示。

```
Algorithm 2 Atom decomposition time step, without redundant calculations 1: send/
    receiveparticlepositionsto/fromallotherprocessors
2: （ifnonbondedcutoffsareused）determinewhichnonbondedforcesneedtobecomputed
3: computepartialforcesforparticlesassignedtothisprocessor
4: sendparticleforcesneededbyotherprocessorsandreceiveparticleforcesneededbythisprocessor 5:
    updatepositions（integration）forparticlesassignedtothisprocessor
```

如果额外的通信量被节省的计算量所抵消，这种算法是有利的。请注意，一般情况下，通信量会增加一倍。

### 7.2.2 力的分解

在力的分解中，力被分配到各处理器中进行计算。做到这一点的直接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 7.3（a）说明了 16 个粒子和 16 个处理器的情况。粒子也需要被分配给处理器（如在原子分解中），目的是让处理器被分配来更新粒子的位置。在图中的例子中，处理器 i 被分配来更新粒子 i 的位置；在实际问题中，一个处理器会被分配来更新许多粒子的位置。请注意，同样，我们首先考虑的是倾斜对称的力矩阵的情况。

现在我们来看看一个时间步骤中力的分解所需的通信。考虑处理器 3，它计算粒子 0、1、2、3 的部分力，并需要粒子 0、1、2、3 以及 12、13、14、15 的位置。因此，处理器 3 需要与处理器 0、1、2、3，以及处理器 12、13、14、15 进行通信。在所有处理器计算完力之后，处理器 3 需要收集由其他处理器计算的对粒子 3 的力。因此，处理器 2 需要再次与处理器 0、1、2、3 进行通信。

<img src="graphics/fig-force1sym.jpg" alt="fig-force1" style="zoom: 53<img src="graphics/fig-force1sym.jpg" alt="fig-force1sym" style="zoom:50

算法 3 显示了从一个处理器的角度来看，在一个时间步骤中所进行的工作。在时间步骤开始时，每个处理器持有分配给它的所有粒子的位置。

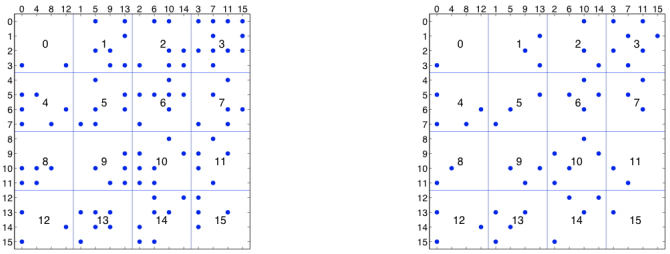
```
Algorithm 3 Force decomposition time step
1: send positions of my assigned particles which are needed by other processors; receive row particle
positions needed by my processor （this communication is between processors in the same processor
row, e.g., processor 3 communicates with processors 0, 1, 2, 3）
2: receivecolumnparticlepositionsneededbymyprocessor（thiscommunicationisgenerallywithpro-
cessors in another processor row, e.g., processor 3 communicates with processors 12, 13, 14, 15）
3: （ifnonbondedcutoffsareused）determinewhichnonbondedforcesneedtobecomputed
4: computeforcesformyassignedparticles
```

```
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: updatepositions（integration）formyassignedparticles
```

一般来说，如果有 $p$ 个处理器（为简单起见，$p$ 为正方形），那么力矩阵被 $\sqrt{p}$ 个块分割成 $\sqrt{p}$ 个。刚刚描述的力的分解需要处理器分三步进行通信，每一步都有 $\sqrt{p}$ 个处理器。这比需要在所有 $p$ 处理器之间进行通信的原子分解要有效得多。

我们还可以利用牛顿第三定律进行力的分解。像原子分解一样，我们首先选择一个修改过的力矩阵，其中只有 $f_i$ 和 $f_{ji}$ 中的一个被计算。粒子 $i$ 上的力是由一排处理器计算的，现在也由一列处理器计算。因此，每个处理器需要一个额外的通信步骤，从一列处理器中收集分配给它的粒子的力。以前有三个通信步骤，现在利用牛顿第三定律时有四个通信步骤（在这种情况下，通信不是像原子分解那样加倍的）。

对力的分解的修改可以节省一些通信。在图 7.4 中，各列被重新排序，使用的是块-循环排序。再考虑一下处理器 3，它计算粒子 0、1、2、3 的部分力。它需要来自粒子 0、1、2、3 的位置，和以前一样，但现在也需要处理器 3、7、11、15。后者是与处理器 3 同列的处理器。因此，所有通信都在同一处理器行或处理器列内，这在基于网状的网络架构上可能是有利的。修改后的方法显示为算法 4。



## 7.2.3 空间分解

在空间分解中，空间被分解为单元。每个单元被分配给一个处理器，负责计算位于该单元内的粒子上的力。图 7.5（a）说明了在二维模拟的情况下，空间被分解为 64 个单元。（这是空间的分解，不能与力矩阵相混淆）。通常情况下，单元的数量被选择为与处理器的数量相等。由于粒子在模拟过程中移动，粒子在单元中的分配也随之改变。这与原子和力的分解相反。

图 7.5（b）显示了一个单元（中间的正方形）和包含可能在截止半径 rc 内的粒子的空间区域（阴影），这些粒子在给定的单元中。阴影区域通常被称为导入区域，因为给定单元必须导入位于该区域的粒子的位置来进行受力计算。请注意，并不是给定单元中的所有粒子都必须与导入区域中的所有粒子相互作用，特别是如果导入区域与截止半径相比很大的话。

```
Algorithm 4 Force decomposition time step, with permuted columns of force matrix
1: send positions of my assigned particles which are needed by other processors; receive 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: receivecolumnparticlepositionsneededbymyprocessor（thiscommunicationisgenerallywithprocessors the same processor column, e.g., processor 3 communicates with processors 3, 7, 11, 15）
```

```
3: （ifnonbondedcutoffsareused）determinewhichnonbondedforcesneedtobecomputed
4: computeforcesformyassignedparticles
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: updatepositions（integration）formyassignedparticles
```

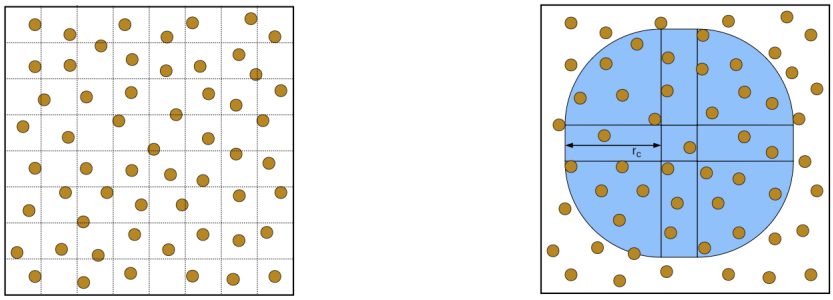



图 7.2: 空间分解，显示了二维计算盒中的粒子

算法 5 显示了每个处理器在一个时间步骤中的执行情况。我们假设在时间步骤开始时，每个处理器持有其单元中的粒子的位置。

为了利用牛顿第三定律，进口区域的形状可以减半。现在，每个处理器只计算其单元中的粒子的部分力，并需要接收来自其他处理器的力来计算这些粒子的总力。因此涉及到一个额外的通信步骤。我们把这个问题留给读者去解决修改后的导入区域和这种情况下的伪代码的细节。

在空间分解方法的实施中，每个单元与它的导入区域的粒子列表相关联，类似于 Verlet 邻居列表。与 Verlet 邻居列表一样，如果进口区域略有扩大，则不必在每个时间步骤中更新该列表。这使得进口区域列表可以在几个时间步中重复使用，这与粒子穿越扩展区域的宽度所需的时间是一致的。这与 Verlet 邻居列表完全类似。

总之，空间分解方法的主要优点是它们只需要在对应于附近粒子的处理器之间进行通信。空间分解方法的一个缺点是，对于非常多的处理器来说，与每个单元内包含的粒子数量相比，导入区域很大。

```
Algorithm 5 Spatial decomposition time step
1: sendpositionsneededbyotherprocessorsforparticlesintheirimportregions;receivepositionsfor particles in my import region
2: computeforcesformyassignedparticles
3: updatepositions（integration）formyassignedparticles
```

### 7.2.4 Neutral Territory Methods

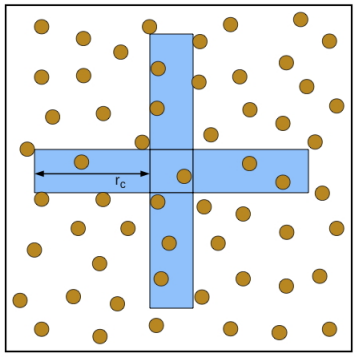
我们对 Neutral Territory Methods 方法的描述与 Shaw[182] 的描述非常接近。Neutral Territory Methods 可以被视为结合了空间分解和力分解的各个方面。为了并行化集成步骤，粒子根据空间的划分被分配到处理器。为了使力的计算并行化，每个处理器计算两组粒子之间的力，但这些粒子可能与被分配给处理器进行整合的粒子没有关系。由于这种额外的灵活性，Neutral Territory Methods 需要的通信量可能比空间分解方法少得多。

图 7.6 是一个二维模拟的 Neutral Territory Methods 的例子。在图中所示的方法中，给定的处理器被指派计算位于横条的粒子与位于竖条的粒子之间的力。因此，这两个区域构成了这个方法的导入区域。通过与图 7.6（b）的比较，这种中性领土方法的导入区域比相应的空间分解方法的导入区域小很多。当每个处理器所对应的单元的大小与截止半径相比很小时，这种优势就更大了。

在计算出力之后，给定的处理器将其计算出的力发送给需要这些力进行整合的处理器。因此，我们有了算法 6。

```
Algorithm 6 Neutral territory method time step
1: sendandreceiveparticlepositionscorrespondingtoimportregions
2: computeforcesassignedtothisprocessor
3: sendandreceiveforcesrequiredforintegration
4: updatepositions（integration）forparticlesassignedtothisprocessor
```

像其他方法一样，Neutral Territory Methods 的进口区域可以被修改以利用牛顿第三定律。我们参考 Shaw [182]，以了解更多的细节和中立领土方法在三维模拟中的图示。




## 7.3 并行快速傅立叶变换

许多计算长程力的方法的一个共同组成部分是用于解决三维网格上泊松方程的三维 FFT。傅里叶变换将泊松算子（称为拉普拉斯）对角化，在求解中需要一个正向和一个反向的 FFT 变换。考虑离散拉普拉斯算子 $L$（具有周期性边界条件）和 $\phi$ 在 $-L\phi - \rho$ 的解。让 $F$ 表示傅里叶变换。原问题相当于

$$-\left(FLF^{-1}\right)F\phi = F\rho$$

$$\phi = -F^{-1}\left(FLF^{-1}\right)^{-1}F\rho.$$

矩阵 $FLF^{-1}$ 是对角线。对 $\rho$ 进行正向傅里叶变换 $F$，然后用对角线矩阵的逆值对傅里叶空间分量进行缩放，最后，应用反傅里叶变换 $F^{-1}$，得到解 $\phi$。

对于现实中的蛋白质大小，通常使用大约 1 埃斯特朗的网格间距，从而导致一个按许多标准来说相当小的三维网格。64Œ64Œ64，或 128Œ128Œ128。并行计算通常不会应用于这种大小的问题，但必须使用并行计算，因为数据 已经分布在并行处理器之间（假设使用了空间分解）。三维 FFT 是通过沿三个维度依次计算一维 FFT 来计算的。对于 64Œ64Œ64 的网格大小，这就是 4096 个 64 维的一维 FFT。并行 FFT 计算通常受到通信的约束。FFT 的最佳并行化取决于变换的大小和计算机网络的结构。下面，我们首先描述平行一维 FFT 的一些概念，然后描述平行三维 FFT 的一些概念。对于目前致力于大型一维变换的并行化和高效计算（使用 SIMD 操作）的软件和研究，我们参考了 SPIRAL 和 FFTW 软件包。这些软件包使用自动调谐来生成对用户的计算机结构来说是有效的 FFT 代码。

### 7.3.1 并行一维 FFT

#### 7.3.1.1 无转置的一维 FFT

图 7.7 显示了 16 点 radix-2 十进频 FFT 算法的输入（左边）和输出（右边）之间的数据依赖关系（数据流图）。（图中未显示的是计算中可能需要的位反转变换）。图中还显示了计算在四个处理器之间的划分。在这种并行化中，初始数据不在处理器之间移动，但在计算过程中会发生通信。在图中所示的例子中，通信发生在前两个 FFT 阶段；最后两个阶段不涉及通信。当通信发生时，每个处理器正好与另外一个处理器进行通信。

#### 7.3.1.2 转置的一维 FFT

使用转置的方式来实现 FFT 计算的并行化是很常见的。图 7.8（a）显示了与图 7.7 相同的数据流图，但为了清晰起见，删除了水平线并增加了额外的索引标签。和以前一样，前两个 FFT 阶段是在没有通信的情况下进行的。然后，数据在各处理器之间进行转置。有了这个转置的数据布局，最后两个 FFT 阶段可以在没有通信的情况下进行。最终的数据不按原来的顺序排列；可能需要额外的转置，或者数据可以按这个转置的顺序使用。图 7.8（b）显示了转置前后指数在四个处理器中的分布情况。从这两个图中可以看出，前两个阶段的数据依赖性只涉及同一分区的指数。转置后的后两个阶段的分区情况也是如此。还要注意的是，转置前后的计算结构是相同的。

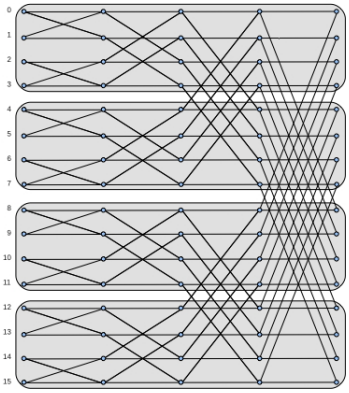
### 7.3.2 并行三维 FFT

#### 7.3.2.1 块状分解的三维 FFT

图 7.9（a）显示了当使用空间分解时，FFT 输入数据的块状分解，其大小为 8Œ8Œ8，分布在以 4Œ4Œ4 拓扑结构排列的 64 个处理器中。并行一维 FFT 算法可以在每个维度上应用。对于图中所示的例子，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）。在每个处理器中，如果穿越其中一个维度，数据是连续排列的，因此在其他两个维度的计算中，数据访问是分层的。分层数据访问可能很慢，因此在计算每个维度的 FFT 时，可能值得在每个处理器中重新排序数据。

#### 7.3.2.2 三维 FFT 与板块分解

图 7.9（b）是 4 个处理器情况下的板块分解图。每个处理器持有输入数据的一个或多个平面。如果输入数据已经分布在板块中，或者可以以这种方式重新分布，就可以使用这种分解。板块平面内的两个一维 FFT 不需要通信。其余的一维 FFT 需要通信，可以使用上述两种平行一维 FFT 的方法之一。板块分解的缺点是，对于大量的处理器来说，处理器的数量可能会超过三维 FFT 中沿任何一个维度的点的数量。另一个选择是下面的铅笔分解。

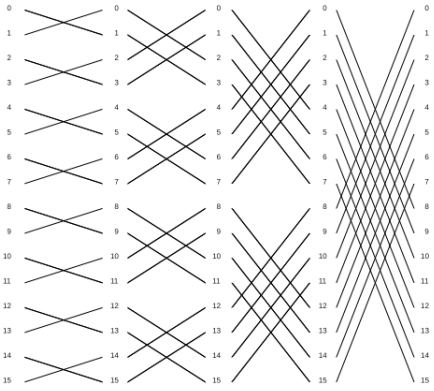


#### 7.3.2.3 三维 FFT 与铅笔式分解

图 7.9（c）显示了 16 个处理器的情况下的铅笔分解情况。每个处理器持有一个或多个输入数据的铅笔。如果原始输入数据像图 7.9（a）那样分布在块中，那么一排处理器之间的通信（在一个三维处理器网中）可以将数据分布到铅笔分解中。然后可以在没有通信的情况下执行一维 FFT。为了在另一个维度上执行一维 FFT，数据需要被重新分配到另一个维度的铅笔中。在整个三维 FFT 计算中，总共需要四个通信阶段。

## 7.4 分子动力学整合

为了对分子动力学中的常微分方程系统进行数值积分，需要采用特殊的方法，与第四章中研究的传统 ODE 求解器不同。这些特殊的方法被称为对称方法，在产生具有恒定能量的解方面比其他方法更好，例如，对于那些被称为哈密尔顿的系统（包括分子动力学中的系统）。当哈密顿系统在很长的时间间隔内以许多时间步长进行积分时，保留结构（如总能量）往往比方法的精度顺序更重要。在这一节中，我们激励了一些想法，并给出了 Störmer-Verlet 方法的一些细节，该方法足以用于简单的分子动力学模拟。



哈密顿系统是一类保存能量的动力系统，它可以被写成一种叫做哈密顿方程的形式。为了对称起见，考虑

一下简单的谐波振荡器

$$u'' = -u$$

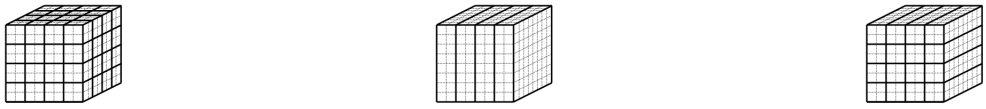
其中 $u$ 是单个粒子从平衡点的位移。这个方程可以模拟一个具有单位质量的粒子在一个具有单位弹簧常数的弹簧上。处于 $u$ 位置的粒子受到的力是 $-u$。这个系统看起来并不像一个分子动力学系统，但对说明几个观点很有用。

上述二阶方程可以写成一阶方程系统

$$q' = pp' = -q$$

其中 $q = u, p = u'$，这是经典力学中常用的符号。一般解决方案是

$$\begin{pmatrix} q \\ p \end{pmatrix} = \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix}$$



简谐振荡器的动能是 $p^2/2$，势能是 $q^2/2$（势能的负梯度是力，$-q$）。因此，总能量与 $q^2 + p^2$ 成正比。

现在考虑用三种方法解决一阶方程组，显式欧拉法、隐式欧拉法和一种叫做 Störmer-Verlet 的方法。初始条件是 $(q, p) \neq 1, 0$。我们使用 $h = 0.05$ 的时间步长，走 500 步。我们将 $q$ 和 $p$ 分别绘制在横轴和纵轴上（称为相位图）。如上所述，准确的解决方案是一个以原点为中心的单位圆。图 7.10 显示了这些解。对于显式欧拉，解决方案是向外螺旋式的，这意味着解决方案的位移和动量随时间而增加。隐式欧拉方法的情况则相反。总能量的图表将显示这两种情况下的能量分别增加和减少。当采取较小的时间步长或使用高阶方法时，解决方案会更好，但这些方法完全不适合在长时间内对共轭系统进行积分。图 7.10（c）显示了使用一种叫做 Störmer-Verlet 方法的对称方法的解。该解显示，$q^2 + p^2$ 的保存情况比其他两种方法要好得多。

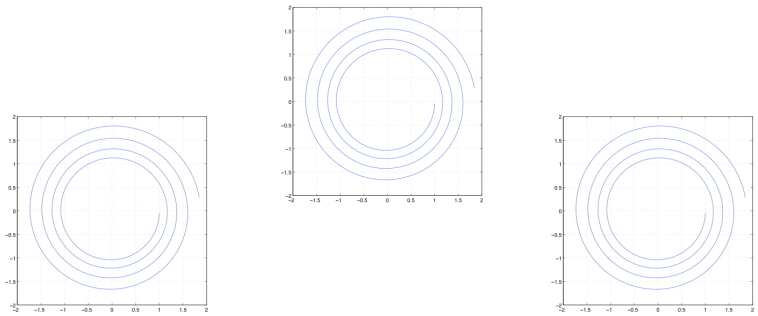


**图 7.3:** 初值（1,0）、时间步长 0.05、步长 500 的三种方法的简谐振荡器解的相位图。对于显式 Euler，解是向外旋的；对于隐式 Euler，解是向内旋的；总能量在 Störmer-Verlet 方法中保存得最好。我们推导出二阶方程的 Störmer-Verlet 方法

$$u'' = ft, u$$

通过简单地用有限差分近似值取代左手边

$$\frac{u_{k+1}-2u_k+u_{k-1}}{h^2} = f(t_k, u_k)$$

这可以重新排列，得到的方法是

$$u_{k+1} = 2u_k - u_{k-1} + h^2 f(t_k, u_k)$$

该公式可以等效地从泰勒级数中导出。该方法与线性多步骤方法类似，需要一些其他技术来提供该方法的初始步骤。该方法也是时间可逆的，因为如果 $k+1$ 和 $k-1$ 互换，公式是一样的。遗憾的是，要解释为什么这种方法是对称性的，已经超出了本介绍的范围。

上面写的方法有一些缺点，最严重的是小 $h^2$ 项的添加会受到灾难性的取消。因此，这个公式不应该以这种形式使用，已经开发了一些数学上等效的公式（可以从上面的公式推导出来）。

一个替代公式是跃迁法。

$$u_{k+1} = u_k + hv_{k+1/2}$$

$$v_{k+1/2} = v_{k-1/2} + hf\left(t_k, u_k\right)$$

其中 v 是第一导数（速度），与位移 u 相差半步。这个公式没有同样的四舍五入问题，而且还提供了速度，尽管它们需要与位移重新对中，以计算给定步骤的总能量。这对方程中的第二个基本上是一个有限差分公式。

Störmer-Verlet 方法的第三种形式是速度 Verlet 变体。

$$u_{k+1} = u_k + hv_k + \frac{h^2}{2} f\left(t_k, u_k\right)$$

$$v_{k+1} = v_k + \frac{h^2}{2}\left(f\left(t_k, u_k\right) + f\left(t_{k+1}, u_{k+1}\right)\right)$$

其中，现在速度是在与位移相同的点上计算的。这些算法中的每一种都可以被实现，从而只需要存储两组量（两个先前的位置，或者一个位置和一个速度）。Störmer-Verlet 方法的这些变体由于其简单性而受到欢迎，每一步只需要一个昂贵的力评估。高阶方法通常不实用。速度 Verlet 方案也是分子动力学的多时间步长算法的基础。在这些算法中，慢速变化的（典型的长程）力的评估频率较低，更新位置的频率也比快速变化的（典型的短程）力低。最后，许多最先进的分子动力学集成了一个经过修改的哈密尔顿系统，以控制模拟温度和压力。对于这些系统，已经开发出了复杂得多的对称性方法。

# 第 8 章 组合算法

内容提要

- 排序算法概要
- 奇偶交换排序
- 快速排序
- 基数排序
- 样本排序
- 双调排序
- 素数查找

本部分我们将简要考虑一些组合算法，如：排序算法以及埃拉托色尼筛选法寻找素数法等等。

在科学计算中，排序并不常见，人们通常在数据库（无论是金融数据库还是生物数据库）中考虑排序。当考虑如**自适应网格加密**（Adaptive Mesh Refinement，AMR）和其他对数据结构的改进中，排序才会凸显其重要之处。

在这一节中，我们将简要介绍一些基本的算法，以及其并行表达。更多的细节，见 [131] 和其中的参考文献。

## 8.1 排序算法概要

### 8.1.1 复杂度

众多排序算法以其计算复杂度来区分：即给定 $n$ 元数组，我们需要对它操作多少次能完成排序。

部分排序算法可以达到的最低复杂度为：$O(n\log n)$，例如快速排序（Quicksort）的最佳情况即为 $O(n\log n)$。但由于快速排序算法由于需要选择“基准元素”，如果每次选择的基准元素都是最差的，那么其时间复杂度可以达到 $O(n^2)$。

```
while the input array has length > 1 do
  Find a pivot element of intermediate size
  Split the array in two, based on the pivot
  Sort the two arrays.
\\Algorithm 2: The quicksort algorithm
```

另一方面，非常简单的**冒泡排序**（bubble sort）算法由于其静态结构，因此时间复杂度恒定。

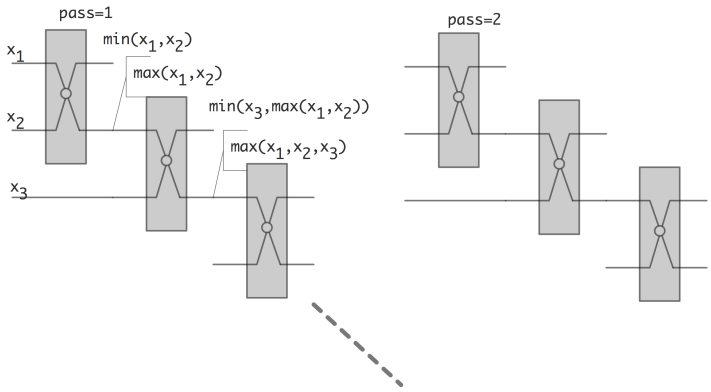
```
for pass from 1 to n - 1 do
  for e from 1 to n - pass do
    if elements e and e + 1 are ordered the wrong way then
      exchange them
\\Algorithm 3: The bubble sort algorithm
```

很容易看出，这个算法的复杂度为 $O(n^2)$：内循环做了 $t$ 的比较和最多 $t$ 的交换。从 1 到 $n-1$ 相加，可以得到大约 $n^2/2$ 的比较和最多相同数量的交换。

### 8.1.2 排序网络

上面我们看到，部分算法的复杂度不受输入数据的影响，而有些算法则并非如此。前者被称为**排序网络**（sorting network），可以认为排序网络是实现算法的专用硬件，其基本元件为比较、交换元素。它有两个输入 $x$，$y$；两个输出 $\max(xy)$ 与 $\min(xy)$。

下图展示了由比较和交换元素构成的冒泡排序。




下面我们将讨论排序网络中的一个例子：双调排序。

### 8.1.3 并行复杂度

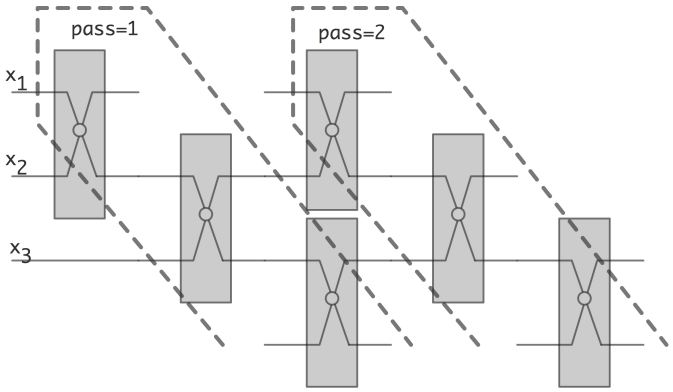
串行排序算法最理想的时间复杂度为 $O(N \log N)$，理想情况下，如果同时使用 $P = N$ 个处理器计算，我们希望并行排序的时间复杂度为 $O(\log N)$。如果并行时间大于这个值，我们将所有处理器上的操作总数定义为**串行复杂度**（sequential complexity）。

这相当于如果并行算法由一个进程执行。理想情况下，这与单进程算法的操作数相同，但如果线程数过大，该算法就会收到并行算法的额外时间开销惩罚。

例如，下面我们将看到，排序算法的串行复杂度通常为 $O(N \log_2 N)$。

## 8.2 奇偶交换排序

观察上图我们发现，pass 2 实际上可以在 pass 1 完成前开始，在给定的时间下，这实际上就是**奇偶交换排序**（odd-even transposition sort）。




奇偶交换排序是一种简单的并行排序算法，其主要优点是在线性面积的处理器上容易实现；缺点是效率并不理想。

该算法的一个步骤由两个子步骤组成：

- 偶数的处理器与它的右邻进行比较和交换；然后
- 奇数处理器与它的右邻进行比较和交换。

**定理 8.1 (分析)**

该算法经过 $N$ 步完成（其中每步由刚才给出的两个子步骤组成）。通过归纳法进行证明：在每个三联体 $2i, 2i+1, 2i+2$ 中，经过一个偶数和一个奇数步骤后，最大的元素将处于最右的位置。

在并行时间为 $N$ 的情况下，可以得到串行复杂度为 $N^2$。♡

✍ **练习 8.1** 讨论交换排序的速度和效率，我们对 $N$ 个处理器的 $P$ 个数字进行排序；设 $N = P$，即每个处理器都包含一个数字。用比较和交换操作表示执行时间。

1. 这个并行代码总共需要多少次比较和交换操作?
2. 该算法需要多少个串行步骤？什么是 $T_1, T_p, T_\infty, S_p, E_p$ 的 $N$ 数排序？平均并行量是多少?
3. 交换排序可以被认为是冒泡排序的并行实现。现在让 $T_1$ 指的是（顺序的）冒泡排序的执行时间。这对 $S_p$ 和 $E_p$ 有何影响?

## 8.3 快速排序

快速排序是一种递归算法，与冒泡排序不同，它的复杂度并不确定。该算法由两部分组成，是基于序列的排列。

```
Algorithm: Dutch National Flag ordering of an array
Input : An array of elements, and a 'pivot' value
Output: The input array with elements ordered as red-white-blue, where red elements are
larger than the pivot, white elements are equal to the pivot, and blue elements are less than the
    pivot
```

我们无需证明，这可以在 $O(n)$ 操作中完成。这样一来，快速排序就变成了

```
Algorithm: Quicksort
Input : An array of elements
Output: The input array, sorted
while The array is longer than one element do
pick an arbitrary value as pivot
apply the Dutch National Flag reordering to this array
  Quicksort( the blue elements )
  Quicksort( the red elements )
```

该算法的不确定性来源于基准元素的选择。最坏情况下，算法所选择的基准元素每次都是数组中唯一最小的元素，如此需要再次进行 $n-1$ 次递归操作。不难看出，此时的运行时间为 $O(n^2)$。另一方面，如果基准元素总是（接近）中位数，即大小适中的元素，那么递归调用的运行时间将大致相等，可以得到递归公式:

$$T_n = 2T_{n/2} + O(n)$$

复杂度为 $O(n \log n)$。现在考虑快速排序的并行实现。

### 8.3.1 在共享内存中进行快速排序

通过共享内存模型和递归调用的线程实现，可以将快速排序进行简单并行化，然而这种方法并不高效。

在一个长度为 $n$ 的数组中，如果有理想的基准元素，在算法的最后阶段会有 $n$ 个线程在活动。最理想的情况是，我们希望并行算法在 $O(\log n)$ 的时间内运行，但是在这里，时间被第一个线程对数组的初始重新排序所支配。

✍ **练习 8.2** 精确论证这个结论。这样并行化快速排序算法的总运行时间、速度提升和效率是多少？是否有办法使分割数组的效率更高？答案是肯定的，关键是要使用并行的前缀操作，见附录 20。如果数组中的数值是 $x_1, ..., x_n$，我们使用并行前缀来计算有多少元素小于基准 $\pi$。

$$X_i = \#\left\{x_j : j < i \wedge x_j < \pi\right\}$$

有了这个方法，如果一个处理器看了 $x_i$，并且 $x_i$ 小于基准，那么它需要被移到数组中的 $X_{i+1}$ 的位置，那里的元素是根据基准分割的。同样地，我们可以知道有多少元素大于基准元素，并相应地移动这些元素。

这表明每一个基准步骤可以在 $O(\log n)$ 的时间内完成，由于排序算法有 $\log n$ 的步骤，整个算法的运行时间为 $O((\log n)^2)$。快速排序算法的串行复杂度 $(\log_2 N)^2$。

### 8.3.2 超立方体上的快速排序

从上一节可以看出，为了使快速排序算法有效地并行化，我们也需要使 Dutch National Flag reordering 并行化。让我们假设数组已经被划分到维度为 $p$ 的超立方体的 $d$ 个处理器上（意味着 $p = 2d$）。

在并行算法的第一步，我们选择一个基准元素，并将其广播给所有处理器。然后，所有处理器将在其本地数据上独立地应用重新排序。

为了将第一层中的红色和蓝色元素聚集在一起，现在每个处理器都与一个二进制地址相同的处理器配对，但最重要的一位除外。在每一对中，蓝色元素被送到在该位上有 1 值的处理器；红色元素被送到在该位上有 0 值的处理器。

经过这次交换（这是本地的，因此是完全并行的），地址为 1xx 的处理器拥有所有的红色元素，地址为 0xx 的处理器拥有所有的蓝色元素。前面的步骤现在可以在子库中重复进行。

这种算法在每一步都能保持所有的处理器工作；但是，如果选择的基准元素远离中位数，就容易出现负载不平衡的情况。此外，在排序过程中，这种负载不均衡也不会减少。

### 8.3.3 在一般的并行处理器上进行快速排序

快速排序也可以在任何具有处理器线性排序的并行机器上进行。我们首先假设每个处理器正好持有一个数组元素，而且，由于标志重排，排序将总是涉及一组连续的处理器。

一个数组（或递归调用中的子数组）的并行快速排序开始于在存储数组的处理器上构建一个二进制树。选择一个基准元素值并在树上广播。然后，树状结构被用来在每个处理器上计算左、右子树中有多少元素小于、等于或大于基准元素值。

有了这些信息，根处理器可以计算出红色/白色/蓝色区域将被储存在哪里。这个信息被下发到树上，每个子树都计算出其子树中元素的目标位置。

如果我们忽略了网络竞争，现在可以在单位时间内完成重新排序，因为每个处理器最多发送一个元素。这意味着每个阶段只需要花费时间来计算子树中蓝色和红色元素的数量，在顶层是 $O(\log n)$，在下层是 $O(\log n/2)$，依此类推。这使得速度几乎完美。

## 8.4 基数排序

大多数排序算法都是基于比较完整的项值。相比之下，**基数排序**（radix sort）在数字的位数上做一些部分排序阶段。每个数字值都有一个"bin " 被分配，数字被移入这些 bin。将这些仓连接起来就得到了一个部分排序的数组，通过移动数字的位置，数组的排序会越来越多。

考虑一个最多两位数的例子，所以需要两个阶段。

| array | 25 | 52 | 71 | 12 |
|---|---|---|---|---|
| last digit | 5 | 2 | 1 | 2 |
| | (only | bins | 1, 2, 5receive | data) |
| sorted | 71 | 52 | 12 | 25 |
| next digit | 7 | 5 | 1 | 2 |
| sorted | 12 | 25 | 52 | 71 |

重要的是，一个阶段的部分排序在下一个阶段得到了保留。归纳起来，我们最后会得到一个完全排序的数组。

### 8.4.1 并行基数排序

分布式内存排序算法已经对数据进行了明显的" 分档"，所以基数排序的并行实现是基于使用 ，即进程的数量作为基数。

我们用两个处理器上的例子来说明这一点，也就是说，我们看的是数值的二进制表示。

| | proc0 | | proc1 | | |
|---|---|---|---|---|---|
| array | 2 | 5 | 7 | 1 | |
| binary | 010 | 101 | 111 | 001 | |
| stage 1: sort by least significant bit | | | | | |
| last digit | 0 | 1 | 1 | 1 | |
| | (this serves as bin number) | | | | |
| sorted | 010 | | 101 | 111 | 001 |
| stage 2: sort by middle bit | | | | | |
| next digit | 1 | | 0 | 1 | 0 |
| | (this serves as bin number) | | | | |
| sorted | 101 | 001 | 010 | 111 | |
| stage 3: sort by most significant bit | | | | | |
| next digit | 1 | 0 | 0 | 1 | |
| | (this serves as bin number) | | | | |
| sorted | 001 | 010 | 101 | 111 | |
| decimal | 1 | 2 | 5 | 7 | |

分析：

- 确定所考虑的数字，以及确定有多少本地值进入哪个仓都是本地操作。我们可以将其视为一个连接矩阵 $C$，其中 $C[i,j]$ 是进程 $i$ 将发送至进程 $j$ 的数据量。每个进程拥有其在该矩阵中的一行。
- 为了在洗牌中接收数据，每个进程都需要知道将从其他每个进程接收多少数据。这需要对连接矩阵进行" 转置"。在 MPI 术语中，这是一个全对全操作。MPI_Alltoall。
- 在这之后，实际的数据可以在另一个 all-to-all 操作中进行洗牌。然而，由于每个 $i,j$ 组合的数量不同，我们需要 MPI_Alltoallv 程序。

### 8.4.2 有效小数的基数排序

完全可以让基数排序的阶段从最重要的数字到最小的数字进行，而不是相反。按顺序，这不会改变任何重要的东西。

然而

- 我们不是完全洗牌，而是在越来越小的子集里洗牌，所以即使是串行算法也会增加**空间局部性**（spatial locality）。
- 一个共享内存的并行版本将显示出类似的局部性改进。
- 一个 MPI 版本不再需要全对全的操作。如果我们认识到每一个下一阶段都将在一个子集的进程内，我们可以使用通信器分割机制。

✍ **练习 8.3** 上面的位置性论证做得有点手忙脚乱。论证该算法可以以广度优先和深度优先的方式进行。讨论一下这种区别与位置性论证的关系。

## 8.5 样本排序

我们在快速排序算法（第 8.3 节）中看到，有可能在排序算法中使用概率元素。我们可以将快速排序法中挑选一个基准的想法扩展为挑选有多少个处理器的基准元素。这不是对元素的一分为二，而是将元素分成与处理器数量一样多的" 桶"。然后，每个处理器对其元素进行完全并行的排序。

$Input: p: the number of processors, N: the numbers of elements to sort; \{x_i\}_{i<N}$ the elements to sort Let $x_0 = b_0 < b_1 < \cdots < b_{p-}$

显然，如果不仔细选择桶的话，这种算法会出现严重的负载不均衡。随机抽取 $p$ 元素可能还不够好；相反，需要对元素进行某种形式的抽样。相应地，这种算法被称为**样本排序**（Samplesort）[17]。

虽然桶排序一旦分配即为完全并行，但仍有一些问题：首先，采样是该算法的一个串行瓶颈。另外，将桶分配给处理器的步骤本质上是一个全对全的操作。

为了对此进行分析，假设有 $P$ 个进程首先作为映射器，然后作为规约器发挥作用。让 为数据点的数量，并定义一个块大小 $b \equiv N/P$。该算法的处理步骤的成本是：

- 局部确定每个元素的 bin，需要时间 $O(b)$；以及
- 局部排序，我们可以假设其最佳复杂度为 $b \log b$。

然而，洗牌步骤是不简单的。除非数据被部分预排序，否则我们可以预期洗牌是一个完全的全对全，其时间复杂度为 $P\alpha + b\beta$。另外，这可能会成为一个网络瓶颈。请注意，在超立方体上的快速排序法中，从来没有出现过对导线的争夺。

✍ **练习 8.4** 论证对于少量的进程，$P << N$，这个算法有完美的加速，串行复杂度（见上文）为 $N \log N$。

将这种算法与双调排序等排序网络相比较，这种排序算法看起来相当简单：它只有一个步骤的网络。前面的问题认为，在" 理想扩展"（工作可以增加，同时保持处理器数量不变）中，串行复杂性与串行算法相同。然而，在我们按比例增加功和处理器的弱缩放分析中，串行复杂度要差很多。

✍ **练习 8.5** 考虑我们同时扩展 $NP$ 的情况，保持 $b$ 不变。论证在这种情况下，洗牌步骤在算法中引入了一个 $N^2$ 项。

### 8.5.1 通过 MapReduce 排序

terasort 基准涉及在一个基于文件的大型数据集上进行排序。因此，它在某种程度上是大数据系统的一个标准。特别是，MapReduce 是一个主要的候选者，见 http://perspectives.mvdirona.com/2008/07/hadoop-wins-terasort/。

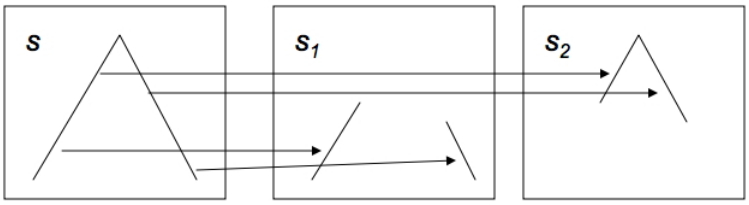
使用 MapReduce，该算法的过程如下。

- 通过抽样或优先信息确定一组键值：键值是这样的，预计每对键值之间的记录数量大致相等。间隔的数量等于规约器进程的数量。
- 然后，映射器进程产生键/值对，其中键是区间或规约器编号。
- 然后，规约器进程执行局部排序。

我们看到，除了术语的变化之外，这实际上是 sampleort。

## 8.6 双调排序

为了激励双调排序，假设一个序列 $x =< x_0, ....., x_{n-1} >$ 由升序和降序组成。现在把这个序列分成两个等长的子序列，定义为：



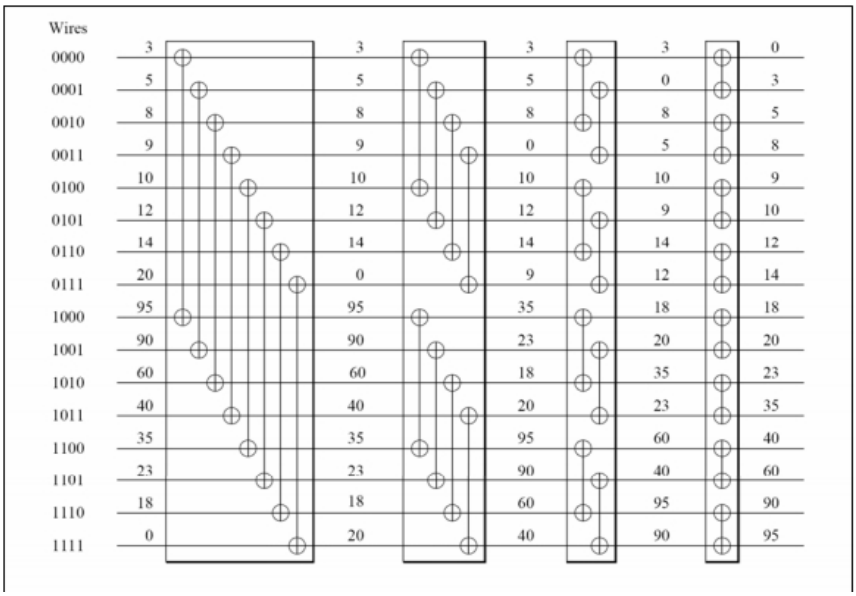

$$s_1 = \langle \min\{x_0, x_{n/2}\}, \ldots \min\{x_{n/2-1}, x_{n-1}\}$$
$$s_2 = \langle \max\{x_0, x_{n/2}\}, \ldots \max\{x_{n/2-1}, x_{n-1}\}$$

从图中不难看出，$s_1, s_2$ 又是具有升序和降序部分的序列。此外，$s_1$ 的所有元素都小于 $s_2$ 的所有元素。

我们称 (8.1) 为升序双调拣器，因为第二个子序列中的元素比第一个子序列中的元素大。同样地，我们可以通过颠倒最大和最小的角色来构造一个降序分拣器。

不难想象这是排序算法中的一个步骤：从这个形式的序列开始，递归应用公式（8.1）得到一个排序的序列。图 8.4 显示了 4 个位元排序器，分别在 8,4,2,1 的距离上，如何对一个长度为 16 的序列进行排序。

双调序列的实际定义要稍微复杂一些。如果一个序列由一个升序部分和一个降序部分组成，或者是这样一个序列的循环变异，那么它就是比特序列。




✍ **练习 8.6** 证明根据公式 (8.1) 分割一个位数序列可以得到两个位数序列。

所以问题是如何得到一个比特尼序列。答案是使用越来越大的双调网络。两个元素的位数排序给你一个排序的序列。如果你有两个长度为 2 的序列，一个向上排序，另一个向下排序，这就是一个双调序列。因此，这个长度为四的序列可以通过两个比特级的步骤进行排序。

两个长度为四的排序形成一个长度为四的比特序列。可以用三个比特步骤进行排序；等等。

从这个描述中你可以看到，你要用 $\log_2 N$ 个阶段来排序 $N$ 个元素，其中第 $i$ 个阶段的长度是 $\log_2 i$。这使得位数排序的总顺序复杂性为 $(\log_2 N)^2$。

图 8.5 中的操作序列被称为排序网络，由简单的比较-交换元素构成。元素组成的排序网络。与快速排序的情况不同，它不依赖于数据元素的值。

## 8.7 素数查找

埃拉托色尼筛选法是一种非常古老的寻找素数的方法。在某种程度上，它仍然是许多现代方法的基础。

写下从 2 到某个上限 $N$ 的所有自然数，我们要把这些数字标记为素数或绝对非素数。所有的数字最初都是没有标记的。

- 第一个没有标记的数是 2：把它标记为质数，并把它的所有倍数标记为非质数。
- 第一个没有标记的数字是 3：把它标记为质数，并把它的所有倍数标记为非质数。
- 下一个数字是 4，但它已经被标记过了，所以标记 5 和它的倍数。
- 下一个数字是 6，但它已经被标记过了，所以标记 7 和它的倍数。
- 数字 8，9，10 已经被标记过了，所以继续标记 11。
- 依此类推。

# 第 9 章 图论分析

内容提要

- 传统图算法
- 并行化
- 现实世界的图
- 超文本算法
- 大型计算图论

科学计算中的各种问题都可以被表述为图问题（关于图论的介绍见附录 18）；例如，我们已经遇到了负载平衡（第 2.10.4 节）和寻找独立集（第 6.8.2 节）的问题。

许多传统的图算法不能立即适用，或者至少不能有效地适用，因为图往往是分布式的，而传统的图理论假设了对整个图的全局知识。此外，图论通常关注的是寻找最佳算法，而这通常不是一种并行的算法。因此，并行图算法本身就是一个研究领域。

最近，在科学计算中出现了新的图计算类型。这里的图不再是工具，而是研究对象本身。例如，万维网或 Facebook 的社交图，或一个生物体内所有可能的蛋白质相互作用的图。

由于这个原因，组合计算科学正在成为一门独立的学科。在这一节中，我们看一下图分析：大型图的计算。我们首先讨论一些经典的算法，但我们在代数框架中给出这些算法，这将使并行实现更加容易。

## 9.1 传统图算法

我们首先看一下几个" 经典" 的图算法，并讨论如何以并行方式实现它们。图和稀疏矩阵之间的联系（见附录 18）在这里至关重要：许多图算法具有稀疏矩阵-向量乘法的结构。

### 9.1.1 最短路径算法

有几种类型的最短路径算法。例如，在单源最短路径算法中，人们想知道从一个给定节点到任何其他节点的最短路径。在全对最短路径算法中，人们想知道任何两个节点之间的距离。计算实际路径不是这些算法的一部分；但是，通常很容易包括一些信息，通过这些信息可以在以后重建路径。

我们从一个简单的算法开始：寻找非加权图中的单源最短路径。用广度优先搜索（BFS）来做这件事很简单。

```
Input : A graph, and a starting node s
Output: A function d(v) that measures the distance from s to v
Let s be given, and set d(s) = 0
Initialize the finished set as  = {s}
Set c = 1
while not finished do
  Let V the neighbours of  that are not themselves in
  if V =\phi then
    We' re done
   else
     Set d(v)=c+1 for all v \inV. \leftarrowV
     Increase c \leftarrow c + 1
```

这种制定算法的方式在理论上是很有用的：通常可以制定一个谓词，在 while 循环的每一次迭代中都是真的。这样，你就可以证明该算法终止了，并且它计算了你想要计算的东西。在传统的处理器上，这的确是你对图算法的编程方式。然而，现在的图，如来自 Facebook 的图，可能是巨大的，而你想对你的图算法进行并行编程。

在这种情况下，传统的表述方式就显得不够了。

- 它们通常基于队列，节点被添加或减去；这意味着存在某种形式的共享内存。
- 关于节点和邻居的声明是在不知道这些是否满足任何形式的空间定位的情况下做出的；如果一个节点被触及一次以上，就不能保证时间上的定位。

### 9.1.2 Floyd-Warshall 最短路径

Floyd-Warshall 算法是一个全对最短路径算法的例子。它是一种动态编程算法，是基于逐步增加路径的中间节点集。具体来说，在步骤 k 中，所有的路径 $u \to v$ 都被考虑，它们的中间节点在集合 $k = \{0, ....\Delta k(u, v)\}$ 被定义为从 $u$ 到 $v$ 的路径长度，其中所有中间节点都在 $k$。最初，这意味着只考虑图边，当 $k \equiv |V|$ 时，我们已经考虑了所有可能的路径，算法完成。

计算的步骤是

$$\Delta_{k+1}(u, v) = \min\{\Delta_k(u, v), \Delta_k(u, k) + \Delta_k(k, v)\}$$

也就是说，对距离 $\Delta(u, v)$ 的第 $k$ 个估计值是旧的估计值和一条新的路径的最小值，现在我们正在考虑节点 $k$ 的可行性。这条路径是由 $u \to k$ 和 $k \to v$ 两条路径连接而成的。

用算法来写:

```
for k from zero to |V| do
  for all nodes u, v do
    \Delta uv \leftarrow f(\Delta uv,\Delta uk,\Delta kv)
```

我们看到这个算法的结构与高斯消除法相似，只是在那里，内循环是" 对于所有的 $uv > k$"。

在第 5.4.3 节中，你看到稀疏矩阵的因式分解会导致填充，所以这里也会出现同样的问题。这需要灵活的数据结构，而且这个问题在并行时变得更加严重，见第 9.5 节。

代数上:

```
for k from zero to |V| do
  D \leftarrow D.min[D(, k) min + D(k, )]
```

Floyd-Warshall 算法并不告诉你实际的路径。在上面的距离计算过程中，存储这些路径的成本很高，无论是时间还是内存。一个更简单的解决方案是可能的：我们存储第二个矩阵 n(i, j)，它有 i 和 j 之间路径的最高节点号。

✍ **练习 9.1** 将 $n(i, j)$ 的计算包含在 Floyd-Warshall 算法中，并描述在已知 $d(,)$ 和 $n(,)$ 的情况下，如何利用它来寻找 $i$ 和 $j$ 之间的最短路径。

### 9.1.3 生成树

在一个无向图 $G =< V, E >$ 中，如果 $T \subset E$ 是连接和无环的，我们称之为" 树"。如果它的边包含所有的顶点，则被称为生成树。如果图的边权重为 $w_i i \in E$，则树的权重为 $\sum_{e \in T} w_e$，如果树的权重最小，我们称它是最小生成树。一个最小生成树不需要是唯一的。

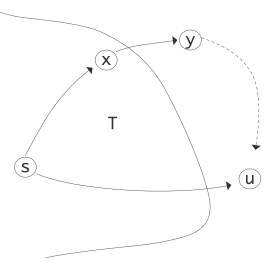
Prim 算法是 Dijkstra 最短路径算法的一个小变种，它从一个根开始计算生成树。根的路径长度为零，而所有其他节点的路径长度为无穷大。在每一步中，所有与已知树节点相连的节点都会被考虑，并更新它们的最佳已知路径长度。

```
for all vertices v do l(v) \leftarrow \infty
l(s) \leftarrow 0
Q \leftarrow V - {s} and T \leftarrow {s} whileQ \phi do
let u be the element in Q with minimal l(u) value remove u from Q, and add it to T
forv \inQwith(u,v)\in E do
```

```
if l(u) + wuv < l(v) then Set l(v) \leftarrow l(u) + wuv
```

**定理 9.1 (分析)**

上述算法计算出每个节点到根节点的最短距离。




**证明** 本算法正确性的关键在于我们选择 $u$ 为最小 $l(u)$ 值。把到顶点的真正最短路径长度称为 $Lv$。由于我们从一个无限大的 $l$ 值开始，并且只减少它，我们总是有 $L(v) \leqslant l(v)$。

我们的归纳假设是，在算法的任何阶段，对于当前租用的 $T$ 中的节点，路径长度的确定是正确的。

$$u \in T \Rightarrow L(u) = \ell(u)$$

当树只由根 $s$ 组成时，这显然成立。现在我们需要证明归纳步骤：如果对于当前树中的所有节点，路径长度都是正确的，那么我们也将得到 $L(u) = l(u)$。假设这不是真的，有另一条路径更短。这条路径需要经过目前不在 $T$ 中的某个节点 $y$；这在图 9.1 中得到说明。假设 $x$ 是 $T$ 中的节点，位于 $y$ 之前的据称最短路径。现在我们有 $l(u) > L(u)$，因为我们还没有正确的路径长度，并且 $L(u) > L(x) + w$，因为在 $y$ 和 $u$ 之间至少有一条边（有正权）。但 $x \in T$，所以 $L(x) = l(x)$，$L(x) + y = l(x) + y$。现在我们注意到，当 $x$ 被添加到 $T$ 时，它的邻居被更新，所以 $l(y)$ 是 $lx + y$ 或更少。将这些不等式，我们发现

$$\ell(y) < \ell(x) + w_{xy} = L(x) + w_{xy} < L(u) < \ell(u)$$

这与我们选择 $u$ 为最小 $l$ 值的事实相矛盾。

为了并行化这个算法，我们观察到内循环是独立的，因此可以并行化。然而，外循环有一个选择，即最小化一个函数值。计算这个选择是一个减法运算，随后它需要被广播。这种策略使得顺序时间等于 $d \log P$，其中 $d$ 是生成树的深度。

在单个处理器上，寻找数组中的最小值是一个 $O(N)$ 的操作，但通过使用优先级队列，这可以减少到 $O(\log N)$。对于生成树算法的并行版本，相应的项是 $O(\log(N/P))$，还不包括减少的 $O(\log P)$ 成本。

### 9.1.4 图的切割

有时，你可能想对一个图进行分区，例如出于并行处理的目的。如果这是通过划分顶点来实现的，那么你就是在切割边，这就是为什么这被称为顶点切割分区。对于什么是好的顶点切割，有各种标准。例如，你希望切割的部分大小大致相等，以平衡平行工作。由于顶点通常对应于通信，你希望顶点的数量（或在加权图中它们的权重之和）要小。图 Laplacian（第 18.6.1 节）是这方面的一个流行算法。

图的切割的另一个例子是二方图的情况：一个有两类节点的图，而且只有从一类到另一类的边。例如，这样的图可以模拟一个人口和一组属性：边表示一个人有某种兴趣。现在你可以做一个边的切割，将边的集合进行分割。这可以给你一些具有类似兴趣的人的集合。这个问题很有意思，比如说，用于定位在线广告。

## 9.2 并行化

许多图的算法，如第 9.1 节中的算法，并不容易并行化。这里有一些考虑。

首先，与许多其他算法不同，很难针对最外层的循环层，因为这通常是一个"while " 循环，使得并行停止测

试成为必要。另一方面，通常有一些宏观步骤是连续的，但其中的一些变量是独立考虑的。因此，确实存在着可以利用的并行性。

图算法中的独立工作是一种有趣的结构。虽然我们可以确定"for all " 循环，它是并行化的候选者，但这些循环与我们以前看到的不同。

- 传统的公式往往具有逐步建立和删除的变量集的特征。这可以通过使用共享数据结构和任务队列来实现，但这限制了某种形式的共享内存的实现。
- 接下来，虽然每个迭代操作都是独立的，但其运行的动力学原理意味着将数据元素分配给处理器是很棘手的。固定分配可能会导致很多空闲时间，但动态分配会带来很大的开销。
- 在动态任务分配的情况下，算法将几乎没有空间或时间上的定位。

由于这些原因，线性代数的表述可能是比较好的。一旦考虑到分布式内存，我们肯定需要这种方法，但即使在多核架构上，它也能为鼓励局部性而付出代价。

在第 6.5 节中，我们讨论了稀疏矩阵-向量乘积的并行评估问题。由于稀疏性，只有按块行或块列进行划分才有意义。实际上，我们让分区由问题变量之一来决定。这也是唯一对单源最短路径算法有意义的策略。

✍ **练习 9.2** 你能为将并行化建立在向量的分布上做一个先验的论证吗？这个操作涉及多少数据，多少工作，多少个顺序步骤？

### 9.2.1 策略

下面是三种图算法的并行化方法，按明显程度递减，按可扩展性递增。(另见 [118])。

#### 9.2.1.1 动态规划

许多图算法建立了一个要处理的顶点的数据结构 $V$；他们执行一个包含循环的超步序列

```
for all v in V:
    // do something with v
```

如果对一个顶点 $v$ 的处理不影响 $V$ 本身，那么这个循环是并行的，可以通过动态调度来执行。

这实际上是将数据动态分配给处理元素。在这个意义上，它是高效的，没有任何处理元素会与不需要处理的数据元素发生冲突，因此计算机的所有处理能力都得到了充分的利用。另一方面，动态分配带有操作系统的开销，而且会导致大量的数据传输，因为顶点不可能在处理元素的本地内存（如缓存）中。

另一个问题是，这个循环可能看起来像

```
for all v in V:
    for all neighbours u of v:
      // update something on u
```

现在可能发生的情况是，两个节点 $v_1$ 和 $v_2$ 都在更新一个共享的邻居 $u$，这个冲突需要通过缓存一致性来解决。这带来了延迟上的损失，甚至可能需要使用锁，这带来了操作系统上的损失。

#### 9.2.1.2 线性代数解释

现在我们将表明，图算法通常可以被视为稀疏矩阵算法，这意味着我们可以应用我们为这些算法开发的所有概念和分析。

如果 $G$ 是图的邻接矩阵，我们也可以将最短路径算法类似于一系列矩阵-向量乘法（见附录 18.5.4 节）。让 $x$ 是列出与源的距离的向量，也就是说，$x_i$ 是节点 $i$ 与源的距离。对于 $i$ 的任何邻居 $j$，到源的距离是 $x_i + G_{ij}$，

除非已经知道更短的距离。换句话说，我们可以定义

$$y^t = x^t G \equiv \forall_i : y_j \min\{x_j, \min_{i:G_{ij}\neq 0} \{x_i + 1\}\}$$

而上述 while 循环的迭代对应于此定义下的后续矩阵-向量积。

这种算法之所以有效，是因为我们可以在第一次访问时将 $d(v)$ 设置为其最终值：这恰恰发生在等于路径长度的外循环次数之后。内循环执行的总次数等于图中边的数量。加权图就比较麻烦了，因为一个有更多阶段的路径实际上可以用阶段的权重之和来衡量更短。下面是贝尔曼-福特算法。

```
Let s be given, and set d(s) = 0 Set d(v) = \infty for all other nodes v for |E | - 1 times do
for all edges e = (u,v) do
Relax: if d(u) + wuv < d(v) then
Set d(v) \leftarrow d(u) + wuv
```

这个算法是正确的，因为对于一个给定的节点 $u$，在外迭代的 $k$ 步之后，它已经考虑了所有路径 $s \to u$ 的 $k$ 阶段。

✍ **练习 9.3** 这个算法的复杂度是多少？如果你对图形有一定的了解，外循环的长度是否可以减少？

我们可以再次将其写成一系列的矩阵-向量积，如果我们将积定义为

$$y^t = x^t G \equiv \forall_i : y_j \min\{x_j, \min_{i:G_{ij}\neq 0} \{x_i + g_{ij}\}\}$$

这与上述基础基本相同：到 j 的最小距离是已经确定的距离的最小值，或者到任何节点 i 的最小距离加上过渡 $g_{ij}$。

### 9.2.2 全对算法的并行化

在上面的单源最短路径算法中，我们没有太多选择，只能通过分布向量而不是矩阵来实现并行化。这种类型的分布在这里也是可能的，它对应于 $D(,)$ 量的一维分布。

✍ **练习 9.4** 画出该算法变体的并行实现。证明每个 $k$ 次迭代都涉及到以处理器 $k$ 为根的广播。

然而，这种方法遇到了与使用一维矩阵分布的矩阵-向量乘积相同的缩放问题；见 6.2.2 节。因此，我们需要使用二维分布。

✍ **练习 9.5** 做缩放分析。在内存不变的弱缩放情况下，渐近效率是多少？

✍ **练习 9.6** 使用二维分布的 $D(,)$ 量勾画 Floyd-Warshall 算法。

✍ **练习 9.7** 并行的 Floyd-Warshall 算法对零点进行了相当多的操作，当然是在早期阶段。你能设计一种算法来避免这些操作吗？

### 9.2.3 分割

传统的图划分算法 [146] 并不是简单的可并行化。相反，有两种可能的方法。

- 使用图 Laplacian；第 18.6.1 节。
- 使用多层次的方法：
  1. 分割图的粗化版本。

2. 逐步解除图的粗化，调整分区。

## 9.3 现实世界的图

在诸如第 4.2.3 节的讨论中，你已经看到了 PDEs 的离散化是如何导致具有图的方面的计算问题的。这种图的特性使它们适合于某些类型的问题。例如，使用 FDMs 或 FEMs 对二维或三维物体进行建模，会导致每个节点只与几个邻居相连的图形。这使得它很容易找到分离器，这反过来又允许像嵌套剖分这样的解决方法；见 6.8.1 节。

然而，有一些应用的计算密集型图问题看起来并不像 FEM 图。我们将简要地看一下全球网络的例子，以及像谷歌的 PageRank 这样试图寻找权威节点的算法。

现在，我们将称这种图为随机图，尽管这个术语也有技术含义 [61]。

### 9.3.1 随机图的性质

我们在本课程的大部分内容中看到的图形，其属性源于它们是我们三维世界中物体的模型。因此，两个节点之间的典型距离通常是 $O(N^{1/3})$，其中 $N$ 是节点的数量。随机图的表现并非如此：它们通常具有小世界的特性，典型的距离是 $O(\log N)$。一个著名的例子是电影演员和他们在同一电影中出现的联系图：根据" 六度分隔"，在这个图中没有两个演员的距离超过六。在图形方面，这意味着图形的直径是六。

小世界图还具有其他特性，例如存在小团体（尽管这些特性在高阶有限元问题中也有）和枢纽：高度的节点。这导致了如下的影响：在这样的图中删除一个随机的节点不会对最短路径产生很大的影响。

✍ **练习 9.8** 考虑机场图和它们之间存在的航线。如果只有枢纽和非枢纽，论证删除一个非枢纽对其他机场之间的最短路径没有影响。另一方面，考虑到节点在二维网格中的排序，并删除一个任意的节点。有多少条最短路径会受到影响?

## 9.4 超文本算法

有几种基于线性代数的算法用于衡量网站的重要性 [139]。我们将简要地定义几个算法并讨论其计算意义。

### 9.4.1 HITS

在 HITS（Hypertext-Induced Text Search）算法中，网站有一个衡量它指向多少其他网站的枢纽得分，以及一个衡量有多少网站指向它的权威得分。为了计算这些分数，我们定义了一个发生率矩阵 $L$，其中

$$L_{ij} = \begin{cases} 1 & \text{document } i \text{ points to document } j \\ 0 & \text{otherwise} \end{cases}$$

权威分数 $x$ 被定义为所有指向 $i$ 的中心分数 $y_j$ 的总和，反之亦然。因此

$$x = L^t y \quad y = Lx$$

或 $x = LL^t x$，$y = L^t L y$，表明这是一个特征值问题。我们需要的特征向量只有非负项；这被称为非负矩阵的 Perron 向量，见附录 13.4。佩伦向量是通过幂方法计算的；见 13.3 节。

一个实用的搜索策略是。

- 找到所有包含搜索词的文档。
- 建立这些文档的子图，以及可能的与之相关的一到两层的文档。
- 计算这些文档的权威性和枢纽得分，并将其作为一个有序的列表呈现给用户。

### 9.4.2 PageRank

PageRank[166] 的基本思想与 HITS 相似：它的模型是" 如果用户不断点击页面上某种程度上最理想的链接，那么总体上最理想的链接集合是什么"。这是通过定义一个网页的排名是所有连接到它的网页的排名之和来建模的。该算法通常是以迭代的方式表述

```
while Not converged do for all pages i do
ranki \leftarrow \epsilon+ (1 - \epsilon) \sumj  connectedj\rightarrowi rankj
```

其中的排名是这个算法的固定点。$\varepsilon$ 项解决了这样一个问题：如果一个页面没有出站链接，那么一个会在那里停留的用户就不会离开。

✍ **练习 9.9** 论证这个算法可以用两种不同的方式来解释，大体上响应雅可比和高斯-赛德尔迭代法；5.5.3 节。

✍ **练习 9.10** 在 PageRank 算法中，每个页面都给它所连接的页面" 提供排名"。给出这个变体的伪代码。说明它对应于一个列的矩阵向量乘积，而不是上述表述的行。在共享内存并行中实现这个方法会有什么问题?

对于这个方法的分析，包括它是否收敛的问题，最好是完全用线性代数的术语来表述。我们再次定义一个连接矩阵

$$M_{ij} = \begin{cases} 1 & \text{if page } j \text{ links to } i \\ 0 & \text{otherwise} \end{cases}$$

$e = (1, ..., 1)$，向量 $d^t = e^t M$ 计算一个页面上有多少个链接。$d_i$ 是页面 $i$ 上的链接数量。我们构建一个对角矩阵 $D = diag(d_1, ...)$ 我们将 $M$ 归一为 $T = MD^{-1}$。

现在 $T$ 的列和（即任何一列的元素之和）都是 1，我们可以表示为 $e^t T = e^t$ 其中 $e^t = 1...1$。这样的矩阵被称为随机矩阵。它有如下解释。

如果 $p$ 是一个概率向量，即 $p_i$ 是用户正在查看页面 $i$ 的概率，那么 $T_p$ 是用户点击了一个随机链接后的概率向量。

✍ **练习 9.11** 从数学上讲，概率向量的特点是其元素之和为 1。说明随机矩阵与概率向量的乘积确实又是一个概率向量。

上面制定的 PageRank 算法对应于取一个任意的随机向量 $p$，计算权重法 $T_p T_p^2 T_p^3 ...$ 并看这个序列是否收敛到什么。

这个基本算法有一些问题，例如没有外向链接的网页。一般来说，在数学上我们是在处理" 不变子空间"。例如，考虑一个只有 2 个页面的网络，并有以下邻接矩阵。

$$A = \begin{pmatrix} 1/2 & 0 \\ 1/2 & 1 \end{pmatrix}$$

自己检查一下，这对应于第二页没有外链的情况。现在让 $p$ 成为起始向量 $p_t = (1, 1)$，并计算几个迭代的功率法。你是否看到，用户在第二页的概率上升到了 1? 这里的问题是，我们正在处理一个可还原矩阵。

为了防止这个问题，PageRank 引入了另一个元素：有时候，用户会因为点击而感到无聊，会去一个任意的页面（对于没有出站链接的页面也有规定）。如果我们把 s 称为用户点击一个链接的机会，那么进入一个任意页面的机会就是 1-s。一起来看，我们现在有这样一个过程

$$p' \leftarrow sTp + (1 - s)e$$

也就是说，如果 $p$ 是一个概率向量，那么 $p'$ 就是一个概率向量，它描述了用户在进行了一次页面转换（无论是通过点击一个链接还是通过" 传送"）之后的位置。

PageRank 向量是这个过程的静止点；你可以把它看作是用户进行了无限次转换之后的概率分布。PageRank 向量满足以下条件

$$p = sTp + (1-s)e \Leftrightarrow (I - sT)p = (1-s)e$$

因此，我们现在不得不想，$I - sT$ 是否有一个逆。如果逆的存在，它满足

$$(I - sT)^{-1} = I + sT + s^2T^2 + \ldots$$

不难看出逆的存在：利用格什哥林定理（附录 13.5），你可以看到 $T$ 的特征值满足 $|\lambda| \leqslant 1$。现在利用 $s < 1$，所以部分和的序列收敛。

上述逆的公式也指出了一种通过使用一系列矩阵-向量乘法来计算 PageRank 向量 $p$ 的方法。

✍ **练习 9.12** 写出计算 PageRank 向量的伪代码，给定矩阵 $T$。证明你从来不需要明确计算 $T$ 的幂。(这是霍纳规则的一个实例)。

在 $s = 1$ 的情况下，也就是说，我们排除了远程传输，PageRank 向量满足 $p = Tp$，这又是寻找 Perron 向量的问题；见附录 13.4。

我们通过幂级迭代找到佩伦向量（第 13.3 节）。

$$p^{(i+1)} = Tp^{(i)}$$

这是一个稀疏的矩阵向量乘积，但与 BVP 的情况不同，稀疏性不太可能有带状结构。在计算上，人们可能必须使用与密集矩阵相同的并行论据：矩阵必须是二维分布的 [162]。

## 9.5 大型计算图论

在前面的章节中，你已经看到许多图算法有一个计算结构，使矩阵-向量积成为它们最重要的内核。由于大多数图相对于节点数来说是低度的，所以该乘积是一个稀疏的矩阵-向量乘积。

然后在很多情况下，我们可以像第 6.5 节那样，对矩阵进行一维分布，由图节点的分布诱导：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，往往计算是非常不平衡的。例如，在单源最短路径算法中，只有沿着前面的顶点是活跃的。由于这个原因，有时边的分布而不是顶点的分布是有意义的。为了平衡负载，甚至可以使用随机分布。

Floyd-Warshall 算法的并行化（第 9.1.2 节）沿着不同的路线进行。在这里，我们不计算每个节点的数量，而是计算作为节点对的函数的数量，也就是一个类似矩阵的数量。因此，我们不是分配节点，而是分配节点对的距离。

### 9.5.1 节点与边的分布

有时我们在处理稀疏图矩阵时需要超越一维分解。我们假设我们要处理的图的结构或多或少是随机的，比如说，对于任何一对顶点来说，有一条边的机会是相同的。另外，假设我们有大量的顶点和边缘，那么每个处理器都会存储一定数量的顶点。那么结论是，任何一对处理器需要交换信息的机会都是一样的，所以信息的数量是 $O(P)$。(另一种可视化的方法是看到非零点随机地分布在矩阵中)。这并不能提供一个可扩展的算法。

出路是把这个稀疏矩阵当作密集矩阵，并援引第 6.2.2 节的论据来决定矩阵的二维分布。(参见 [207] 中对 BFS 问题的应用；他们以图的形式制定了他们的算法，但是二维矩阵-向量乘积的结构是清晰可辨的)。二维乘积算法只需要处理器行和列的集合体，所以涉及的处理器数量为 $O(\sqrt{P})$。

### 9.5.2 超稀疏数据结构

在稀疏矩阵的二维分布下，可以想象一些过程会出现行数为空的矩阵。从形式上看，如果非零的数量（渐进地）小于维度，那么一个矩阵被称为超稀疏。

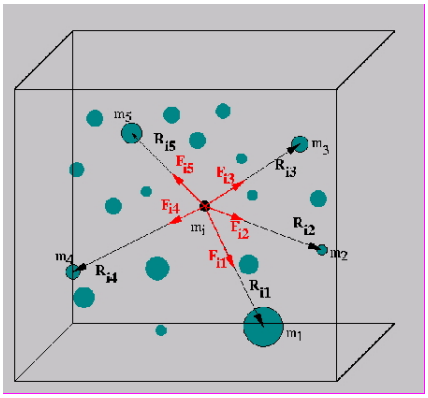
在这种情况下，CRS 格式可能是低效的，可以使用称为双压缩存储的东西 [24]。

# 第 10 章 N 体问题

内容提要

- 巴恩斯-胡特算法
- 快速多极点算法
- 全程计算
- 执行

在第 4 章中，我们研究了连续现象，例如在整个区间 [0，1] 中，受热棒在一定时间内的行为。也有一些应用，你可能对有限数量的点感兴趣。一个这样的应用是研究粒子的集合，可能是非常大的粒子，如行星或恒星，在重力或电力的影响下。(也可能有外力，我们将忽略这一点；我们还假设没有碰撞，否则我们需要纳入最近邻的相互作用)。这种类型的问题被称为 N 体问题；关于介绍，见 http://www.scholarpedia.org/article/N-body_simulations (引力)。




这个问题的基本算法是很容易的

- 选择一些小的时间间隔。
- 根据所有粒子的位置，计算每个粒子上的力。
- 移动每个粒子的位置，就像它所受的力在整个时间间隔内保持不变一样。

对于一个足够小的时间间隔，这个算法给出了一个合理的近似事实。最后一步，更新粒子的位置，很容易而且完全并行：问题在于评估力。以一种天真的方式，这种计算足够简单，甚至是完全并行的。

```
for each particle i
for each particle j
let    be the vector between i and j; ij
  then the force on i because of j is
  ij ij |rij|
  (where mi, mj are the masses or charges) and
  fji = -fij.
```

该算法的主要缺点为复杂度太高：对于 $N$ 个粒子，操作的数量是 $O(N^2)$。

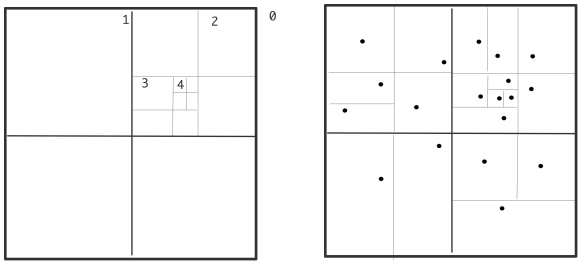
**练习 10.1** 如果我们有 $N$ 个处理器，一个更新步骤的计算将需要时间 $O(N)$。通信复杂度是多少？提示：是否可以用集合运算？

已经发明了几种算法来使顺序复杂度降低到 $O(N \log N)$ 甚至是 $O(N)$。正如可以预期的那样，这些方法比原始的算法更难实现。我们将讨论一种流行的方法：Barnes-Hut 算法 [8]，其复杂度为 $O(N \log N)$。

## 10.1 巴恩斯-胡特算法

导致复杂性降低的基本观察是以下几点。如果我们正在计算两个靠近的粒子 $i_1, i_2$ 的力，这些力来自两个同样靠近的粒子 $j_1, j_2$，你可以将 $j_1, j_2$ 合并为一个粒子，并将其用于 $i_1, i_2$。

接下来，该算法使用了空间的递归划分，在二维的象限和三维的八度空间中；见图 10.2。

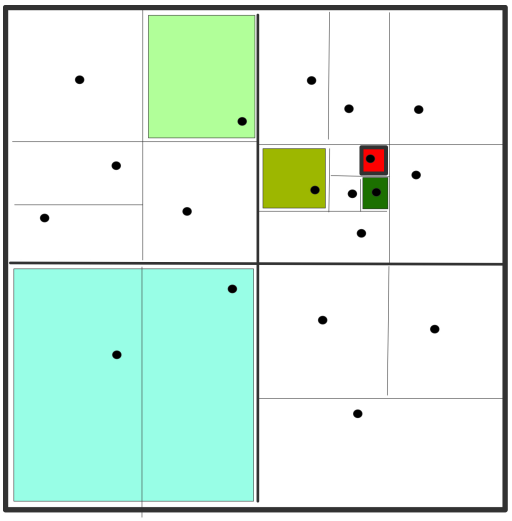



该算法如下。首先计算所有层的所有单元的总质量和质量中心。

```
for each level l, from fine to coarse: for each cell c on level l:
compute the total mass and center of mass for cell c by considering its children
if there are no particles in this cell, set its mass to zero
```

这些级别被用来计算与每个粒子的相互作用。

```
for each particle p:
for each cell c on the top level
if c is far enough away from p:
use the total mass and center of mass of c;
otherwise consider the children of c
```



对一个细胞是否足够远的测试通常是以其直径与距离足够小的比率来实施的。这有时被称为" 细胞开放标准"。以这种方式，每个粒子与若干同心环的细胞相互作用，每个下一个环的宽度是双倍的；见图 10.3。

如果细胞被组织成一棵树，这种算法就很容易实现。在三维的情况下，每个单元有八个孩子，所以这被称为八叉树。

质量中心的计算必须在粒子移动后每次进行。更新可能比从头开始计算的成本要低。另外，可能会发生一个粒子穿过一个单元的边界，在这种情况下，数据结构需要被更新。在最坏的情况下，一个粒子移动到一个原来是空的单元。

## 10.2 快速多极点法

快速多极法（FMM）计算每一点上的势的表达式，而不是像巴恩斯-胡特那样计算力。FMM 使用的信息比盒子里的粒子的质量和中心还要多。这种更复杂的扩展更准确，但也更昂贵。作为补偿，FMM 使用一组固定的盒子来计算势，而不是一组随精度参数 和质心位置变化的盒子。

然而，在计算上，FMM 很像 Barnes-Hut 方法，所以我们将共同讨论它们的实施。

## 10.3 全程计算

尽管有上述明智的近似方法，但也有对 $N^2$ 相互作用进行全面计算的努力；例如，见 Sverre Aarseth 的 NBODY6 代码；见 http://www.ast.cam.ac.uk/ sverre/web/pages/home.htm。这些代码使用高阶积分器和自适应时间步长。在 Grape 计算机上的快速实现是存在的；由于需要定期的负载平衡，一般的并行化通常是困难的。

## 10.4 执行

八叉树方法在高性能架构上提供了一些挑战。首先，问题是不规则的，其次，不规则性是动态变化的。第二方面主要是分布式内存中的问题，它需要负载再平衡；见 2.10 节。在本节中，我们集中讨论了单步的力计算。

### 10.4.1 矢量化

如图 10.2 的问题结构是很不规则的。这对于小规模的 SSE/AVX 指令和大规模的矢量流水线处理器的矢量化来说都是一个问题（关于两者的解释见 2.3.1 节）。程序步骤" 为某一盒子的所有孩子做某事" 将是不规则的长度，而且数据可能不会以规则的方式存储。

这个问题可以通过细分网格来缓解，即使这意味着有空盒子。如果底层被完全划分，将始终有八个（三维）粒子可以操作。更高的层次也可以被填满，但这意味着低层的空盒子数量越来越多，所以在增加工作和提高效率之间要进行权衡。

### 10.4.2 共享内存的实现

在顺序结构上执行，这个算法的复杂度为 $O(N \log N)$。很明显，如果每个粒子变成一个任务，这个算法在共享内存上也可以工作。由于不是所有的单元都包含粒子，任务将有不同的运行时间。

### 10.4.3 分布式内存的实现

上述 Barnes-Hut 算法的共享内存版本不能立即用于分布式内存环境，因为每个粒子原则上可以从总数据的任何部分获取信息。使用哈希八叉树有可能实现这种思路的实现，但我们不会坚持这样做。

我们观察到，数据访问的结构化程度比最初看起来要高。考虑一个粒子 $p$ 和它所交互的第 l 层的单元。位于 p 附近的粒子将与相同的细胞互动，因此我们可以通过查看第 l 层的细胞和与之互动的同一层的其他细胞来重新安排互动。

这给我们提供了以下算法 [123]：质心的计算变成了计算粒子 $g(l)$ 对第 $l$ 层施加的力。

```
for level l from one above the finest to the coarsest:
for each cell c on level l
let g(l) be the combination of the g(l+1) for all children i of c
```

有了这个，我们就可以计算出一个细胞上的力。

```
for level l from one below the coarses to the finest: for each cell c on level l:
let f(l) be the sum of
1. the force f(l-1) on the parent p of c, and p
2. the sums g(l) for all i on level l that
satisfy the cell opening criterium
```

我们看到，在每一层，每个细胞现在只与该层的少量邻居互动。在算法的前半部分，我们只使用单元格之间的父子关系来向上攀登。据推测这是很容易的。

算法的后半部分使用更复杂的数据访问。第二项中的单元格 i 都与我们要计算力的单元格 c 有一定距离。在图的术语中，这些单元可以被描述为表亲：c 的父母的兄弟姐妹的孩子。如果开放的标准更加明确，我们就用二表兄妹：$c$ 的祖父母的兄弟姐妹的孙子，等等。

✍ **练习 10.2** 论证这个力计算操作在结构上与稀疏矩阵-向量乘积有很多共同之处。

在共享内存的情况下，我们已经说过，不同的子树需要不同的时间来处理，但是，由于我们可能有更多的任务，而不是处理器核心，这一切都会扯平。对于分布式内存，我们缺乏将工作分配给任意处理器的可能性，所以我们需要谨慎地分配负载。空间填充曲线（SFC）可以在这里得到很好的应用（见 2.10.5.2 节）。

### 10.4.4 完整方法的 1.5D 实施

有可能通过分布粒子，让每个粒子评估来自其他每一个粒子的力，来直接实现完整的 $N^2$ 方法的并行化。

✍ **练习 10.3** 由于力是对称的，我们可以通过让粒子 pi 只与粒子 pj > i 相互作用来节省 2 倍的工作量。在这个方案中，数据（如粒子）的平等分布会导致工作（如相互作用）的不均衡分布。粒子需要如何分布才能得到均匀的负荷分布？

假设我们不关心使用力的对称性所带来的两个系数的收益，而使用粒子的均匀分布，这个方案有一个更严重的问题，即它渐进地没有规模。

这方面的通信成本是

- $O(P)$ 延迟，因为需要从所有处理器接收信息。
- $O(N)$ 带宽，因为所有粒子都需要被接收。

然而，如果我们分配力的计算，而不是粒子，我们就会得出不同的界限；见 [54] 和其中引用的参考文献。

随着每个处理器计算边长为 $N/\sqrt{P}$ 的区块中的相互作用，存在粒子的复制，需要收集力。因此，现在的成本是

- $O(\log p)$ 广播和还原的延迟
- $O(N/\sqrt{P}\log P)$ 带宽，因此，每个处理器对最终的总和贡献了多少力。

#### 10.4.4.1 问题描述

我们对 N 体问题进行如下抽象

- 我们假设有一个粒子电荷/质量的向量 $c()$ 和位置信息。
- 为了计算位置更新，我们需要基于存储图元 $C_{ij} =< c_i, c_j >$ 的对等相互作用 $F(,)$。
- 然后，这些相互作用被汇总为一个力矢量 $f()$。

在并行性综合模型 (IMP) 模型中，如果我们知道各自的数据分布，就可以充分地描述算法；我们并不立即关注局部计算的情况。

### 10.4.4.2 颗粒分布

基于粒子分布的实现以分布为 c 的粒子向量 $c(u)$ 为起点，并直接计算同一分布上的力向量 $f(u)$。我们将计算描述为三个内核的序列，两个是数据移动，一个是局部计算。

$$\begin{cases} \{\alpha : c(u)\} & \text{initial distribution for} cis u \\ C(u,*) = c(u) \cdot \times c(*) & \text{replicate} con\ each\ processor \\ \text{local computation of} F(u,*) \text{from} C(u,*) & \\ f(u) = \sum_2 F(u,*) & \text{local reduction of partial forces} \\ \text{particle position update} & \end{cases}$$

最后的内核是一个具有相同的 $\alpha$ 和 $\beta$ 分布的还原，所以它不涉及数据运动。本地计算内核也没有数据运动。这就留下了从初始分布 $u(u)$ 收集 $C(u,)$ 的问题。如果没有任何关于分布 $u$ 的进一步信息，这就是对 $N$ 元素的全收集，其成本为 $\alpha \log p + \beta N$。特别是，通信成本不会随着 $p$ 的下降而下降。

通过一个合适的基于分布的编程系统，方程组（10.1）可以被转化为代码。

### 10.4.4.3 工作分布

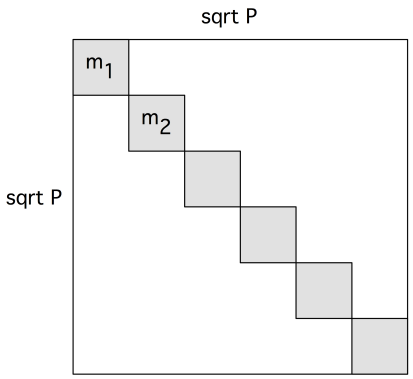
粒子分布的实现使用了一个一维分布 $F(u)$，用于与粒子分布一致的力。由于 $F$ 是一个二维物体，所以也可以使用二维分布。我们现在将探讨这个选项，它早先在 [54] 中描述过；我们在这里的目的是要说明如何在 IMP 框架中实现和分析这个策略。我们不是把算法表达为分布式计算和复制的数据，而是使用分布式的临时，我们把复制方案推导为子通信器上的集体。

对于二维分布的 $F$，我们需要 $(N/b) \times (N/b)$ 处理器，其中 $b$ 是一个块大小参数。为了便于说明，我们使用 $b = 1$，给出处理器的数量 $P = N^2$。

现在我们将进行必要的推理。图 10.4 中给出了接近可实现形式的完整算法。为了简单起见，我们使用识别分布 $Ip \to \{p\}$。

$$\begin{aligned} &\{\alpha : C(D : \langle I, I \rangle)\} \quad \text{initial distribution on the processor diagonal} \\ &C(I,I) = C(I,p) + C(q,I) \quad \text{row and column broadcast of the processor diagonal} \\ &F(I,I) \leftarrow C(I,I) \\ &f(I,I) \leftarrow C(I,I) \quad \text{local interaction calculation} \end{aligned}$$

初始存储在处理器网格的对角线上我们首先考虑收集图元 $C_{ij} =< c_i, c_j >$。最初，我们决定让 $c$ 向量存储在处理器网格的对角线上；换句话说，对于所有 $p$，处理器 $< p, p >$ 包含 $C_p$，而所有其他处理器的内容则未被定义。




为了正式表达这一点，我们让 $D$ 为对角线 $\{< p, p > p \in P\}$。利用部分定义分布的机制，最初的 $\alpha-$ 分布就是

$$C(D < I, I >) \equiv D \ni < p, q > \mapsto C(I(p), I(q)) = C_{pq}.$$

力计算的复制本地力计算 $f_{pq} = f(pq)$ 对每个处理器 $< p, q >$ 的数量 $pq$，所以我们需要一个 $\beta$ 分布的 $C(I,I)$ 。

关键是认识到

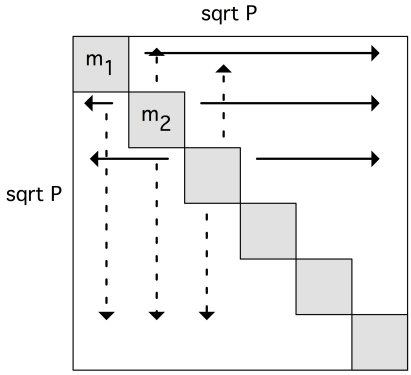
$$C_{qp} = C_{pp} + C_{qq}$$

以至于

$$C(I,I) = C(I,p) + C(q,I)$$

(其中 $p$ 代表将每个处理器映射到索引值 $p$ 的分布，$q$ 同理)。

为了找到从 $\alpha$ 到 $\beta$ 分布的转化，我们考虑表达式 $C$ 的转化 $(D :< I,I >)$。一般来说，任何集合 $D$ 都可以写成从第一个坐标到第二个坐标的数值集合的映射。




$$D \equiv p \mapsto D_p \quad \text{where} \quad D_p = q < p,q > \in D.$$

在我们对角线的特殊情况下，我们有

$$D \equiv p \mapsto p.$$

有了这个，我们就可以写出

$$C(D < I,I >) = C(I,D_pI) = C(I,pp).$$

请注意，方程（10.4）仍然是 $\alpha$ 分布。$\beta$ 分布是 C(I,p)，这是一个模式匹配的练习，可以看到这是由每一行的广播来实现的，其成本为 $\alpha \log \sqrt{p} + \beta N/\sqrt{P}$。

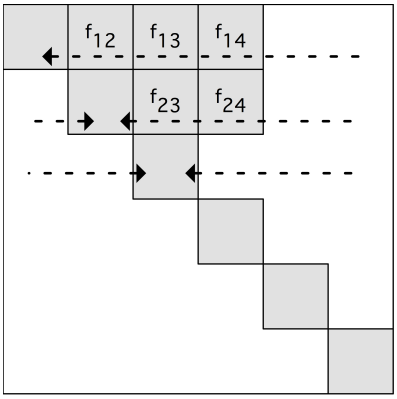
同样地，$C(q,I)$ 是通过列广播从 $\alpha$ 分布中找到的。我们的结论是，这个变体确实有一个通信成本，并随着处理器数量的增加而成比例地下降。

局部交互计算 $F(I,I) \leftarrow C(I,I)$ 具有相同的 $\alpha$ 和 $\beta$ 分布，因此它是并行的。

力的求和我们必须将力矢量 $f()$ 扩展到我们的二维处理器网格，即 $f(,)$。然而，为了符合对角线上的初始粒子分布，我们只在对角线上求和。该指令

$$f(D < I,I >) = \sum F(I,D)$$

有一个 $\beta$ -分布的 $I,D$ ，它是由 $\alpha$ -分布的 $I$,$I$ 通过聚集在行中形成。


f12
f13
f14
f23
f24

# 第 11 章 蒙特卡洛方法

内容提要

- 奖励
- 示例

蒙特卡洛模拟是一个广义概念，指的是使用随机数和统计抽样来解决问题的方法，而非精确建模。从这种抽样的性质来看，结果会有一些不确定性，但统计学上的" 大数定理" 会确保不确定性随着样本数量的增加而下降。

统计抽样的一个重要工具是随机数发生器。参见教程 32，了解随机数生成。

## 11.1 奖励

在边长为 2 的正方形中随机选取一个点，它落入正方形内切圆中的概率为 $\pi/4$。因此我们可以通过大量随机选取点来估计 $\pi$ 的值。我们甚至可以做一个简单的物理实验：假设院子中存在一个形状不规则的池塘，而院子本身是面积已知的长方形。如果现在向院子里扔卵石，使它们在任何给定的地方都有同样的可能性，那么落在池塘里的卵石和落在外面的卵石的比例就等于面积的比例。

从数学角度讲，令 $\Omega \in [01]^2$ 为某个区域，函数 $f(\bar{x})$ 为 $\Omega$ 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 $\tilde{x}_0, \tilde{x}_1, \tilde{x}_2 [0,1]^2$，我们可以通过计算 $f(\bar{x}_i)$ 为正或负的频率来估计 $\Omega$ 的面积。

我们可以把这个想法扩展到积分。一个函数在一个区间 $(a,b)$ 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，我们也可以估计其平均数为

$$\langle f \rangle \approx \frac{1}{N} \sum_{i=1}^{n} f(x_i)$$

如果点 $x$ 的分布合理，并且函数 $f$ 不是太苛刻的话。可以得出

$$\int_a^b f(x) dx \approx (b-a) \frac{1}{N} \sum_{i=1}^{n} f(x_i)$$

统计理论告诉我们，积分中的不确定性 $\sigma_I$ 与下列因素有关。标准差 $\sigma_f$ 的表达为

$$\sigma_I \sim \frac{1}{\sqrt{N}} \sigma_f$$

且服从正态分布。

### 11.1.1 吸引人的地方是什么?

到目前为止，蒙特卡洛积分看起来与经典的黎曼和积分没有什么区别，但当增加空间维度时，差别就较为明显。此时，经典积分需要在每个维度上有 $N$ 个点，因此有 $d$ 维上有 $N^d$ 个点。另一方面，在蒙特卡洛方法中，点是从 $d$ 维空间中随机抽取的，而且点的数量少得多。

在计算方面，蒙特卡洛方法很有吸引力，因为所有的函数评估都可以并行进行。

对标准差为 $\sigma$ 的事件进行 $N$ 次取样观察，其平均标准差为：$\sigma/\sqrt{N}$。这意味着更多采样观测将降低平均标准差以提高精度；使蒙特卡洛方法的有趣之处在于，这种准确性的提高与原始问题的维度无关。

蒙特卡洛方法是模拟具有统计性质现象的不二选择，如放射性衰变或布朗运动。而蒙特卡洛方法其他应用

范畴则不属于科学计算领域，例如，用于股票期权定价的 Black-Scholes 模型 [15] 就使用了蒙特卡洛模拟。

线性方程组也可以通过蒙特卡洛方法求解，但这并不常见。下面我们将以一个用精确方法解答十分复杂的应用为例，说明蒙特卡罗方法所代表的统计抽样手段的便捷之处。

## 11.2 示例

### 11.2.1 伊辛模型的蒙特卡洛模拟

伊辛模型（介绍见 [33]）最初是为了模拟铁磁性而提出的。磁性是原子排列其" 自旋" 方向的结果：假设自旋只能是" 向上" 或" 向下"，那么如果有更多的原子自旋向上，而不是向下，那么这种材料就具有磁性，反之亦然。这些原子被称为" 晶格" 结构中的原子。

现在想象一下，加热一种材料会使原子松动。如果在材料上施加一个外部场，原子将开始与场对齐，如果场被移除，磁性又会消失。然而在某个临界温度以下，材料将保留其磁性。我们将使用蒙特卡洛模拟来寻找保留的稳定构型。

假设晶格 $\Lambda$ 有 $N$ 个原子，我们把原子的构型表示为 $\sigma = (\sigma_1 ... \sigma_N)$，其中每个 $\sigma_i = \pm 1$。晶格的能量模型为

$$H = H(\sigma) = -J \sum_i \sigma_i - E \sum_{ij} \sigma_i \sigma_j$$

第一个项模拟单个自旋 $\sigma_i$ 与强度为 $J$ 的外部场的相互作用。第二项是对近邻对的求和，模拟原子对的排列：如果原子具有相同的自旋，则乘积 $\sigma_i \sigma_j$ 为正，如果相反则为负。

在统计力学中，构型的概率是

$$P(\sigma) = \exp(-H(\sigma))/Z$$

其中" 分区函数"$Z$ 定义为

$$Z = \sum_\sigma \exp(H(\sigma))$$

其中，总和在所有 $2^N$ 配置上运行。

如果一个构型的能量在微小扰动下没有减少，那么它就是稳定的。为了探索这一点，我们在晶格上进行迭代，探索改变原子的自旋是否会降低能量。我们引入了一个偶然因素来防止人为的解决方案。(这就是 Metropolis 算法 [155])。

```
for fixed number of iterations do for each atom i do
calculate the change \Delta E from changing the sign of \sigma i
if \Delta E < or exp(-\Delta E) greater than some random number then
accept the change
```

如果我们注意到与稀疏矩阵-向量乘积结构的相似性，这种算法可以被并行化。在该算法中，我们也是通过结合几个近邻的输入来计算一个局部数量。这意味着我们可以对晶格进行分区，在每个处理器收集到一个**幽灵区域**（ghost region）后计算局部更新。

让每个处理器在晶格中的局部点上迭代，相当于晶格的一个特定的全局排序；为了使并行计算等同于串行计算，我们还需要一个并行随机发生器（第 32.3 节）。

# 第 12 章 机器学习

内容提要

- 神经网络
- 深度学习网络
- 计算层
- Stuff

**机器学习**（Machine Learning，ML）是一些处理" 智能" 问题的技术的统称，如图像识别。抽象地讲，这些问题是从一个**特征向量**（features）空间（如图像中的像素值）到另一个结果向量空间的映射。在图像识别字母的情况下，这个最终空间可能是 26 维的，第二个分量的最大值将表明一个"B " 被识别。

ML 技术的基本特征是，这种映射是由大量的内部参数描述的，而且这些参数是逐步完善的。这里的学习方面是，通过将输入与基于当前参数的预测输出和预期输出进行比较来完善参数。

## 12.1 神经网络

现在最流行的 ML 形式是深度学习（DL），或**神经网络**（neural networks）。神经网络，或深度学习网络，是一个计算数字输出的函数，给定一个（通常是多维的）输入点。假设这个输出被归一化为区间 [0, 1]，我们可以通过在输出上引入一个阈值来使用神经网络作为一个分类工具。

为什么是" 神经的" 网络?

在生物体内，神经元是一个" 发射" 的细胞，也就是说，如果它收到某些输入，就会发出一个高电压。在 ML 中，我们把它抽象为一个感知器：一个根据某些输入而输出数值的函数。具体来说，输出值通常是输入的线性函数，其结果限制在（0，1）范围内。

### 12.1.1 单个数据点

在其最简单的形式中，我们有一个输入，其特征是一个特征向量 $s\bar{x}$，和一个标量输出 $y$。我们可以通过使用相同大小的权重向量和标量偏置 b 来计算 $y$ 作为 $x$ 的线性函数。

$$y = \bar{w}\bar{x} + b$$

### 12.1.2 激活函数

为了有一定的尺度不变性，我们引入了一个被称为激活函数的函数 $\sigma$，它映射了 $\mathbb{R} \to (01)$，我们实际上计算了标量输出 $y$ 为

$$\mathbb{R} \ni y = \sigma\left(\bar{w}^t\bar{x} + b\right)$$

一种常见的 sigmoid 函数为

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

该函数有一个有趣的特性，即

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

所以计算函数值和导数并不比只计算函数值开销大多少。

对于向量值的输出，我们以点的方式应用 sigmoid 函数。

```
// funcs.cpp
template <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 positive
      a.vals[i] = 1 / (1 + exp(-a.vals[i]));
 }
}
```

其他激活函数有：$y = tanh(x)$ 或'ReLU'（矫正线性单元）。

$$f(x) = \max(0, x).$$

在其他地方（如 DL 网络的最后一层），softmax 函数可能更合适。

### 12.1.3 多维输出

由 $\bar{w}$、$\bar{b}$ 定义的单层可以实现我们对神经网的所有要求，这一点很罕见。通常情况下，我们使用一个层的输出作为下一个层的输入。这意味着我们计算的不是一个标量 $y$，而是一个多维向量 $\bar{y}$。

现在我们对输出的每个分量都有权重和偏置，所以

$$\mathbb{R}^n \ni \bar{y} = \sigma(W\bar{x} + \bar{b})$$

其中 $W$ 现在是一个矩阵。

有几点看法：

- 如上所述，输出向量通常比输入的分量少，所以矩阵不是正方形，特别是不能倒置。
- sigmoid 函数使整个映射变得非线性。
- 神经网络通常有多个层次，每个层次都是以 $x \to y$ 的形式进行的映射。

以上。

### 12.1.4 卷积

上面关于应用权重的讨论将输入视为一组没有进一步结构的特征。然而，在图像识别等应用中，输入向量是一个图像，有一个结构需要被承认。对输入向量进行线性化后，如果输入向量中的像素在水平方向上很接近，但在垂直方向上则不接近。

因此，我们的动机是找到一个能反映这种位置性的权重矩阵。我们通过引入内核来做到这一点：一个小的"stencil" 被应用于图像的各个点。(见第 4.2.4 节关于 PDEs 背景下的 stencil 的讨论）这样的内核通常是一个小的正方形矩阵，应用它是通过取模版值和图像值的内积来完成的。(这是对信号处理中卷积一词的不精确使用）。

例如：https://aishack.in/tutorials/image-convolution-examples/。

## 12.2 深度学习网络

现在我们将介绍一个完整的神经网络。这种介绍遵循 [105]。

使用一个具有 $L \geqslant 1$ 层的网络，其中 $\ell = 1$ 层是输入层，$\ell = L$ 层是输出层。

对于 $\ell = 1, ..., L$，第 $\ell$ 层计算

$$z^{(\ell)} = W^{(\ell)} a^{(\ell)} + b^{(\ell)}$$

$$y^{(\ell)} = \sigma\left(y^{(\ell)}\right)$$

其中，$a^{(1)}$ 是输入，$z^{(L+1)}$ 是最终输出。我们将其简洁地写成

$$y^{(L)} = N_{\{W^{(\ell)}\}_\ell, \{b^{(\ell)}\}_e ll}\left(a^{(1)}\right)$$

其中我们通常会省略网对 $W^{(\ell)}$、$b^{(\ell)}$ 集的依赖。

```
// 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);
}
```

### 12.2.1 分类

在上述描述中，输入 $x$ 和输出 $y$ 都是向量值。也有一些情况是需要不同类型的输出的。例如，假设我们想描述数字的位图图像；在这种情况下，输出应该是一个整数 09。

我们通过让输出 $y$ 在 $\mathbb{R}^{10}$ 中来解决这个问题，如果 $y_5$ 比其他输出成分足够大，我们就说网络能识别数字 5。通过这种方式，我们保持整个故事仍然是实值的。

### 12.2.2 误差最小化

通常我们有数据点 $x_{ii=1,}$ 有已知的输出 $y_i$，我们想让网络预测尽可能好地再现这种映射。从形式上看，我们寻求最小化成本，或者说误差

$$C = \frac{1}{N} L\left(N\left(x_i\right), y_i\right)$$

在所有选择 $W$，$b$ 中。(通常我们不会明确说明这个成本是所有 $W^{[\ell]}$ 权重矩阵和 $b^{[\ell]}$ 偏差的函数。)

```
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;
}
```

成本最小化意味着选择权重 $\{W^{(\ell)}\}_\ell$ 和偏置 $\{b^{(\ell)}\}_\ell$，使得对于每个 $x$。

$$\left[\left\{W^{(\ell)}\right\}_\ell, \left\{b^{(\ell)}\right\}_\ell\right] = \underset{\{W\},\{b\}}{\operatorname{argmin}} L\left(N_{\{W\},\{b\}}(x), y\right)$$

其中 $L(N(x), y)$ 是一个损失函数，描述计算出的输出 $N(x)$ 与预期输出 $y$ 之间的距离。

我们使用梯度下降法找到这个最小值。

$$w \leftarrow w + \Delta w$$

其中

$$\Delta W = \frac{\partial L}{\partial W_{ij}^{(\ell)}}$$

这是一个复杂的表达式，我们现在将不做推导。

### 12.2.3 系数的计算

我们对成本相对于各种权重、偏差和计算量的部分导数感兴趣。对于这一点，引入一个简短的方法是很方便的。

$$\delta_i^{[\ell]} = \frac{\partial C}{\partial z_i^{[\ell]}} \quad \text{for} 1 \le i \le n_\ell \text{and} 1 \le \ell < L$$

现在应用连锁法则（完整的推导见上面引用的论文），我们得到，用 $x \circ y$ 表示点式（或哈达玛）向量-向量积 $\{x_i y_i\}$。

- 在最后层

$$\delta^{[L-1]} = \sigma'\left(z^{[L-1]}\right) \cdot \left(a^{[L]} - y\right)$$

- 递归到前一层级

$$\delta^{[\ell]} = \sigma'\left(z^{[\ell]}\right) \circ \left(W^{[\ell+1]^t} \delta^{[\ell+1]}\right)$$

- 对偏置量的敏感性

$$\frac{\partial C}{\partial b_i^{[\ell]}} = \delta_i^{[\ell]}$$

- 对权重的敏感性

$$\frac{\partial C}{\partial w_{ik}^{[\ell]}} = \delta_i^{[\ell]} a_k^{[\ell-1]}$$

使用特殊形式

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

给出

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

| | | |
|---|---|---|
| Input layer $\ell = 1$ starts with: | | |
| $a^{(\ell)} = x$ | network input | $n_{ell} \times b$ |
| For layers $\ell = 1, \ldots, L$ | | |
| $a^{(\ell)}$ | layer input | $n_\ell \times b$ |
| $W^{(\ell)}$ | weights | $n_{\ell+1} \times n_\ell$ |
| $b^{(\ell)}$ | biases | $n_{\ell+1} \times b$ |
| $z^{(\ell)} \leftarrow W^{(\ell)} a^{(\ell)} + b^{(\ell)}$ | biased product | $n_{\ell+1} \times b$ |
| $a^{(\ell+1)} = y^{(\ell)} \leftarrow \sigma(z^{(\ell)})$ | activated product | $n_{\ell+1} \times b$ |
| The final output: | | |
| $y = a^{(L+1)} = z^{(L)}$ | | $n_{L+1} \times b$ |
| For layers $\ell = L, L-1, \ldots, 1$ | | |
| $D^{(\ell+1)} \leftarrow \text{diag}(\sigma'(z^{(\ell)}))$ | | $n_{\ell+1} \times n_{\ell+1}$ |
| $\delta^{(\ell)} \leftarrow \begin{cases} D^{L+1}(a^{L+1} - y) \\ D^{(\ell+1)} W^{(\ell+1)^t} \delta^{(\ell+1)} \end{cases}$ $\ell < L$ | | $n_{\ell+1} \times b$ |
| $\Delta W^{(\ell)} \leftarrow \delta^{(\ell)} a^{(\ell-1)^t}$ | weights update | $n_{\ell+1} \times n_\ell$ |
| $w^{(\ell)} \leftarrow w^{(\ell)} - \Delta W^{(\ell)}$ | | |
| $\Delta b^{(\ell)} \equiv \delta^{(\ell)}$ | bias update | $n_{\ell+1} \times b$ |

### 12.2.4 算法

现在我们在图 12.1 中介绍完整的算法。我们的网络有 $\ell = 1, ..., L$，其中参数 $n_\ell$ 表示层 $\ell$ 的输入大小。

第 1 层有输入 $x$，第 $L$ 层有输出 $y$。考虑到 minibatch 的使用，我们让 $x$，$y$ 表示一组大小为 $b$ 的输入/输出，因此它们的大小分别为 $n_1 \times b$ 和 $n_{L+1} \times b$。

## 12.3 计算层

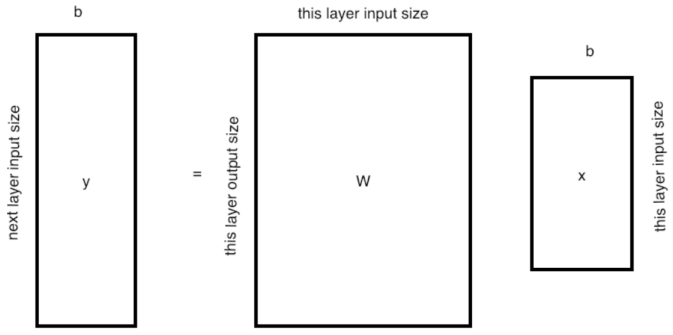
在本节中，我们将讨论深度学习的高性能计算方面。在标量意义上，我们论证了矩阵-矩阵乘积的存在，它可以被高效率地执行。我们还将讨论并行性，重点是数据并行，其基本策略是分割数据集。

- 模型并行，其基本策略是分割模型参数；以及
- 流水线化，在这种情况下，指令可以按照比原始的模型更多的串行执行。

### 12.3.1 重量矩阵乘积

无论是在应用层，即前向传递，还是在学习，即后向传递中，我们都要对权重矩阵进行矩阵乘向量的乘积。这种操作没有太多的缓存重用，因此性能不高；1.7.11 节。

另一方面，如果我们将一些数据点捆绑在一起--这有时被称为小批量--并对它们进行联合操作，我们的基本操作就变成了矩阵乘以矩阵乘积，它能够有更高的性能；1.6.1.2 节。



我们在图 12.2 中描述了这一点。

- 输入批和输出批由相同数量的向量组成。
- 权重矩阵 $W$ 的行数等于输出大小，列数等于输入大小。相当于输入规模。

gemm 内核的这种重要性（1.6.1 和 6.4.1 节）导致人们为其开发专用硬件。

另一种方法是使用权重矩阵的特殊形式。在 [143] 中，研究了用托普利茨矩阵的近似。这在节省空间方面有优势，而且可以通过 FFT 进行乘积。

### 12.3.2 权重矩阵乘积的并行性

我们现在可以考虑有效计算 $N(x)$。前面已经说过，矩阵-矩阵多运算是一个重要的内核，但除此之外，我们还可以使用并行处理。图 12.3 给出了两种并行化策略。

在第一种策略中，批处理被划分到各个进程中（或者说，多个进程同时处理独立的批处理）；我们把这称为数据并行。

✍ **练习 12.1** 考虑在共享内存环境下的这种情况。在这段代码中

```
for b = 1, ... , batchsize
  for i = 1,...,outsize
    yi,b \leftarrow \sumj Wi,j  xj,b
```

假设每个线程计算 1，... 的部分内容。, batchsize 的范围。把这一点翻译成你最喜欢的编程语言。你是按行还是列来存储输入/输出向量？为什么？这两种选择的影响是什么？

✍ **练习 12.2** 现在考虑分布式内存背景下的数据并行，每个进程都在批处理的一个片断（块列）上工作。你看到一个直接的问题了吗？

还有第二种策略，被称为模型并行，其中模型参数，也就是权重和偏差是分布式的。正如在图 12.3 中看到的，这立即意味着该层的输出向量是分布式计算的。

✍ **练习 12.3** 概述一下分布式内存中第二个分区的算法。

这些策略的选择取决于模型是否很大，权重矩阵是否需要被分割，或者输入的数量是否很大。当然，也可以把这些结合起来，模型和批处理都是分布式的。

### 12.3.3 权重更新

权重更新的计算

$$\Delta W^{(\ell)} \leftarrow \delta^{(\ell)} a^{(\ell-1)t}$$

是一个等级为 $b$ 的外积。它需要两个向量，并从它们中计算出一个低秩矩阵。

✍ **练习 12.4** 根据 $n_\ell$ 和 $b$ 之间的关系，讨论该操作中（潜在）的数据重用量。为简单起见，假设 $n_{\ell+1} \approx n_\ell$。

✍ **练习 12.5** 讨论图 12.3 的两种分区策略中所涉及的数据移动结构。

除了这些方面，当我们考虑并行处理小批量时，这个操作变得更加有趣。在这种情况下，每个批次都独立地计算一个更新，我们需要对它们进行平均。在假设每个进程计算一个完整的 $\Delta W$ 的情况下，这就变成了一个全还原。这种"HPC 技术" 的应用被开发成 Horovod 软件 [76, 181, 111]。在一个例子中，在涉及 40 个 GPU 的配置上显示了相当大的速度。

另一个选择是延迟更新，或以异步方式执行。

✍ **练习 12.6** 讨论在 MPI 中实现延迟或异步更新的问题。

### 12.3.4 流水线

最后一种类型的并行可以通过在各层上应用流水线来实现。简要说明这如何能提高训练阶段的效率。

### 12.3.5 卷积

应用卷积相当于乘以一个托普利茨矩阵。这比完全通用的矩阵-矩阵乘法的复杂度要低。

### 12.3.6 稀疏矩阵

权重矩阵可以通过忽略小条目而被稀疏化。这使得稀疏矩阵乘以密集矩阵的乘积成为主要的操作 [72]。

### 12.3.7 硬件支持

从上述内容中，我们可以得出结论，gemm 计算内核的重要性。将一个普通的 CPU 专门用于这一目的是对硅和电力的相当大的浪费。至少，使用 GPU 是一个节能的解决方案。

然而，通过使用特殊用途的硬件，甚至可以达到更高的效率。下面是一个概述：https://blog.inten.to/hardware-for-deep-learning-part-4-asic-96a542fe6a81 在某种程度上，这些特殊用途的处理器是收缩阵列的再世。

### 12.3.8 降低精度

见 3.7.4.2 节。

## 12.4 Stuff

通用近似定理设 $\varphi()$ 是一个非常数、有界、单调增加的连续函数。让 $Im$ 表示 $m$ 维的单位超立方体 $[0,1]^m$。$Im$ 上的连续函数空间用 $CI_m$ 表示。那么，给定任何函数 $f \in C(I_m)$ 和 $\varepsilon > 0$，存在一个整数 $N$，实常数 $v_i, b_i \in \mathbb{R}$ 和实向量 $w_i \in \mathbb{R}^m$，其中 $i = 1, N$，这样我们可以定义。

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

作为函数 $f$ 的近似实现，其中 $f$ 与 $\varphi$ 无关；即。

$$|F(x) - f(x)| < \varepsilon$$

对于所有 $x \in Im$。换句话说，$F(x)$ 形式的函数在 $C(I_m)$ 中是密集的。NN 可以近似于乘法吗?

https://stats.stackexchange.com/questions/217703/can-deep-neural-network-approximate

传统的神经网络由线性图和 Lipschitiz 激活函数组成。作为 Lischitz 连续函数的组合，神经网络也是 Lipschitz 连续的，但乘法不是 Lipschitz 连续。这意味着，当 $x$ 或 $y$ 中的一个数字过大时，神经网络工作不能近似于乘法。

# 第三部分

# 附录（Appendices）

# 第 13 章 线性代数

学习本节之前，我们将默认您有矩阵和向量的基础知识，并了解简单的算法如：矩阵与向量乘积、矩阵的可逆性等等。附录介绍了一些通常不属于线性代数第一课程的概念和定理。

## 13.1 范数

**范数**（norm）是将绝对值的概念概括到多维对象（如向量和矩阵）的一种方式。有许多定义范数的方法，也有关于不同范数之间关系的理论。这里我们只给出基本的定义；更多的细节可以参见任何一本线性代数教科书，例如 [82]。

### 13.1.1 向量范数

向量空间 $V$ 上的任意函数 $n()$ 的范数有如下性质：

- $x \in V$ 时 $n(x) \geqslant 0$ 且 $x = 0$ 时 $n(x) = 0$.
- 对于 $x \in V, \lambda \in \mathbb{R}$，总有 $n(\lambda x) = |\lambda| n(x)$.
- $n(x + y) \leqslant n(x) + n(y)$

对于任意的 $p \geqslant 1$，向量范数的定义为：

$$|x|_p = \sqrt[p]{\sum_i |x_i|^p}$$

常见的范数有 $\|\cdot\|_1$（" 绝对值之和"）和 $\|\cdot\|_2$（" 平方之和的平方根"）；$\|\cdot\|_\infty$ 定义为 $\lim_{p\to\infty} \|\cdot\|_p$，不难看出，这等于

$$\|x\|_\infty = \max_i |x_i|$$

### 13.1.2 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。

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}$$

然而，我们将主要研究相关的矩阵范数。

$$\|A\|_p = \sup_{\|x\|_p = 1} \|Ax\|_p = \sup_x \frac{\|Ax\|_p}{\|x\|_p}$$

根据定义，可以得出

$$\|Ax\| \leqslant \|A\| \|x\|$$

为相关范数。容易得出：

- $\|A\|_1 = \max_j \sum_i |a_{ij}|$
- $\|A\|_\infty = \max_j \sum_i |a_{ij}|$

观察 $\|A\|_2 = \sup_{\|x\|_2 = 1} x^t A^t A x$，不难得出 $\|A\|_2$ 是 $A$ 的最大奇异值，是 $A^t A$ 的最大特征值的根。

矩阵的条件数定义为：

$$\kappa(A) = \|A\| \left\|A^{-1}\right\|$$

在对称情况下，使用 2-norm，这就是最大和最小特征值之间的比率。

## 13.2 格兰-施密特正交化

Gram-Schmidt（GS）算法将一系列的向量进行归纳正交。这可以用来把一个向量空间的任意基变成一个正交基；也可以看成是把一个矩阵 $A$ 变成一个具有正交列的矩阵。如果 $Q$ 有正交列，那么 $Q^tQ$ 就是对角线，这通常是一个很方便的属性。

GS 算法的基本原理可以用两个向量 $u, v$ 来证明。假设我们想要一个向量 $v$，使 $u, v$ 和 $u, v$ 横跨同一空间，但 $v \perp u$。为此，我们令

$$v' \leftarrow v - \frac{u^t v}{u^t u} u$$

很容易看出，这满足要求。

假设我们有一组向量 $u_1, ..., u_n$，我们想将其正交化，通过连续应用上述变换来做到这一点。

```
For i = 1,...,n:
For j = 1, ... i - 1:
let c_{ji} = u_{jt} u_i/u_{jt} u_j
  For i = 1,...,n:
update u_i <- u_i - u_{j}c_{ji}
```

通常情况下，上述算法中的向量 $v$ 被归一化；这就增加了一条线

$$u_i \leftarrow u_i / \|u_i\|$$

到算法中。GS 正交化与这种归一化，应用于矩阵，也被称为 QR 因式分解。

**练习 13.1** 假设我们将 GS 算法应用于一个矩形矩阵 $A$ 的列，得到一个矩阵 $Q$。证明有一个上三角矩阵 $R$，使得 $A = QR$。(提示：看上面的 $c_{ji}$ 系数。) 如果我们按照上面的算法将正交向量归一，$Q$ 有一个额外的属性：$Q^tQ = I$。也要证明这一点。

上面给出的 GS 算法可以计算出所需的结果，但是只能用精确的算术。如果 $v$ 和其中一个 $u_i$ 之间的角度很小，计算机的实现就会很不准确。在这种情况下，修改后的 Gram-Schmidt（MGS）算法会表现得更好。

```
For i = 1,...,n:
For j = 1, ... i - 1:
let cji = ujt ui/ujt uj update ui  ui - ujcji
```

为了与 MGS 形成对比，最初的 GS 算法也被称为经典的 Gram-Schmidt（CGS）。为了说明这两种方法的区别，请考虑矩阵

$$A = \begin{pmatrix} 1 & 1 & 1 \\ \epsilon & 0 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon \end{pmatrix}$$

其中 $\epsilon$ 是机器精度的顺序，因此在机器算术中 $1 + \epsilon^2 = 1$。CGS 方法如下：

- 在机器运算中，第一列的长度为 1，因此

$$q_1 = a_1 = \begin{pmatrix} 1 \\ \epsilon \\ 0 \\ 0 \end{pmatrix}$$

- 第二列被正交为 $v \leftarrow a_2 - 1q_1$，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

$$\begin{cases} c_1 = q_1^t a_3 = 1 \\ c_2 = q_2^t a_3 = 0 \end{cases} \Rightarrow v = \begin{pmatrix} 0 \\ -\epsilon \\ 0 \\ \epsilon \end{pmatrix}; \quad \text{normalized:} \quad q_3 = \begin{pmatrix} 0 \\ \frac{\sqrt{2}}{2} \\ 0 \\ \frac{\sqrt{2}}{2} \end{pmatrix}$$

很容易看出，$q_2$ 和 $q_3$ 根本就不是正交的。相比之下，MGS 方法的不同之处在于最后一步。

- 如前所述，$q_1^t a_3 = 1$，所以

$$v \leftarrow a_3 - q_1 = \begin{pmatrix} 0 \\ -\epsilon \\ \epsilon \\ 0 \end{pmatrix}$$

然后，$q_2^t v = \sqrt{2}\epsilon$（注意之前 $q_2^t a_3 = 0$），所以第二次更新得出

$$v \leftarrow v - \frac{\sqrt{2}}{2}\epsilon q_2 = \begin{pmatrix} 0 \\ \frac{\epsilon}{2} \\ -\frac{\epsilon}{2} \\ \epsilon \end{pmatrix}, \quad \text{normalized:} \begin{pmatrix} 0 \\ \frac{\sqrt{6}}{6} \\ -\frac{\sqrt{6}}{6} \\ 2\frac{\sqrt{6}}{6} \end{pmatrix}$$

现在所有的 $q_i^t q_j$ 对于 $i \neq j$ 的顺序都是 $\epsilon$。

## 13.3 幂法

向量序列

$$x_i = A x_{i-1}$$

其中 $x_0$ 是某个起始向量，被称为幂法，因为它计算的是随后的矩阵幂与向量的乘积。矩阵幂乘以一个向量，所以称为幂法。

$$x_i = A^i x_0$$

有些情况下，$x_i$ 向量之间的关系很简单。例如，如果 $x_0$ 是一个特征向量，对于某些标量 $\lambda$，我们有

$$A x_0 = \lambda x_0 \quad \text{and} \quad x_i = \lambda^i x_0$$

然而，对于一个任意的向量 $x_0$，序列 $\{x_i\}_i$ 很可能由独立的向量组成，直到一个点。

✍ **练习 13.2** 让 $A$ 和 $x$ 为 $n \times n$ 矩阵和维度 $n$ 向量

$$A = \begin{pmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & \ddots & \ddots & \\ & & & 1 & 1 \\ & & & & 1 \end{pmatrix}, \quad x = (0, \ldots, 0, 1)^t$$

说明对于 $i < n$ 序列 $[x, Ax, ..., A_i x]$ 是一个独立的集合。为什么在 $i \geqslant n$ 的情况下这不再是真的？现在考虑矩阵 $B$：

$$A = \begin{pmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & \ddots & \ddots & \\ & & & 1 & 1 \\ & & & & 1 \end{pmatrix}, \quad x = (0, \ldots, 0, 1)^t$$

说明在 $i < n/2$ 的情况下，$[y, By, ..., B_i y]$ 是一个独立的集合，但对于任何更大的 $i$ 值都不是。

虽然在一般情况下，向量 $xAxA_2x.....$ 可以预期是独立的，但在计算机运算中，这个故事不再那么清晰。假设矩阵有特征值 $\lambda_0 > \lambda_1 \geqslant \lambda_{n-1}$ 和相应的特征向量 $u_i$，因此

$$Au_i = \lambda_i u_i$$

让向量 $x$ 写为

$$x = c_0 u_0 + \cdots + c_{n-1} u_{n-1}$$

那么

$$A^i x = c_0 \lambda_0^i u_i + \cdots + c_{n-1} \lambda_{n-1}^i u_{n-1}$$

如果我们把它写成

$$A^i x = \lambda_0^i \left[ c_0 u_i + c_1 \left( \frac{\lambda_1}{\lambda_0} \right)^i + \cdots + c_{n-1} \left( \frac{\lambda_{n-1}}{\lambda_0} \right)^i \right]$$

我们看到，从数值上看，$A^i x$ 将逐渐接近于主导特征向量 $u_0$ 的倍数。因此，任何使用 $A^i x$ 向量的独立性的计算都可能是不准确的。

## 13.4 非负矩阵；佩伦向量

如果 A 是一个非负矩阵，最大的特征值具有其特征向量为非负的特性：这就是佩伦-弗洛本纽斯定理。

**定理 13.1 (定理)**

如果一个非负矩阵 $A$ 是不可逆的，其特征值满足

- 幅度最大的特征值 $\alpha_1$ 是实数和简单的。

$$\alpha_1 > |\alpha_2| \geqslant ...$$

- 相应的特征向量为正数。

## 13.5 格什戈林定理

寻找矩阵的特征值通常很复杂。然而，有一些工具可以估计特征值。在本节中，我们将看到一个定理，在某些情况下，它可以提供关于特征值的有用信息。

令 $A$ 是一个方形矩阵，$x$ 是一个特征对：$Ax = \lambda x$。从一个分量来看，我们有

$$a_{ii} x_i + \sum_{j \neq i} a_{ij} x_j = \lambda x_i$$

范数：

$$(a_{ii} - \lambda) \leqslant \sum_{j \neq j} |a_{ij}| |\frac{x_j}{x_i}|$$

取决于 $i$ 的值，在此情况下 $|x_i|$ 是最大的，我们发现

$$(a_{ii} - \lambda) \leqslant \sum_{j \neq i} |a_{ij}|.$$

这句话可以解释如下。

特征值 $\lambda$ 位于以 $a_i$ 为半径的圆内 $\sum_{j \neq i} |a_{ij}|$.

由于我们不知道哪一个 $i$ 的值是最大的，我们只能说存在某个 $i$ 的值，使得 $\lambda$ 位于这样一个圆中。这就是 Gershgorin 定理。

**定理 13.2 (定理)**

令 $A$ 是一个正方形矩阵，让 $D_i$ 是中心为 $a_i$，半径为 $\sum j \neq i|a_{ij}|$ 的圆，那么特征值包含在圆 $\cup_i D_i$ 的联盟中。如果常数向量不是特征向量，我们可以得出结论，特征值在这些圆盘的内部。

## 13.6 Householder 反射器

在某些情况下，问题出现了，如何将一个子空间转化为另一个子空间。在某种意义上，Householder reflectors 是这个问题的最小解决方案。考虑一个单位向量 $u$，并让

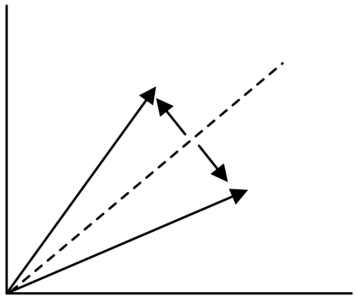
$$H = I - 2uu^t$$

对于这个矩阵，我们有 $Hu = -u$，如果 $u \perp v$，那么 $Hv = v$。换句话说，$v$ 的倍数的子空间

的倍数子空间被翻转，而正交子空间则保持不变。现在我们来看看将一个空间映射到另一个空间的原始问题。让原空间被一个向量 $x$，所产生的空间为 $y$，则注意到

$$\begin{cases} x = (x+y)/2 + (x-y)/2 \\ y = (x+y)/2 - (x-y)/2 \end{cases}$$

换句话说，我们可以根据 $u = (x-y)/2$，用反射器将 $x$ 映射为 $y$。



我们可以将 Householder 反射器概括为一种形式

$$H = I - 2uv^t$$

在 LU 因子化中使用的矩阵 $L_i$（见第 5.3 节）可以被看作是 $L_i = I - \ell_i e_i^t$ 的形式。

其中 $e_i$ 在 $i$ 的位置上有一个一，而 $\ell_i$ 在该位置以下只有非零。这种形式也使得我们很容易看到 $L_i^{-1} = I = I + \ell_i e_i^t$:

$$(I - uv^t)(I + uv^t) = I - uv^t uv^t = 0$$

若 $v^t u = 0$。

# 第 14 章 复杂度

在本书的不同地方，我们对一个算法需要多少次运算感兴趣。这些操作取决于上下文，但通常我们计算加法（或减法）和乘法。这被称为一个算法的算术或计算复杂性。例如，对 $n$ 数字求和需要 $n-1$ 次加法。另一个我们可能想要描述的数量是需要的空间（计算机内存）的数量。有时，容纳一个算法的输入和输出的空间就是全部需要的，但有些算法需要临时空间。所需的总空间被称为一个算法的空间复杂度。

算术复杂度和空间复杂度都取决于对输入的一些描述，例如，对于一个数组的求和，只需要数组的长度 $n$。我们用" 数组求和的时间复杂度为 $n-1$，其中 $n$ 为数组的长度" 来表达这种依赖关系。

求和算法所花费的时间（或空间）并不依赖于其他因素，如数字的值。相比之下，一些算法，如计算整数阵列的最大公除数，可能取决于实际数值。

✍ **练习 14.1** 将两个大小为 $n \times n$ 的正方形矩阵相乘的时间和空间复杂度是多少？假设加法和乘法花费的时间相同。

通常我们的目的是简化描述时间或空间复杂性的公式。例如，如果一个算法的复杂度是 $n^2+2n$，我们可以看到，对于 $n>2$，复杂度小于 $2n^2$，对于 $n>4$，复杂度小于 $(3/2)n^2$。另一方面，对于所有的 $n$ 值，其复杂度至少为 $n^2$。显然，二次项 $n^2$ 是最重要的，而线性项 $n$ 按比例变得越来越不重要。我们非正式地表示，当 $n \to \infty$ 时，复杂度对 $n$ 是二次的：存在常数 $c$，$C$，所以对于 $n$ 足够大时，复杂度至少是 $cn^2$，最多是 $Cn^2$。

简而言之，就是复杂度为 $n^2$，当 $n \to \infty$ 时，写为 $O(n^2)$。在第四章中，我们会看到想要描述的现象是一个归零的参数的阶数。在这种情况下，例如我们写成 $f(h)=O(h^2)$，当 $h \downarrow 0$ 时，意味着 $f$ 被 $ch^2$ 和 $Ch^2$ 所约束，对于某些常数 $c$、$C$ 和 $h$ 足够小。

# 第 15 章 偏微分方程

偏微分方程是大部分高性能计算问题的根源。这里是两个最重要的公式的快速推导。

## 15.1 偏导数

函数 $u(x)$ 的导数是对变化率的度量。偏导数相同，但对于函数 $u(x,y)$ 有两个变量。标记为 $u_x$ 和 $u_y$ 的这些偏导数表示，如果只有一个变量变化，而另一个保持不变，则变化率。

在形式上，我们定义 $u_x, u_y$:

$$u_x(x,y) = \lim_{h\to 0} \frac{u(x+h,y)-u(x,y)}{h}, \quad u_y(x,y) = \lim_{h\to 0} \frac{u(x,y+h)-u(x,y)}{h}$$

## 15.2 泊松或拉普拉斯方程

设 $T$ 为材料的温度，则其热能与其成正比。一段长度 $\Delta x$ 具有热能 $Q = c\Delta x u$。如果这段的热能是恒定的

$$\frac{\delta Q}{\delta t} = c\Delta x \frac{\delta u}{\delta t} = 0$$

但它也是部分的流入和流出之间的差异。由于流量与温差成比例，也就是与 $u_x$ 成比例，我们也看到了这一点

$$0 = \left.\frac{\delta u}{\delta x}\right|_{x+\Delta x} - \left.\frac{\delta u}{\delta x}\right|_{x}$$

在 $\Delta x \downarrow 0$ 的极限下，得到 $u_{xx} = 0$，这被称为拉普拉斯方程。如果我们有一个初始项，例如对应于外部施加的热量，方程就变成 $u_{xx} = f$，这被称为泊松方程。

## 15.3 热方程

设 $T$ 为材料的温度，则其热能与其成正比。一段长度 $\Delta x$ 具有热能 $Q = c\Delta x u$。这一段的热能变化率是

$$\frac{\delta Q}{\delta t} = c\Delta x \frac{\delta u}{\delta t}$$

但它也是部分的流入和流出之间的差异。由于流量与温差成比例，也就是与 $u_x$ 成比例，我们也看到了这一点

$$\frac{\delta Q}{\delta t} = \left.\frac{\delta u}{\delta x}\right|_{x+\Delta x} - \left.\frac{\delta u}{\delta x}\right|_{x}$$

在 $\Delta x \downarrow 0$ 的限制下，这会得到 $u_t = \alpha u_{xx}$。

## 15.4 恒稳态

IBVP 的解决方案是函数 $u(x,t)$。在强迫函数和边界条件不依赖于时间的情况下，解将随时间收敛到一个称为稳态解的函数:

$$\lim_{t\to\infty} u(x,t) = u_{steadystate}(x)$$

这个解满足一个 BVP，这个 BVP 可以通过设置 $u_t \equiv 0$ 得到。例如，对于热方程

$$u_t = u_{xx} + q(x)$$

稳态解满足 $-uxx = q(x)$。

# 第 16 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。在某种意义上，泰勒展开式定理要求函数可以用多项式逼近得多好，也就是说，对于一个给定的函数 f，我们能否找到 $i = 1$ 的系数 $c_i \cdots n$, 这样

$$f(x) \approx c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

这个问题显然需要改进。我们说的“近似相等”是什么意思? 这个近似公式并不适用于所有的函数 f 和所有的 x: 函数 $\sin x$ 对于所有的 x 是有界的，但是对于 $x \to +\infty$，任何多项式都是无界的，因此任何多项式逼近 $\sin x$ 函数都是无界的。很明显，我们只能在一个区间上近似。

我们将证明一个具有足够多导数的函数 f 可以近似如下: 如果 $n - th$ 导数 $f(n)$ 在区间 $I$ 上是连续的，则存在系数 $c_0 \cdots c_{n-1}$

$$\forall_{x \in I} : \left| f(x) - \sum_{i<n} c_i x^i \right| \leq c M_n \quad \text{where} M_n = \max_{x \in I} \left| f^{(n)}(x) \right|$$

很容易得到这些系数应该是多少。假设

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots$$

（在那里我们不会担心收敛的问题和点持续多久）然后填入

$$x = 0 \quad \text{gives} \quad c_0 = f(0).$$

接下来，求一阶导数

$$f(x) = c_1 + 2c_2 x + 3c_3 x^2 + \cdots$$

填入

$$x = 0 \quad \text{gives} \quad c_1 = f'(0).$$

二阶导数

$$f''(x) = 2c_2 + 6c_3 x + \cdots$$

给定 $x = 0$ 得到:

$$c_2 = f''(0)/2$$

同样，在三阶导数中

$$\text{filling in} \quad x = 0 \text{gives} c_3 = \frac{1}{3!} f^{(3)}(0).$$

现在我们需要更精确一点。柯西形式的泰勒定理说

$$f(x) = f(a) + \frac{1}{1!} f'(a)(x-a) + \cdots + \frac{1}{n!} f^{(n)}(a)(x-a)^n + R_n(x)$$

其中，余项 $R_n$ 为:

$$R_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-a)^{n+1} \quad \text{where} \xi \in (a,x) \text{or} \xi \in (x,a) \text{depending.}$$

如果 $f^{(n+1)}$ 是有界的，$x = a + h$，那么我们经常使用泰勒定理的形式是

$$f(x) = \sum_{k=0}^{n} \frac{1}{k!} f^{(k)}(a) h^k + O\left(h^{n+1}\right)$$

我们现在用一个多项式在某一区间上逼近函数 $f$，其误差随多项式次数的倒数呈几何级数递减。为了证明泰勒定理，我们用分部积分法。首先，我们写

$$\int_a^x f'(t)dt = f(x) - f(a)$$

为

$$f(x) = f(a) + \int_a^x f'(t)dt$$

然后给出分部积分法

$$\begin{aligned} f(x) &= f(a) + [xf'(x) - af'(a)] - \int_a^x tf''(t)dt \\ &= f(a) + [xf'(x) - xf'(a) + xf'(a) - af'(a)] - \int_a^x tf''(t)dt \\ &= f(a) + x\int_a^x f''(t)dt + (x-a)f'(a) - \int_a^x tf''(t)dt \\ &= f(a) + (x-a)f'(a) + \int_a^x (x-t)f''(t)dt \end{aligned}$$

给出了分部积分法的另一应用

$$f(x) = f(a) + (x-a)f'(a) + \frac{1}{2}(x-a)^2 f''(a) + \frac{1}{2}\int_a^x (x-t)^2 f'''(t)dt$$

归纳起来，这就给出了泰勒定理

$$R_{n+1}(x) = \frac{1}{n!}\int_a^x (x-t)^n f^{(n+1)}(t)dt$$

根据中值定理，这是

$$\begin{aligned} R_{n+1}(x) &= \frac{1}{(n+1)!} f^{(n+1)}(\xi)\int_a^x (x-t)^n f^{(n+1)}(t)dt \\ &= \frac{1}{(n+1)!} f^{(n+1)}(\xi)(x-a)^{n+1} \end{aligned}$$

# 第 17 章 最小化

## 17.1 下降法

我们考虑多变量函数 $fR^N \to$ 以及为求最小值的向量 $\bar{x}$ 的问题。即使是光滑函数，我们也会立即面对存在局部极小值和全局极小值的事实; 现在我们满足于找到一个局部最小值。

我们不是在一次大型计算中寻找最小值，而是使用迭代策略，从一个点 $\bar{x}$ 开始，将其更新为 $\bar{x}+\bar{h}$，并重复这个过程。更新向量 $\bar{h}$ 是通过直线搜索策略找到的：我们确定搜索方向 $\bar{h}$，沿着该方向找到步长 $\tau$，然后更新

$$\bar{x} \leftarrow \bar{x} + \tau\bar{h}.$$

### 17.1.1 最陡下降

利用泰勒展开公式的多维形式（第 16 节），我们可以将任何足够光滑的函数写成

$$f(\bar{x}+\bar{h}) = f(\bar{x}) + \bar{h}^t\nabla f + \cdots$$

不难看出，选择 $\bar{h} = -\nabla f$ 会得到最大的最小化，所以我们设置

$$x_{\text{new}} \equiv x - \tau\nabla f$$

这种方法被称为梯度下降或最陡下降。

对于新函数的值

$$f(\bar{x} - \tau\nabla f) \approx f(\bar{x}) - \tau\|\nabla f\|^2$$

因此，对于 $\tau$ 足够小的情况，这使得新函数的值既为正，又小于 $f(\bar{x})$。

对于二次函数 $f$，步长 $\tau$ 可以计算出来，否则近似为:

$$f(\bar{x}+\tau\bar{h}) = f(\bar{x}) + \tau\bar{h}^t\nabla f + \frac{\tau^2}{2}h^t(\nabla\cdot\nabla f)h + \cdots$$

忽略高阶项并设置 $\delta f/\delta\tau = 0$ 给出

$$\tau = -\bar{h}^t\nabla f / h^t(\nabla\cdot\nabla f)h$$

另一种寻找合适步长的策略叫做**回溯**（backtracking）:

- 从默认的步长开始，例如 $\tau \equiv 1$。
- 然后直到 $f(\bar{x}+\tau\bar{h}) < f(\bar{x})$，执行

$$\tau \leftarrow \tau/2$$

### 17.1.2 随机梯度下降法

虽然梯度在任何给定的步骤中都是最优的搜索方向，但总的来说它不一定是最好的。例如，二次问题，其中包括许多问题的偏微分方程，一个更好的选择是正交搜索方向。另一方面，在 ML 应用中，随机梯度下降 (SGD) 是一个很好的选择，其中坐标方向被用作搜索方向。在这种情况下

$$f\left(x+\tau e_i\right) = f(x) + \tau\frac{df}{de_i} + \frac{\tau^2}{2}\frac{\delta^2 f}{\delta e_i^2}$$

那么

$$\tau = -(\nabla f)_i / \frac{\delta^2 f}{\delta e_i^2}$$

### 17.1.3 代码

#### 17.1.3.1 预知识

我们声明了一个 vector 类，它是一个标准 vector 类，并在其上定义了诸如加法和旋转等操作。有一个派生类 unit_vector 做了显而易见的工作。

#### 17.1.3.2 框架

我们首先将函数定义为纯虚类，这意味着任何函数都需要支持这里提到的方法：

```
// 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;
};
```

使用这样的函数可以定义各种更新步骤。例如，最陡的下降阶梯使用梯度：

```
// minimlib.cxx
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，随机下降是基于单位向量:

```
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;
};
```

#### 17.1.3.3 健全性测试

在圆周上使用最陡的下降可以在一步内收敛：

```
Code:
// ellipse.cxx
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;
```

另一方面，椭圆上最陡的下降需要多次迭代：

```
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;}
```

## 17.2 牛顿法

牛顿法（或牛顿-拉弗森法）是一个求函数零点的迭代过程，即求出 $f(x) = 0$ 的值 $x$。它需要知道函数的导数 $f'$，可以从 17.1 等数字中证明。

另一个理由来自于最小化: 如果函数 $f$ 是两次可微的，我们可以这样写

$$f(x+h) = f(x) + h^t \nabla f + \frac{1}{2} h^t \left(\nabla^2 f\right) h$$

最小值为

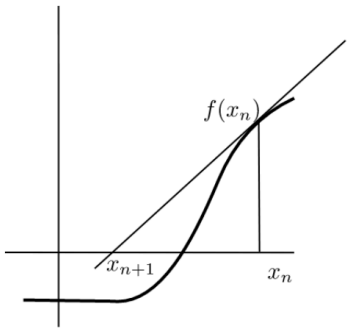
$$x_{\text{new}} = x - \left(\nabla^2 f\right)^{-1} \nabla f$$

这相当于求出梯度的零点：

$$0 = \nabla f(x+h) = \nabla f(x) + h^t \left(\nabla^2 f(x)\right) h$$

✍ **练习 17.1** 让 $f(x_1,x_2)=(10x_1^2+x_2^2)/2+5\log(1+e^{-x_1-x_2})$。梯度下降法和牛顿法的收敛速度有多快？为了了解它们的不同行为，可以根据函数的关卡曲线绘制若干次迭代。

这个练习告诉我们梯度下降是错误的：它总是垂直于当前的水平曲线。然而，最小值并不一定在那个方向上。




迭代：

$$x_{n+1}=x_n-f(x_n)/f'(x_n)$$

没有证明，我们声明：

- 如果迭代的起始点足够接近于 0，并且函数在 0 处是可微的，那么牛顿法将收敛到 0。
- 对于许多函数，牛顿方法不会收敛，但可以通过引入阻尼或做不精确的更新来获得收敛：

$$x_{n+1}=x_n-\alpha f(x_N)/f'(x_n)$$

其中 $\alpha<1$。

✍ **练习 17.2** 牛顿的方法有可能是循环的。假设这是一个长度为 2 的循环：

$$x_0\to x_1\to x_2=x_0$$

如果写出这个循环的方程，就会得到 $f$ 的微分方程。解决方案是什么？为什么牛顿法在这个函数中不收敛？

在多维情况下，即以函数 $fR^N\to R$ 牛顿法成为一个迭代线性系统解：

$$\bar{x}_{n+1}=\bar{x}_n-F\left(\bar{x}_n\right)^{-1}f\left(\bar{x}_n\right)$$

其中 $F$ 是 $f$ 的雅可比矩阵。

由于牛顿法是一个迭代过程，我们不需要对这个线性系统的解达到完全精度，所以不精确的解，例如通过迭代求解，往往是可行的。

### 17.2.1 不精确牛顿法

牛顿方法不收敛的原因有很多（事实上，对于合适的函数，从它的收敛点绘制区域可以得到很好的分形）。由于这个原因，在实践中使用的是不精确的牛顿法。这种不精确表现在两个方面：

- 代替“最佳”步长，我们使用一个分数。这通常是通过“回溯”来选择的，并采用了至少观察到一些函数值下降的选择。
- 我们不用雅可比矩阵的逆而是用这个算子的近似值。例如，我们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。

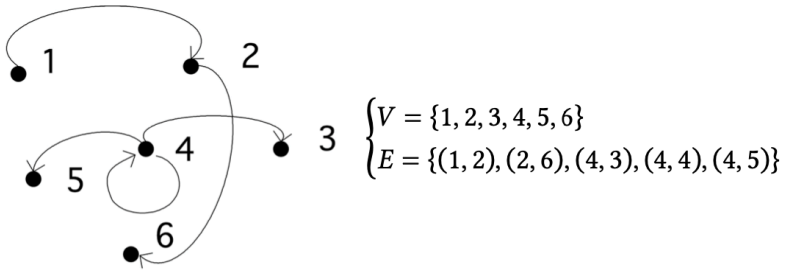
可以证明，这些技术的组合可以保证收敛 [209]。

# 第 18 章 图论

图论是研究物体之间成对关系的数学分支。图既可以作为分析 HPC 中问题的工具，也可以作为研究对象。本附录介绍了基本概念和相关理论。

## 18.1 定义

图由一组对象以及它们之间的一组关系组成。这些对象，称为图的节点或顶点，通常形成一个有限集，所以我们通常用连续的整数 $1 \cdots n$ 或 $0 \cdots n-1$。节点之间的关系由图的边来描述：如果 $i$ 和 $j$ 是相关的，我们就说 $(ij)$ 是图的边。这个关系不需要是对称的，例如“小于”关系。那么，正式图是一个元组 $G =< V, E >$ 其中 $V = \{1, \cdots, n\}$，且 $E \subset \{(i,j) : 1 \leqslant i, j \leqslant n, i \neq j\}$。




如果 $(i,j) \in E \Leftrightarrow (ji) \in E$，则图为“无向图”。另一种是有向图，我们用从 $i$ 到 $j$ 的箭头指示边缘 $(ij)$。图论中经常出现的两个概念是图的度数和直径。

**定义 18.1 (度与直径)**

度表示连接到任意节点的最大节点数：

$$d(G) \equiv \max_i |\{j : j \neq i \wedge (i,j) \in E\}|.$$

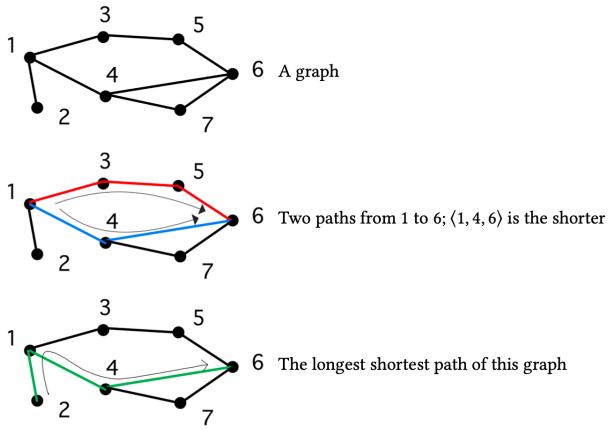
图的直径是图中最长的最短路径的长度，其中路径定义为一组顶点 $v_1 \cdots v_{k+1}$，即对于所有 $i \neq j$ 和 $v_i \neq v_j$。

$$\forall_{1 \leqslant i \leqslant k} : (v_i, v_{i+1}) \in E.$$

路径长度为 $k$。

♣

直径的概念如图 18.2 所示



除 $v_1 = v_k + 1$ 外，所有节点都不相交的路径称为圈。有时我们只对边的存在感兴趣 $(ij)$，有时我们给边附加一个值或“权重”$w_{ij}$。带加权边的图称为加权图。这样的图可以表示为元组 $G =< V, E, W >$，其中 $E$ 和 $W$ 具有相同的基数。相反，一个未加权图的所有权值都是相同的，在这种情况下我们忽略权值。

## 18.2 图的一般类型

### 18.2.1 有向无环图

没有圈的图称为无环图。这类图的一个特殊情况是有向无环图（DAG）。例如，这种类型的图可以用于建模任务之间的依赖关系：如果在 $i$ 和 $j$ 之间有一条边，这意味着任务 $i$ 必须在任务 $j$ 之前完成。

### 18.2.2 树

DAGs 的一种特殊情况是树形图: 在这里，任何节点都可以有多个传入边，但只有一条传出边。没有入边的节点是叶节点; 没有出线边的节点称为根（可以有多个根吗?），而所有其他节点称为内部节点。

### 18.2.3 二部图

如果一个图可以划分为两个节点集，其中边只从一个节点集延伸到另一个节点集，而不是在一个集合内，我们称之为**二部图**（bipartite graph）。

## 18.3 图着色和独立集

我们可以给图的节点分配标签，这相当于把一组节点划分成不相交的子集。其中一种有趣的标记是图形着色：这里选择的标签（或“颜色”）是这样的，如果节点 $i$ 和 $j$ 具有相同的颜色，则没有边连接它们：$i, j \notin E$。

图的颜色很简单，每个节点都有自己的颜色。更有趣的是，可以给图着色的最小颜色数称为图的色数。

✍ **练习 18.1** 证明如果一个图的度数为 $d$，则颜色数最多为 $d + 1$。

一个著名的图着色问题是“**四色定理**”: 如果一个图在二维地图（所谓的平面图）上描绘国家，那么颜色数最多为 4。一般来说，找到颜色数字是困难的（事实上，这是一个 np 难题）。

图着色的颜色集也被称为**独立集**（independent sets），因为在每个颜色中，没有节点连接到相同颜色的节点。有一个简单的方法可以找到独立的集合：声明每个节点都有自己独特的颜色。另一方面，在独立集合中寻找“最佳”除法，例如通过寻找图的颜色数，是很困难的。然而，通常只需找到将节点合理划分为独立集就足够了，例如在构建并行 ILU 预处理器时。下面的算法做到了 [116,149]:

- 给每个节点一个唯一的随机数。
- 现在找到一组节点，其数量高于其所有邻居；称之为第一个独立集。
- 从图中删除这个集合，并再次找到比它们所有的节点数量都高的节点邻居；这将是第二组。
- 重复此过程，直到所有节点都在一个独立的集合中。

✍ **练习 18.2** 试着证明以这种方式找到的集合确实是独立的。

## 18.4 图论算法

在本节中，我们只简单地接触图算法；完整的讨论可以在 9.1 节中找到。

- **距离算法**: 在诸如流量路由这样的应用中，知道从一个给定节点到所有其他节点的最短距离是很有意义的: 单源最短路径问题; 或者任意一对的最短距离全对最短路径问题。

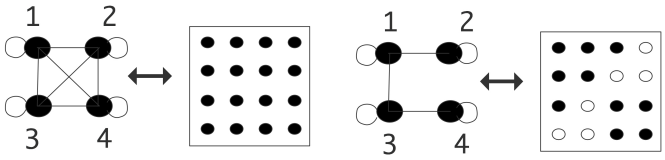
- **连接算法**：在社交网络中，我们很有兴趣知道两个人是否有联系，这个图是否可以划分为不联系的子图，或者某个人是否两个群体之间的关键桥梁。

## 18.5 图和矩阵

图可以用多种方式呈现。当然，您可以只列出节点和边，但这种方法几乎不能得到深入的了解。简单的图可以通过绘制顶点和边来可视化，但对于大的图来说，这就变得笨拙了。另一种方法是构造图的邻接矩阵。对于图 $G =< V, E >$，定义邻接矩阵 $M$（其大小 $n$ 等于顶点的数量 $|V|$）

$$M_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

相反，如果我们有一个矩阵，特别是一个稀疏矩阵，我们可以构造它的邻接图。这在图 18.3 中对密集矩阵和稀疏矩阵进行了说明。在这个例子中，矩阵在结构上是对称的，所以我们在图中使用直线而不是箭头。每个顶点上都有一条边对应于对角线元素；这条边通常会被省略在插图中。




对于带边权值的图，我们将邻接矩阵的元素设为权值：

$$M_{ij} = \begin{cases} w_{ij} & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

如果一个矩阵没有零元素，它的邻接图在每一对顶点之间有一条边。这样的图叫做**小团**（clique）。如果图是无向的，邻接矩阵是**对称的**（structurally symmetric），反之，如果一个矩阵是结构对称的，它的邻接图是无向的。

### 18.5.1 排列

图通常用来表示现实世界中物体之间的关系。Facebook 上的“朋友关系”就是一个例子。在这种情况下，图中的节点没有自然编号: 它们由名称标识，任何编号都是人为的。因此，如果我们采用不同的编号方式，我们就会想知道哪些图形属性保持不变，哪些属性会改变。

对一组对象重新编号可以通过应用一个排列矩阵进行代数建模。排列矩阵是一个方阵，其中每一行和每一列都恰好有一个元素等于 1；其他所有元素都是零。

✍ **练习 18.3** 让一组 $N$ 对象 $x_1 \cdots x_N$ 得到。排列为 $x_1, x_3, \cdots, x_2, x_4 \cdots$ 的排列矩阵是什么？也就是说，求矩阵 $P$，使得：

$$\begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = P \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix}$$

✍ **练习 18.4** 证明矩阵的特征值在置换下是不变的。

### 18.5.2 不可约性

作为在邻接矩阵中易于解释的图概念的一个例子，考虑可重延性。

**定义 18.2 (不可约图)**

如果一个图的每个节点对 $ij$ 都有一条从 $i$ 到 $j$ 和从 $j$ 到 $i$ 的路径，则该图称为不可约图。如果一个图不是不可约的，它就是可约的。

♣

✍ **练习 18.5** 令矩阵 $A$：

$$A = \begin{pmatrix} B & C \\ \varnothing & D \end{pmatrix}$$

其中 $B$ 和 $D$ 为方阵。证明它所对应的图的可约性邻接矩阵。

如果我们置换图，它的可约性或不可约性是不变的。然而，从邻接矩阵来看，它现在可能不再明显。

### 18.5.3 图的闭包

这是邻接矩阵如何简化图的推理的另一个例子。

✍ **练习 18.6** 令 $G =< V, E >$ 为无向图，让 $G' = V, E'$ 为具有相同顶点的图，但顶点由定义

$$(i, j) \in E' \Leftrightarrow \exists_k : (i, k) \in E \wedge (k, j) \in E$$

若 $M$ 为 $G$ 的邻接矩阵，则证明 $M^2$ 为 $G'$ 的邻接矩阵，其中我们使用布尔乘法对元素：$11 = 1$，$1 + 1 = 1$。

### 18.5.4 线性代数中的图运算

在上述大多数情况下，邻接矩阵不过是一个表。现在我们将证明我们可以用它来做线性代数，使它配得上“矩阵”这个名字。作为在邻接矩阵 $G$ 上使用线性代数的一个简单示例，设 $e$ 为所有 1 的向量，则 $Ge$ 是列出节点的度数的向量。

我们可以用这种方法做很多操作。考虑一个加权图 $G$。为每个节点 $i$ 寻找最大权重 $g_{ij}$ 可以描述为

$$y = G \otimes e \quad \text{where} \quad y_i = \max_j g_{ij} \cdot 1$$

这看起来像正规的矩阵向量乘积 $Ge$，但是用最大值计算代替了总和。

在很多情况下，我们实际上需要左积，也就是说，从左到右的邻接矩阵乘以一个行向量。例如，$e_i$ 是位置 $i$ 为 1，其他位置为 0 的向量。那么 $e_i^t G$ 在每 $j$ 中都有一个 1，在 $G$ 的邻接图中是 $i$ 的一个邻居。

#### 18.5.4.1 马尔可夫链

这个左矩阵向量积有一个巧妙的应用：马尔可夫链。假设我们有一个系统（参见实例 19）可以是任何 $n$ 状态。然后我们可以使用邻接矩阵来建模状态转换。

让 $G$ 通过一个邻接矩阵，其属性是所有元素都是非负的，并且每一行的元素和为 1。我们现在将 $g_{ij}$ 解释为从状态 $i$ 到状态 $j$ 的概率。设 $x$ 为概率向量，即 $x_i$ 为处于状态 $i$ 的非负概率，则 $y^t = x^t G$ 描述了这些概率，在一次状态转换之后。

✍ **练习 18.7** 一个向量 $x$ 要成为一个合适的概率向量，其元素的和需要为 1。表明 $y$ 元素的和再次为 1。

#### 18.5.4.2 一般的矩阵-向量乘积

上面的例子表明，有时我们对具有矩阵-向量乘积结构的邻接矩阵进行运算，但不一定使用加法和乘法（更多的例子可以在 9.1 节中找到）。

基于此，我们定义了一般乘积

$$y = G \oplus \cdot_{\otimes} x$$

其中 $\otimes$ 为二元算子，$\oplus$ 为约简算子，为

$$y_i = \bigoplus_j (g_{ij} \otimes x_j)$$

在这个符号中，求出最大的权值

$$w = G \max \cdot_{\times} e$$

这将在第 9 章的几个算法中使用; 其他应用请参见 [133]。

## 18.6 谱图理论

对于图 $G$ 及其邻接矩阵 $A_G$，我们可以通过缩放 $A_G$ 定义随机矩阵或马尔可夫矩阵，得到单位行和:

$$W_G = D_G^{-1} A_G \quad \text{where } (D_G)_{ii} = \deg(i)$$

为了了解如何解释这一点，让我们看一个简单的例子。我们取一个带邻接矩阵的非加权图

$$A_G = \begin{pmatrix} 1 & & 1 & 1 \\ & & 1 & 1 \\ 1 & & 1 & 1 \\ & 1 & 1 & \end{pmatrix}$$

再看第二行，这里有边 (2,3) 和 (2,4) 这意味着如果在节点 2 上，我们可以到节点 3 和 4。对这个矩阵进行缩放

$$W_G = \begin{pmatrix} 1/3 & & 1/3 & 1/3 \\ & & 1/2 & 1/2 \\ 1/3 & & 1/3 & 1/3 \\ & 1/2 & 1/2 & \end{pmatrix}$$

第二行表示从节点 2 得到节点 3 和节点 4 的概率相等。我们也可以从数学上推导出这个命题:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} W_G = \begin{pmatrix} 0 & 0 & 1/2 & 1/2 \end{pmatrix}$$

可以很简单地推断: 如果 $p$ 是一个向量，其中第 $i$ 组件给出了节点 $i$ 存在的概率，那么 $(p^t W_G)_i$ 是节点 $i$ 的概率，如果沿着图沿再走一步的话。

✍ **练习 18.8** 证明 $p^t W_G$ 确实是一个概率向量。提示:可以表达出来 $p$ 是一个概率向量 $p^t e = e$,其中 $e$ 是所有 1 的向量。

### 18.6.1 图拉普拉斯算子

另一个与图相关联的矩阵是图拉普拉斯矩阵

$$L_G = D_G - A_G$$

这个矩阵的行和为零，且对角项为正，因此根据 Gershgorin 定理（第 13.5 节），它的所有特征值都在复右半平面上。

✍ **练习 18.9** 证明所有 1 的向量是一个特征值为 1 的特征向量。这个拉普拉斯矩阵给出了一个二次形式

$$x^t L_G x = \sum_{(i,j)\in E} (x_i - x_j)^2$$

### 18.6.2 通过拉普拉斯矩阵进行区域分解

有许多有趣的定理与图邻接和拉普拉斯矩阵有关。这些对于区域分解有非常实际的应用。我们受到椭圆偏微分方程的启发。

与拉普拉斯方程 $-\Delta u = f$ 相联系的是一个算子 $Lu = -\Delta u$。在单位区间 [0,1] 上，该算子的特征函数，即 $Lu = \lambda u$ 的函数为 $n > 0$ 的 $u_n(x) = \sin \pi x$。它们的性质是 $u_n(x)$ 在间隔的内部有 $n-1$ 个零，并且它们分割了函数为正或负的 $n$ 连接区域中的间隔。因此，如果我们想在 $p$ 处理器上划分域 $\Omega$，您可以考虑 $\Omega$ 上拉普拉斯算子的第 $p$ 特征函数，并找到其为正或为负的连通区域。

这个关于偏微分方程的陈述在费德勒定理的两个版本中具有图等价性。(在这部分我们不会给出任何证明; 参见 [185])。

**定理 18.1 (定理)**

*设 $G$ 为 $n$ 顶点上的加权路径图，设 $L_P$ 特征值为 $0 = \lambda_1 < \lambda_2 \leqslant \cdots \leqslant \lambda_n$，设 $v_k$ 为 $\lambda_k$ 的特征向量。然后 $v_k$ 修改“$k-1$”。*

♡

第二个定理更有用 [64]:

**定理 18.2 (定理)**

*设 $G = (V, E, w)$ 为加权连通图，设 $L_G$ 为其拉普拉斯矩阵。令 $0 = \lambda_1 < \lambda_2 \leqslant \cdots \leqslant \lambda_n$ 的特征值 $L_G$ 让 $v_1, \cdots, v_n$ 为对应的特征向量。对于任何 $k \geqslant 2$，让 $W_k = \{i \in V : v_k(i) \geqslant 0\}$。那么由 $G$ 在 $W_k$ 上生成的图最多有 $k-1$ 个连通分量。*

♡

这一结果的重要结果是，特征向量的第一个非平凡特征值可以用来把图分割成两个连接的部分——一个是特征向量为正的节点，一个是特征向量为负的节点。这个特征向量被称为**费德勒向量**。邻接矩阵是非负的，对于这种类型的矩阵 [12] 有一个广泛的理论；见第 13.4 节的 Perron-Frobenius 定理。

一般来说，通过测量边数的比例，并不能保证这种分解有多好，但在实践中，可以证明这种行为是相当好的 [186]。

### 18.6.3 Cheeger 不等式

上面我们讨论了图拉普拉斯的第一个非平凡特征值与图分成两部分有关。齐格常数和齐格不等式将这个特征值与分割的某种质量度量联系起来。

设 $V$ 为顶点的集合，$S \subset V$，则定义一个图的 cheger 常数为

$$C = \min_S \frac{e(S, V-S)}{\min \operatorname{vol}(S), \operatorname{vol}(V-S)}$$

其中 $e(SV-S)$ 表示连接 $S$ 到 $V-S$ 的边数，一组节点的容量定义为

$$\operatorname{vol}(S) = \sum_{e\in S} d(e)$$

Cheeger 不平等是这样说的:

$$2C \geqslant \lambda \geqslant \frac{C^2}{2}$$

其中 $\lambda$ 是图拉普拉斯矩阵的第一个非平凡特征值。

# 第 19 章 自动机理论

自动机是机器的数学抽象。自动机有一个广泛的理论; 这里我们只讲基本概念。让我们从一个简单的例子开始。

## 19.1 有限状态自动机

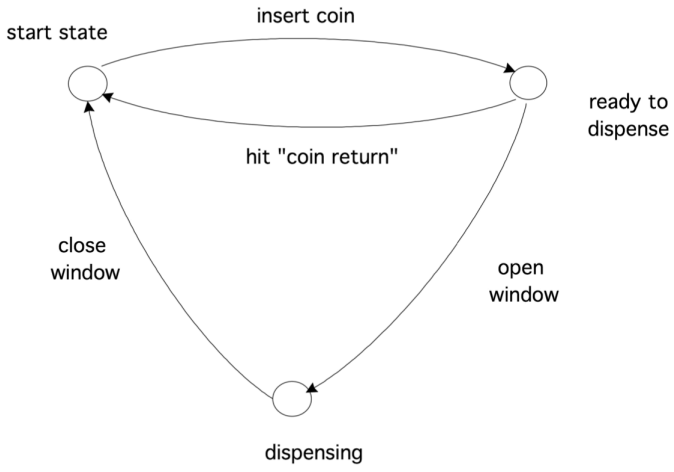
有限状态自动机（FSA）是一种非常简单的机器，它与自动售货机一样，插入硬币后就会自动吐出一块糖果棒。自动贩卖机有四种操作方式：插入硬币，按“硬币返还”键要求归还所有插入的钱，打开窗口取糖果棒，然后再次关闭窗口。一个操作（特别是第三个）是否可能取决于机器所处的状态。这里有三种状态：开始状态，四分之一被插入并且窗口被解锁的状态（让我们称之为“准备分配”），以及窗口打开的状态（我们称之为“分配”）。

在某些状态下，某些行为是不可能的。例如，在开始状态下，窗口不能打开。这台自动售货机的数学描述包括 1。州的列表，2。一个表格，显示可能的操作如何使机器从一种状态转到另一种状态。但是，与记录表格相比，图形表示通常更有洞察力。

## 19.2 一般讨论

从自动售货机的例子中，你可以看到自动售货机的一个重要特征: 它允许某些操作，但只在某些情况下，最终它有一个对应于“成功”的状态，这只有在采取某些操作序列时才会达到。

对这一过程的正式描述如下: 我们称个体行为为“字母表”，而基于该字母表的一系列行为为“单词”。然后，机器的“成功”结果对应于一个判断，即某个单词属于自动机器所接受的“语言”。因此，自动机理论与语言理论之间存在着某种对应关系。




练习 19.1 考虑 alpha$\{ab\}$，即只包含 $ab$ 的字母表，并考虑语言 $\{a^m b^n : mn > 0\}$，这是由一个或多个 $as$ 后跟一个或多个方根组成的单词。画出接受这种语言的自动机。

FSA 之所以是最简单的类型，是因为它没有内存。大多数自动贩卖机不会抱怨如果你放入超过 1 / 4 的硬币: 它们除了“被插入了 1 / 4”外没有任何记忆。更复杂的机器会计算你插入了多少枚 25 美分硬币，然后让你打开那么多窗口，看到不同的糖果棒。在上述正式的描述方式中，该机器将接受语言 $q^n w^n n \geqslant 0$，也就是说，您存入的硬币数 ('$q$') 与您打开的窗口数 ('$w$') 相同。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是常规语言。

这两种语言类型属于乔姆斯基语言的四级层次结构。识别递归可枚举语言类型的著名图灵机位于层次结构的顶层。缺失的步骤具有上下文敏感的语言类型，该类型由线性有界自动机识别。

# 第 20 章 并行前缀

为了使操作能够并行执行，它们需要是独立的。这使得并行计算递归是困难的。递归出现在很明显的地方，比如解三角方程组（第 5.3.5 节），但它们也可以出现在排序和许多其他操作中。

在本附录中，我们将介绍并行前缀操作: 并行执行由涉及关联操作符的递归定义的操作（请参阅第 6.10.2 节，了解并行化递归的“递归加倍”方法）。计算元素数组的和就是这种类型操作的一个例子（暂时忽略非结合性）。设 $\pi(xy)$ 为二进制和运算符：

$$\pi(x,y) \equiv x+y$$

然后定义 $n \geqslant 2$ 项的前缀和为

$$\Pi(x_1,\ldots,x_n) = \begin{cases} \pi(x_1,x_2) & \text{if} n=2 \\ \pi(\Pi(x_1,\ldots,x_{n-1}),x_n) & \text{otherwise} \end{cases}$$

作为一个非明显的前缀操作，我们可以计算具有特定属性的数组元素的数量。

**练习 20.1** 假设 $p()$ 为一个谓词，$p(x)=1$，如果对 $x$ 成立，则为 0。定义一个二元运算符 $\pi(xy)$，以便它对一个数字数组进行缩减，得到 $p$ 为真的元素数量。

现在假设存在一个关联操作符 $\oplus$，它是一个值为 $x_1 \cdots x_n$ 的数组。然后我们将前缀问题定义为 $X_1 \cdots, X_n$，

$$\begin{cases} X_1 = x_1 \\ X_k = \oplus_{i \leq k} x_i \end{cases}$$

## 20.1 并行前缀

并行化的关键是实现我们可以并行计算部分约简:

$$x_1 \oplus x_2, \quad x_3 \oplus x_4, \cdots$$

都是独立的。此外，这些削减的部分削减，

$$(x_1 \oplus x_2) \oplus (x_3 \oplus x_4), \cdots$$

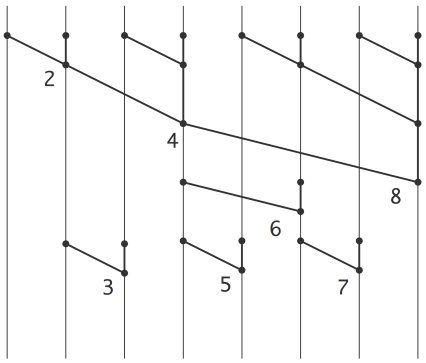
也是独立的。我们用这个符号

$$X_{i,j} = x_i \oplus \cdots \oplus x_j$$

对这些部分削减。

我们在之前的 2.1 节中已经看到: 可以在 $\lceil \log 2n \rceil$ 步骤中减少 $n$ 数字数组。要使这个操作成为一个完整的前缀操作，需要计算所有中间值。

观察一下，例如，$X_3 = (x_1 \oplus x_2) \oplus x_3 = X_2 \oplus x_3$，我们现在可以想象整个过程; 关于 8 个元素的情况，请参见图 20.1。要计算，比如 $X_{13}$，可以表示 $13 = 8+4+1$ 并进行计算 $x_{13} = x_8 \oplus X_{9,12} \oplus x_1 3$.



在此图中，相同“距离”上的操作与 SIMD 类型执行相对应，已经水平对齐。如果按照任务图执行，有些步骤可以在图中建议的时间之前执行; 例如 $X_3$ 可以与 $X_6$ 同时计算。

不管计算步骤的安排如何，不难看出整个前缀计算可以在 $2\log_2 n$ 步骤中完成：$\log_2 n$ 步骤用于计算最终约简 $X_n$，然后另一个 $\log_2 n$ 步骤用于填充中间值。

## 20.2 并行前缀的稀疏矩阵向量积

研究发现，稀疏矩阵向量积可以看作是一个前缀运算；参见 [16]。这里的推理是先计算所有 $y_{ij} \equiv a_{ij}x_j$，然后用前缀运算计算 $y_i = \sum_j y_{ij}$ 的和。

上面解释的前缀和不能计算正确的结果。前两个 $y_{ij}$ 术语的总和确实为 $y_1$，但是继续前缀 sum 得到 $y_1 + y_2$，而不是 $y_2$。使此工作的技巧是考虑双组件数量 $< y_{ij}s_{ij} >$，其中

$$s_{ij} = \begin{cases} 1 & \text{if } j \text{is the first nonzero index in row} i \\ 0 & \text{otherwise} \end{cases}$$

现在我们可以定义每次 $s_{ij} = 1$ 时的前缀和为’ reset ’。

## 20.3 Horner 法则

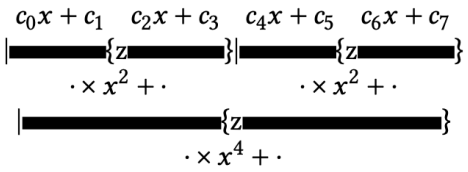
评估多项式的霍纳规则是一个简单递归的例子：

$$y = c_0x^n + \cdots + c_nx^0 \equiv \begin{cases} t_0 \leftarrow c_0 \\ t_i \leftarrow t_{i-1} \cdot x + c_i \quad i = 1, \ldots, n \\ y = t_n \end{cases}$$

或者写得更明确些

$$y = (((c_0 \cdot x + c_1) \cdot x + c_2) \cdots)$$

像许多其他递归一样，这个看似连续的操作可以并行：




然而，我们在这里看到需要一些聪明的方法：我们需要 $xx_2x_4$ 等来相乘子结果。霍纳氏法则上面解释的前缀和不能计算正确的结果。前两个 $y_{ij}$ 术语的总和确实为 $y_1$，但是继续前缀 sum 得到 $y_1 + y_2$，而不是 $y_2$。使此工作的技巧是考虑双组件数量 $< y_{ij}s_{ij} >$，其中将霍纳规则解释为前缀方案是失败的：“霍纳操作符”$hx(ab) = ax + b$ 很容易被认为是没有联想性的。从上面的按树计算中可以看出，我们需要携带并更新 $x$，而不是将其附加到操作符。

一些小实验证明了这一点

$$h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}\right) \equiv \begin{bmatrix} ay + b \\ xy \end{bmatrix}$$

可以达到：

$$h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right) = \left\{ \begin{array}{c} h\left(h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} b \\ y \end{bmatrix}\right), \begin{bmatrix} c \\ z \end{bmatrix}\right) = h\left(\begin{bmatrix} ay+b \\ xy \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right) \\ h\left(\begin{bmatrix} a \\ x \end{bmatrix}, h\left(\begin{bmatrix} b \\ y \end{bmatrix}, \begin{bmatrix} c \\ z \end{bmatrix}\right)\right) = h\left(\begin{bmatrix} a \\ x \end{bmatrix}, \begin{bmatrix} ayz+bz+c \\ yz \end{bmatrix}\right) \end{array} \right\} = \begin{bmatrix} ayz+bz+c \\ xyz \end{bmatrix}$$

这样我们就可以总结霍纳规则：

$$h\left(\begin{bmatrix} c_0 \\ x \end{bmatrix}, \begin{bmatrix} c_1 \\ x \end{bmatrix}, \ldots, \begin{bmatrix} c_n \\ x \end{bmatrix}\right)$$

顺便说一句，这种特殊形式的“霍纳运算符”对应于程序设计语言 APL 中的“rho”运算符，它通常被表述为具有变基数的数字系统的求值。

# 第四部分

# HPC Carpentry

# 写在 HPC 之前

如果编程仅仅包括设计算法与数据结构，那么前面的内容完全就可以概括，但现实往往并非如此。我们往往需要一些了解一些基础工具，以帮助我们成为一个高效的科学计算程序员。

绝大多数科学编程都是在 Unix 平台上完成的，所以我们将首先讲述 Unix，并在随后的章节中讲解编译器和连接器如何处理代码等内容。

接下来，我们将了解一些可以提高生产力和效率的工具：

- 实用的 **Make** 用于管理项目的构建；
- 源码控制系统以一种可以撤消更改或维护多个版本的方式存储代码；第 23 章中，我们将看到 **subversion** 软件。
- 一旦程序开始运行产生结果，数据的存储和交换就变得十分重要；在第 24 章我们将学习 **HDF5** 的使用。
- 程序数据的可视化：这个话题太宽泛，无法在此详细讨论；第 25 章我们将了解 **gnuplot** 包的基础知识，适用于简单的数据绘图。

我们还考虑了程序设计本身：第 26 章我们将考虑如何避免编程错误，第 27 章将会使用 **gdb** 调试代码。第 28 章包含一些关于如何编写使用多种编程语言的程序。

最后，第 30 章我们会讲述有关 LaTeX 文档系统的知识，以便我们在工作报告中排版出精美的文章。

这些教程的非常实用，快来学习吧！

# 第 21 章 Unix 简介

## 21.1 Unix 简介

Unix 是一种**操作系统**（Operating System，OS），即介于用户或用户程序与硬件之间的软件层。它负责文件和屏幕输出，并确保许多进程可以并存在一个系统上。但是，它不会立即对用户可见。我们使用 Unix 的大多数时候，我们都是在输入由 shell 的解释器执行的命令。Shell 使实际操作系统调用。可能一共有几种 Unix shell 可用，但在本教程中，我们假设我们是使用 sh 或 bash shell，尽管许多命令对于现有的各种 shell 是通用的。

本教程的大部分内容都适用于任何类 Unix 平台，但是 Unix 并非只有一个：

- 传统的有几种主要风格的 Unix：ATT 和 BSD。苹果的达尔文更接近 BSD；IBM 和 HP 有自己版本的 Unix，Linux 就是另一个变体。它们之间存在着巨大的差异，因此我们正在学习本教程时，可能很长时间看不到它们。
- Linux 中有各种 Linux 发行版，例如 Red Hat 和 Ubuntu。这些主要是在系统文件的组织存在不同，因此不必对它们担心。
- 正如刚才所提到的，有不同的 shell，它们确实有很大不同。在这里我们会学习 bash shell，它是旧的 sh shell 的改进版本。由于种种原因，bash 优于 csh 和 tcsh shell。其他 shell 是 ksh 和 zsh，它们是本身是对 bash shell 的改进。

## 21.2 文件及相关

**目的**：了解由存储文件目录组成的 Unix 文件系统，了解用于显示数据文件的可执行文件和命令。

### 21.2.1 查看文件

**目的**：学习用于显示文件内容的命令。

| 命令 | 作用 |
|---|---|
| ls | 列出当前目录下的文件和文件夹 |
| touch | 创建新文件或者更新已存在的文件的时间 |
| cat > filename | 将文本输入文件 |
| cp | 拷贝文件 |
| mv | 重命名文件 |
| rm | 删除文件 |
| file | 声明文件类型 |
| cat filename | 展示文件 |
| head,tail | 展示部分文件 |
| less,more | 渐进展示文件 |

#### 21.2.1.1 ls

没有任何参数，*ls* 命令为我们提供了当前位置中的文件列表。

✍ **练习 21.1** 输入 *ls* 会展示什么?

预期结果：如果目录中有文件，则将列出它们：如果没有，则不会给出输出结果。这是标准的 Unix 行为：没有输出并不意味着出了问题，只意味着没有什么可报告的。

✍ **练习 21.2** 如果 *ls* 命令显示有文件，则选择其中一个文件上输入 *ls name*。使用一个选项，例如，*ls - s name*，我们可以得到更多的文件名称。

**注释** 如果我们指定了不存在的文件的名称，我们将得到报错信息。

#### 21.2.1.2 cat

*cat* 命令通常用于显示文件，但它也可用于创建一些简单的内容。

✍ **练习 21.3** 输入 *cat > newfilename*（可以选择任意文件名）然后输入一些文本。使用 *Control-d* 独占一行来表示终止，现在使用 *cat* 来展示文件内容：*cat newfilename*

预期结果：在 *cat* 的第一次使用中，文本被从终端串联到一个文件。第二种情况下，文件被 *cat* 到终端输出。我们在屏幕上应该可以准确地看到我们输入到文件中的内容。

注意事项：请确保在最后一行的输入是 *Control-d* 。如果卡住，可以使用 *Control-c* 解决。试着用 *cat > filename* 创建一个文件然后再在某一行的中间按 *Control-c*。文件的内容是什么?

**注释** 我们经常会看到用 *D* 的符号代替 *Control -d*。大写字母是出于历史原因：我们使用控制键和小写字母。

#### 21.2.1.3 man

Unix 命令的主要来源（尽管可能看起来很麻烦）是 *man* 命令，意思是'' 手册''。通过这种方式获得的描述被称为手册页。

✍ **练习 21.4** 阅读 *ls* 命令的手册页：*man ls*。找出一些文件的大小和时间/日期（例如我们刚刚创建的文件）。我们是否发现了 *ls-sandls-loptions*? 第一个选项列出了每个文件的大小，通常以千字节为单位，另一个选项给出了关于一个文件的各种信息，包括我们以后要学习的东西。

预期结果：找到 *ls -s* 和 *ls -l* 选项了吗? 第一个选项列出了每个文件的大小通常是以千字节为单位，另一个选项给出了关于文件的各种信息，包括我们以后要学习的东西。

注意事项：*man* 命令使我们进入查看长文本文件模式。这种查看器在 Unix 系统中很常见（它可以作为 *more* 和 *or* 系统命令使用），所以记住以下命令：使用空格键前进，*u* 键后退；用 *g* 键到文本的开头，*G* 键到文本的结尾，*q* 键退出。如果我们被卡住，可以使用 *Control-c* 。

**注释** 有一些与文件相关的日期，对应着内容的变化、权限的变化和任何形式的访问。stat 命令给出了所有这些日期。

**注释** 如果我们已经知道我们要找的是什么命令，我们可以用 *man* 来获取关于在线信息。如果我们忘记了某个命令的名字，*man -k* 关键字可以帮助我们找到它。

#### 21.2.1.4 touch

*touch* 命令用于创建一个空文件，或者使用 *ls -l* 命令更新一个已经存在的文件的时间戳。

#### 21.2.1.5 cp, mv, rm

*cp* 命令可以用来复制一个文件（或目录）：*cp file1 file2* 复制了一个 file1 的副本，并将其命名为 file2 。

✍ **练习 21.5** 使用 *cp file1 file2* 来复制一个文件。确认这两个文件有相同的内容。如果我们改变原始文件，复制的文件会发生变化吗?

预期结果：我们应该看到，如果原始文件改变或被删除，副本不会发生变化。
注意事项：如果 file2 已经存在，我们会得到一个报错信息。
文件可以用 *mv* 重命名，*mv* 的意思是”move’’。

✍ **练习 21.6** 重命名一个文件。如果目标名称已经存在，会发生什么?

文件可以用 *rm* 删除。这个命令很危险，原因是没有撤消命令。

#### 21.2.1.6 head, tail

还有更多的命令用于显示一个文件、一个文件的部分内容或关于一个文件的信息。

✍ **练习 21.7** 执行 *ls /usr/share/words* 或 *ls /usr/share/dict/words* 来确认我们的系统中存在一个带有字的文件存在。现在用这个文件试验一下 *head*、*tail*、*more* 和 *wc* 命令。

预期结果：*head* 显示文件的前几行，*tail* 显示最后一行，而 *more* 使用的是与 *man* 页相同的查看器。阅读这些命令的 *man* 页并尝试增加和减少输出的数量。*wc*（’word count’）命令报告一个文件中的字数、字符数和行数。另一个有用的命令是 *file*：它告诉我们正在处理的是什么类型的文件。

✍ **练习 21.8** 对不同的 *foo* 执行 *file foo*：一个文本文件，一个目录，或*/bin/ls* 命令。

预期结果：有些信息我们可能无法理解，但要注意的词是要注意的是’’ 文本’’、’’ 目录’’ 或’’ 可执行’’。在这一点上，建议我们学会使用文本编辑器，如 emacs 或 vi 。

### 21.2.2 目录

目的：将学习 Unix 目录树的操作和使用方法。

| 命令 | 作用 |
|---|---|
| ls | 列出目录的内容 |
| mkdir | 创建新的目录 |
| cd | 改变目录 |
| pwd | 展示当前工作目录 |

一个 Unix 文件系统是一棵目录树，其中一个目录是文件或更多目录的容器。我们将以如下方式显示目录

```
/ ..........................................目录树的根
bin.........................................二进制程序
home........................................用户目录的地址
```

Unix 目录树的根部用斜线表示。执行 *ls /*来查看根目录下有哪些文件和目录看看根目录下有哪些文件和目录。注意，根目录不是我们重新启动我们的个人电脑时开始的位置，也不是我们登录服务器时开始的位置，而是登录到服务器时开始的位置。

✍ **练习 21.9** 查找我们当前工作目录的命令是 *pwd*。我们的主目录是我们在登录时的工作目录，找出我们的主目录。

**预期结果**：我们通常会看到类似 */home/yourname* 或*/Users/yourname* 的东西，结果取决于系统。

输入 *ls* 来查看工作目录的内容。在本节的显示中，目录名称后面会有一个斜杠：*dir/*，但这个字符不是其名称的一部分。我们可以通过使用 *ls -F* 来获得这种输出，我们可以通过在会话开始时声明 *alias ls = ls -F* 来告诉我们的 shell 持续

```
/home/you/
--adirectory/
--afile
```

建立一个新目录的命令是 *mkdir*

✍ **练习 21.10** 用 *mkdir newdir* 建立一个新目录，用 ls 查看当前目录

**预期结果**：我们应该看到这种结构：

```
/home/you/
                newdir/.............................新目录
```

- *cd* ('change directory') 命令是进入另一个目录，也就是把它变成我们的工作目录。它可以在以下方面使用：
- *cd* 没有任何参数，cd 将我们带到我们的主目录。
- *cd* 一个绝对路径从目录树的根部开始，也就是说，从 */*. 开始 *cd* 命令带我们到那个位置。
- *cd* 一个相对路径是指不从根开始的路径。这种形式的的 *cd* 命令将我们带到/。

✍ **练习 21.11** 使用 *cd newdir* ，用 pwd 查出我们在目录树中的位置。用 *ls* 确认确认该目录是空的。我们将如何使用绝对路径到达这个位置?

预期结果：*pwd* 应该告诉我们 /home/you/newdir，然后 *ls* 没有输出。意味着没有任何东西可以列出。绝对路径是 /home/you/newdir。

✍ **练习 21.12** 让我们在这个目录下快速创建一个文件：*touch onefile*，和另一个目录：*mkdir otherdir*。做 *ls* 并确认有一个新的文件和目录。

预期结果：我们应该会得到如下结果

```
/home/you/
newdir/.............................我们在这
onefile
otherdir/
```

*ls* 命令有一个非常有用的选项：通过 *ls -a*，我们可以看到我们的普通文件和隐藏文件，隐藏文件的名称以点开头。在我们的新目录中进行 *ls -a* 操作，我们会发现有以下文件：

```
/home/you/
newdir/.............................我们在这
.
..
onefile
otherdir/
```

单点是当前目录，双点是后一级的目录。

✍ **练习 21.13** 预测我们在 *cd ./otherdir/..* 之后的位置，并检查我们的判断是否正确。

预期结果：单点将我们送到当前目录，所以这并没有改变任何东西。otherdir 部分使该子目录成为我们的当前工作目录。最后，.. 回到了上一级，换句话说，这个命令让我们回到我们开始的地方。

由于我们的主目录是一个特殊的地方，所以有一些 *cd* 的快捷方式：*cd* 不带参数、*cd ~* 和 *cd $HOME* 都能让我们回到我们的 home 目录。

进入我们的主目录，在那里做 ls newdir 来检查我们创建的第一个目录的内容，而不需要去那里。

✍ **练习 21.14** *ls ...* 是做什么的?

预期结果：回顾一下，.. 表示树上的一级目录：我们应该看到我们自己的主目录，以及任何其他用户的目录。

✍ **练习 21.15** 我们能用 *ls* 来查看其他用户的主目录的内容吗？在前面的练习中我们看到我们的系统中是否存在其他用户。如果是的话，就用 *ls .../thatotheruser* 。

预期结果：如果这是我们的私人电脑，我们可能可以查看其他用户的目录内容。其他用户的目录内容。如果这是一台大学的计算机，那么另一个目录很可能是受保护的。保护权限问题将在下一节讨论--我们会得到：../ otheruser: Permission denied 尝试用 *cd* 移动到别人的主目录，起作用吗?

我们可以用 cp 复制一个目录，但我们需要添加一个标志，以表明我们递归地复制了内容：*cp -r*。在我们的 home 目录中建立另一个目录 somedir ，这样我们就有了以下结构：

```
/home/you/
newdir/.............................you have been working in this one
somedir/.....................................you just created this one
```

*cp -r newdir somedir* 和 *cp -r newdir thirddir* 的区别是什么（thirddir 是一个不存在的目录名)?

### 21.2.3 权限

**目的**：在这一节中，我们将了解到如何给系统中的各种用户提供对我们的文件执行（或不执行）各种事务的权限。

Unix 文件，包括目录，都有权限，表明'' 谁可以对这个文件做什么''。可以对文件进行的操作可以对一个文件进行的操作分为三类：

- 读 r：对文件的任何访问（显示、获取文件的信息），但不改变文件。
- 写 w：对文件的访问，改变其内容，或甚至其元数据，如'' 修改日期''。

- 执行 x：如果文件是可执行的，可以运行它；如果它是一个目录，可以进入它。有可能访问一个文件的人也被分为三类。
- 用户 u：拥有该文件的人。
- 组 g：所有者所属的用户组。
- 其他 o：其他所有人。这九种权限是按顺序呈现的

| user | group |
|---|---|
| rwx | rwx |

例如，*rw-r--r--*意味着所有者可以读写一个文件，所有者的组和其他人都只能读文件。权限也是以三比特为一组的数字表示的，*r=4*，*w=2*，*x=1*。

常见的代码是 7 = rwx 和 6 = rw 。我们会发现许多文件的权限是 755，这代表每个人都可以运行的可执行文件，但只有所有者可以改变。或者 644 代表一个每个人都可以看到的数据文件，但同样只有所有者可以改变。我们可以用 *chmod* 命令来设置权限：

```
chmod  file     # just one file
chmod -R  directory     # directory, recursively
```

示例如下：

```
chmod 766 file  # set to rwxrw-rw-
chmod g+w file  # give group write permission
chmod g=rx file     # set group permissions
chod o-w file   # take away write permission from others
chmod o= file    # take away all permissions from others.
chmod g+r,o-x file  # give group read permission # remove other execute permission
```

*man* 页给出了所有的选项。

**练习 21.16** 建立一个文件 foo ，并执行 *chmod u-r foo*。我们现在能检查它的内容吗？让这个文件再次可读，这次使用数字代码。现在让我们的同学也能读到这个文件，检查方法是让他们中的一个人查看文件内容。

预期结果：当我们把文件设置为不可读时，我们仍然可以 *ls* 文件，但是不能 *cat* 文件：*cat* 时会给出权限拒绝的信息。创建一个 *com* 文件，内容如下：

```
#!/bin/sh
echo: "Hello world!"
```

这是一个合法的 shell 脚本。当我们输入./com 时会发生什么？我们能使这个脚本可执行吗？在这三个权限类别中，’’ 我们’’ 和’’ 其他人’’ 指的是谁很清楚。那么’’ 组’’ 呢？我们将在第 20.10 节中讨论这个问题。

**注释** 还有更多不明显的权限。例如，*setuid* 位声明程序应该以创建者的权限运行，而不是执行它的用户的权限。这对系统实用程序，如 *passwd* 或 *mkdir*，它们改变了密码文件和目录结构，因此需要 root 权限。多亏了 setuid 位，一个用户才可以运行这些程序，这些程序被设计为用户只能分别对自己的密码文件和自己的目录进行修改。编码 setuid 位是用 *chmod* 设置的：*chmod 4ugo* 文件。

我们已经看到 *ls filename* 给我们提供了关于那个文件的信息，而 *ls* 给我们提供了当前目录下的所有文件。为了查看名字上有某些条件的文件，存在通配符机制。通配符机制如下：

| | |
|---|---|
| * | 可代表任意数量的字符。 |
| ? | 可代表任意单个字符 |

例如：

```
%% ls
s         sk         ski         skiing         skill
%% ls ski
ski skiing skill
```

第二个选项列出了所有文件名以 *ski* 开头，后面是任意数量的其他字符的文件'。下面我们会看到，在不同的情况下，*ski* 的意思是" sk 后面有任意数量的 *i* 字符''。现象令人困惑，但事实就是这样的。

## 21.3 文本搜索和正则表达式

**目的**：在本节中，我们将学习如何在文件中搜索文本。对于这一部分，我们至少需要一个包含一定数量文本的文件。例如，我们可以从以下网站获得随机的文本：http://www.lipsum.com/feed/html。

*grep* 命令可以用来搜索文件中的文本表达式。

**练习 21.17** 在我们的文本文件中用 *grep q yourfile* 搜索字母 q，并在我们目录的所有文件中用 *grep q* 搜索它。尝试一些其他的搜索。

预期结果：在第一种情况下，我们得到一个包含 q 的所有行的列表；在第二种情况下，*grep* 还报告了在哪个文件名中发现了匹配的内容：qfile--这一行有 q 在里面。

注意事项：如果我们要找的字符串没有出现，*grep* 将根本不输出任何东西，这是 Unix 命令的标准行为。除了搜索字面字符串外，我们还可以寻找更多的一般表达式。

| 字符 | 作用 |
|---|---|
| ^ | 行首 |
| $ | 行末 |
| . | 任意字符 |
| * | 任意数量的字符 |
| [*xyz*] | 任意字符例如 xyz |

这看起来像我们刚才看到的通配符机制（第 20.1.4 节），但它有细微的不同。与上面的例子相比：

```
%% cat s
sk
ski
skill
skiing
```

```
%% grep "ski*" s
  sk
 ski
  skill
   skiing
```

在第二种情况下，我们搜索一个由 sk 和任何数量的 i 字符组成的字符串，包括零个字符的字符串。还有一些例子：我们可以找到

- 所有包含字母"q" 的行，用 *grep q yourfile*。

- 所有以'a' 开头的行，用 *grep "a" yourfile*（如果我们的搜索字符串包含特殊字符，最好使用引号将其括起来）。
- 所有以数字结尾的行，用 *grep "[0-9]$" yourfile*。

✍ **练习 21.18** 构建搜索字符串，用于查找

- 以大写字母开头的行，以及
- 恰好包含一个字符的行。

预期结果：对于第一种情况，使用范围字符 [ ]，对于第二种情况，使用句号来匹配任何字符。

✍ **练习 21.19** 添加几行 $x = 1$, $x = 2$, $x = 3$（也就是说，在 $x$ 和等号之间有不同数量的空格）到我们的测试文件中。并使用 *grep* 命令来搜索所有对 $x$ 的赋值。

上表中的字符有特殊含义。如果我们想搜索那个实际的字符，我们必须对它进行转义。

✍ **练习 21.20** 做一个测试文件，其中有 abc 和 a.c，分别在不同的行中。尝试使用命令 *grep"a.c" file*，*grep a\\.c file*，*grep "a\.c " file*。

预期结果：我们会看到，句号需要转义，搜索字符串需要加引号。如果没有这两种情况，我们会看到 *grep* 也能找到 abc 字符串。

#### 21.3.0.1 用 sed 编辑流

Unix 有各种工具用于逐行处理文本文件。流媒体编辑器 sed 就是一个例子。如果我们使用过 vi 编辑器，我们可能已经习惯了像 *s/foo/bar/*这样的语法来进行修改。有了 *sed* ，我们可以在命令行上做这些事情。比如说：

```
sed 's/foo/bar/' myfile > mynewfile
```

将把替代命令 *s/foo/bar/* 应用到 myfile 的每一行。输出会显示在我们的所以我们应该在一个新文件中捕获它；关于输出重定向的更多信息，请参见第 20.3.2 节。

### 21.3.1 用 cut 的方式切起线条

另一个编辑行的工具是 *cut*，它将切行并显示它的某些部分。例如：

```
cut -c 2-5 myfile
```

将显示 myfile 中每一行的第 2-5 位的字符，制作一个测试文件并验证这个例子。也许更有用的是，我们可以给 *cut* 一个分割符，让它在出现分割符的情况下分割一个行。例如，我们的系统很可能有一个*/etc/passwd* 文件，其中包含用户的信息，以每一行都由冒号分隔的字段组成。例如：

```
daemon::1:1:System Services:/var/root:/usr/bin/false
nobody::-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root::0:0:System Administrator:/var/root:/bin/sh
```

第七个也是最后一个字段是用户的登录 shell；*/bin/false* 表示用户无法登录。我们可以用以下方法显示用户和他们的登录 shell。

```
cut -d ":" -f 1,7 /etc/passwd
```

这告诉 *cut* 使用冒号作为分隔符，并输出字段 1 和 7。

## 21.4 其他有用的命令：tar

*tar* 命令代表’’ 磁带归档’’，也就是说，它最初是用来将文件打包到磁带上的（’ 存档’’ 部分来自于 ar 命令）。现在，它被用来将文件打包在一起，以便在网站上发布：如果您想发布一个包含数百个文件的库，它将它们捆绑到一个文件中。最常见的两个选项是

1. 创建 *tar* 文件：

```
tar fc package.tar directory_with_stuff
```

声明一个 *tar file create*

2. 解压 *tar* 文件：

```
tar fx package.tar
# this creates the directory that was packaged
```

文本文件通常可以被压缩到很大程度，所以为 *gzip* 添加 z 压缩是一个好主意：

```
tar fcz package.tar.gz directory_with_stuff
tar fx package.tar.gz
```

命名为”*gzip*’’ 文件包。*tgz* 也很常见。

## 21.5 命令的执行

### 21.5.1 搜索路径

**目的**：在这一节中，我们将学习 Unix 如何决定当我们输入一个命令名时，Unix 是如何决定做什么的。如果我们输入了 *ls* 这样的命令，shell 并不只是依靠一个命令列表：它实际上会去搜索一个名为 *ls* 的程序。这意味着我们可以有多个同名的不同命令。这意味着我们可以有多个同名的不同命令，哪一个被执行取决于哪一个先被找到。

✍ **练习 21.21** 我们可能认为的”Unix 命令’’ 往往只是系统目录中的可执行文件，输入 *which ls*，然后输入 *ls -l* 查看结果。

预期结果：*ls* 的位置是类似于 /bin/ls 的东西。如果我们输入了 *ls* 命令，我们会看到它可能是由 root 拥有的。它的可执行位可能是为所有用户设置的。unix 搜索命令的位置是搜索路径，它被存储在环境的变量（更多细节见下 PATH）。

✍ **练习 21.22** 输入 *echo $PATH*。我们能找到 *cd* 的位置吗？是否有其他命令在相同的位置吗？当前目录’.’ 在路径中吗？如果没有，做 *export PATH=".:$PATH"*。现在创建一个可执行文件 *cd*（见上面的基础知识），然后执行 *cd* 。

预期结果：路径将是一个用冒号分隔的目录列表。例如，*/usr/bin:/usr/local/bin:/usr/X11R6/bin*。如果工作目录在路径中，它可能会在最后。*/usr/X11R6/bin:*。但通常情况下工作目录不在路径中。如果我们在路径的开头加上’.’，unix 会在系统命令之前找到本地的 *cd* 命令。

有些人认为将工作目录放在路径中是一种安全风险。如果我们的目录是可写的，有人可以在我们的目录中放置一个名为 *cd* 的恶意脚本（或任何其他系统命令），而我们就会在不知不觉中执行它。

可以将我们自己的命令定义为现有命令的别名。

✍ **练习 21.23** 使用 *alias chdir=cd*，确保现在 *chdir* 的工作方式和 *cd* 一样。使用 *alias rm='rm -i'*；在手册中查找它的含义。有些人认为这个别名是个好主意；我们能明白为什么吗?

预期结果：*rm* 的*-i*'' 交互式'' 选项使命令每一次删除之前都要求确认。由于 unix 没有一个需要明确清空的垃圾桶（如 Windows 或 Mac OS 上的），这可能是一个好主意。

### 21.5.2 命令序列

### 21.5.3 重定向

**目的**：在本节中，我们将学习如何将一条命令输入另一条命令，以及如何将命令与输入和输出文件连接起来。

到目前为止，我们所使用的 unix 命令都是从我们的键盘上，或从一个名为的文件；它们的输出到了我们的屏幕上。还有其他的可能性，即从文件中提供输入或将输出存储在一个文件中的其他可能性。

### 21.5.4 命令排序

有多种方法可以在一个命令行上有多个命令。

#### 21.5.4.1 简单排序

首先，我们可以输入：

```
command1 ; command2
```

如果我们多次重复相同的两个命令，这很方便：我们只需要向上述的方式一次就可以重复这两个命令。但是有一个问题，如果我们输入：

```
cc -o myprog myprog.c ; ./myprog
```

然后编译失败，程序仍然会被执行，如果存在旧版本的可执行文件，则使用该版本的存在。这是非常令人困惑的，更好的办法是：

```
cc -o myprog myprog.c && ./myprog
```

这条命令只有在第一条命令成功的情况下才会执行第二条命令。

#### 21.5.4.2 管道

不从文件中获取输入，也不向文件发送输出，而是将两个命令连接在一起，使第二个命令将第一个命令的输出作输入。这方面的语法是 *cmdone* | *cmdtwo*。这就是所谓的管道。例如，*grep a yourfile* | *grep b* 可以找到所有同时包含 a 和 b 的行。

✍ **练习 21.24** 构造一个管道，计算文件中包含字符串 *th* 的行数。使用 *wc* 命令（参见上面）进行计数。

#### 21.5.4.3 反引用

还有一些组合命令的方法。假设我们想把 wc 的结果以一种很好地呈现。输入以下命令：

```
echo The line count is wc -l foo
```

其中 foo 是一个现有文件的名称。获得实际行数输出的方法是通过反引用：

```
echo The line count is ‘wc -l foo ‘
```

在反引用之间的任何内容都会在命令行的其他部分被评估之前被执行。

✍ **练习 21.25** 这里使用 *wc* 的方式是，它打印文件名。你能找到防止这种情况发生的方法吗?

还有另一种无序评估机制：

```
echo "There are $( cat Makefile | wc -l ) lines"
```

#### 21.5.4.4 子 shell 中分组

假如我们想把输出重定向应用于一连串的几个命令中：

```
configure ; make ; make install > installation.log 2>&1
```

这只抓住了最后一条命令。例如，我们可以把这三条命令放在一个子 shell 里，然后捕获它的输出。

```
( configure ; make ; make install ) > installation.log 2>&1
```

这种机制使嵌套命令成为可能，但出于兼容性和遗留目的，当不需要嵌套时，倒引号可能仍然更可取。

### 21.5.5 退出状态

命令可能会失败。如果我们在命令行上键入一条命令会看到错误，并在键入下一条命令时采取相应的行动，当我们输入下一条命令时，我们就会采取相应的行动。当这个失败的命令发生在一个脚本中时，我们必须告诉脚本如何采取相应的行动。为此，我们可以使用命令的退出状态：这是一个值（成功时为零，否则为非零）存储在一个内部变量中，我们可以用 $? 访问它。

示例：假设我们有一个不可写的目录如下

```
[testing] ls -ld nowrite/
dr-xr-xr-x 2 eijkhout 506 68 May 19 12:32 nowrite//
[testing] cd nowrite/
```

然后写

```
[nowrite] cat ../newfile
#!/bin/bash
touch $1 echo "Created file: $1"
[nowrite] newfile myfile
bash: newfile: command not found
[nowrite] ../newfile myfile
touch: myfile: Permission denied
Created file: myfile
[nowrite] ls
[nowrite]
```

脚本将报告说文件被创建了，尽管它并没有被创建。改进的脚本：

```
[nowrite] cat ../betterfile
#!/bin/bash touch $1
if [ $? -eq 0 ] ; then
    echo "Created file: $1"
else
    echo "Problem creating file: $1"
fi
[nowrite] ../betterfile myfile
touch: myfile: Permission denied
Problem creating file: myfile
```

### 21.5.6 进程和工作

| 命令 | 简介 |
|---|---|
| ps | 列出（所有）进程 |
| kill | 杀死一个进程 |
| CTRL -c | 杀死前台的工作 |
| CTRL -z | 挂起前台的工作 |
| jobs | 给出所有工作的状态 |
| fg | 将最后一个暂停的工作带到前台 |
| fg %3 | 将一个特定的工作带到前台 |
| bg | 在后台运行最后一个暂停的工作 |

Unix 操作系统可以同时运行许多程序，方法是在列表中轮流运行，每次只给每个程序几分之一的时间。每次只给每个程序几分之一秒的时间来运行。命令 ps 可以告诉我们当前正在运行的所有程序。

**练习 21.26** 输入 *ps*。目前有多少个程序在运行？默认情况下，*ps* 只给出我们明确启动的程序我们明确启动的程序。使用 *ps guwax* 可以获得所有正在运行的程序的详细列表。有多少个程序正在运行？有多少属于根用户，有多少属于我们？

预期结果：要计算属于某个用户的程序，可以用管道将 *ps* 命令通过一个适当的 *grep* ，然后将其输送到 *wc*。在这个长长的 ps 列表中，第二列包含了进程号。有时，拥有这些进程号是很有用的：如果一个程序行为不正常，我们可以使用以下命令杀死：

```
kill 123456
```

其中，12345 是进程号。

上面解释过的 cut 命令可以从一行中剪切某些位置：输入 *ps guwax | cut -c 10-14* 要获得所有运行进程的动态信息，可以使用 *top* 命令，通过阅读手册来了解如何按 CPU 使用率对输出进行排序。

在 shell 中启动的进程被称为 *jobsjob*（unix）。除了进程号之外，它们还有也有一个作业号。现在我们将探讨如何操作作业。

当我们输入一条命令并按下返回键时，该命令在其运行期间成为前台的进程。其他同时运行的东西都是后台进程。制作一个可执行文件 hello，内容如下：

```
\#!/bin/sh
while [ 1 ] ; do
    sleep 2
    date
done
```

然后输入 *./hello* 。

✍ **练习 21.27** 输入 *Control-z*。这个命令会将前台进程暂停，它将给我们一个数字，如 [1] 或 [2]，表明它是第一个或第二个被暂停或置于后台的程序。现在输入 bg，把进程放到后台。确认没有前台进程，点击返回键，然后输入 *ls* 命令。

预期结果：在我们把一个进程放到后台后，终端又可以接受前台命令。如果我们点击回车键，我们应该看到命令提示符。然而。后台进程仍然不断地产生输出。

✍ **练习 21.28** 输入 *jobs* 来查看当前会话中的进程。如果我们刚才放在后台的进程是 1 号，输入 *fg %1*，确认它又是一个前台进程。

预期结果：如果 shell 在前台执行一个程序，它将不接受命令输入，所以点击返回键应该只产生空行。

✍ **练习 21.29** 当我们使 hello 脚本再次成为前台进程时，我们可以用 *Control-c* 杀死它。尝试这样：再次启动该脚本，这次是以 *./hello &* 的形式，这样就可以立即把它放到后台。我们也应该得到类似于 [1] 12345 的输出，它告诉我们这是我们放在后台的第一个工作，而 12345 是它的进程 ID。用 kill %1 杀死这个脚本。再次启动它，通过它的进程号。

预期结果：使用进程号的 kill 12345 命令通常足以来杀死一个正在运行的程序，但有时需要使用 *kill -9 12345*。

### 21.5.7 Shell 自定义

上面提到 *ls -F* 是查看哪些文件是常规文件、可执行文件或目录的一个简单方法。通过输入别名 *ls='ls -F'*，*ls* 命令每次被调用时都会自动扩展为 *ls -F*。如果我们想在每个登录会话中都有这种行为，我们可以将别名命令添加到我们的*.profile* 文件中。除了 *sh/bash* 之外，其他 shells 也有其他的文件可以进行这样的自定义。

## 21.6 输入/输出重定向

**目的**：在本节中，我们将学习如何将一个命令提供给另一个命令，以及如何将命令连接到输入和输出文件。到目前为止，我们使用的 unix 命令都是从我们的键盘或命令行上命名的文件中获取输入；他们的输出到你的屏幕上。从文件中提供输入或将输出存储在文件中还有其他可能性。

#### 21.6.0.1 输入重定向

*grep* 命令有两个参数，第二个参数是一个文件名。我们也可以写 *grep string < yourfile*，其中的小于号意味着输入将来自命名的文件，即 yourfile，这就是所谓的输入重定向。

### 21.6.1 标准文件

Unix 有三个标准文件来处理输入和输出：

| 标准文件 | 目的 |
|---|---|
| stdin | 为进程提供输入的文件。 |
| stdout | 是写入进程输出的文件。 |
| stderr | 是写入错误输出的文件。 |

在交互式会话中，所有三个文件都连接到用户终端。使用输入或输出重定向意味着将输入或输出发送到与终端不同的文件。

#### 21.6.1.1 输出重定向

反之，*grep string yourfile* > *outfile* 会把通常进入终端的内容重定向输出到 outfile。如果输出文件不存在，就会被创建，否则就会被覆盖掉。(如果要追加，使用 *grep text yourfile* >> *outfile* )。

✍ **练习 21.30** 从上一节中选取一个 *grep* 命令，并将其输出发送到一个文件。检查该文件的内容是否与之前出现在屏幕上的内容相同。搜索一个文件中没有出现的字符串，并将其输出发送到一个文件中。这对输出文件意味着什么?

预期结果：搜索一个不在文件中出现的字符串，没有终端输出。如果我们把这个 *grep* 的输出重定向到一个文件，会给出一个零大小的文件。使用 *ls* 和 *wc* 命令进行检查。

有时我们想运行程序，但忽略输出。为此，您可以将输出重定向到系统空设备：*/dev/null*

```
yourprogram >/dev/null
```

下面是一些常用的习语：

| 习语 | 含义 |
|---|---|
| program 2>/dev/null | 只发送错误到空设备 |
| program >/dev/null 2>&1 | 将输出发送到 dev-null，并输出错误。注意违反直觉的规范顺序! |
| program 2>&1 \|less | 发送输出和错误到更少 |

## 21.7 Shell 环境变量

在上面我们遇到了 PATH，它是一个 shell 或环境变量的示例。这些变量是 shell 所知道的，shell 所运行的所有程序都可以使用它们。虽然 PATH 是一个内置变量，但您也可以定义自己的变量，并在 shell 脚本中使用它们。Shell 变量大致分为以下几类：

- 特定于 shell 的变量，如 HOME 或 PATH。
- 特定于某些程序的变量，例如 TEX/LATEX 的 TEXINPUTS。
- 自己定义的变量；见下。
- 控制结构定义的变量，例如 for；见下文。

我们可以通过键入 *env* 查看 *shell* 已知的所有变量的完整列表。

注释 这并不包括你自己定义的变量，除非我们导出它们；见下文。

✍ **练习 21.31** 通过输入 *echo $HOME*，检查 *HOME* 变量的值。也可以通过 *grep* 管道输入 *env* 来查找 HOME 的值。

### 21.7.1 shell 变量的使用

我们可以通过在 shell 变量前面加上一个美元符号来获得它的值。输入以下内容并检查输出：

```
echo x
echo $x
x=5
echo x
echo $x
```

可以看到，shell 将所有内容都视为一个字符串，除非我们通过在名称前面加上 $ 来显式地告诉它取一个变量的值。以前没有定义的变量将打印为空白字符串。Shell 变量可以通过多种方式设置。与其他编程语言一样，最简单的方法是赋值。当进行下一个练习时，最好记住 shell 是一种基于文本的语言。

✍ **练习 21.32** 在命令行上输入 *a=5* 。这就定义了一个变量 *a* ；通过使用 *echo* 命令检查其值。

预期结果：shell 将通过输入的数值 5 来响应。
注意事项：注意等号周围不要有空格；也要注意用美元符号来打印该值。

### 21.7.2 导出变量

这样设置的变量会被我们在这个 shell 中发出的所有后续命令所知，但不会被我们启动的新 shell 中的的命令所知。为此，我们需要使用导出命令。重现以下内容会话（方括号内为命令提示）：

```
[] a=20
[] echo $a
20
[] /bin/bash
[] echo $a
[] exit
exit
[] export a=21
[] /bin/bash
[] echo $a
21
[] exit
```

我们也可以临时设置一个变量。重现这个场景：

1. 找到一个没有数值的环境变量：

```
[] echo
$b []
```

2. 写一个简短的 shell 脚本来打印这个变量：

```
[] cat > echob
#!/bin/bash
echo $b
```

当然，还要使其可执行：*chmod +x echob* 。

3. 现在调用该脚本，在其前面设置变量 b：

```
[] b=5 ./echob
5
```

我们设置值的语法，作为一个前缀而不使用单独的命令，设置值只为这一条命令。

4. 证明该变量仍然是未定义的：

```
[] echo $b
[]
```

也就是说，我们定义了这个变量只是为了执行一个单一的命令。

## 21.8 控制结构

像任何好的编程系统一样，shell 有一些控制结构。它们的语法需要一点时间来适应习惯 (不同的 shell 有不同的语法；在本教程中我们只讨论 bash shell)。

### 21.8.1 条件语句

bash shell 的条件语句可以被称为 *if*，它可以写成几行上：

```
if [ $PATH = "" ] ; then
    echo "Error: path is empty"
fi
```

或在单行上：

```
if [ `wc -l file ` -gt 100 ] ; then echo "file too long" ; fi
```

有一些测试被定义，例如 *-f somefile* 测试一个文件是否存在。更改我们的脚本，使它在文件不存在时报告 -1。这方面的语法是很微妙的:

- if 和 elif 后面是一个条件，后面是一个分号
- 条件的括号周围需要有空格
- else 的 then 后面没有分号

✍ **练习 21.33** Bash 条件语句有一个 elif 关键字。但我们仍然可以写出以下序列 else if，如：

```
if [ something ] ; then
    foo
else if [ something_else ] ; then
    bar
fi
```

我们能预测这里的错误是什么吗?

### 21.8.2 循环

一个 for 循环的形式如下：

```
for var in listofitems ; do
    something with $var
done
```

这将以以下方式工作:

- 对于 listofitems 中的每个项目，变量 var 被设置为该项目
- 并且循环主体被执行。

举一个简单的例子:

```
[] for x in a b c ; do echo $x ; done
a
b
c
```

有一个更有意义的例子，以下是我们如何对所有的.c 文件进行备份:

```
or cfile in .c ; do
cp $cfile $cfile.bak
done
```

Shell 变量可以以多种方式进行操作。执行下面的命令，我们就会发现可以删除变量的尾部字符:

```
[] a=b.c
[] echo ${a%.c}
b
```

以此为提示，写一个循环，将所有的 *.c* 文件重命名为 *.x* 文件。上面的结构很容易在单词上进行循环，比如 *ls* 的输出。要进行数字循环，可以使用命令 *seq* :

```
[shell:474] seq 1 5
1
2
3
4
5
```

循环处理一个数字序列，通常如下:

```
for i in `seq 1 ${HOWMANY}` ; do echo $i ; done
```

注意回车键，这是为了在评估循环之前执行 seq 命令所必需的。

## 21.9 脚本

Unix shells 也是编程环境，我们将在本节中进一步了解 Unix 的这一方面。

### 21.9.1 如何执行脚本

可以编写 Unix shell 命令的程序：首先，我们需要知道如何把一个程序放在一个文件中，并让它被执行。做一个包含以下两行的文件 script1:

```
#!/bin/bash
echo "hello world"
```

并在命令行上输入./script1 。结果如何？使该文件可执行，然后再试一次。

为了编写我们想从任何地方调用的脚本，人们通常会把它们放在主目录下的 bin 目录中。然后我们会把这个目录添加到我们的搜索路径中，包含在 PATH 中；见第 20.3.1 节。

### 21.9.2 脚本参数

我们可以用选项和参数来调用一个 shell 脚本：

```
./my_script -a file1 -t -x file2 file3
```

现在我们将学习如何在我们的脚本中纳入这一功能。首先，所有的命令行参数和选项都可以在脚本中作为变量 $1 ,2′#’ 来表示：

```
#!/bin/bash
echo "The first argument is $1"
echo "There were $# arguments in all"
```

官方定义：

| 变量 | 含义 |
|---|---|
| $# | 参数个数 |
| $0 | 脚本名称 |
| $1，$2，... | 参数 |
| $ ，$@ | 所有参数的列表 |

✍ **练习 21.34** 编写一个脚本，将一个文件名作为输入参数，并报告该文件中有多少个行。

编辑我们的脚本以测试文件是否少于 10 行 (使用 *foo -lt bar* 测试)，如果是的话，就使用 cat 查看该文件。提示：我们需要在测试中使用反引号。在我们的脚本中添加一个测试，如果我们在没有任何参数的情况下调用它，它将给出一个有用的信息。

解析参数的标准方法是使用 shift 命令，它将从参数列表中弹出第一个参数。然后，依次解析参数包括查看 *$1* 、移动和查看新的 *$1* 。代码：

```
// arguments.sh
while [ $# -gt 0 ] ; do
    echo "argument: $1"
        shift
done
```

✍ **练习 21.35** 写一个 say.sh 脚本，输出它的文本参数。

然而，如果我们调用它：

```
./say.sh -n 7 "Hello world"
```

它应该是按照我们指定的次数来输出的。使用选项-u:

```
./say.sh -u -n 7 "Goobye cruel world"
```

应该以大写字母输出该信息。确保参数的顺序并不重要，并对任何不被识别的选项给出一个错误信息。变量 *$@* 和 *$* 在双引号方面有不同的行为。比方说，我们评估 myscript ”1 2” 3, 然后

- 使用 *$** 是去掉引号后的参数列表：myscript 1 2 3。
- 使用”$*” 是去除引号后的参数列表：myscript ”1 2 3’’。
- 使用”$@” 保留了引号：myscript ”1 2” 3。

## 21.10 扩展

shell 对命令行进行各种扩展，即用不同的文本替换命令行的一部分。花括号扩展：

```
[] echo a{b,cc,ddd}e
abe acce addde
```

例如，这可以用来删除一些基本文件名的所有扩展名：

```
[] rm tmp.{c,s,o}    # delete tmp.c tmp.s tmp.o
```

波浪号扩展给出了我们自己的，或别人的主目录：

```
[] echo
/share/home/00434/eijkhout
[] echo eijkhout
/share/home/00434/eijkhout
```

参数扩展给出了 shell 变量的值：

```
[] x=5
[] echo $x
5
```

未定义的变量不会出现错误信息：

```
[] echo $y
```

在参数扩展方面有许多变化。上面我们已经看到，我们可以剥离尾部的字符：

```
[] a=b.c
[] echo ${a%.c}
b
```

以下是处理未定义变量的方法：

```
[] echo ${y:-0}
0
```

反引号机制被称为命令替换。它允许我们对一条命令的一部分进行评估，并将其作为另一条命令的输入。例如，如果我们想查看文件是什么类型，请执行 ls 命令：

```
[] file `which ls`
```

这首先评估了 *which ls*，给出了 */bin/ls*，然后评估 *file /bin/ls*。另一个例子，这里我们反引用整个管道，并对结果做一个测试：

```
[] echo 123 > w
[] cat w
123
[] wc -c w
       4 w
[] if [ `cat w | wc -c` -eq 4 ] ; then echo four ; fi
four
```

Unix shell 编程在很大程度上是面向文本操作的，但也有可能进行算术。算术替换告诉 shell 将一个参数扩展作为一个数字来处理：

```
[] x=1
[] echo $((x2))
2
```

整数范围可按以下方式使用：

```
[] for i in {1..10} ; do echo $i ; done
1
2
3
4
5
6
7
8
9
10
```

## 21.11 启动文件

在本教程中，我们已经看到了几种自定义 shell 行为的机制。比如说，通过设置 *PATH* 变量，我们可以扩展 shell 寻找可执行文件的位置。其他环境变量我们可以为自己的目的引入。这些定制的变量中的许多需要适用于每个会话，所以我们可以在任何会话开始时读取 shell 启动文件。

在启动文件中最普遍的事是定义 *alias*：

```
alias grep=' grep -i'
alias ls=' ls -F'
```

并设置一个自定义的命令行提示。

不巧的是，有几个启动文件的读取是一个复杂的情况功能。这里有一个很好的常识性指南：

- 有一个 *.profile* 文件，除了读取*.bashrc* 文件之外什么都不做：

```
# /.profile
if [ -f /.bashrc ]; then
    source /.bashrc
fi
```

- 我们的*.bashrc* 文件会进行实际的自定义：

```
\# /.bashrc
\# make sure your path is updated
if [ -z "$MYPATH" ]; then
    export MYPATH=1
    export PATH=$HOME/bin:$PATH
fi
```

## 21.12 交互式 Shell

Unix 的交互式使用，与编写脚本不同，是用户和 shell 之间复杂的对话。我们作为用户输入一行按下回车，然后 shell 试着解释。有以下几种情况：

- 我们输入行只包含一个完整的命令，例如 *ls foo*： shell 将执行这个命令。
- 我们可以在一行中输入一个以上的命令，用分号隔开：*mkdir foo ; cd foo*，shell 将按顺序执行这些命令。
- 我们的输入行不是一个完整的命令，例如 *while [1]*。shell 会识别出识别出还有更多的命令，并使用一个不同的提示来显示它正在等待命令的剩余部分。
- 我们的输入行是一个合法的命令，但我们想在第二行输入更多的内容。在这种情况下在这种情况下，我们可以用一个反斜杠字符来结束我们的输入行，shell 将知道它需要暂缓执行我们的命令。但实际上，反斜杠会隐藏（逃避）返回。

当 shell 通过使用我们的一个或多个输入行或只使用一个输入行的一部分来收集要执行的命令行时，如刚才所述，它将对命令行进行扩展。然后，它将把命令行解释为一个命令和参数，并继续用找到的参数来调用该命令。

有一些微妙的地方。如果我们输入 ls .c，那么 shell 会重新识别通配符并将其扩展为一个命令行，例如 ls foo.c bar.c 。注意，ls 不接受.c 作为参数！在我们想让 unix 命令接收一个带有通配符的参数情况下，我们需要转义它，这样 shell 就不会将它扩展。例如，*find . -name \.c* 将使 shell 调用带有参数.-name .c 的 find 。

## 21.13 系统和其他用户

Unix 是一个多用户操作系统。因此，即使我们在自己的个人设备上使用它，我们也是一个有账户的用户，偶尔需要输入我们的用户名和密码。

如果我们是在我们的个人设备上，我们可能是唯一登录的用户。在大学机器或其他服务器上，经常会有其他用户，这里有一些与他们有关的命令。

| 命令 | 作用 |
|---|---|
| whoami | 显示我们的登录名 |
| who | 显示当前登录的其他用户 |
| finger otheruser | 获取另一个用户的信息；我们可以在这里指定一个用户的登录名，或他们的真实姓名，或系统知道的其他 |
| top | 哪些进程正在系统上运行；使用 top -u 来获得这个排序的 cpu 数量时间。(在 Linux 上，也可以尝试使用 v |
| uptime | 距离上一次重启的时间 |

### 21.13.1 组

当我们的账户被创建时，我们的系统管理员会把我们分配到一个或多个组。(如果我们管理自己的机器，我们会在一些默认的组中；请继续阅读，以便将自己添加到更多的组种)。

*groups* 命令告诉我们所在的组，*ls -l* 告诉我们每个文件属于哪个组。与 chmod 类似，我们可以使用 *chgrp* 来改变一个文件所属的组，以便与该组中的用户共享。

创建一个新的组，或将一个用户添加到一个组，需要系统权限。创建一个组命令如下：

```
sudo groupadd new_group_name
```

添加一个用户到一个组命令如下：

```
sudo usermod -a -G thegroup theuser
```

### 21.13.2 超级用户

即使我们拥有我们的设备，也尽可能地使用普通用户的帐户工作，只有在严格需要时才使用 root 权限 ( root 账户也被称为超级用户)。如果我们有 root 权限，我们也可以用它来'' 成为另一个用户''，用他们的的特权，使用 *sudo*（'*superuser do*'）命令。

- 以另一个用户的身份执行一个命令：

```
sudo -u otheruser command arguments
```

- 以根用户身份执行命令：

```
sudo command arguments
```

- 成为另一个用户：

```
sudo su - otheruser
```

- 成为超级用户：

```
sudo su -
```

## 21.14 其他系统：SSH 和 SCP

没有人是一座孤岛，计算机也是如此。有时我们相用我们的笔记本电脑去链接超级计算机，如果我们已经在一台 Unix 计算机上，我们可以用' 安全 shell' 的命令 ssh 登录到另一台计算机。它是旧的'' 远程 shell'' 命令 rsh 的一个更安全的变种：

```
ssh yourname@othermachine.otheruniversity.edu
```

其中 yourname 可以省略，如果我们在两台机器上有相同的名字的话。如果只想从一台设备复制一个文件到另一台设备，我们可以使用'' 安全复制"scp，这是'' 远程复制" rcp 的一个安全变体。scp 命令的语法与 cp 很相似，只是源文件或目标文件有一个设备前缀。要将一个文件从当前设备复制到另一台设备，请输入：

```
scp localfile yourname@othercomputer:otherdirectory
```

其中 yourname 也可以省略，otherdirectory 可以是一个绝对路径，也可以是一个相对于我们的主目录的路径的相对路径。

```
# absolute path:
scp localfile yourname@othercomputer:/share/
# path relative to your home directory:
scp localfile yourname@othercomputer:mysubdirectory
```

把目标路径留空，会把文件放在远程主目录下：

```
scp localfile yourname@othercomputer:
```

注意这个命令结尾的冒号：如果我们把它漏掉，就会得到一个名称中带有'at' 的本地文件。我们也可以从远程设备上复制一个文件，例如，要复制一个文件，保留名称：

```
scp yourname@othercomputer:otherdirectory/otherfile .
```

## 21.15 sed 和 awk 工具

除了相当小的实用工具，如 tr 和 cut ，Unix 还有一些更强大的工具，在本节中我们将看到两个对文本文件进行逐行转换的工具。

### 21.15.1 sed

流媒体编辑器 sed 就像一个远程控制的编辑器，用一个命令行接口做简单的行编辑。大多数时候，我们会按以下方式使用 sed：

```
cat somefile | sed ' s/abc/def/:g' > newfile
```

(这里使用 cat 并不是严格意义上的需要。) s/abc/def/部分的效果是在每一行中用 def 替换 abc；g 修改器将其应用于每一行中的每一个实例，而不仅仅是第一行。

- 如果我们有一个以上的编辑，我们可以用：

  ```
  sed -e ' s/one/two/' -e ' s/three/four/'
  ```

- 如果一个编辑只需要在某些行上进行，我们可以通过在编辑的前缀加上匹配字符串。比如说：

  ```
  sed ' /a/s/b/c/'
  ```

  只对以 a 开头的行进行编辑（见第 20.2 节正则表达式）。
- 传统上，sed 只能在一个流中起作用，所以输出文件总是必须与输入不同。GNU 版本，即 Linux 系统的标准版本，有一个标志-i，可以就地进行编辑： sed -e 's/ab/cd/' -e 's/ef/gh/' -i thefile

### 21.15.2 awk

*awk* 工具也是对每一行进行操作，但它可以说是有一个内存。一个 *awk* 程序由一连串的对子组成，每个对子由一个匹配字符串和一个行为组成。最简单的 *awk* 程序是:

```
cat somefile | awk ' { print }'
```

其中匹配字符串被省略，意味着所有的行都是匹配的，并且动作是打印该行。*awk* 将每一行分成由空白分隔的字段，*awk* 的一个常见应用是打印某个字段：

```
awk ' {print $2}' file
```

输出每一行的第二个字段。假设我们想打印一个 Fortran 程序中的所有子程序，可以通过以下方法完成：

```
awk ' /subroutine/ {print}' yourfile.f
```

✍ **练习 21.36** 建立一个命令管道，在每个子程序的标题下只输出子程序的名称。为此，我们首先用 sed 把括号换成空格，然后用 *awk* 来打印子程序名称域。*awk* 有一些变量，它可以记住一些东西。例如，我们可以将每一行的第二个字段打印出来，而不是只打印的第二个字段，我们可以把它们做成一个列表，然后再输出：

```
cat myfile | awk ' BEGIN {v="Fields:"} {v=v " " $2} END {print v}'
```

作为使用变量的另一个例子，下面是如何如何 BEGIN 和 END 行之间的所有行：

```
cat myfile | awk ' /END/ {p=0} p==1 {print} /BEGIN/ {p=1} '
```

✍ **练习 21.37** 与 BEGIN 和 END 匹配的位置可能看起来很奇怪。重新安排 *awk* 程序，进行测试，并解释我们得到的结果。

## 21.16 复习题

✍ **练习 21.38** 假设我们是一个教授，为提交作业编写一个脚本：如果一个学生调用这个脚本，它就会把学生的文件复制到某个标准位置。

```
submit_homework myfile.txt
```

为了简单起见，我们通过建立一个提交目录和两个不同的文件 student1.txt 和 student2.txt 进行模拟。然后

```
submit_homework student1.txt
submit_homework student2.txt
```

在提交的目录中应该有这两个文件的副本。开始写一个简单的脚本；如果我们用错了方法，它应该给出一个有用的信息。

尝试检测一个学生是否作弊。探索 *diff* 命令，看看提交的文件是否与已提交的文件相同：循环查看所有已提交的文件然后

1. 首先输出所有的差异；
2. 计算差异个数；
3. 测试该计数是否为零。

现在通过捕捉作弊学生是否随机插入了一些空格来完善我们的测试。对于一个更难的测试：尝试检测作弊的学生是否插入了换行。这不可能不能用 diff 来做，但我们可以尝试用 *tr* 来删除换行线。

# 第 22 章 编译器和库

## 22.1 当我们运行一个程序时，会执行什么?

**目的**：在本节中，我们将了解在编程时遇到的不同类型的文件。

### 22.1.1 源与程序

有两种类型的编程语言：

1. 解释型语言：我们写的是人可读的源代码，我们直接执行它：计算机在遇到我们的源代码时逐行翻译。
2. 编译语言：我们的整个源代码首先被编译成一个程序，然后我们再执行它。

解释型语言的例子有 *Python*、*Matlab*、*Basic*、*Lisp*。解释型语言有一些优势：通常我们可以在一个交互式环境中编写它们，使我们可以非常迅速地测试代码。它们还允许我们在运行时动态地构建代码。然而，所有这些灵活性是有代价的：如果一个源码行被执行两次，它就会被翻译两次。那么，在本书中，我们将侧重于编译语言，用 *C* 和 *Fortran* 作为原型例子。

因此，现在我们已经区分了可读的源代码和不可读的、但可执行的程序代码。在本教程中，我们将了解从一个到另一个的翻译过程。进行这种翻译的程序被称为编译器，编译器内部发生的事情是计算机科学的另一个分支。可以这么说，本教程将是一本编译器的'' 用户手册''。

### 22.1.2 二进制文件

程序可以会非常大，所以我们不想为每一个改动都重新编译所有的东西：通常我们会把我们的源文件分成多个文件，并分别编译它们。

首先，一个源文件可以被编译成一个目标文件，这有点像一个可执行文件的一部分：它本身什么都不做，但它可以和其他文件结合起来。它本身不做任何事情，但它可以与其他目标文件结合起来，形成一个可执行文件。

如果我们有一个目标文件的集合，而我们需要在一个以上的程序中使用，通常是一个好主意，即做一个库：一个可以用来形成可执行文件的目标文件包。

通常情况下，库是由专家编写的含有专门用途的代码，如线性代数操作。库重要到它们可以是用于商业，我们可以购买我们需要某种目的的专业代码。

现在我们将学习这些类型的文件是如何创建和使用的。

## 22.2 简单汇编

**目的**：在本节中，我们将学习可执行文件和目标文件。

### 22.2.1 编译器

我们把源代码变成程序的主要工具是编译器，编译器是专门针对一种语言的。我们对 *C* 语言使用不同的编译器，而对 *Fortran* 使用不同的编译器，我们也可以为同一种语言有两个来自不同的供应商的编译器。例如，虽然许多人使用开源的 *gcc* 或 *clang* 编译器系列，而像英特尔和 *IBM* 这样的公司提供的编译器可能会在其处理器上提供更有效的代码处理器上提供更有效的代码。

### 22.2.2 编译单个文件

让我们从一个简单的程序开始，它的全部源代码都在一个文件中。一个 *hello.c* 文件：用我们喜欢的编译器编译这个程序；本教程中我们将使用 gcc。但也可以根据需要使用我们自己的编译器，编译的结果是创建了一个

文件 *a.out*，这就是可执行文件。

```
%% gcc hello.c
%% ./a.out
hello world
```

我们可以用 *-o* 选项得到一个更合理的程序名称：

```
%% gcc -o helloprog hello.c
%% ./helloprog
hello world
```

### 22.2.3 多个文件：编译和链接

现在我们转到一个不止一个源文件中的程序。

主程序：*fooprog.c*

```
#include<stdlib.h>
#include<stdio.h>
extern void bar(chzar *);
int main() {
    bar("hello world\n");
    return 0;
}
```

子程序：*fooprog.c*

```
#include<stdlib.h>
#include<stdio.h>
void bar(char *s) {
    printf("%s",s);
    return;
}
```

和以前一样，我们可以用一个命令来运行程序。

**Output**

**[compile/c] makeoneprogram:**

```
clang -o oneprogram fooprog.c foosub.c
./oneprogram
hello world
```

然而，我们也可以分步进行，分别编译每个文件，然后把它们连接起来。

**Output**

**[compile/c] makeseparatecompile:**

```
clang -o oneprogram fooprog.o foosub.o
./oneprogram
hello world
```

*-c* 选项告诉编译器对源文件进行编译，并给出一个目标文件。第三条命令不止作为链接器，将目标文件连接成一个可执行文件。(对于分布在多个文件中的程序，总是存在编辑子程序定义的危险，然后忘记更新使用过的的地方。参见教程第 22 节，了解处理这个问题的方法)。

每个目标文件都定义了一些程序名称，并使用了一些在其中未定义的其他名称，但这些名称将在其他目标文件或系统库中被定义，但在其他目标文件或系统库中会被定义。使用 *nm* 命令来显示这些:

```
[c:264] nm foosub.o
0000000000000000 T _bar
                 U _printf
```

带 *T* 的行表示已定义的程序；带 *U* 的行表示已使用但未在本文件中定义的程序。在这种情况下，*printf* 是一个在链接器阶段提供的系统程序。

## 22.3 库

**目的**：在本节中，我们将学习 Unix 中的库。

如果我们写了一些子程序，并且我们想与其他人分享它们（或是售卖它们），那么交出单个目标文件是不方便的。相反，解决方案是将它们合并成一个库。首先，我们看一下归档工具 *ar* 使用的静态库，一个静态库被链接到我们的可执行文件中，成为它的一部分可能会导致可执行文件变大；接下来我们会了解到共享库，它不存在这个问题。

创建一个包含我们的库的目录（取决于我们的库是什么，这可以是一个系统的目录，如*/usr/bin* ），并在那里创建库文件。

**Output**

**[compile/c] makestaticlib:**

```
Making static library

for o in foosub.o ; do \
    ar cr libs/libfoo.a ${o} ; \
done
```

*nm* 命令告诉我们库中有什么，就像它对对象文件所做的那样，但现在它还告诉我们哪些对象文件在库中:

```
%% nm ../lib/libfoo.a

../lib/libfoo.a(foosub.o):
00000000 T _bar
         U _printf
```

该库可以通过明确给出其名称或指定一个库的路径来链接到我们的可执行文件中:

**Output**

**[compile/c] makestaticlib:**

```
for o in foosub.o ; do \
    ar cr libs/libfoo.a ${o} ; \
done

Linking to static library
```

```
clang -o staticprogram fooprog.o -Llibs -lfoo

.. note the size of the program

-rwxr-xr-x 1 eijkhout staff 8464 Jan 23 13:07 staticprogram

.. running:

hello world
```

虽然使用起来比较复杂，但共享库有几个优点。例如，由于它们没有被链接到可执行文件中，而只是在运行时被加载，所以它们会导可执行文件更小。它们不是用 *ar* 创建的，而是通过编译器创建的：

**Output**

**[compile/c] makedynamiclib:**

```
Making shared library

clang -o libs/libfoo.so -shared foosub.o
```

我们可以再次使用 *nm* 命令：

```
%% nm ../lib/libfoo.so

../lib/libfoo.so(single module):
00000fc4 t \__dyld_func_lookup _
0000000 t \_mh_dylib_header
00000fd2 T _bar U _printf
00001000 d dyld__mach_header
00000fb0 t dyld_stub_binding_helper*
```

共享库实际上并没有被链接到可执行文件中；相反，可执行文件需要的信息是在执行时找到库的位置。一种方法是用 *LD_LIBRARY_PATH* 来做到这一点：

**Output**

**[compile/c] dynamicprogram:**

```
Linking to shared library

clang -o libs/libfoo.so -shared foosub.o
clang -o dynamicprogram fooprog.o -Llibs -lfoo

.. note the size of the program

-rwxr-xr-x 1 eijkhout staff 8432 Jan 23 13:07 dynamicprogram

.. note unresolved link to a library
```

```
make[3]: ldd: No such file or directory
make[3]: [run_dynamicprogram] Error 1 (ignored)

.. running by itself:

hello world

.. running with updated library path:

LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:./libs ./dynamicprogram
hello world
```

另一个解决方案是让路径包含在可执行文件中:

```
%% gcc -o foo fooprog.o -L../lib -Wl,-rpath, `pwd `/../lib -lfoo
%% ./foo
hello world*
```

现在的链接行包含了库的路径两次:

1. 一次是-L 指令，这样链接器就可以解析所有的引用，以及
2. 另一次是使用链接器指令 *-Wl,-rpath,'pwd'/. . . /lib* 将路径存储到可执行文件中，以便在运行时可以找到它。

使用 *ldd* 命令来获取关于我们的可执行文件使用了哪些共享库的信息。(在 *Mac OS X* 上，使用 *otool -L* 代替)。

# 第 23 章 Make 与 Makefile

## 23.1 使用 Make 命令管理项目

*Make* 工具可以帮助我们管理项目的构建：它的主要任务是方便重建一个多文件项目中只需要重新编译或者重建的部分。这可以节省大量的时间，因为它可以用一个文件的编译来取代长达几分钟的完整安装。*Make* 还可以帮助维护一个程序在一台机器上的多个安装，例如用多个编译器编译一个库，或者在调试和优化模式下编译一个程序。

*Make* 是一个历史悠久的 *Unix* 工具，传统上有一些行为略有不同的变种，例如在不同的 *Unix* 版本上，如 *HP-UX*、*AUX*、*IRIX*。现在，无论在什么平台上，都建议使用 *GNU* 版的 *Make*，它有一些非常强大的扩展。它在所有的 *Unix* 平台上都可用（在 *Linux* 上它是唯一可用的变体），而且它是一个事实的标准。该手册可在 http://www.gnu.org/software/make/manual/make.html，或者我们可以阅读这本书。

还有其他的构建系统，最引人注目的是 *Scons* 和 *Bjam*。我们不在这里讨论这些。本教程中的例子是针对 *C* 和 **Fortran** 语言的，但 *Make* 可以在任何语言中工作，而事实上，像 LaTeX 这样的根本就不能算得上一种语言；见第 22.6 节。

#### 23.1.0.1 C

*Make* 下面的文件：*foo.c* 与 *bar.c*

```
#include "bar.h"
int c=3;
int d=4;
int main()
{
int a=2;
return(bar(a*c*d));
}
```

```
#include "bar.h"
        int bar(int a){
              int b=10;
             return(b*a);
            }
```

*bar.h*

```
extern int bar(int);
```

*Makefile*

```
fooprog : foo.o bar.o
        cc -o fooprog foo.o bar.o
foo.o : foo.c
        cc -c foo.c
bar.o : bar.c
        cc -c bar.c
clean :
        rm -f *.o fooprog
```

makefile 有许多规则，如：

```
foo.o : foo.c
<TAB>cc -c foo.c
```

其一般形式为:

```
target : prerequisite(s)
<TAB>rule(s)
```

其中规则行以 TAB 字符缩进。

如上所述的规则指出一个'' 目标" 文件 *foo.o* 是由'' 先决条件"*foo.c* 生成的，即通过执行 *cc -c foo.c* 命令。该规则的精确定义是:

- 如果目标文件 *foo.o* 不存在或者比先决条件文件 *foo.c* 存在时间更早。
- 则执行该规则的命令部分：*cc -c foo.c*
- 如果先决条件本身是另一条规则的目标，那么该规则将首先被执行。

也许解释规则的最好方法是:

- 如果任何先决条件发生了变化，
- 那么目标就需要重新 make，
- 而这是通过执行该规则的命令来完成的，
- 检查先决条件需要递归应用 make:
  - 如果先决条件不存在，找到一个规则来创建它;
  - 如果前提条件已经存在，检查适用的规则，看是否需要重新制作。

✍ **练习 23.1** 调用 *make* 命令

预期结果：应用上述规则：不带参数的 *make* 命令尝试构建第一个目标，*fooprog*，建立这个目标需要原先不存在的先决条件 *foo.o* 和 *bar.o*。然而，有一些可以生成它们的规则，*make* 会递归地调用这些规则。因此我们可以看到两个编译，分别为 *foo.o* 和 *bar.o* ，以及一个针对 *fooprog* 的链接命令。注意事项: *makefile* 或文件名中的错别字会导致各种错误，特别是要确保我们在规则行中使用制表符而不是空格。调试一个 makefile 文件并不简单，*make* 会给我们发送 *make* 文件中发现错误的行号信息。

✍ **练习 23.2** 执行 *make clean*，然后执行 *mv foo.c boo.c* 后再执行 *make* 。解释一下错误信息，并恢复原来的文件名。

预期结果：*make* 会报错说没有按照规则来构建 *foo.c*，这个错误是由于 *foo.c* 是构建 *foo.o* 的先决条件，但是 *foo.c* 并不存在。*make* 后去寻找构建 *foo.c* 的规则，但不存在创建 *.c* 文件的规则。

现在给函数 *bar* 添加第二个参数，这需要我们编辑 *bar.c* 和 *bar.h* ： 编辑后并 *make* 它们。然而，这也要求我们编辑 *foo.c*，但现在让我们'' 忘记'' 这样做，我们将看到 *make* 如何帮助我们找到所产生的错误。

✍ **练习 23.3** 调用 *make* 来重新编译我们的程序，它是否重新编译了 *foo.c* ?

预期结果：即使在概念上 *foo.c* 需要被重新编译，因为它使用了 bar 函数，但 *make* 并没有这样做，因为 *makefile* 中没有任何规则强制它。

在 makefile 中，改变这一行

```
foo.o : foo.c
```

为

```
foo.o : foo.c bar.h
```

这意味着在这种情况下，*foo.o* 已经存在。*make* 将检查 *foo.o* 是否比它的任何先决条件都要早，由于 *bar.h* 已经被编辑过，它比 *foo.o* 更早，所以 *foo.o* 需要被重构。

✍ **练习 23.4** 确认新的 *makefile* 确实使 *foo.o* 在 *bar.h* 被改变时被重新编译。这个编译现在会出现错误，因为我们已经'' 忘记'' 编辑 *bar* 函数的使用。

### 23.1.1 Fortran

*make* 以下文件：

*foomain.F*

```
program test
use testmod
call func(1,2)
end program
```

*foomod.F*

```
module testmod
contains
subroutine func(a,b)
integer a,b
print *,a,b,c
end subroutine func
end module
```

和一个 *makefile*: *Makefile*

```
fooprog : foomain.o foomod.o
    gfortran -o fooprog foo.o foomod.o
foomain.o : foomain.F
    gfortran -c foomain.F
foomod.o : foomod.F
    gfortran -c foomod.F
clean :
    rm -f *.o fooprog
```

调用 *make*，*makefile* 中的第一条规则被执行，解释会发生什么。

✍ **练习 23.5** 调用 *make*。

预期结果：应用上述规则：调用没有参数的 *make* 构建第一个目标，即 *foomain* ，构建它需要先前不存在的先决条件 *foomain.o* 和 *foomod.o*。然而，有一些规则可以用来生成它们，*make* 会递归地调用这些规则。因此，我们可以看到两个编译，分别为 *foomain.o* 和 *foomod.o*，还有一个链接命令为 *fooprog* 的链接命令。

注意事项：*makefile* 或文件名中的错别字会导致各种错误，调试一个 makefile 并不简单，我们将不得不理解并改正这些错误。

✍ **练习 23.6** 先执行 *make clean*，然后执行 *mv foomod.c boomod.c* 后再执行 *make*。解释一下错误的信息并恢复原来的文件名。

预期结果：make 会报错说没有构建 *foomod.c* 的规则，由于 *foomod.c* 作为 *foomod.o* 的先决条件不存在。make 后去寻找构建它的规则，但不存在构建 *.F* 文件的规则。

现在在 *foomod.F* 中给 *func* 添加一个额外的参数，然后重新编译。

✍ **练习 23.7** 调用 *make* 来重新编译我们的程序，它是否重新编译了 *foomain.F* ？

预期结果：即使从概念上讲，*foomain.F* 也需要被重新编译。但 *make* 并没有这样做，因为 *makefile* 中没有任何规则强迫它。

更改这一行

```
foomain.o : foomain.F
```

为

```
foomain.o : foomain.F foomod.o
```

将 *foomod.o* 添加为 *foomain.o* 的先决条件。这意味着，在这种情况下，*foomain.o* 已经存在的情况下，*make* 将检查 *foomain.o* 是否比它的任何先决条件都早。然后，*make* 将递归地检查 *foomode.o* 是否需要更新。在重新编译 *foomode.F* 后，*foomode.o* 比 *foomain.o* 更晚存在，所以 *foomain.o* 将被重构。

✍ **练习 23.8** 确认修正后的 *makefile* 确实使 *foomain.F* 被重新编译。

### 23.1.2 关于 make 文件

*make* 文件被称为 *makefile* 或 *Makefile*；名称的文件在同一目录下是不可取的。如果我们想让 *Make* 使用一个不同的文件作为 *make* 文件，请使用语法 *make -f My_Makefile*。

## 23.2 变量和模板规则

在我们的 *makefile* 中引入变量是很方便。例如，不需要每次都明确说明编译器的明确说明，而是在 makefile 中引入一个变量：

```
CC = gcc
FC = gfortran
```

并在编译行中使用 *CC * *{FC}* 。

```
foo.o : foo.c
        ${CC} -c foo.c
foomain.o : foomain.F
        ${FC} -c foomain.F
```

✍ **练习 23.9** 按照指示编辑我们的 *makefile* 文件。首先执行 *make clean*，然后执行 *make foo* (C) 或 *make fooprog*(Fortran)。

预期结果：我们应该看到和以前完全一样的编译和链接行。

注意事项：不像在 shell 中，大括号是可选的，makefile 中的变量名必须在大括号或小括号中。试验一下，如果我们忘记了变量名周围的大括号，会发生什么？

使用变量的一个好处是，我们现在可以从命令行上改变编译器：

```
make CC="icc -O2"
make FC="gfortran -g"
```

✍ **练习 23.10** 按照建议调用 *make*（在 *make clean* 之后），我们看到我们的屏幕输出有什么不同吗？

预期结果：编译行现在显示添加的编译器选项 *-O2* 或 *-g*。*make* 也有局部作用域变量：@ * * 先决条件的列表。在程序的链接行中也使用这个。< ** * 在模板规则中（第 22.2.2 节），这与模板部分相匹配，即与% 对应的部分。使用这些变量，*fooprog* 的规则变为：

```
fooprog : foo.o bar.o
        ${CC} -o $@ $
```

而一个典型的编译行变成了：

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

我们也可以定义一个变量:

```
THEPROGRAM = fooprog
```

并在我们的 **makefile** 中使用这个变量而不是程序名称，这使我们以后更容易改变可执行文件的名称。

✍ **练习 23.11** 编辑我们的 *makefile*，添加这个变量定义，并使用它来代替字面的程序名。构建一个命令行，使我们的 *makefile* 能够构建可执行的 *fooprog v2*。

预期结果：我们需要在命令行中指定 *THEPROGRAM* 变量，使用语法 *make VAR=value*。

注意事项：确保我们的命令行中的等号周围没有空格。

要查看全部局部作用域变量的列表可以登录:https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

### 23.2.1 模板规则

到目前为止，我们为每个需要编译的文件写了一个单独的规则，而各种 *.c* 文件的规则是非常相似的。

- 规则头（*foo.o : foo.c*）指出，源文件是对象文件的先决条件，具有相同的基名。
- 而我们正在使用 *make* 的内置变量编译的指令（*${CC} -c $<* ）甚至在字符上都是一样的。
- 规则唯一有区别的是

```
foo.o : foo.c bar.h
        ${CC} -c $<
```

我们可以把这些共性总结为一个模板规则:

```
%.o : %.c
        ${CC} -c $<
%.o : %.F
        ${FC} -c $<
```

这说明任何对象文件都依赖于具有相同基础名称的 *C* 或 *Fortran* 文件。要重新生成对象文件，请用 *-c* 标志调用 *C* 或 *Fortran* 编译器。这些模板规则可以作为替代上述 *makefile* 中的多个特定目标，但 *foo.o* 的规则除外。

*foo.o* 对 *bar.h* 的依赖，或者 *foomain.o* 对 *foomod.o* 的依赖，可以通过添加一个规则来处理:

```
# C
foo.o : bar.h
# Fortran
foomain.o : foomod.o
```

这条规则指出，如果文件 *bar.h* 或 *foomod.o* 改变了，文件 *foo.o* 或 *foomain.o* 也需要更新''。然后，*make* 会在 *makefile* 中搜索另一条规则，说明如何进行更新的不同规则，如何找到模板规则。

✍ **练习 23.12** 修改我们的 *makefile* 文件以纳入这些想法，并进行测试。

### 23.2.2 通配符

我们的 *makefile* 现在使用一个一般规则来编译任何源文件。通常，我们的源文件是我们目录中所有的*.c* 或 *.F* 文件，那么是否有办法说明'' 编译此目录中的所有文件''？ 实际上是有的。

在我们的 *makefile* 中添加以下几行，并在适当的地方使用变量 *COBJECTS* 或 *FOBJECTS*。*wildcard* 命令给出了 *ls* 的结果，我们可以用以下方法来处理 *patsubst* 的文件名:

```
# wildcard: find all files that match a pattern
CSOURCES := ${wildcard *.c}
# pattern substitution: replace one pattern string by another
COBJECTS := ${patsubst %.c,%.o,${SRC}}

FSOURCES := ${wildcard *.F}
FOBJECTS := ${patsubst %.F,%.o,${SRC}}
```

### 23.2.3 条件式

有多种方法可以使 *makefile* 的操作动态化。例如，我们可以把一个 *shell* 条件放在一个操作行中。然而，这可能会使 *makefile* 变得杂乱无章；有一个更简单的方法是使用 *makefile* 条件。有两种类型的条件：对字符串相等的测试和对环境变量的测试。第一种类型如下形式：

```
ifeq "${HOME}" "/home/thisisme"
    # case where the executing user is me
else
    # case where it's someone else
endif
```

而在第二种情况下，形式如下：

```
ifdef SOME_VARIABLE
```

*true* 和 *false* 部分的文本可以是 *makefile* 中的大多数部分。例如，可以让规则中的一个规则中的一个动作行被有条件地包括在内。然而，在大多数情况下，我们将使用条件式来使变量的定义依赖于某些条件。

✍ **练习 23.13** 假设我们想在家里和工作中使用我们的 *makefile*，在工作中，我们的雇主有一个付费的许可证的英特尔编译器 *icc*，但在家里我们使用开源的 *Gnu* 编译器 *gcc*。写一个 *makefile* 在两个地方都能使用，并为 *CC* 设置适当的值。

## 23.3 杂录

### 23.3.1 makefile* 是做什么的?

上面我们已经知道，发出 *make* 命令会自动执行 *makefile* 中的第一条规则。这在某种意义上是很方便的，但在另一种意义上是不方便的：要想知道 *makefile* 允许哪些可能的操作的唯一方法是阅读 *makefile* 本身，否则阅读文档通常是不够充分的。

一个更好的方式是用一个目标来启动 *makefile*：

```
info :
        @echo "The following are possible:"
        @echo " make"
        @echo " make clean"
```

没有明确目标的 *make* 会告知我们 *makefile* 的能力。

如果我们的 *makefile* 变长了，我们可能想这样记录每个部分。这就遇到了一个问题。我们不能有两条目标相同的规则，上述的 *info* 就是这种情况。然而，如果我们使用双冒号，则是可以的。我们的 *makefile* 将有如下结构：

```
info ::
        @echo "The following target are available:"
        @echo " make install"
install :
        # ..... instructions for installing
info ::
        @echo " make clean"
clean :
        # ..... instructions for cleaning
```

### 23.3.2 Phony 目标

*makefile* 包含一个 *clean* 目标的例子。它使用 make 机制来完成一些与文件创建无关的操作：调用 *make clean* 导致 *make* 推理出'' 没有名为 *clean* 的文件所以需要执行下面的指令''。然而，这实际上并没有使 *clean* 文件出现，所以再次调用 make clean 会使同样的指令被执行。

为了表明这个规则实际上并产生成为目标，我们可以使用 *.PHONY* 关键字:

```
.PHONY : clean
```

大多数情况下，没有声明 *makefile* 实际上也能正常工作，但声明 *phony* 目标的的主要好处是，即使我们有一个名为" *clean*'' 的文件（或文件夹），*make* 规则仍然可以工作。

### 23.3.3 预定义的变量和规则

调用 *make -p yourtarget* 会使 *make* 输出它的所有操作，以及所有变量和规则的值，包括我们的 *makefile* 中的和预定义的。如果我们在一个没有 *makefile* 的目录下做这个动作，我们会看到 *make* 实际上已经知道如何编译*.c* 或 *.F* 文件。找到这个规则，并找到其中的变量定义。

我们会看到，我们可以通过设置 *CFLAGS* 或 *FFLAGS* 等变量来自定义 *make*，通过一些实验来证实这一点。如果我们想为相同的源代码 *make* 第二个 *makefile* ，我们可以调用 *make -fothermakefile* 来代替默认的 *makefile* 。

顺便注意，*makefile* 和 *Makefile* 都是默认 *makefile* 的合法名称。在我们的目录中尽量不要同时拥有 *makefile* 和 *Makefile* 。

### 23.3.4 将目标作为先决条件

假设我们有两个同等对待的不同目标，我们会想这样写:

```
PROGS = myfoo other
${PROGS} : $@.o # this is wrong!!
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

然后说 *make myfoo* 会导致

```
cc -c myfoo.c
cc -o myfoo myfoo.o ${list of libraries}
```

同样地，对于 *make other* 也是如此。这里出问题的是使用 *$@.o* 作为先决条件。在 *Gnu Make* 中，我们可以按以下方式修复它:

```
.SECONDEXPANSION:
${PROGS} : $$@.o
        ${CC} -o $@ $@.o ${list of libraries goes here}
```

**练习**：编写第二个主程序 *foosecond.c* 或 *foosecond.F*，并修改我们的 *makefile*，以便使调用 *make foo* 和 *make foosecond* 都使用同一规则。

## 23.4 Makefile 中的 shell 脚本

目的：在这一节中，我们将看到一个出现在 *makefile* 规则中较长的 *shell* 脚本例子。

在我们到目前为止看到的 *makefile* 中，命令部分是单行的。实际上，只要我们想可以有许多行。例如，让我们制定一个为我们正在构建的程序做备份的规则。在我们的 *makefile* 中添加一个备份规则，它需要做的第一件事是建立一个备份目录：

```
.PHONY : backup
backup :
        if [ ! -d backup ] ; then
            mkdir backup
        fi
```

我们输入这个了吗？但这并不会奏效：*makefile* 规则的命令部分的每一行都会作为一个单独的程序被执行。因此，我们需要把整个命令写在一行上。

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
```

否则 *if* 行变得太长：

```
backup :
        if [ ! -d backup ] ; then \
            mkdir backup ; \
        fi
```

接下来我们进行实际的复制：

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

但这种备份方案只保存一个版本。让我们做一个其名称中的日期为保存的程序名称中的日期的版本。

*Unix* 的 *date* 命令可以通过接受一个格式字符串来定制其输出。输入以下内容：*date* 这可以在 *makefile* 中使用。

**练习 23.14** 编辑 *cp* 命令行，使备份文件的名称包括当前日期。

**练习 23.15** 编辑 cp 命令行，使备份文件的名称包括当前日期。

预期结果：提示：需要反引号。如果我们不记得反引号的作用，请查阅 *Unix* 教程第 *20.3.3* 节。

如果我们在 *makefile* 规则的命令部分定义 *shell* 变量，我们需要注意的是以下几点。用一个循环来扩展我们的备份规则，复制对象文件：

```
#### This Script Has An ERROR!
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
```

```
        for f in ${OBJS} ; do \
            cp $f backup ; \
        done
```

(这不是最好的复制方式，但我们为了演示而使用它)。这导致了一个报错信息，原因是 *Make* 将 *$f* 解释为外部进程的一个环境变量。解决办法是：

```
backup :
        if [ ! -d backup ] ; then mkdir backup ; fi
        cp myprog backup/myprog
        for f in ${OBJS} ; do \
            cp $$f backup ; \
        done
```

(在这种情况下，*Make* 在扫描命令行时将两个连续的美元符号替换为单个的美元符号，在执行命令行时，*$f* 会扩展为适当的文件名)。

## 23.5 使用 Make 的实用技巧

这里有几个实用的提示。

- 调试一个 *makefile* 通常是令人沮丧的困难，唯一的工具就是 *-p* 选项。它可以根据当前的 *makefile* 打印出 *Make* 正在使用的所有规则。
- 我们经常会发现自己先输入一个 *make* 命令，然后再调用程序。大多数 *Unix shell* 允许我们通过使用向上的箭头键来使用 *shell* 的历史命令。不过，这可能会让人感到厌烦，所以我们可能会想写以下内容：

  ```
  make myprogram ; ./myprogram -options
  ```

  然后不断地重复这一点。这样做有一个危险：如果 *make* 失败，例如因为编译问题，我们的程序仍然会被执行。相反，如果写

  ```
  make myprogram && ./myprogram -options
  ```

  该程序执行的条件是 *make* 成功结束。

## 23.6 LaTeX 的 Makefile

Make 工具通常用于编译程序，但其他用途也是可能的。在本节中，我们将讨论 LaTeX 文档的 Makefile。我们从一个非常基本的 makefile 开始：

```
info :
@echo "Usage: make foo"
@echo "where foo.tex is a LaTeX input file"
%.pdf : %.tex
pdflatex $<
```

命令 *make myfile.pdf* 将调用 p*dflatexmyfile.tex*，如果需要的话，就调用一次。接下来我们重复调用 *pdflatex* ，直到日志文件不再报告说需要进一步运行。

```
%.pdf : %.tex
pdflatex $<
while [ `cat ${basename $@}.log | grep "Rerun to get" \
| wc -l ` -gt 0 ] ; do \
pdflatex $< ; \
done
```

我们使用 *${basename fn}* 宏来从目标名称中提取不带扩展名的基名。

如果该文件有书目或索引，我们就运行 *bibtex* 和 *makeindex* 。

```
%.pdf : %.tex
pdflatex ${basename $@}
-bibtex ${basename $@}
-makeindex ${basename $@}
while [ `cat ${basename $@}.log | grep "Rerun to get" \
| wc -l ` -gt 0 ] ; do \
pdflatex ${basename $@} ; \
done
```

行首的减号意味着如果这些命令失败，*make* 不会中止。

最后，我们想使用 *make* 的工具来考虑到依赖关系，我们可以写一个有通用规则的 *makefile* :

```
mainfile.pdf : mainfile.tex includefile.tex
```

但我们也可以明确地发现包含文件，下面的 *makefile* 是用:

```
make pdf TEXTFILE=mainfile
```

然后，*pdf* 规则使用一些 *shell* 脚本来发现 *include* 文件（但不是递归的），并再次调用 *make*，调用另一个规则，并明确地传递依赖性。

```
pdf :
export includes= `grep ".input " ${TEXFILE}.tex \
| awk ' {v=v FS $$2".tex"} END {print v}' ` ; \
${MAKE} ${TEXFILE}.pdf INCLUDES="$$includes"
%.pdf : %.tex ${INCLUDES}
pdflatex $< ; \
while [ `cat ${basename $@}.log \
| grep "Rerun to get" | wc -l ` -gt 0 ] ; do \
pdflatex $< ; \
done
```

这种 *shell* 脚本也可以在 *makefile* 之外进行，动态地生成 *makefile*。

## 23.7 Cmake

写 *makefile* 可能很麻烦，它们可能需要为任何特定的安装进行自定义。由于这个原因，我们使用 *Cmake* 构建系统，它首先生成 *makefiles*，然后可以用正常的方式使用。

Cmake 主要组成部分为:

- 源文件目录中有一个描述应用程序结构的文件 *CMakeLists.txt* ；
- 生成的文件可以与源文件分开保存：

```
sourcedir=....   # source location
builddir=....    # place for temporaries
installdir=...   # here goes the finished stuff
cd ${builddir}
cmake \
    -DCMAKE_PREFIX_PATH=${installdir} \
    ${sourcedir} # do the setup
make             # compile
make install     # move finished stuff in place
```

### 23.7.1 依赖性分析

上面我们学到了 *make* 如何用于具有复杂依赖关系的项目，我们可以用子句告诉 *Cmake* ：

```
include_directories(include)
file(GLOB SOURCES "*/*.c")
file(GLOB HEADERS "*/*.h")
add_executable(hello_world hello.c ${SOURCES} )
```

当被调用时，*Cmake* 会报告

```
Scanning dependencies of target hello_world
```

并相应地生成 *makefile* 文件。

这种设置的一个好处是，只有当应用程序的结构发生变化时，才需要重新进行 *Cmake* 阶段，例如增加了文件。对于程序开发过程中的任何普通编辑，只需重复 *make* 和 *make install* 部分即可。

例子：考虑一个主文件 *hello.c* ，它使用了两个辅助文件 *compute.c* 和 *output.c*，*Cmake* 后调用 *make*；*make install* 给出的输出是：

```
Scanning dependencies of target hello_world
[ 25%] Building C object CMakeFiles/hello_world.dir/hello.c.o
[ 50%] Building C object
                CMakeFiles/hello_world.dir/compute/compute.c.o
[ 75%] Building C object CMakeFiles/hello_world.dir/output/output.c.o
[100%] Linking C executable hello_world
[100%] Built target hello_world
[100%] Built target hello_world
Install the project...
```

在编辑完 *compute.c* 文件后，只有该文件被重新编译，并且项目被重新连接：

```
Scanning dependencies of target hello_world
[ 25%] Building C object

CMakeFiles/hello_world.dir/compute/compute.c.o
[100%] Built target hello_world
[100%] Built target hello_world
Install the project...
```

显然，*cmake* 已经生成了具有正确结构的 *makefile*，而使用 *cmake* 的缺点是自动生成的 *makefile* 几乎是不可读的。

### 23.7.2 Re-Cmaking

刚才概述包括独立的构建和安装的目录结构，意味着完全删除这些库就足够从没有历史记录的开始。对于一个更温和的方法，我们可以删除位于构建目录中的 *CMakeCache.txt* 文件。

### 23.7.3 系统依赖性

我们可以给 *Cmake* 提供选项，使其进入 *makefile* 结构。例如，在编写本书的苹果笔记本上，直接调用 *Cmake* 就能找到本地的 *C* 编译器：

```
-- The C compiler identification is AppleClang 9.1.0.9020039
```

然而，指定：

```
cmake -DCMAKE_C_COMPILER=gcc
```

会给出：

```
-- The C compiler identification is GNU 7.4.0
```

### 23.7.4 详情

*CMake* 有的变量：

```
set(myvar myvalue)
message(STATUS "my variable has value ${myvar}")
```

# 第 24 章 源码控制

## 24.1 源码控制

源码控制系统，也叫版本控制系统，是一种存储软件的方式。它不仅存储当前版本，而且还存储所有以前的版本，这是通过维护一个包含所有版本的存储库来实现的，同时一个或多个用户在一个'*check out*' 的最新版本上工作。那些作为开发者的用户可以将他们的修改提交到存储库中，其他用户然后更新他们的本地拷贝。版本库通常存在远程设备上，并有可靠的备份。将我们的源代码保存在资源库里有各种原因：

- 如果我们在一个团队中工作，它是与我们的同事同步工作的最佳方式。它也是记录哪些改动是由谁做的，以及为什么这样做的。
- 它可以让我们把有缺陷的代码回滚到可以工作的版本。
- 它允许我们有一些分支，例如用于需要保持在主线之外的定制化的开发线之外。如果我们在一个团队中工作，分支是开发一个主要功能的一种方式，可以保持分支是一种开发主要功能的方式，可以及时了解同事们所做的修改，并在充分测试后将功能添加到主开发中。
- 如果我们独自工作，它是一种在多台机器之间进行同步的方式。(我们甚至可以想象在旅行中不带所有的文件，而从版本库中安装到借来的的机器上)。
- 拥有一个源代码库是备份我们工作的一种方式。

有各种源码控制系统；在本教程中，我们可以学习 *Subversion* 的基本知识 ( 也叫 *svn* )，它可能是最流行的传统源码控制系统，和 *Mercurial*（或 *hg* ），它是新一代分布式源码控制系统的示例。

## 24.2 源码控制系统的工作流程

源码控制系统是围绕着资源库的概念建立的：一个项目的文件的中央存储，以及它们的全部历史。因此，版本库允许我们与多人共享文件，也可以回滚修改，在旧版本上打补丁。对存储库的基本操作是：

- 创建版本库；这需要我们在一些服务器上有空间和写权限，也许我们的系统管理员必须为我们做这件事。
- 检查版本库，也就是在我们自己的空间里制作一份本地内容的拷贝。
- 将我们的修改添加到版本库中，并且
- 用别人的修改来更新我们的本地拷贝。

添加自己的修改并不总是可能的：有很多项目的开发者允许我们可以查看版本库，但不能加入修改，这样的版本库是只读的。

可以编辑文件，并使用提交命令检查新版本；要获得其他用户提交的修改，则使用更新命令。其他用户提交的修改，我们可以使用 *update*。提交的用途之一是，如果我们意识到我们犯了错误或引入了一个错误，我们可以将我们的代码回滚到一个较早的版本。它还可以让我们轻松地看到不同的代码版本的区别。然而，提交许多小改动可能会让其他开发者感到困惑，例如，如果他们会依赖我们引入的东西，而我们后来又将其删除。由于这个原因，分布式源代码控制系统使用两个级别的存储库。

它仍然有一个权威的顶层，但现在有一个较低的层级，通常是本地副本，在那里我们可以提交我们的修改，并积累到最后添加到中央版本库中。这也使得为只读版本库做贡献变得更容易：我们在本地做修改，当完成后，告诉开发者检查我们的修改，并把它们拉到顶级版本库中。这个结构如图 23.2 所示。

## 24.3 Mercurial( hg ) 和 Git

*Mercurial* 和 *Git* 是新一代分布式源代码控制系统中最著名的。许多命令与 *Subversion* 相同，但也有一些新的命令，与新的先进水平相对应。*Mercurial* 和 *Git* 共享一些命令，但也有区别。后期使用 *Git* 更加强大，但 *Mercurial*

一开始更容易使用。

下面是两个系统之间的转换：https://github.com/sympy/sympy/wiki/Git-hg-rosetta-stone.

*Git* 的最低限度介绍：http://rogerdudler.github.io/git-guide/

这个实验应该由两个人完成，来模拟一群程序员在一个联合项目上工作。我们也可以通过使用两个版本库的克隆，最好在我们的电脑上打开两个窗口来做这个实验。

分布式版本控制的最佳实践：https://homes.cs.washington.edu/mernst/advice/version-control.html

### 24.3.1 创建和填充一个存储库

**目的**：在这一节中，我们将创建一个资源库，并制作一个本地副本来工作。

| Mercurial | Git |
|---|---|
| hg clone [ localdir ] | git clone [ localdir ] |

首先，我们需要有一个版本库。在实践中，我们通常会使用一个已经建立好的版本库，但也有几种方法可以自己建立一个版本库。有一些商业和免费的托管服务，如 http://bitbucket.org。(学术用户可以有更多的私人资源库)。我们假设一个学生在 *Bitbucket* 上创建了一个版本库 *your-project*，两个学生都可以然后克隆：

```
%% hg clone https://YourName@bitbucket.org/YourName/your-project
updating to branch default
0 files updated, 0 files merged, 0 files removed,
    0 files unresolved
```

或:

```
%% git clone git@bitbucket.org:YourName/yourproject.git
Cloning into 'yourproject' ...
warning: You appear to have cloned an empty repository.
```

然后我们就会有一个空目录 *your-project*。

✍ **练习 24.1** 进入项目目录，看看它是否真的为空。

预期结果：有一个隐藏的目录 *.hg* 或 *.git* 。

### 24.3.2 新文件

创建一个未监控的文件，**目的**：在本节中，我们将做一些简单的修改：创建一个新文件和编辑一个现有的文件。

| Mercurial | Git |
|---|---|
| hg status [ path] | git status [ path ] |
| hg add [ files ] | git add [ files ] |
| 每个文件都使用一次 | 每次文件被改变时 |

一个学生现在做了一个文件添加到资源库中：

```
%% cat > firstfile
a
b
```

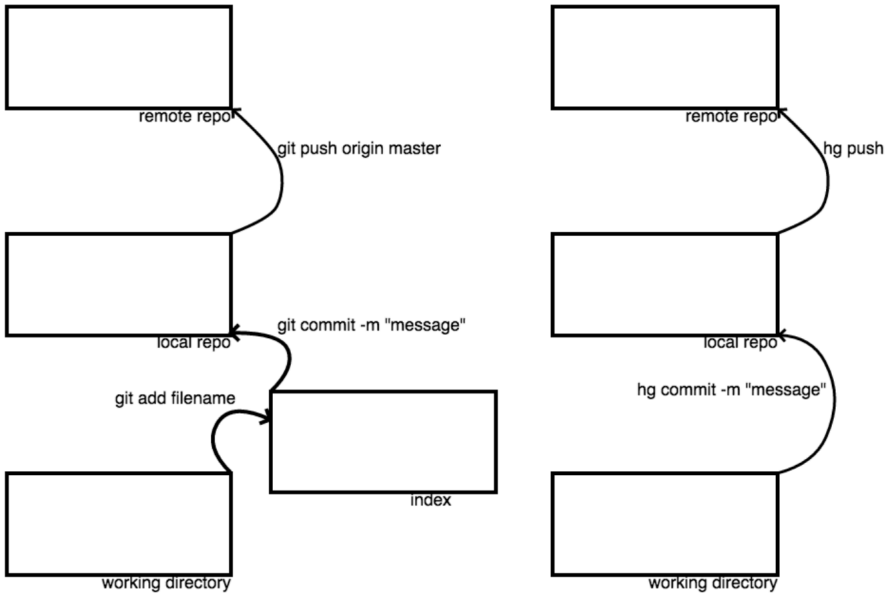
```
c
d
e
f
D
```

(其中 *D* 代表 *control-D*，它终止了输入)。这个文件对 *hg* 来说是未知的：

```
%% hg status
? firstfile
```

Git 会更麻烦一点：

```
git status
On branch master
Initial commit
Untracked files:
    (use "git add <file>..." to include in what will be committed)
firstfile
nothing added to commit but untracked files present
    (use "git add" to track)
```




添加文件到版本库我们需要声明该文件属于版本库；随后的一个 *hgcommit* 命令就会把它复制到版本库中。

```
%% hg add firstfile
%% hg status
A firstfile
%% hg commit -m "made a first file"
```

或是

```
%% git add firstfile
%% git status
On branch master
Initial commit
Changes to be committed:
    (use "git rm --cached <file>..." to unstage)
        new file: firstfile
%% git commit -a -m "adding a first file"
    [master (root-commit) f4b738c] adding a first file
    1 file changed, 5 insertions(+)
    create mode 100644 firstfile
```

| Mercurial | git |
|---|---|
| hg commit -m <message> | git commit -m <message> |
| hg push | git push origin master |
| 每个文件都使用一次 | 每次文件被改变时 |

与 *Subversion* 不同的是，现在该文件只被复制到本地版本库中，这样我们就可以回滚我们的修改。如果我们想把这个文件添加到主版本库，我们需要使用 *hg push* 命令。

```
%% hg push https://YourName@bitbucket.org/YourName/your-project
pushing to https://YourName@bitbucket.org/YourName/your-project
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files
remote: bb/acl: YourName is allowed. accepted payload.
```

在 *push* 步骤中，我们可能被要求提供我们的密码。我们可以通过在 *$HOME/.hgrc* 文件中设置几行来防止这种情况:

```
[paths]
projectrepo = https://YourName:yourpassword@bitbucket.org/YourName/my-project
[ui]
username=Your Name <you@somewhere.youruniversity.edu>
```

现在，*hg push projectrepo* 命令将把本地的修改推送到全局版本库，而不需要我们的密码。当然，如果我们有一个带有明文密码的文件，我们应该正确设置这个文件的权限。使用 *Git* ，我们需要更明确，因为我们的本地拷贝和'' 上游'' 仓库之间的联系可能更不稳定。

```
git remote add origin git@bitbucket.org:YourName/yourrepo.git
git push origin master
```

第二个学生现在需要更新他们的版本库。就像上传需要两个命令一样，这个过程也需要两个命令。首先，我们需要 *hg pull* 来更新我们的本地版本库。这并不更新我们所拥有的本地文件：因此我们需要 *hg update* 。

✍ **练习 24.2** 如上这样做并检查文件的内容是否正确。

预期的结果：为了执行更新命令，我们必须在一个检验过的拷贝中储存库的。

✍ **练习 24.3** 让两个学生创建一个带有几个文件的新目录。声明该目录并提交，*pull* 之后使用 *update*，以获得另一个 *mde* 的变化。

预期结果：我们可以对该目录进行 *hg* 添加，这也会添加其中的文件。

由于我们大多会在 *pull* 后立即进行 *update*，我们可以把它们结合起来：

```
hg pull -u
```

*Git* 会报告哪些文件被更新了；对于 *Hg* 来说，我们需要获取更改的编号然后进行查询:

```
hg status --change 82ffb99c79fd
```

注释 为了让 *Mercurial* 追踪我们的文件，我们不应该对版本库中的文件执行 *shell* 中 *cp* 命令或 mv 命令。相反，应该使用 *hg cp* 或 *hg mv*。同样地，也有一个 *hg rm* 命令。

### 24.3.3 哎呀! 撤销!

拥有源码控制的原因之一是为了能够恢复修改，最简单的撤销方式是回到回到版本库中最后存储的版本。

| Mercurial | Git |
|---|---|
| hg revert <yourfile> | git checkout – <yourfile> |

### 24.3.4 冲突

**目的**：在本节中，我们将了解如何处理同一版本库的两个用户的冲突编辑的两个用户在同一个版本库中的冲突编辑。

现在让我们看看当两个人编辑同一个文件时会发生什么。让两个学生都对 *firstfile* 进行编辑。但一个在上面，另一个在下面。一个学生提交编辑后，另一个可以提交修改。毕竟，这些只影响本地仓库。然而，试图推送该修改时却出现了错误；

```
%% emacs firstfile # make some change
%% hg commit -m ''first again''
%% hg push test
abort: push creates new remote head b0d31ea209b3!
(you should pull and merge or use push -f to force)
```

解决的办法是获得另一个编辑，并再次提交。这需要几个命令:

```
%% hg pull myproject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)

%% hg merge
merging firstfile
```

```
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)

%% hg status
M firstfile
%% hg commit -m ''my edit again''
%% hg push test
pushing to https://VictorEijkhout:***@bitbucket.org/VictorEijkhout/my-project

searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 2 changesets with 2 changes to 1 files
remote: bb/acl: VictorEijkhout is allowed. accepted payload.
```

这可能看起来很复杂，但我们看到 *Mercurial* 提示我们要执行什么命令，如果我们参考图 23.2，工作流程很清楚。

✍ **练习 24.4** 在我们们两个人都在编辑的文件上使用 *cat* 命令。我们应该发现，两个人的编辑都合并了。这就是 *Mercurial* 提到的'' 合并''。

如果两个学生都对文件的同一个部分进行编辑，版本控制就不能再解决冲突。例如，让一个学生在第一行和第二行之间插入一行，并让第二个学生编辑第二行。无论谁尝试推送第二条，都会得到这样的信息：

```
%% hg pull test
added 3 changesets with 3 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
%% hg merge
merging firstfile
warning: conflicts during merge.
merging firstfile incomplete!
    (edit conflicts, then use 'hg resolve --mark')
0 files updated, 0 files merged, 0 files removed, 1 files   unresolved
use 'hg resolve' to retry unresolved file merges
    or 'hg update -C .' to abandon
```

现在有以下选项：

1. 通常有一种方法可以表明是使用本地版本还是远程版本。
2. 有一些图形化的程序来解决冲突。它们通常会向我们显示三栏，分别是两个版本和我们的解决方案。然后我们可以表示'' 从本地版本中取这个，从远程版本中取那个''。
3. 我们也可以自己编辑文件来解决冲突，我们很快会讨论这个问题。

两者都会给我们几个选项。用文本编辑器来解决冲突是最容易的。如果我们打开有冲突的文件我们会看到类似的东西：

```
<<<<<<< local
aa
bbbb
=======
```

```
aaa
a2
b
>>>>>>> other
c
```

表明本地版本和另一版本之间的差异，也就是我们 *pull* 过并试图合并的版本，我们需要编辑该文件来解决冲突。在这之后，我们告诉 *hg* 冲突已经解决：

```
  hg resolve --mark
%% hg status
M firstfile
? firstfile.orig
```

或是

```
git add <name of that file>
```

在这之后，我们可以提交并再次 *push*，另一个学生就需要再一次更新来更正。不是所有的文件都可以合并：对于二进制文件，*Mercurial* 会询问我们：

```
%% hg merge
merging proposal.tex
merging summary.tex
merking references.tex
 no tool found to merge proposal.pdf
keep (l)ocal or take (o)ther? o
```

这意味着唯一的选择是保留我们的本地版本（输入 *l* 并点击回车）或采用另一个版本（输入 *o* 并按回车键）。在二进制文件显然是自动生成的情况下，有些人认为他们不应该在版本库中出现。

### 24.3.5 检查历史

**目的**：在本节中，我们将学习如何查看和比较版本库中的文件。

如果我们想知道我们是从哪里克隆的版本库，可以在 *.hg/hgrc* 文件中查看。

关于版本库的主要信息来源是 *hg log* 和 *hgid*，后者给我们的是否是全局信息，取决于我们使用的选项。例如，*hg id -n* 给出了本地版本号。*hg log* 给我们一个到目前为止所有更改集的列表，以及我们输入的注释。*hg log -v* 告诉我们在每个变化集中有哪些文件被影响。*hg log -r 5* 或 *hg log -r 6:8* 给出了一个或多个变化集的信息。

要查看单个文件的不同修订版的差异，可以使用 *hg diff*。首先确保我们的版本是最新的，输入 *hg diff firstfile*，没有输出，对吗？再在 *firstfile* 中做一个编辑，然后输入 *hg diff firstfile*，这样我们就能得到最后一个提交的版本和工作副本之间的差异。

| Mercurial | Git |
|---|---|
| hg diff <file> | git diff HEAD <file> |
| hg diff -r A -r B <file> | git diff A_..B <file> |

自行检查一下，当我们做了提交但没有推送，并发出上述 *diff* 命令时会发生什么。我们也可以用 *hg diff -r 4:6 firstfile* 询问提交版本之间的差异。*diff* 命令的输出有点隐秘，但我们可以轻松就理解它。还有也有针对每个平台的 *hg* 的不同 *GUI* 实现，以更好的方式显示差异。

如果我们只是想看看一个文件以前是什么样子的，可以输入 *hg cat -r 2 firstfile*。要获得一份仓库的某个版本，输入 *hg export -r 3 . . . /rev3*，它将仓库在当前目录下（' *dot* '）的版本库导出到 *./rev3* 目录。

### 24.3.6 传输

*Mercurial* 和 *Git* 可以使用 *ssh* 或 *http* 作为传输。对于 *Git*，我们可能需要重新定义推送的运输方式来进行推送。

```
git remote rm origin
git remote add origin git@github.com:TACC/pylauncher.git
```

# 第 25 章 科学数据存储

## 25.1 科学数据存储

有许多存储数据的方法，特别是以数组形式出现的数据。有很多人在电子表格中存储数据，然后将它们导出到带有逗号或制表符的 *ascii* 文件中，并希望其他人（或由他们自己编写的其他程序）再次读入这些数据，这样一个过程在以下几个方面的浪费：

- 一个数字的 *ascii* 表示法要比内部的二进制表示法占用更多的空间，理想情况下，你希望一个文件和内存中的表示一样紧凑。
- *ascii* 的转换很慢；它也可能导致精度的损失。

由于这样的原因，最好能有一种基于二进制存储的文件格式。还有一些对一个有用的文件格式的要求：

- 由于二进制存储在不同的平台之间会有差异，所以一个好的文件格式是与平台无关的。例如，这将防止混淆大端字节序和小端字节序，以及 32 位和 64 位浮点数的约定。
- 应用程序数据可能是异质的，包括整数、字符和浮点数据，理想情况下，所有这些数据应该被存储在一起。
- 应用数据也是有结构的，这种结构应该反映在存储的形式中。
- 一个文件格式最好是自带文档的，如果文件本身能告诉你哪些存储的数字是矩阵，哪些是向量，以及这些对象的大小是什么难道不是一件很好的事吗?

本教程将介绍 *HDF5* 库，它可以满足这些要求。*HDF5* 是一个庞大而复杂的库，所以本教程将只涉及到基础知识。欲知更多信息，请查阅 http://www.hdfgroup.org/HDF5/当你做这个教程的时候，请保持你的浏览器在 http://www.hdfgroup.org/HDF5/doc/或 http://www.hdfgroup.org/HDF5/RM/RM_H5Front.html 来了解这些例程的确切语法。

## 25.2 HDF5 简介

如上所述，*HDF5* 是一种与机器无关的文件格式，并且是自我记录的。每个 *HDF5* 文件的设置就像一个目录树，有子目录和包含实际数据的叶子结点。这意味着可以通过参考文件的名称，而不是文件中的位置来找到数据。本节中，你将学习如何编写向 *HDF5* 文件写入和从 *HDF5* 文件中读取的程序。为了检查文件是否符合你的目的，你可以在命令行中使用 *h5dump* 工具。

*HDF5* 格式与旧版本的 *HDF4* 并不兼容，后者已不再开发。由于历史原因，你仍然可以碰到有人使用 *HDF4* 。本教程是基于 *HDF5* 的 1.6 版本。在当前的 1.8 版本中，一些接口发生了变化；为了在 1.8 版本的软件中使用 1.6 版本的 *API*，请在你的编译行中添加一个 *-DH5_USE_16_API* 标志。许多 *HDF5* 例程是关于创建对象的：文件句柄、数据集中的成员等等。一般语法是：

```
hid_t h_id;
h_id = H5Xsomething(...);
```

创建对象的失败是由一个负的返回参数表示的，所以最好是创建一个包含 *myh5defs.h* 的文件。

```
#include "hdf5.h"
#define H5REPORT(e) \
    {if (e<0) {printf("\nHDF5 error on line %d\n\n",__LINE__); \
    return e;}}
```

并将此作为：

```
#include "myh5defs.h"
```

```
hid_t h_id;
h_id = H5Xsomething(...); H5REPORT(h_id);
```

## 25.3 创建一个文件

首先，我们需要创建一个 *HDF5* 文件。

```
hid_t file_id;
herr_t status;
file_id = H5Fcreate( filename, ... );
    ...
status = H5Fclose(file_id);
```

这个文件将是一些数据项的容器，像目录树一样组织。

✍ **练习 25.1** 通过编译和运行下面的 *create.c* 例子创建一个 *HDF5* 文件。
预期结果：应该会创建一个 *file.h5* 文件。
注意事项：请确保添加 *HDF5* 的包含目录和库目录。

```
cc -c create.c -I. -I/opt/local/include
```

然后

```
cc -o create create.o -L/opt/local/lib -lhdf5.
```

*include* 和 *lib* 目录将取决于系统。

```
/*
* File: create.c
* Author: Victor Eijkhout
*/
#include "myh5defs.h"
#define FILE "file.h5"

main() {

    hid_t file_id; /* file identifier */
    herr_t status;

    /* Create a new file using default properties. */
    file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
    H5REPORT(file_id);

    /* Terminate access to the file. */
    status = H5Fclose(file_id);
}
```

你可以在命令行上显示创建的文件：

```
%% h5dump file.h5
HDF5 "file.h5" {
GROUP "/" {
}
}
```

请注意，一个空文件只对应于将容纳数据的目录树的根。

## 25.4 数据集

接下来我们创建一个数据集，在这个例子中是一个二维网格。为了描述这个，我们首先需要构建一个数据空间：

```
dims[0] = 4; dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);
dataset_id = H5Dcreate(file_id, "/dset", dataspace_id, .... );
....
status = H5Dclose(dataset_id);
status = H5Sclose(dataspace_id);
```

注意，数据集和数据空间需要像文件一样被关闭。

**练习 25.2** 通过编译和运行下面的 *dataset.c* 代码创建一个数据集。

预期结果：这将创建一个 *dset.h* 文件，可以用 *h5dump* 显示。

```
/*
* File: dataset.c
* Author: Victor Eijkhout
*/
#include "myh5defs.h"
#define FILE "dset.h5"
main() {

hid_t file_id, dataset_id, dataspace_id; /* identifiers */
hsize_t dims[2];
herr_t status;

/* Create a new file using default properties. */
file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

/* Create the data space for the dataset. */
dims[0] = 4;
dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);

/* Create the dataset. */
dataset_id = H5Dcreate(file_id, "/dset",H5T_NATIVE_INT,dataspace_id,H5P_DEFAULT);
```

```
    /*H5T_STD_I32BE*/

    /* End access to the dataset and release resources used by it. */
    status = H5Dclose(dataset_id);

    /* Terminate access to the data space. */
    status = H5Sclose(dataspace_id);
    /* Close the file. */
    status = H5Fclose(file_id);
}
```

我们再次在线查看创建的文件：

```
    %% h5dump dset.h5
    HDF5 "dset.h5" {
    GROUP "/" {
    DATASET "dset" {
        DATATYPE H5T_STD_I32BE
        DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
        DATA {
        (0,0): 0, 0, 0, 0, 0, 0,
        (1,0): 0, 0, 0, 0, 0, 0,
        (2,0): 0, 0, 0, 0, 0, 0,
        (3,0): 0, 0, 0, 0, 0, 0
        }
    }
}
}
```

数据文件包含诸如你存储的数组的大小等信息，不过你可能还想添加相关的标量信息。例如，如果数组是一个程序的输出，你可以记录用什么输入参数生成的。

```
    parmspace = H5Screate(H5S_SCALAR);
    parm_id = H5Dcreate(file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);
```

**练习 25.3** 通过编译和运行下面的 *parmwrite.c* 代码，在 *HDF5* 文件中添加一个标量数据空间。

预期结果：一个新的 *wdset.h5* 文件被创建。

```
    /*
    * File: parmdataset.c
    * Author: Victor Eijkhout
    */
    #include "myh5defs.h"
    #define FILE "pdset.h5"

    main() {
```

```
    hid_t file_id, dataset_id, dataspace_id; /* identifiers */
    hid_t parm_id,parmspace;
    hsize_t dims[2];
    herr_t status;

    /* Create a new file using default properties. */
    file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create the data space for the dataset. */
    dims[0] = 4;
    dims[1] = 6;
    dataspace_id = H5Screate_simple(2, dims, NULL);

    /* Create the dataset. */
    dataset_id = H5Dcreate(file_id, "/dset", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

    /* Add a descriptive parameter */
    parmspace = H5Screate(H5S_SCALAR);
    parm_id = H5Dcreate(file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);

    /* End access to the dataset and release resources used by it. */
    status = H5Dclose(dataset_id);
    status = H5Dclose(parm_id);

    /* Terminate access to the data space. */
    status = H5Sclose(dataspace_id);
    status = H5Sclose(parmspace);

    /* Close the file. */
    status = H5Fclose(file_id);
}

%% h5dump wdset.h5
HDF5 "wdset.h5" {
GROUP "/" {
    DATASET "dset" {
        DATATYPE H5T_IEEE_F64LE
        DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
        DATA {
        (0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5,
        (1,0): 6.5, 7.5, 8.5, 9.5, 10.5, 11.5,
        (2,0): 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
        (3,0): 18.5, 19.5, 20.5, 21.5, 22.5, 23.5
        }
```

```
        }
    DATASET "parm" {
        DATATYPE H5T_STD_I32LE
        DATASPACE SCALAR
        DATA {
        (0): 37
        }
    }
}
}
```

## 25.5 写入数据

你创建的数据集分配了 *HDF5* 文件中的空间，现在你需要把实际的数据放在里面。这就是用 *H5Dwrite* 调用完成

```
/* Write floating point data */
for (i=0; i<24; i++) data[i] = i+.5;
status = H5Dwrite(dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,data);
/* write parameter value */
parm = 37;
status = H5Dwrite(parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,&parm);

/*
* File: parmwrite.c
* Author: Victor Eijkhout
*/
#include "myh5defs.h"
#define FILE "wdset.h5"

main() {

hid_t file_id, dataset, dataspace; /* identifiers */
hid_t parmset,parmspace;
hsize_t dims[2];
herr_t status;
double data[24]; int i,parm;

/* Create a new file using default properties. */
file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

/* Create the dataset. */
dims[0] = 4; dims[1] = 6;
dataspace = H5Screate_simple(2, dims, NULL);
dataset = H5Dcreate(file_id, "/dset", H5T_NATIVE_DOUBLE, dataspace, H5P_DEFAULT);
```

```
        /* Add a descriptive parameter */
        parmspace = H5Screate(H5S_SCALAR);
        parmset = H5Dcreate(file_id,"/parm",H5T_NATIVE_INT,parmspace,H5P_DEFAULT);

        /* Write data to file */
        for (i=0; i<24; i++) data[i] = i+.5;
        status = H5Dwrite(dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,data); H5REPORT(status)

        /* write parameter value */
        parm = 37;
        status = H5Dwrite(parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,&parm); H5REPORT(status);
        /* End access to the dataset and release resources used by it. */
        status = H5Dclose(dataset);
        status = H5Dclose(parmset);

        /* Terminate access to the data space. */
        status = H5Sclose(dataspace);
        status = H5Sclose(parmspace);

        /* Close the file. */
        status = H5Fclose(file_id);
}

        %% h5dump wdset.h5
        HDF5 "wdset.h5" {
        GROUP "/" {
            DATASET "dset" {
                DATATYPE H5T_IEEE_F64LE
                DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
                DATA {
                (0,0): 0.5, 1.5, 2.5, 3.5, 4.5, 5.5,
                (1,0): 6.5, 7.5, 8.5, 9.5, 10.5, 11.5,
                (2,0): 12.5, 13.5, 14.5, 15.5, 16.5, 17.5,
                (3,0): 18.5, 19.5, 20.5, 21.5, 22.5, 23.5
                }
            }
            DATASET "parm" {
                DATATYPE H5T_STD_I32LE
                DATASPACE SCALAR
                DATA {
                (0): 37
                }
            }
}
```

```
}
```

如果仔细看一下源代码和备份文件系统，我们会发现数据类型被声明为'' 本地''，但是呈现为 *LE* 。'' 本地'' 声明使数据类型的行为与内置的 *C* 或 *Fortran* 数据类型相似，另外，你也可以明确指出数据是小端字节序还是大端字节序。这些术语描述了一个数据项的字节在内存中的排序方式。正如你在备份文件系统输出中所看到的那样，大多数架构都使用小端字节序（*little endian*）。但值得注意的是，*IBM* 使用的是大端字节序。

## 25.6 读取

现在我们有了一个包含一些数据的文件，我们可以从该文件中读取数据。这些基本命令是

```
h5file = H5Fopen( .... )
....
H5Dread( dataset, .... data .... )
```

其中 *H5Dread* 命令的参数与相应的 *H5Dwrite* 相同。

✍ **练习 25.4** 通过编译和运行下面的 *allread.c* 例子，从你在前面的练习中创建的 *wdset.h5* 文件中读取数据。
预期结果：运行可执行文件 *allread* 将输出参数的值 37 和数组的 (1,2) 数据点的值 8.5。
注意事项：确保你运行 *parmwrite* 来创建输入文件。

```
/*
* File: allread.c
* Author: Victor Eijkhout
*/
#include "myh5defs.h"
#define FILE "wdset.h5"

main() {

hid_t file_id, dataset, parmset;
herr_t status;
double data[24]; int parm;

/* Open an existing file */
file_id = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
H5REPORT(file_id);

/* Locate the datasets. */
dataset = H5Dopen(file_id, "/dset"); H5REPORT(dataset);
parmset = H5Dopen(file_id,"/parm"); H5REPORT(parmset);

/* Read data back */
status = H5Dread(parmset,H5T_NATIVE_INT,H5S_ALL,H5S_ALL,H5P_DEFAULT,&parm); H5REPORT(status);
printf("parameter value: %d\n",parm);

status = H5Dread(dataset,H5T_NATIVE_DOUBLE,H5S_ALL,H5S_ALL,H5P_DEFAULT,data); H5REPORT(status);
```

```
printf("arbitrary data point [1,2]: %e\n",data[1*6+2]);

/* Terminate access to the datasets */
status = H5Dclose(dataset); H5REPORT(status);
status = H5Dclose(parmset); H5REPORT(status);
/* Close the file. */
status = H5Fclose(file_id);
}
%% ./allread
parameter value: 37
arbitrary data point [1,2]: 8.500000e+00c
```

# 第 26 章 GNU plot

## 26.1 使用 GNUplot 绘图

*gnuplot* 工具是一个简单的程序，用于绘制点或曲线的集合，这个非常简短的教程将向我们展示一些基础知识。关于更多的命令和选项，请参见手册 http://www.gnuplot.info/docs/gnuplot.html。

## 26.2 使用方法

运行 *gnuplot* 的两种模式分别是交互式和从文件中运行。在交互式模式下，我们从命令行调用 *gnuplot* 的时候，我们可以从命令行中调用 *gnuplot*，输入命令，然后观察输出；我们可以用" *exit*'' 来终止交互式会话。如果我们想保存一个交互式会话的结果，输入 *save"name.plt"* 。这个文件可以被编辑，并通过 *load "name.plt"* 来加载。非交互式绘图，我们可以调用 *gnuplot<yourfile>*。*gnuplot* 的输出可以是我们屏幕上的图片，也可以是文件中的绘图指令。输出的位置取决于终端的设置。默认情况下，*gnuplot* 会尝试画一张图片。这相当于声明了

```
set terminal x11
```

或是 aqua，windows，或是任意选择的图形硬件。对于输出到文件，声明

```
set terminal pdf
```

或 *fig* , *latex*, *pbm* 等等。请注意，这只会导致 *pdf* 命令被写入我们的屏幕上：我们需要把它们引导到文件中，用

```
set output "myplot.pdf"
```

或用以下方法捕获它们:

```
gnuplot my.plt > myplot.pdf
```

## 26.3 绘图

基本的绘图命令是用于 2D 的 *plot* 和用于 3D 绘图的 *splot*（'surface plot'）。

### 26.3.1 绘制曲线

通过指定

```
plot x**2
```

我们会得到一个 $f(x) = x^2$ 的图像；*gnuplot* 会定义 $x$ 的范围通过以下代码:

```
set xrange [0:1]
plot 1-x title "down", x**2 title "up"
```

我们可以在一个图中得到两个图，$x$ 的范围被限制在 $[0, 1]$，并为图形加上适当的图例。变量 $x$ 是绘制函数的默认值。绘制一个函数与另一个函数的对比图，或者说，绘制一个参数曲线是这样的:

```
set parametric
plot [t=0:1.57] cos(t),sin(t)
```

这画出了一个四分之一圆。要在一个图中得到一个以上的图形，请使用 *set multiplot* 命令。

### 26.3.2 绘制数据点

也可以根据数据点来绘制曲线。基本的语法是 *plot 'datafile'*，它从数据文件中抽取两列，并将其解释为 $(x, y)$ 坐标。由于数据文件通常可以有多列数据，常用的语法是 *plot 'datafile' using 3:6* 表示第 3 和第 6 列。进一步的修饰语，如 *with line* 表示点的连接方式。

同样，*splot "datafile3d.dat" 2:5:7* 将把三列解释为指定 $(x, y, z)$ 坐标的三维图。如果一个数据文件要被解释为矩形网格上的水平或高度值，输入 *split "matrix.dat" matrix* 代表数据点；用

```
split "matrix.dat" matrix with lines
```

### 26.3.3 自定义

绘图可以通过许多方式进行自定义，其中一些自定义方式使用了 *set* 命令。例如，

```
set xlabel "time"
set ylabel "output"
set title "Power curve"
```

我们也可以通过以下方式改变默认的绘图样式

```
set style function dots
```

( *dots, lines, dots, points* 等等)，或在单幅图上的变化，用

```
plot f(x) with points
```

## 26.4 工作流程

想象一下，我们的代码产生了一个我们想绘制的数据集，并且我们得到了一些输入。如果能自动绘图就更好了，*gnuplot* 本身并没有这方面的功能。但在 *Shell* 编程的帮助下，这并不难做到。假设我们有以下数据文件

```
data1.dat data2.dat data3.dat
```

而我们想用同样的 *gnuplot* 命令来绘制它们，我们可以编写一个文件 *plot.template*：

```
set term pdf
set output "FILENAME.pdf"
plot "FILENAME.dat"
```

字符串 *FILENAME* 可以用实际的文件名代替，例如用 *sed* 命令：

```
for d in data1 data2 data3 ; do
    cat plot.template | sed s/FILENAME/$d/ > plot.cmd
    gnuplot plot.cmd
done
```

这个基本想法的变种很多。

# 第 27 章 代码规范

## 27.1 良好的代码行为

无论什么时候，每个程序员都可能会遇到代码行为不符合预期的情况。在本节中，我们将学习一些处理这个问题的技巧。首先，我们将看到一些预防错误的工具。在下一章中，我们将讨论调试，即在程序中出现不可避免的错误时，找出这些错误的过程。

## 27.2 防御式编程

在这一节中，我们将讨论一些情况以防止出现编程错误的可能性，或增加它们在运行时被发现的可能性，我们把这种技术称为防御式编程。科学计算中代码通常是十分庞大且复杂的，我们要做好会犯错的准备。良好的编程习惯是使用一些工具：如果有人已经遇到过这种错误了，就没有必要再重蹈覆辙。这些工具中的一些将在其他章节中描述：

- 构建系统，如 *Make*、*Scons*、*Bjam*；见第 22 节。
- 用 *SVN*、*Git* 进行源代码管理；见第 23 节。
- 回归测试和考虑到测试的设计（单元测试）

首先，我们将对运行时进行仔细检查，在这里我们要测试不可能或不应该发生的事情。

### 27.2.1 断言

在一个程序可能出错的地方，我们可以区分 *error* 和 *bug*。*error* 是指合法地发生但不应该发生的事情，文件系统是错误的常见来源：一个程序想打开一个文件，但由于用户输入错误的文件名，事实上文件并不存在。或者是程序写入一个文件中但磁盘已经满了，而其他错误可能来自算术，如溢出错误。

另一方面，程序中的 *bug* 是一种不能合法地发生的现象。当然，这里的'' 合法'' 意味着'' 根据程序员的意图''，*bug* 通常可以被描述为'' 计算机总是按照我们的要求去做，而没有按照我们的想法去做''。断言的作用是检测程序中的 *bug*：断言是一个判断在程序中的某一处是否正确的判断式。因此，断言的失败意味着我们没有按照我们的意图进行编程。一个断言通常是我们的编程语言中的一个声明，或者是一个预处理程序宏；当断言失败时，我们的程序将继续采取错误的行为。

断言的例子：

- 如果一个子程序有一个数组参数，那么在对数组进行索引之前，最好测试一下实际参数是否是一个空指针，然后再对数组进行索引。
- 同样地，我们可以测试一个动态分配的数据结构是否有空指针。
- 如果我们计算一个数字结果，而这个数字结果的某些数学属性是成立的，例如我们正在编写一个正弦函数，其结果一定是在 $[-1, 1]$ 之间，我们应该测试这个属性是否真的适用于该结果。

一旦程序得到充分的测试，断言往往被禁用，这样做的原因是断言执行起来可能很浪费。例如，如果我们有一个复杂的数据结构，我们可以写一个复杂的完整性测试，并在断言中执行该测试，在每次对数据结构的访问之后结构。由于断言在'' 生产'' 版本中的代码中被禁用，所以对任何存储的数据不会产生影响。

#### 27.2.1.1 C 语言的断言宏

C 标准库有一个 *assert.h* 头文件，它提供了一个 *assert* () 宏。插入 *assert*（*foo*）有以下效果：如果 *foo* 是零（假），就会在标准错误上输出一条诊断信息。

```
Assertion failed: foo, file filename, line line-number
```

其中包括表达式的字面文本、文件名和行号；随后程序被中止。下面是一个例子：

```
#include<assert.h>
void open_record (char *record_name) {
         assert (record_name!=NULL) ;
          /* Rest of code */
}
int main (void) {
          open_record (NULL) ;
}
```

断言宏可以通过定义 *NDEBUG* 宏来禁用。

#### 27.2.1.2 Fortran 的断言宏

```
#if  (defined ( GFORTRAN )  || defined ( G95 )  || defined  ( PGI)  )
# define MKSTR (x)  "x"
#else
# define MKSTR (x)  #x
#endif
#ifndef NDEBUG
# define ASSERT (x, msg)  if  (.not.  (x)  )  \ call assert ( FILE , LINE ,MKSTR (x) ,m
#else
# define ASSERT (x, msg)
#endif
subroutine assert (file, ln, testStr, msgIn)
implicit none
character (*)  :: file, testStr, msgIn
integer :: ln
print *, "Assert: ",trim (testStr) ," Failed at ",trim (file) ,":",ln
print *, "Msg:", trim (msgIn)
stop
end subroutine assert
```

这被用作

```
ASSERT (nItemsSet.gt.arraySize,"Too many elements set")
```

### 27.2.2 错误代码的使用

在一些软件库中（例如 *MPI* 或 *PETSc*），每个子程序都会返回一个结果，要么是函数值，要么是参数，以表明该程序的成功或失败。良好的编程实践是检查这些错误参数，即使我们认为不可能出错。以总是有一个错误参数方式编写我们自己的子程序也是一个好主意，让我们考虑一个执行某些数值计算的函数的情况。

```
float compute (float val)  {
         float result;
         result = ... /* some computation */
         return result;
```

```
}
float value,result;
result = compute (value)
```

看起来不错？如果计算可能失败怎么办：

```
result = ... sqrt (val)  ... /* some computation */
```

我们如何处理用户传递一个负数的情况？

```
float compute (float val) {
        float result;
        if  (val<0)  { /* then what? */}
        else result = ... sqrt (val)  ... /* some computation */
        return result;
  }
```

我们可以输出一个错误信息，并提供一些结果，但这个信息可能没有被注意到，而调用的环境并没有真正收到任何关于出了问题的通知。下面的方法更加灵活：

```
int compute (float val,float *result) {
        float result;
        if  (val<0)  {
                return -1;
         } else {
                *result = ... sqrt (val)  ... /* some computation */
         }
         return 0;
  }

  float value,result; int ierr;
  ierr = compute (value,&result) ;
  if  (ierr!=0)  { /* take appropriate action */}
```

我们可以通过编写以下内容来节省大量的输入工作：

```
#define CHECK_FOR_ERROR (ierr)  \
if  (ierr!=0)  { \
printf ("Error %d detected\n",ierr) ; \
return -1 ; }
....
ierr = compute (value,&result) ; CHECK_FOR_ERROR (ierr) ;
```

使用一些 *cpp* 宏，我们甚至可以定义

```
 #define CHECK_FOR_ERROR (ierr)  \
 if  (ierr!=0)  { \
         printf ("Error %d detected in line %d of file %s\n",\
ierr,__LINE__,__FILE__) ; \
    return -1 ; }
```

请注意，这个宏不仅打印出错误信息，而且还做了进一步的返回。这意味着，如果我们将会得到一个完整的调用树的回溯，如果一个错误发生时，我们将得到一个完整的调用树的回溯。(在 *Python* 语言中，这是错误的方法，因为回溯是内置的)。

## 27.3 防范内存错误

在科学计算中，我们不可避免地会处理大量的数据。一些编程语言使管理数据变得容易，而另一些，可以说使数据出错变得容易。以下是一些内存冲突的例子

- 在数组边界外写入。如果地址在用户内存之外，我们的代码可能会以一个分配冲突的错误中止，而且这个错误是很容易发现的。如果该地址在一个数组的范围之外，它将破坏数据，但不会使程序崩溃。这样的错误可能很长时间都不会被发现，因为它可能没有任何影响，只是在我们的计算中引入了微妙的错误值。
- 读取数组边界以外的数据比写入时的错误更难发现，因为它往往不会中止我们的代码，而只是引入错误的值。
- 使用未初始化的内存与超出数组边界的读取类似，并且可以在很长时间内不被发现。这方面的一个变种是通过将内存附加到一个未分配的指针上，这种特殊的错误可以表现为有趣的行为。比方说，我们注意到我们的程序出现了问题，我们用调试模式重新编译找到了错误，现在这个错误不再发生了。这可能是由于在低优化水平下，所有分配的数组都被填满了零。因此，我们的代码原本是在读取一个随机值，但现在却得到的是一个零。

本节包含了一些防止在处理我们已经为数据预留的内存时出现错误的技术。

### 27.3.1 数组边界检查和其他内存的技术

在并行代码中，内存错误经常会通过 *MPI* 例程的崩溃而显示出来，这几乎不可能是一个 *MPI* 问题或我们的集群的问题。*Fortran* 的编译器通常支持数组绑定检查。由于这使得我们的代码变慢，所以我们只能在代码的开发阶段启用它。

### 27.3.2 内存泄漏

如果一个程序分配了内存，但随后失去了对该内存的追踪，我们就说它有内存泄漏。操作系统认为该内存正在使用中，而实际上并没有，因此，计算机的内存会被分配的内存填满，而这些内存却没有任何用处。在这个例子中，数据被分配在一个词法范围内：

```
for (i=.... ) {
    real *block = malloc ( /* large number of bytes */ )
    /* do something with that block of memory */
    /* and forget to call "free" on that block */
  }
```

内存块在每个迭代中都被分配，但是一个迭代的分配在下一个迭代中不再可用，一个类似的例子可以用在条件中分配来做。应该指出的是，这个问题在 *Fortran* 中要轻得多，因为在 *Fortran* 中，当一个变量超出范围时，内存会被自动分配。有各种检测内存错误的工具，如 *Valgrind* , *DMALLOC*，*Electric Fence*。关于 *Valgrind*，见第 27.3.2.1 节。

### 27.3.3 使用自己的内存分配函数

许多编程错误来自不正确使用动态分配的内存：程序写到超出界限，或写入尚未分配的内存，或已经被释放的内存。虽然一些编译器可以在运行时进行边界检查，但这会降低程序的速度。一个更好的策略是编写我们

自己的内存管理。一些如 *PETSc* 库，已经提供了一个增强的 *malloc*。如果我们有这样的功能，我们当然应该利用它（*gcc* 编译器有一个 *mcheck* 函数，定义在 *mcheck.h* 中有类似的功能）。

如果我们用 *C* 语言写代码，我们可能会知道 *malloc* 和 *free* 调用：

```
int *ip;
ip = (int*) malloc(500*sizeof(int));
if (ip==0) {/* could not allocate memory */}
..... do stuff with ip .....
free(ip);
```

我们可以通过以下方式为自己节省一些写代码的时间：

```
#define MYMALLOC(a,b,c) \
a = (c*) malloc(b*sizeof(c)); \
if (a==0) {/* error message and appropriate action */}
int *ip;
MYMALLOC(ip,500,int);
```

运行时检查内存使用情况（通过编译器生成的边界检查，或者通过像 *valgrind* 或 *Rational Purify* 等工具）是很浪费的，但我们可以通过在我们的 *malloc* 中添加一些功能来发现许多问题，我们在这里要做的是在事后检测内存损坏。我们在被分配对象的左边和右边分配几个整数（下面代码的第 1 行），并在其中放入一个可识别的值（第 2 行和第 3 行）以及对象的大小（第 2 行），然后我们返回指针到实际请求的内存区域（第 4 行）。

```
#define MEMCOOKIE 137
#define MYMALLOC(a,b,c) { \
char *aa; int *ii; \
aa = malloc(b*sizeof(c)+3*sizeof(int)); /* 1 */ \
ii = (int*) aa; ii[0] = b*sizeof(c); \
ii[1] = MEMCOOKIE; /* 2 */ \
aa = (char*) (ii+2); a = (c*) aa; /* 4 */ \
aa = aa+b*sizesof(c); ii = (int*) aa; \
ii[0] = MEMCOOKIE; /* 3 */ \
}
```

现在我们可以写我们自己的 *free*，它可以测试对象的边界是否没有被写入过。

```
#define MYFREE(a) { \
char *aa; int *ii,; ii = (int*) a; \
if (*(--ii) !=MEMCOOKIE) printf("object corrupted\n"); \
n = *(--ii); aa = a+n; ii = (int*) aa; \
if (*ii!=MEMCOOKIE) printf("object corrupted\n"); \
}
```

我们可以扩展这个想法：在每一个被分配的对象中，也存储两个指针，这样分配的内存区域成为一个双链表。然后我们可以写一个 *CHECKMEMORY* 宏，测试所有分配的对象是否损坏。这种解决内存损坏问题的方法相当容易编写，而且开销很小。每个对象最多只有 5 个整数的内存开销，而且几乎没有性能损失。（在某些系统上，我们可以不为 *malloc* 写一个装饰器，而是改变系统的行为例程。在 *Linux* 上，*malloc* 调用的 *Hooks* 可以用我们自己的例程来代替；见 http://www.gnu.org/s/libc/manual/html_node/Hooks-for-Malloc.html）。

### 27.3.4 具体技术：Fortran

使用 *Implicit none*。把所有的子程序放在模块中，这样编译器就可以检查缺少的参数和不匹配的类型，它还允许用 *fdepend* 来自动建立依赖关系。使用 *C* 语言预处理器进行条件编译之类的工作。

## 27.4 测试

在测试代码的正确性方面有各种理念。

- 正确性证明：程序员写出描述部分代码预期行为的断言并通过数学方法证明这些断言的正确性。
- 单元测试：每个例程都要单独进行正确性测试，而对于数字码来说，这种方法通常很难做到。因为输入的浮点数，可能本质上是无穷大，并且知道什么能构成足够的输入集并不容易。
- 集成测试：测试子系统
- 系统测试：测试整个代码。这通常适用于数字码，因为我们常常有已知解的问题模型，或者有一些属性，如需要在全局解上保持不变的边界。
- 测试驱动设计：程序开发过程中可以随时进行以驱动为要求的测试。

在并行代码中我们遇到了一类新的测试困难。许多算法被执行并行时，会以稍微不同的顺序执行，导致出现不同的舍入行为。例如，并行的向量和计算会使用部分和。一些算法具有数值误差的固有阻尼，例如静态的迭代方法（第 5.5.1 节），但其他算法没有这种内置的误差修正（非稳态方法；5.5.8 节）。因此，同一个迭代过程可能需要不同的迭代次数，这取决于有多少个处理器被使用。

### 27.4.1 测试驱动设计和开发

在测试驱动设计中，非常强调代码总是可测试性。其基本思想如下：

- 整个代码和部分都应该是可测试的。
- 当扩展代码时，只做允许测试的最小的改变。
- 对于每一个变化，都要进行前后测试。
- 在添加新的功能之前保证正确性。

# 第 28 章 Debug

## 28.1 调试

调试就像在一部犯罪电影中当侦探，而你也是凶手。（Filepe Fortes，2013）

当一个程序出现问题时，调试就是找出问题原因的过程。有多种方法可以找到程序中的错误，最粗暴的一种是通过输出语句进行调试。如果你有一个概念，那就是产生错误的地方，你可以编辑你的代码，插入输入语句，重新编译，重新运行，看看输出是否给你建议。这样做有几个问题：

- 编辑/编译/运行循环是很耗时的。
- 特别是由于早期的代码部分引起的错误，需要你反复地编辑、编译和运行。
- 此外，你的程序产生的数据量可能太大，而导致无法有效显示和检查。
- 而且如果你的程序是并行的，你可能需要输出所有处理器的数据，这使得检查过程非常乏味。

由于这些原因，最好的调试方法是使用交互式调试器，能让你监测和控制运行中的程序。在本节中，我们将熟悉 *gdb* 和 *ldb*，它们分别是 *GNU* 和 *clang* 项目的开源调试器。其他调试器是专有的，通常与编译器配套出现。另一个区别是，*gdb* 是一个命令行的调试器；还有一些图形化的调试器，如 *ddd*（*gdb* 的前端）、*DDT* 和 *TotalView*（并行代码的调试器）。我们只讨论 *gdb* ，因为它包含了所有调试器所共有的基本概念。

在本教程中，你将用 *gdb* 和 *valgrind* 调试一些简单的程序，这些文件可以在资源库 *tutorials/debug_tutorial_files* 目录下找到。

## 28.2 调用调试器

有三种使用 *gdb* 的方法：用它来启动一个程序，把它添加到一个已经运行的程序上，或者用它来检查一个 *coredump*，我们只考虑第一种情况。启动调试器

| gdb | lldb |
| --- | --- |
| $ gdb program (gdb) run \| $ lldb program (lldb) run | |

下面是一个例子，说明如何用没有参数的程序启动 *gdb*（*Fortran* 用户使用 *hello.F*）。

```
      *tutorials/gdb/c/hello.c*

         #include <stdlib.h>
#include <stdio.h>
int main() {
       printf("hello world\n");
       return 0;
 }
 %% cc -g -o hello hello.c
 # regular invocation:
 %% ./hello
 hello world
 # invocation from gdb:
 %% gdb hello
 GNU gdb 6.3.50-20050815 # ..... version info
```

```
        Copyright 2004 Free Software Foundation, Inc. .... copyright info ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

              Program exited normally.
       (gdb) quit
       %%
```

重要提示：该程序是用调试标志 *-g* 编译的，这将导致符号表（即从机器地址到程序变量的转换）和其他调试信息会被包含在二进制文件中。这将使你的二进制文件比严格意义上需要空间的要大，但也会使它更慢，比如编译器将不执行某些优化等原因。为了说明符号表的存在执行以下命令

```
%% cc -g -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

并与不使用 *-g* 标志的情况进行比较。

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

对于一个有命令行输入的程序，我们给运行命令的参数（*Fortran* 用户使用 *say.F*）。

```
*tutorials/gdb/c/say.c*

 #include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
 int i;
 for (i=0; i<atoi(argv[1]); i++)
        printf("hello world\n");
        return 0;
  }
%% cc -o say -g say.c
%% ./say 2
hello world
hello world
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb/c/say 2
Reading symbols for shared libraries +. done
hello world
hello world
              Program exited normally.
```

## 28.3 查找错误

### 28.3.1 C 程序

```
        // square.c
int nmax,i;
float *squares,sum;
fscanf(stdin,"%d",nmax);
for (i=1; i<=nmax; i++) {
        squares[i] = 1./(i*i); sum += squares[i];
 }
 printf("Sum: %e\n",sum);
 %% cc -g -o square square.c
 %% ./square
 5000
 Segmentation fault
```

存储器区块错误（其他信息也有可能）表明我们正在访问我们不允许访问的内存，导致了程序中止，调试器会很快地告诉我们这种情况发生在哪。

```
%% gdb square
(gdb) run
50000
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()
```

| gdb | lldb |
| --- | --- |
| (gdb) where | (lldb) thread backtrace |

显然，错误发生在一个是我们写的 *__svfscanf_l* 函数中，而是一个来自系统的函数。使用 *backtrace*（或 *bt*，也有 *where* 或 *w*）命令，我们显示了调用栈，这通常可以让我们找出错误所在。

```
(gdb) backtrace
#0 0x00007fff824295ca in __svfscanf_l ()
#1 0x00007fff8244011b in fscanf ()
#2 0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at  square.c:7
```

我们仔细看一下第 7 行，发现我们需要将 *nmax* 改为 *&nmax* 。我们的程序中仍有一个错误：

```
(gdb) run
50000
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x0000000100000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9       squares[i] = 1./(i*i); sum += squares[i];
```

我们进一步调查：

```
(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
(gdb) print squares
$2 = (float *) 0x0
```

然后很快就会发现忘记了分配 squares 。如果我们有一个合法的数组，但我们访问它的边界外，也会发生内存错误。

```
// up.c
int nlocal = 100,i;
double s, *array = (double*) malloc(nlocal*sizeof(double));
for (i=0; i<nlocal; i++) {
       double di = (double)i;
       array[i] = 1/(di*di);
 }
 s = 0.;
 for (i=nlocal-1; i>=0; i++) {
       double di = (double)i;
       s += array[i];
       }
  Program received signal EXC_BAD_ACCESS, Could not access memory.
  Reason: KERN_INVALID_ADDRESS at address: 0x0000000100200000
  0x0000000100000f43 in main (argc=1, argv=0x7fff5fbfe2c0) at up.c:15
  15 s += array[i];
  (gdb) print array
  $1 = (double *) 0x100104d00
  (gdb) print i
  $2 = 128608
```

### 28.3.2 Fortran 程序

编译并运行以下程序：

*tutorials/gdb/f/square.F*

```
      Program square
      real squares(1)
      integer i
      do i=1,100
            squares(i) = sqrt(1.*i)
            sum = sum + squares(i)
       end do
       print *,"Sum:",sum
       End
```

它会以’’ 非法指令’’ 这样的信息中止，在 *gdb* 中运行该程序会很快告诉你问题所在：

```
(gdb) run
Starting program: tutorials/gdb//fsquare
Reading symbols for shared libraries ++++. done
Program received signal EXC_BAD_INSTRUCTION,
Illegal instruction/operand.
0x0000000100000da3 in square () at square.F:7
7 sum = sum + squares(i)
```

我们仔细看了一下代码，发现没有正确地分配 *squares*。

## 28.4 内存调试

编程中的许多问题都源于内存错误，我们从最常见类型的排序描述开始，然后讨论如何使用帮助你检测它们的工具。

### 28.4.1 内存错误的类型

#### 28.4.1.1 无效的指针

引用一个没有指向对象的指针会导致错误。如果你的指针指向到有效的内存中，你的计算将继续进行，但结果是不正确的。然而，更有可能的是，你的程序会因为段违规或总线错误而终止。

#### 28.4.1.2 越界错误

在已分配对象的范围之外寻址，不太可能使你的程序崩溃，而更有可能的是带来错误的结果。越界的量足够大时将再次出现段违规，但是，少量的超出边界可能会读取无效的数据，或者破坏其他变量的数据，从而产生可能很长一段时间不被发现的错误结果。

#### 28.4.1.3 内存泄漏

如果所分配的内存变得无法访问，我们称这种情况为内存泄漏。例子如下：

```
if (something) {
        double *x = malloc(10*sizeofdouble);
        // do something with x
 }
```

在 *if* 条件语句后，分配的内存没有被释放，但指向的指针已经消失了。尤其是最后一种类型，可能很难发现，内存泄漏只有在你的程序耗尽时才会出现。因为你的分配会失败，所以可以通过此来发现，经常检查 *malloc* 和 *allocate* 返回的结果是一个好习惯。

### 28.4.2 内存工具

#### 28.4.2.1 Valgrind

在你的程序中插入下面的 *squares* 分配:

```
squares = (float *) malloc( nmax*sizeof(float) );
```

编译并运行你的程序，尽管程序不正确。输出很可能是正确的。你能看到问题吗？要发现这种微妙的内存错误，你需要一个不同的工具：内存调试工具。流行的工具是 *valgrind*（因为开源）；常见的商业工具是 *purify* 。

tutorials/gdb/c/square1.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
        int nmax,i;
        float *squares,sum;
        fscanf(stdin,"%d",&nmax);
        squares = (float*) malloc(nmax*sizeof(float));
        for (i=1; i<=nmax; i++) {
                squares[i] = 1./(i*i);
                sum += squares[i];
         }
         printf("Sum: %e\n",sum);
         return 0;
}
```

用 *cc -o square1 square1.c* 编译这个程序，用 *valgrind square1* 运行它（你需要输入变量值）。你会得到很多输出，首先是:

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==53695== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==53695== Command: a.out
==53695==
10
==53695== Invalid write of size 4
==53695== at 0x100000EB0: main (square1.c:10)
==53695== Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695== at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695== by 0x100000E77: main (square1.c:8)
==53695==
==53695== Invalid read of size 4
==53695== at 0x100000EC1: main (square1.c:11)
==53695== Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695== at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695== by 0x100000E77: main (square1.c:8)
```

*Valgrind* 尽管提供的信息丰富，但信息晦涩难懂，因为它是在裸露的内存上工作，而不是变量。因此，这些报错信息需要一些解释。第 10 行中，在分配了 40 字节的块之后，立即写入了一个 4 字节的对象。换句话说：这段代码是在分配的数组的边界外写入的。你能知道代码中的问题是什么吗？注意，*Valgrind* 也会在程序运行结束时报告有多少内存仍在使用，也就是没有适当释放的内存。如果你修复了数组的出界问题，并重新编译并运行程序，*Valgrind* 仍然会报错：

```
==53785== Conditional jump or move depends on uninitialised value(s)
==53785== at 0x10006FC68: __dtoa (in /usr/lib/libSystem.B.dylib)
==53785== by 0x10003199F: __vfprintf (in /usr/lib/libSystem.B.dylib)
==53785== by 0x1000738AA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==53785== by 0x1000A1006: printf (in /usr/lib/libSystem.B.dylib)
==53785== by 0x100000EF3: main (in ./square2)
```

虽然没有给出行号，但提及 *printf* 表明了问题的所在。提到'' 未初始化的值'' 又令人费解：唯一被输出的值是 *sum*，而这并不是未被初始化：它已经被添加了几次。你知道为什么 *Valgrind* 说它未被初始化了吗?

#### 28.4.2.2 Electric fence

electric fence 库是许多提供具有调试支持的新 *malloc* 的工具之一，这些工具被连接起来，代替了标准 *libc* 的 *malloc*。

```
cc -o program program.c -L/location/of/efence -lefence
```

## 28.5 逐步调试程序

通常情况下，程序中的错误足够隐蔽，你需要详细调查程序的运行情况。编译以下程序：

tutorials/gdb/c/roots.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
float root(int n){
     float r;
     r = sqrt(n);
     return r;
  }
  int main() {
  feenableexcept(FE_INVALID | FE_OVERFLOW);
      int i;
      float x=0;
      for (i=100; i>-100; i--)
             x += root(i+5);
             printf("sum: %e\n",x);
             return 0;
       }
```

然后运行程序：

```
%% ./roots
sum: nan
```

像以前一样在 *gdb* 中启动它:

```
%% gdb roots
GNU gdb 6.3.50-20050815
Copyright 2004 Free Software Foundation, Inc.
....
```

但在你运行程序之前，你在 *main* 处设置一个断点。在主程序中，这告诉执行器停止执行，或者说是'' 中断''。

```
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
```

现在，程序将在 *main* 的第一个可执行语句处停止：

```
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14 float x=0;
```

大多数时候，你会在一个特定的行上设置断点：

| gdb | clang |
|---|---|
| break foo.c:12 | breapoint set -f |

如果执行在断点处停止，你可以做各种事情，如发出 *step* 命令：

```
Breakpoint 1, main () at roots.c:14
14 float x=0;
(gdb) step
15 for (i=100; i>-100; i--)
(gdb)
16 x += root(i);
(gdb)
```

(如果你只是按了回车键，先前发出的命令会被重复)。连续执行若干步骤按回车键，你注意到这个函数和循环是什么了吗?

从执行 *step* 切换到执行 *next*，现在你注意到这个循环和这个函数是什么了吗? 设置另一个断点：*break 17*，然后执行 *cont* 。会发生什么? 在你在调用 *sqrt* 的那一行设置断点后，重新运行程序，当执行停止时在那执行 *where* 和 *list* 。

## 28.6 观察变量值

在 *gdb* 中再次运行之前的程序：在实际调用 *run* 之前，在调用 *sqrt* 的那一行设置一个断点。当程序运行到第 8 行时，你可以执行 *print n*。执行 *cont*，程序在哪里停止? 如果你想修复一个变量，你可以做 *set var=value* 。改变变量 *n* 并确认新值的平方根被计算出来，你要执行哪些命令?

## 28.7 断点

如果在一个循环中出现问题，不断地输入 *cont* ，用 *print* 检查变量会很繁琐。相反，你可以在现有的断点上增加一个条件。首先，你可以使断点受到条件约束：通过

```
condition 1 if (n<0)
```

只有当 *n<0* 为真时，断点 1 才会被遵从。你也可以有一个只由某些条件激活的断点。例如以下语句

```
break 8 if (n<0)
```

意味着在遇到条件 *n<0* 后，8 号断点会（无条件地）激活。另一种可能是使用 *ignore 1 50*，这样就可以不停地执行断点 1 在接下来的五十次。删除现有的断点，用 *n<0* 的条件重新定义断点，然后重新运行你的程序。当程序中断时，找出哪个值是循环变量，你所使用的命令的顺序是什么？你可以通过各种方式设置断点：

- *break foo.c* 当达到某个文件的代码时停止。
- *break 123* 在当前文件的某一行停止。
- *break foo* 在子程序 *foo* 处停止。
- 或各种组合，如 *break foo.c:123* 。
- 最后，如果你设置了多断点，你可以用 *info breakpoints* 来找出它们。
- 你可以用 *delete n* 删除断点，*n* 是断点的编号。
- 如果你在不退出 *gdb* 的情况下用 *run* 重启你的程序，断点会一直有效。
- 如果你退出 *gdb*，断点将被删除，但你可以保存它们：*save breakpoints <file>*。在下一次运行 *gdb* 时使用 *source <file>* 来读入它们。
- 在有 *exceptions* 的语言中，比如 *C++* 中你可以设置一个 *catchpoint*:

| gdb | clang |
|---|---|
| catch throw | break set -E C++ |

最后，你可以在断点处执行命令：

```
break 45
command
print x
cont
end
```

这说明在第 45 行要输出变量 *x*，并立即继续执行。如果你想在同一个程序上重复运行 *gdb* 会话，你可能想保存和重新加载断点。可以用

```
save-breakpoint filename
source filename
```

## 28.8 并行调试

并行调试比顺序调试更难，因为我们遇到只由于进程间的交互而产生的错误，例如死锁；见 2.6.3.6 节。考虑这段 *MPI* 示例代码：

```
        MPI_Init(0,0);
        // set comm, ntids, mytid
        for (int it=0; ; it++) {
              double randomnumber = ntids * ( rand() / (double)RAND_MAX );
              printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);
              if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
                    MPI_Finalize();
               }
               MPI_Finalize();
```

每个进程都计算随机数，直到某个条件得到满足，然后退出。然而，考虑引入一个 *barrier*（或类似于 *barrier* 的东西，如 *reduction*）：

```
        for (int it=0; ; it++) {
                double randomnumber = ntids * ( rand() / (double)RAND_MAX );
                printf("[%d] iteration %d, random %e\n",mytid,it,randomnumber);
                if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
                MPI_Finalize();
MPI_Barrier(comm);
}
MPI_Finalize();
```

现在执行会挂起，这不是由于任何特定的进程造成的：每个进程都有一个代码路径从 *init* 到 *finalize* 且不会产生任何内存错误或其他运行时错误的代码路径。然而，一旦一个进程到达条件中的 *finalize* 调用时，它就会停止，而所有其他进程将在 *barrier* 等待。

图 27.1 在这段代码停止的地方展示了 *Allinea DDT* 调试器的主要显示（http://www.allinea.com/product/ddt）。在源面板的上方，你看到有 16 个进程，并给出了进程 1 的状态。在底部显示中，你看到 16 个进程中 15 个进程在第 19 行调用 *MPI_Barrier*，而一个进程在第 18 行。在右边的显示中，你可以看到一个列表的局部变量：进程 1 的特定值。一个基本的图表显示了处理器上的进程的值：*ntids* 的值是恒定的，*mytid* 的值是线性增加的，除了一个进程外，其他都是恒定的。

✍ **练习 28.1** 编写并运行 *ring_1a* 程序。该程序没有终止，也不会崩溃。在调试器中，你可以中断执行，并看到所有进程都在执行一个接收语句。这可能是一个死锁的案例，诊断并修复这个错误。

✍ **练习 28.2** *ring_1c* 的作者对 *MPI* 的工作原理非常迷惑。运行该程序，虽然它的终止没有问题，但输出是错误的。设置一个断点在发送和接收语句中设置断点，以弄清发生了什么。

## 28.9 延伸阅读

教程：http://www.dirac.org/linux/gdb/ 参考手册：http://www.ofb.net/gnu/gdb/gdb_toc.html

# 第 29 章 C、Fortran 与 Python

## 29.1 C/Fortran 的互用性

大多数情况下，一个程序是用一种语言编写的，但在某些情况下，有可能会在一个可执行文件中混合使用一种以上的语言。其中一种情况是当一个库是用一种语言编写的时候，但却被另一种语言的程序使用。在这种情况下，库的编写者可能让我们很轻易地使用这个库；本节是让我们发现自己处于库作者的位置。我们将集中讨论 *C/C++* 和 *Fortran* 之间的互用性。

这个问题很复杂，因为这两种语言已经存在了很长时间，而最近各种的语言标准已经引入了一些机制来促进互用性。然而，仍然有很多旧代码存在，而且并非所有的编译器都支持最新的标准。因此，我们同时讨论旧的和新的解决方案。

## 29.2 链接器规定

如上所述，编译器将源文件变成二进制文件，其中不再有任何源语言的痕迹：它实际上包含了机器语言的功能。然后，链接器将匹配可能在不同的文件中的调用和定义。使用多种语言的问题是编译器对如何将有不同的概念的函数名称从源文件翻译成二进制文件。

让我们看看代码（我们可以在 *tutorials/linking* 中找到示例文件）：

```
// C:
Subroutine foo()
Return
End Subroutine
! Fortran
void foo() {
return;
}
```

编译后，我们可以用 *nm* 来查看二进制对象文件：

```
%% nm fprog.o
0000000000000000 T _foo_
....
%% nm cprog.o
0000000000000000 T _foo
....
```

我们看，在内部，*foo* 例程有不同的名字：*Fortran* 的名字后面有一个下划线。这使得我们很难从 *C* 语言中调用 *Fortran* 例程，反之亦然。可能名称不匹配的是：

- *Fortran* 编译器附加了一个下划线，这是最常见的情况。
- 有时它可以附加两个下划线。
- 通常情况下，例程名称在对象文件中是小写的，但也有可能是大写的。

由于 *C* 是一种流行的编写库的语言，这意味着这个问题经常在 *C* 库中解决：

- 在所有的 *C* 函数名称上加一个下划线；
- 或包括一个简单的封装器调用：

```
int SomeCFunction(int i,float f){
      .....
 }
 int SomeCFunction_(int i,float f){
     return SomeCFunction(i,f);
 }
```

### 29.2.1 Fortran 2003 中的 C 语言绑定

在最新的 *Fortran* 标准中，有明确的 *C* 语言绑定，使我们可以声明变量和例程的外部名称：

```
module operator
      real, bind(C) :: x
contains
      subroutine s() bind(C,name=' s' )
      return
 end subroutine
 end module
 %% ifort -c fbind.F90
 %% nm fbind.o
 .... T _s
 .... C _x
```

也可以将数据类型声明为与 C 语言兼容：

```
Program fdata
use iso_c_binding
type, bind(C) :: c_comp
     real (c_float) :: data
     integer (c_int) :: i
     type (c_ptr) :: ptr
end type
end Program fdata
```

### 29.2.2 C++ 连接

用 *C++* 编写的库提供了进一步的问题，*C++* 编译器通过在一个 *name mangling* 的进程合并类和方法的名称来制造外部符号。我们可以迫使编译器生成对其他语言可理解的名称，方法是：

```
#ifdef __cplusplus
    extern"C" {
#endif
 .
.
place declarations here
.
.
#ifdef __cplusplus}
#endif
```

例如，编译以下内容

```
#include <stdlib.h>
int foo(int x) {
     return x;
     }
```

并用 *nm* 检查输出，得到的结果是:

```
0000000000000010 s EH_frame1
0000000000000000 T _foo
```

另一方面，将相同的程序编译为 *C++* 语言后，可以得到

```
0000000000000010 s EH_frame1
0000000000000000 T __Z3fooi
```

我们可以看到 *foo* 的名字是一些杂乱无章的东西，所以我们不能从一个不同语言的程序中调用这个程序。另一方面，如果我们加上 *extern* 声明:

```
#include <stdlib.h>
#ifdef __cplusplus
extern"C" {
#endif
int foo(int x) {
    return x;
}
#ifdef __cplusplus
}
#endif
```

我们又得到了与 *C* 语言相同的链接器符号，因此该例程可以从 *C* 和 *Fortran* 中调用。如果我们的主程序是 *C* 语言，我们可以使用 *C++* 编译器作为链接器。如果主程序是用 *Fortran* 编写的，我们需要使用 *Fortran* 编译器作为链接器。然后有必要为 *C++* 的系统例程连接额外的库。例如，在英特尔编译器中，需要在链接行中加入*-lstdc++ -lc* 。如果我们将其他语言链接到 *C++* 主程序中，也需要使用 *extern* 的方法。例如，一个 *Fortran* 子程序 *foo* 应该被声明为:

```
extern "C" {
        void foo_();
}
```

在这种情况下，我们再次使用 *C++* 编译器作为链接器。

### 29.2.3 复数

*C/C++* 和 *Fortran* 中的复杂数据类型是相互兼容的。下面是一个 *C++* 的例子程序与 *Lapack* 的复杂向量缩放例程 *zscal* 连接的例子。

```
        // zscale.cxx
extern "C" {
     void zscal_(int*,double complex*,double complex*,int*);
```

```
}
complex double *xarray,*yarray, scale=2.;
xarray = new double complex[n]; yarray = new double complex[n];
zscal_(&n,&scale,xarray,&ione);
```

## 29.3 数组

*C* 和 *Fortran* 对于存储多维数组有不同的规定，当我们在不同语言编写的程序之间传递一个数组时，我们需要意识到 *Fortran* 以列为主的顺序存储多维数组，见图 28.1。对于二维数组 *A(i,j)*，这意味着每一列中的元素是连续存储的：一个 2Œ2 的数组被存储为 *A(1,1), A(2,1), A(1,2), A(2,2)*。三维和更高维的数组是一个明显的延伸：有时会说'' 左边的索引变化最快''。*C* 语言的数组是以行为主的顺序存储的：每一行的元素都是连续存储的，然后列是按顺序放在内存中。在内存中按顺序排列。一个 2Œ2 的数组 A[2][2] 被存储为 A[1][1],A[1][2], A[2][1] ,A[2][2]。

关于 *C* 语言中的数组的一些备注：

- *C*（在 *C99* 标准之前）只在有限的意义上有多维数组。我们可以声明它们，但如果我们把它们传递给另一个 *C* 语言函数，它们就不再是多维的了：它们变成了普通的 *float* *（或其他类型）数组。
- C 语言中的多维数组看起来好像是 *float* ** 类型的，也就是说，一个指针数组指向（单独分配的）行数组。我们肯定可以实现这一点：

```
        float **A;
A = (float**)malloc(m*sizeof(float*));
for (i=0; i<n; i++)
      A[i] = (float*)malloc(n*sizeof(float));
```

仔细阅读标准可以发现，多维数组实际上是一个单一的内存块，不涉及进一步的指针。鉴于上述对传递多维数组的限制，以及 *C* 程序无法分辨它是由 *Fortran* 还是 *C* 语言调用的事实，最好不要在 C 语言中使用多维数组，而应仿效它们:

```
float *A;
A = (float*)malloc(m*n*sizeof(float));
#define SUB(i,j,m,n) i+j*m
for (i=0; i<m; i++)
for (j=0; j<n; j++)
       ... A[SUB(i,j,m,n)] ....
```

其中，为了实现互用性，我们以列为主的方式存储这些元素。

### 29.3.1 数组排列

由于 *SIMD* 向量指令等原因，使用对齐分配（*aligned allocation*）可能是有利的。例如，"16 字节对齐'' 意味着我们的数组的起始地址，用字节表示，是 16 的倍数。在 *C* 语言中，我们可以用 *posix memalign* 强制进行这种对齐。在 *Fortran* 中，没有通用的机制。英特尔的编译器允许这样写：

```
double precision, allocatable :: A(:), B(:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A, B
```

## 29.4 字符串

编程语言在处理字符串的方式上有很大不同。

- 在 *C* 语言中，字符串是一个字符数组；字符串的结束是由一个空字符表示的，即 *ascii* 码 0，它有一个全零的比特模式。被称为空终止（*null termination* ）
- 在 *Fortran* 中，一个字符串是一个字符数组。长度被保存在一个内部变量中，作为一个隐藏参数传递给子程序。
- 在 *Pascal* 中，一个字符串是一个数组，在第一个位置有一个表示长度的整数。由于只有一个字节，所以在 *Pascal* 中，字符串的长度不能超过 255 个字符。

正如我们所看到的，在不同语言之间传递字符串是充满危险的。这种情况由于将字符串作为子程序参数传递并不是标准的，而变得更加糟糕。例子：*Fortran* 中的主程序传递一个字符串

```
        Program Fstring
character(len=5) :: word = "Word"
call cstring(word)
end Program Fstring
```

和 C 程序接收一个字符串和它的长度：

```
#include <stdlib.h>
#include <stdio.h>.
void cstring_(char *txt,int txtlen) {
printf("length = %d\n",txtlen);
printf("<<")。
for (int i=0; i<txtlen; i++)
      printf("%c",txt[i])
      printf(">>\n");
 }
```

这输出了：

```
length = 5
<<Word >>
```

要将一个 *Fortran* 字符串传递给 *C* 程序，我们需要附加一个空字符：

```
call cfunction (’ A string’ //CHAR(0))
```

一些编译器支持扩展，为促进这一点，编写：

```
DATA forstring /’ This is a null-terminated string.’ C/
```

最近，” *C/Fortran* 互用作性标准’’ 为此提供了一个系统的解决方案。

## 29.5 子程序参数

在 *C* 语言中，我们可以向该函数传递一个它需要的 *float* 参数，或当该函数要在调用环境中修改变量的值时传递 *float* *。*Fortran* 没有这种区别：每个变量都是通过引用传递的。这有一些奇怪的后果：如果我们把一个值 37 传给一个子程序，编译器会用这个值分配一个无名变量，并传递它的地址，而不是该值。对于 *Fortran* 和 *C* 程

序的接口，这意味着一个 *Fortran* 程序在 *C* 程序看来，其所有的参数都是"*" 参数。反过来说，如果我们想让一个 *C* 语言的子程序可以从 *Fortran* 中调用，那么它所有的参数都必须是"*" 的。这意味着一方面，我们有时会通过引用传递一个变量而我们却想用值来传递。

更糟的是，这意味着 *C* 语言的子程序如下：

```
void mysub(int **iarray) {
*iarray = (int*)malloc(8*sizeof(int));
return;
}
```

不能从 *Fortran* 中调用。有一个方法可以解决这个问题（请看 *Fortran77* 的接口到 *Petsc* 例程 *VecGetValues* 的接口），如果更聪明的话，我们可以使用 *POINTER* 变量来解决这个问题。

## 29.6 输入/输出

两种语言都有自己的处理输入/输出的系统，而且不可能真正遇到。基本上，如果 *Fortran* 例程做 *I/O*，主程序就必须是 *Fortran* 的。因此，最好是尽可能地隔离 *I/O* ，并在混合语言编程中使用 *C* 来处理 *I/O* 。

## 29.7 Fortran2003 中的 Fortran/C 互用性

目前许多编译器都不支持最新版本的 *Fortran*，但它有与 *C* 语言对接的机制。

- 有一个模块包含了命名的种类，这样就可以声明

```
INTEGER,KIND(C_SHORT) :: i
```

- *Fortran* 的指针是比较复杂的对象，所以把它们传递给 C 是很困难的；*Fortran2003* 有一个机制来处理 *C* 语言的指针，它只是地址。
- 可以使 *Fortran* 派生类型与 *C* 结构兼容。

## 29.8 Python 调用 C 代码

由于计算效率高，*C* 语言是用于程序最低层的合理语言。另一方面，由于它的表达能力，*Python* 是最上层的良好候选语言。那么想从一个 *Python* 程序中调用 *C* 程序是一个合乎逻辑的想法。这可以通过 *python ctypes* 模块来实现。

1. 我们写好我们的 *C* 语言代码，并如上所述将其编译成一个动态库。
2. *python* 代码动态地加载库，例如 *libc*：

```
path_libc = ctypes.util.find_library("c")
libc = ctypes.CDLL(path_libc)
libc.printf(b"%s\n", b"Using the C printf function from Python ... ")
```

3. 我们需要在 python 中声明 C 程序的类型是什么：

```
test_add = mylib.test_add
test_add.argtypes = [ctypes.c_float, ctypes.c_float]
test_add.restype = ctypes.c_float
test_passing_array = mylib.test_passing_array
test_passing_array.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int]
test_passing_array.restype = None
```

4. 标量可以简单传递，数组则需要构建：

```
data = (ctypes.c_int * Nelements)(*[x for x in range(numel)])
```

# 第 30 章 位操作

## 30.1 位操作

在本书的大部分内容中，我们将数字，如实数的整数或浮点表示，作为我们最基本的构建模块。然而，有时候，我们有必要更深入地考虑这些数字的实际这些数字在比特方面的表示。各种编程语言都支持位操作。我们将探讨各种选择。

## 30.2 结构和展示

### 30.2.1 C/C++

显示的内置可能性是有限的：

```
                printf("Octal: %o",i);
printf("Hex : %x",i);
```

提供了八进制和十六进制的表示方法，但没有二进制的格式说明。取而代之的是以下：

```
                printBits(size_t const size, void const * const ptr){
                unsigned char *b = (unsigned char*) ptr;
                unsigned char byte;
                for (int i=size-1; i>=0; i--) {
                          for (int j=7; j>=0; j--) {
                                    byte = (b[i] >> j) & 1;
                                    printf("%u", byte);
                           }
                 }
                 }
/* ... */
printBits(sizeof(i),&i);
```

### 30.2.2 Python

- *python* 的 *int* 函数将一个字符串转换为 *int* ，第二个参数可以指出字符串的基数是字符串的解释：

```
five = int(’101’,2)
maxint32 = int(’0xffffffff’,16)
```

- 函数 *bin hex* 分别将一个 *int* 转换为以 2,8,16 为基准的字符串。
- 由于 *Python* 中的整数可以是无限长的，所以有一个函数可以确定位的长度 ( *Python 3.1* 版)：*i.bit_length()*。

## 30.3 位操作

布尔运算通常被应用于编程语言的布尔数据类型，有些语言允许你将它们应用于实际的比特。

| | boolean | bitwise (C) | bitwise (Py) |
|---|---|---|---|
| and | & | | |
| or | 11 | 1 | 1 |
| not | ! | | ~ |
| xor | | – | |

此外，还有对比特串这样的操作：

| | |
|---|---|
| left shift | ≪ |
| right shift | ≫ |

✍ **练习 30.1** 使用位操作来测试一个数是奇数还是偶数

移位操作有时被用作算术的有效速记，例如，一个左移一个位置对应于乘以 2。

✍ **练习 30.2** 给定一个整数 $n$，求小于 $n$ 的 8 的最大倍数

这种机制有时被用来分配对齐的内存。写一个例程

```
aligned_malloc( int Nbytes, int aligned_bits );
```

分配了 *Nbytes* 的内存，其中第一个字节的地址是 *aligned_bits* 的倍数。

# 第 31 章 用于科学文献的 LaTeX

## 31.1 LaTeX 理念及历史

TeX 是一个可以追溯到 1970 年代末的排版系统。在那时候，用图形终端就像微软的 *Word* 那样可以设计一个文档布局并立即查看它是很少见的。相反，*TeX* 使用了一个两步的工作流程，我们首先使用我们最喜欢的文本编辑器在一个 *ascii* 文档中输入我们的格式指令。接下来，我们将调用 LaTeX 程序作为一种编译器，把这个文件翻译成可以打印或查看的形式。

```
%% edit mydocument.TeX
%% LaTeX mydocument
%% # print or view the resulting output
```

这个过程与通过输入 *HTML* 命令制作网页相似。

这种工作方式可能看起来很笨拙，但它有一些优点。例如，*TeX* 的输入文件是纯粹的 *ascii*，所以它们可以很容易地被自动生成，例如从数据库中生成。而且，我们可以用我们喜欢的任何编辑器来编辑它们。另一个有利于 *TeX* 的地方是，布局是由以某种编程语言编写的命令来指定的。这有一些重要的结果：

- 关注点分离：当我们在写文件时，我们不必考虑到布局。我们给出一个'' 章节'' 命令，而该命令的执行将由我们独立决定。例如，由我们选择一个文档样式。
- 通过在输入文件中选择不同的实现方式，可以很容易地改变已完成的文档的布局。在输入文件中选择不同的布局命令，就可以很容易地改变完成的文件的布局：使用相同的'' 章节'' 命令，但通过选择不同的风格，产生的布局是不同的。但通过选择不同的风格，所产生的布局是不同的。这种改变可以简单到只用一行改变文件的样式声明。
- 如果我们有不寻常的排版需求，也可以为此编写新的 TeX 命令。对于许多需求，这样的扩展实际上已经写好了；见第 30.4 节。

### 31.1.1 安装 LaTeX

在我们的系统上安装 *LaTeX* 的最简单方法是下载 *TeXlive* 发行版，地址是 http://tug.org/TeXlive 。苹果用户也可以使 *fink* 或 *macports*，有各种 *TeX* 的前端，例如 *Mac* 上的 *TeXshop*。

### 31.1.2 运行 LaTeX

目的：在本节中，我们将运行 LaTeX 编译器。最初，LaTeX 编译器会输出一种名为 *dvi* 独立于设备的文件格式，，它可以被翻译成 *PostScript* 或 *PDF* 或直接输出。如今，许多人使用 *pdfLaTeX* 程序，直接将 *.TeX* 文件翻译成 *.pdf* 文件。这有一个很大的好处，即生成的 *PDF* 文件有自动交叉链接和带目录的侧板。

让我们做一个简单的例子。

✍ **练习 31.1** 创建一个文本文件 *minimal.TeX*，内容如图 30.1 所示。尝试使用 *pdfLaTeX minimal* 或 LaTeX *minimal* 命令。在第一种情况下，我们是否得到了 *minimal.pdf* 文件，或者在第二种情况下得到 *minimal.dvi* 文件？请分别使用 *pdf* 阅读器，如 *Adobe Reader*，或 *dvips* 来查看输出。

注意事项：如果我们打错了字，*TeX* 可能有些不友好。如果我们得到一个错误信息并且 *TeX* 要求输入，输入 *x* 通常会让我们退出，或者 *Ctrl-C* 。有些系统允许我们输入 *e* ，直接进入编辑器纠正错字。

## 31.2 对 LaTeX 的简单介绍

在这里，我们将得到一个非常简单的 LaTeX 功能的介绍。还有各种更深入的教程，例如 *Oetiker* 的教程。

### 31.2.1 文档结构

每个 LaTeX 文档都需要以下几行：

```
\documentclass{ .... } % the dots will be replaced
\begin{document}
\end{document}
```

'*documentclass*' 一行需要在大括号之间输入一个类别名称；典型的值是'*article*' 或' *book*'。一些组织有自己的风格，例如，'*ieeeproc*' 是指 *IEEE* 的会议记录。

所有的文档文本都在 *begin{document}* 和 *end{document}* 行之间。(匹配的'*begin*' 和'end' 行被称为表示一个'*environment*'，在这里是指文档环境)。

在 *\begin{document}* 之前的部分被称为'' 序言''。它包含了对这个特定文档的定制。例如，一个让整个文档变成双倍行距的命令将被放在序言中。如果我们使用 *pdfLaTeX* 来格式化我们的文档，我们需要有一行

```
\usepackage{hyperref}
```

我们有没有注意到以下几点?

- 反斜线字符很特别：它是 LaTeX 命令的开始。
- 大括号也很特别：它有各种功能，比如表示一个命令的参数。
- 百分号字符表示到行末的所有内容都是注释。

### 31.2.2 一些简单的文本

**目的**：在本节中，我们将学习一些文本格式化的基本知识。

✍ **练习 31.2** 创建一个 *first.TeX* 文件，其中包含图 30.1 的内容。在序言中输入一些文字，也就是在'*begin{document}*' 行之前，在文件中运行 *pdfLaTeX* 。

预期结果：我们应该会得到一个报错信息，因为不允许在序言中出现文字，这里只允许存在命令；所有的文字都必须放在'*begin{document}*' 之后。

当 shell 通过使用我们的一个或多个输入行或只使用一个输入行的一部分来收集要执行的命令行时，如刚才所述，它将对命令行进行扩展（第 20.7 节）。然后，它将把命令行解释为一个命令和参数，并继续用找到的参数来调用该命令。编辑我们的文档：在 *

begindocument 和 enddocument 之间添加一些文字。让我们的文本有一些的持续一段时间的长行，还有一些短行。在字与字之间、行与行之间的开头或结尾处加上空格。在我们的文档上运行 *pdfLaTeX* 并查看输出。

预期结果：我们注意到我们输入的空白处在输出中已经被压缩了。*TeX* 有它自己关于空格应该是什么样子的的概念，我们不需要关心这个问题。

✍ **练习 31.3** 再次编辑我们的文档，剪切并粘贴这段文字，但在两份副本之间留出一行空白。第三次粘贴，留下几个空行。格式化，并查看输出结果。

预期结果：TeX 将一个或多个空行解释为段落之间的分隔。

✍ **练习 31.4** 在序言中加入 *usepackagepsLaTeX* ，然后在我们的文档中重新运行 *pdfLaTeX* 。输出结果有什么变化?

预期的结果：这样做的效果是，将字体从默认的改为 *Times Roman*。

注意事项：字体是出了名的不标准化，试图使用不同的字体可能不起作用。对此，一般来说没有什么可说的。

在第一段前添加以下一行："'latex chapterThis is a section"' 和第二行之前的类似的行。我们可以看到 *LaTeX* 会自动对章节进行编号，并且它对标题后的第一段的缩进处理是不同的。

✍ **练习 31.5** 在 *documentclass* 声明行中用 *artikel3* 替换 *article*，然后重新格式化我们的文档。有什么变化?

预期结果：有许多文档类实现了与 *article*（或其他标准样式）一样的命令，但它们有自己的布局。我们的文档格式应该没有任何问题，但得到了一个更好看的布局。

注意事项：*artikel3* 类是现在大多数发行版的一部分，但如果它是一个未知的文档类，如果缺少这个类，或者我们的环境设置不正确，我们可能会得到一个关于未知文档类的报错信息。这取决于我们的安装，如果该文件丢失了，请从以下网站 http://tug.org/TeXmf-dist/TeX/LaTeX/ntgclass/ 下载文件并把它们放到我们的当前目录中；另见第 30.2.9 节。

### 31.2.3 关于数学的排版

**目的**：在本节中，我们将学习数学排版的基本知识。最初的 *TeX* 系统的目标之一是促进数学的设置，有两种方法在我们的文档中设置数学公式：

- 内联数学是段落的一部分，由美元符号划定。
- 展示公式（*Display math*）模式，顾名思义，就是通过自身显示。

✍ **练习 31.6** 把 '$x+y$' 放在一个段落中的某个地方，然后格式化我们的文件。将 [x+y] 放在段落中的某处，然后格式化。

预期结果：单一美元之间的公式被包含在我们声明它们的段落中。介于 [...] 之间的公式在显示中排版。

对于带有数字的显示方程，使用 equation 环境。尝试如此。

这里有一些数学公式中常见的事情。请务必试一下：

- 下标和上标：'$x_i^2$'，如果下标或上标是一个以上的符号需要进行分组：'$x_{i+1}^{2n}$'。如果我们在公式中需要一个括号，请使用：'{}'。
- 希腊字母和其他符号：'$\alpha \otimes \beta_i$'。
- 所有这些的组合'$\int_{t=0}^{\infty} tdt$'。

✍ **练习 31.7** 以最后一个例子为例，把它排成显示数学。我们看到与内联数学的区别了吗?

预期结果：TeX 试图不包括文本行之间的距离，即使在段落中有数学。由于这个原因，它对积分符号的界限的排版与展示公式不同。

### 31.2.4 引用

**目的**：在本节中，我们将看到 TeX 的交叉引用机制在发挥作用。

到目前为止，我们还没有看到 LaTeX 做了多少能让我们省事的事情。LaTeX 的交叉引用机制肯定会帮我们节省工作：LaTeX 插入的任何计数器（如章节编号）都可以通过标签来引用。因此，参考总是正确的。

从一个至少有两个章节标题的例子文件开始。在我们的第一个章节标题后面，把命令 ** ，并在第二节标题后加上 *。这些标签命令可以和章节命令放在同一行，也可以放在下一行。现在输出

```
As we will see in section\ref{sec:other}.
```

在第二节之前的段落中。( *tilde* 字符表示一个非断裂的空格)

✍ **练习 31.8** 进行这些编辑，并对文件进行格式化。我们看到关于未定义引用的警告了吗？看一下输出文件。再次格式化文件，并再次检查输出。我们的目录中有任何新文件吗?

预期结果：在第一次浏览文档时，TeX 编译器会把所有的标签和它们的值放在一个.aux 文件中。文件将显示一个双问号，表示任何的引用是未知的。在第二遍中，正确的值将被填入。

注意事项：如果在第二遍之后，仍然有未定义的参考文献，那么我们可能是打错了。如果我们使用 bibTeX 工具来处理文献引用，我们将经常需要三遍来解决所有的引用。

上面我们看到，equation 环境给出的是带有方程编号的数学计算。我们可以在这个环境中添加一个标签来指代方程编号。

✍ **练习 31.9** 在一个 equation 环境中写一个公式，并添加一个标签。在文本中的任何地方提到这个标签，格式化（两次）并检查输出。

预期结果：对公式来说，*label* 和 *ref* 命令的使用方法与对章节编号的使用方法相同。请注意，我们必须使用 begin/endequation 而不是用公式 [...] 。

### 31.2.5 列表

**目的**：在本节中，我们将看到列表的基本知识。
通过一个环境提供项目和编号的列表。

```
\begin{itemize}
\item This is an item;
\item this is one too.
\end{itemize}
\begin{enumerate}
\item This item is numbered;
\item this one is two.
\end{enumerate}
```

✍ **练习 31.10** 在我们的文档中添加一些列表，包括嵌套列表并检查输出。
预期结果：嵌套的列表将被进一步缩进，标签和编号样式会随着列表深度的增加而改变。

✍ **练习 31.11** 为 enumerate 列表中的一个项目添加一个标签，并引用它。
预期结果：同样，*label* 和 *ref* 命令像以前一样工作。

### 31.2.6 源代码和算法

作为一名程序员，我们经常希望在我们的著作中包括算法，有时甚至包括源代码。在本教程中，我们已经看到一些字符对 *LaTeX* 有特殊的意义，而不能输入这些字符并期待它们出现在输出中。由于有趣的字符经常出现在我们需要一个工具来解决这个问题：*verbatim mode*。

要在一个段落内显示代码位，我们可以使用 *令。这条令用两个相同的字符来限定其用两个相同的字符来限定其参数，这些字符不能出现在逐字文本中。例如，输出 if (x%5>0) ... 是由 `if (x\%5>0) { ... }`。

✍ **练习 31.12** 本书作者是如何这本书的作者是如何在文本中得到这个逐字命令的?
对于较长的逐字文本，需要单独显示，我们可以使用

```
begin{verbatim}
stuff
end{verbatim}
```

最后，为了将整个文件作为逐字记录包括在内，可以使用以上方法。
逐字文本是显示算法的一种方式，但还有更优雅的解决方案。例如，在本书中本书中使用了以下方法。

```
usepackage[algo2e,noline,noend]{algorithm2e}
```

### 31.2.7 图片

由于我们不能立即看到我们所输入的内容的输出，有时输出可能是一个惊喜。对于图形来说更是如此。*LaTeX* 没有处理图片的标准方法，但以下是一组常用的命令：

```
usepackage{graphicx} % this line in the preamble
includegraphics{myfigure} % in the body of the document
```

图片可以是多种格式中的任何一种，但如果使用 pdfLaTeX，则不能使用 PostScript 图形（扩展名为.ps 或.eps）不能被使用。由于我们的图片往往不合适，通常会有这样的内容：

```
includegraphics[scale=.5]{myfigure}
```

一个更大的问题是，如果图片放在我们声明的地方，它们可能会太大，无法在页面上放置。由于这个原因，它们通常被视为' *floating material* '。下面是一个典型的图片声明：

```
begin{figure}[ht]
    includegraphics{myfigure}
    caption{This is a figure}
    label{fig:first}
end{figure}
```

它包含以下内容:

- figure 环境用于'floating' 的图篇；它们可以直接放在声明的位置上。也可以放在下一页的顶部或底部，或者放在本章的结尾，等等。
- beginfigure 行的 [ht] 参数指出，我们的数字应该尝试放在这里；如果不成功，它应该放在下一页的顶部。其余可能的参数是 b 用于放置在页面的底部，或者 p 用于放置在页面上。例如

```
begin{figure}[hbp]
```

- 在图片的下面有一个包括图片号标题。
- 一个标签，这样我们就可以通过它的标签来参考图号：figreffig:first 。
- 当然还有图的材料，有各种方法来微调图的位置。例如

```
begin{center}
    includegraphics{myfigure}
end{center}
```

给出一个居中的图片。

### 31.2.8 参考文献目录

在我们的文件中引用论文和书籍的机制有点像交叉引用的机制，这里有一个 citethatbook 命令可以插入一个参考文献。但是，由于我们可能会在我们写的不止一个文档中引用一篇论文或一本书，LaTeX 允许我们建立一个文献参考数据库，而不是在文档的某个地方。

制作一个 mybibliography.bib 文件，内容如下：

```
@article{JoeDoe1985,
author = {Joe Doe},
title = {A framework for bibliography references},
journal = {American Library Assoc. Mag.},
year = {1985}
}
```

在我们的 mydocument.TeX 文件中放入

```
For details, refer to Doe\cite{JoeDoe1985} % somewhere in the TeXt

\bibliography{mybibliography} % at the end of the document
\bibliographystyle{plain}
```

格式化我们的文件，然后在命令行上输入：

```
bibTeX mydocument
```

并对我们的文件再进行两次格式化，现在里面应该有一个书目和一个正确的引文，我们还会看到文件 mydocument.bbl 和 mydocument.blg 已经被创建。

### 31.2.9 环境变量

在 Unix 系统中，当 TeX 试图找到一个包含文件时，它将查询 TeXINPUTS 环境变量。因此，我们可以为我们的样式和其他下载的包含文件创建一个目录，并将这个变量为该目录的位置。同样地，BIBINPUTS 变量指示的是 bibTeX 的书目文件（第 30.2.8 节）。

## 31.3 已完成的例子

下面的例子 demo.TeX 包含了上面讨论的许多元素。

```
\documentclass{artikel3}

\usepackage{psLaTeX,graphicx,amsmath,amssymb}
\usepackage{pdfLaTeX}

\newtheorem{theorem}{Theorem}

\newcounter{excounter}
\newenvironment{exercise}
    {\refstepcounter{excounter}
     \begin{quotation}\TeXtbf{Exercise \arabic{excounter}.} }
    {\end{quotation}}

\begin{document}
\title{SSC 335: demo}
\author{Victor Eijkhout}
\date{today}
\maketitle

\chapter{This is a section}
\label{sec:intro}

This is a test document, used in\cite{LaTeXdemo}. It contains a
discussion in section\ref{sec:discussion}.

\begin{exercise}\label{easy-ex}
    Left to the reader.
\end{exercise}
\begin{exercise}
    Also left to the reader, just like in exercise\ref{easy-ex}
\end{exercise}
\begin{theorem}
```

```
    This is cool.
\end{theorem}
This is a formula: $a\Leftarrow b$.
\begin{equation}
    \label{eq:one}
        x_i\leftarrow y_{ij}\cdot x{(k)}_j
\end{equation}
TeXt: $\int_01 \sqrt x\,dx$
\[
    \int_0 1 \sqrt x\,dx
\]
\chapter{This is another section}
\label{sec:discussion}

\begin{table}[ht]
    \centering
    \begin{tabular}{|rl|}
        \hline one&value \\ \hline another&values \\ \hline
    \end{tabular}
    \caption{This is the only table in my demo}
    \label{tab:thetable}
\end{table}

\begin{figure}[ht]
    \centering
    \includegraphics{graphics/caches}
    \caption{this is the only figure}
    \label{fig:thefigure}
\end{figure}
As I showed in the introductory section\ref{sec:intro}, in the
paper \cite{AdJo:colorblind}, it was shown that
equation\eqref{eq:one}
\begin{itemize}
\item There is an item.
\item There is another item
    \begin{itemize}
    \item sub one
    \item sub two
    \end{itemize}
\end{itemize}
\begin{enumerate}
\item item one
\item item two
    \begin{enumerate}
    \item sub one
    \item sub two
    \end{enumerate}
\end{enumerate}
```

```
\tableofcontents
\listoffigures

\bibliography{math}
\bibliographystyle{plain}

\end{document}
```

我们还需要 math.bib 这个文件:

```
@article{AdJo:colorblind,
author = {Loyce M. Adams and Harry F. Jordan},
title = {Is {SOR} color-blind?},
journal = {SIAM J. Sci. Stat. Comput.},
year = {1986},
volume = {7},
pages = {490--506},
abstract = {For what stencils do ordinary and multi-colour SOR have
the same eigenvalues.},
keywords = {SOR, colouring}
}

@misc{LaTeXdemo,
author = {Victor Eijkhout},
title = {Short {\LaTeX}\ demo},
note = {SSC 335, oct 1, 2008}
}
```

以下的命令顺序:

```
pdfLaTeX demo
bibTeX demo
pdfLaTeX demo
pdfLaTeX demo
```