第 8 章　矩阵向量乘法

Anarchy, anarchy! Show m a greater evil!
This is why cities tumble and the great housed rain down,
This is what scatters armies!
Sophocles, Antigone

8.1　概　　述

矩阵向量乘法在许多解决实际问题中都有应用。例如，许多求解线性方程的迭代算法就依赖于矩阵向量乘法，第 12 章中的共轭梯度法就是这样一个算法。

神经网络也是矩阵向量乘法的一个典型应用。神经网络可广泛地应用于诸如手写识别、石油勘探、航班座位分配和信用卡检测【114】等许多领域。要得到一个 k 级神经网络的输出，最直接的方法就是完成 k-1 次矩阵向量乘。而且，神经网络的主要训练方法是反向传播算法，而这个算法的核心也是矩阵向量乘法【98】。

为了完成稠密矩阵与向量相乘的算法，在本章中我们将设计、分析并实现三个 MPI 程序并对它们进行测试。这三个设计基于三种不同的在 MPI 进程间分解矩阵及向量中元素的方式。这些不同的数据分解方式导致了进程间的通信模式不尽相同，这也就是说我们将使用不同的 MPI 函数来分别进行这些程序中所涉及的数据传递。所以这三个程序两两差别都很大。

在开发这三个程序的过程中，我们将逐渐引入下面 4 个功能强大的 MPI 通信函数：

- MPI_Allgaterv，数据全收集函数，不同的进程可以提供不同数量的元素；
- MPI_Scatterv，散发数据操作，不同的进程可以获得不同数目的元素；
- MPI_Gatherv，数据收集操作，从不同进程收集来的元素的个数可以不同；
- MPI_Alltoall，全交换操作，在所有进程间交换数据元素。

我们还用到了 5 个支持面向格状结构的通信域：

- MPI_Dims_create，为平衡的笛卡儿进程网格创建新的维度；
- MPI_Cart_create，创建一个笛卡儿拓扑通信域；
- MPI_Cart_coords，返回笛卡儿进程网格中某个进程的坐标；
- MPI_Cart_rank，返回笛卡儿进程网格中位于某个坐标的进程的进程号；
- MPI_Comm_split，将一个通信域中的进程划分为一个或多个小组。

8.2　串 行 算 法

图 8.1 所示是向量和矩阵相乘的串行算法。又如图 8.2 所示，矩阵向量乘法是一系列

简单内积（或点积）的计算。两个 n 维向量的内积需要 n 次乘法和 $n-1$ 次加法，算法复杂度为$\Theta(n)$。矩阵向量乘法需要完成 m 个内积计算，所以算法复杂度为$\Theta(mn)$。如果矩阵是方阵，那么复杂度就变为$\Theta(n^2)$。

```
输入 : a[0..m-1,0..n-1]——m×n 的矩阵
       b[0..n-1]——n 维向量
输出: c[0..m-1]——m 维向量
for i ← 0 to m - 1
    c[i] ← 0
    for j ← 0 to n - 1
        c[i] ←c[i] + a[i,j] × b[j]
    endfor
    endfor
```

图 8.1 串行矩阵向量乘法

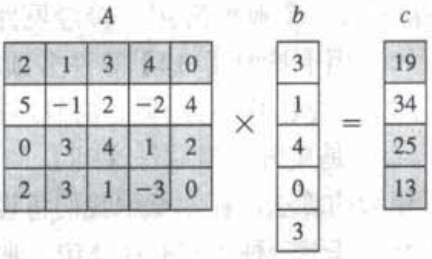

图 8.2 矩阵向量乘法可被看作是一系列内积（点积）操作

例如，c_1=5×3+（−1）×1+2×4+（−2）×0+4×3=34

8.3 数据分解方式

我们使用域分解的策略开发该并行程序。我们有很多种方法可以分解、聚合并映射矩阵和向量中的元素。不同数据分解方式会产生不同的并行算法。

我们有三种较为直接的分解矩阵的方法：按行分解、按列分解和棋盘式分解，如图 8.3 所示。假设待分解矩阵 A 的大小为 $m \times n$。我们已经见过按行分解的方法了：在第 6 章 Floyd 算法的实现中，我们通过这种方法在进程间分配矩阵的元素。在这种分解方法中，p 个进程的每一个都会负责矩阵中相邻的$\lfloor m/p \rfloor$或$\lceil m/p \rceil$行元素。

按列分解与此类似，只不过是将矩阵按照列序号进行分解。p 个进程的每一个都会负责矩阵中相邻的$\lfloor n/p \rfloor$或$\lceil n/p \rceil$列元素。

在棋盘式分解方式中，所有进程构成一个虚拟网格，矩阵按照按照这个网格分为二维的数据块。假设 p 个进程构成一个 r 行 c 列的网格，则每个进程至多将负责$\lceil m/r \rceil$和$\lceil n/c \rceil$列的矩阵元素。

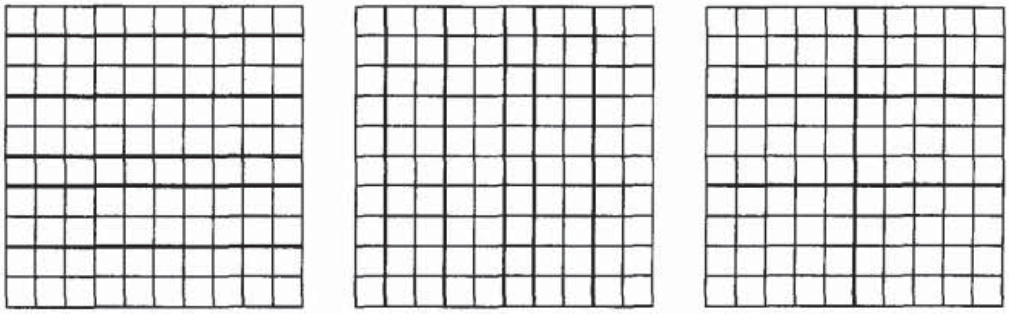

图 8.3　二维矩阵的三种分解方式。在这些例子中，一个 10×10 的矩阵被分解到 6 个进程中（a）按行分解；（b）按列分解；（c）棋盘式分解（进程被组织为一个 3×2 的虚拟网格）

针对向量 b 和 c，我们有两种简单的分解方法。一是将向量元素复制，也就是每个进程都有此向量的一份副本；一是向量元素可以分配到几个或者所有的进程中去。将含有 n 个元素的向量块进行分解，p 个进程中的每一个都会负责原向量中相邻的$\lfloor n/p \rfloor$或$\lceil n/p \rceil$个元素。

为什么一个任务可以保存整个向量 b 和 c，却不能保存整个矩阵 A 呢？为了简化我们的讨论，假设 $m=n$。向量 b 和 c 只有 n 个元素，与矩阵 A 一行或一列的元素数目相同。若每个进程保存 A 的一行或一列和 b 与 c 的一个元素，其存储的量级为$\Theta(n)$。若每个任务保存 A 的一行或一列和 b 与 c 的所有元素，存储的量级仍为$\Theta(n)$。所以，不论向量是被复制到各进程还是被分解到各任务，存储的复杂度是相同的。

由于矩阵有三种分解方法，向量有两种分解方法，所以一共有六种组合。本章我们研究其中三种情况：（1）矩阵按行分解，向量复制；（2）矩阵按列分解，向量分块；（3）矩阵棋盘式分解，向量分块并且只存在于进程网格的第一列进程中。

8.4　矩阵按行分解

8.4.1　设计与分析

本节我们将开发一个并行矩阵向量相乘算法。这个算法基于域分解，每个元任务与矩阵 A 的一行相关联。向量 b 和 c 被复制到所有的进程中。图 8.4 是此算法的一个高阶视图。为了计算内积，每个进程都需要一个行向量和一个列向量。任务 i 中有 A 的第 i 行和 b 的副本，这是用于完成内积计算的所有因素。完成内积计算后，任务 i 就得到了向量 c 的第 i 个元素。当然，不光是向量 b，向量 c 也需要被复制到各个进程中去。所以我们需要进行一次全收集操作以使各个任务都含有 c 中所有的元素，这个操作完毕后算法结束。

在稠密矩阵向量乘法中，完成内积的计算步数是相同的。所以我们可以将连续的几行所对应的元任务进行聚合并与一个进程相对应，这样就形成了矩阵按行分解，如图 8.3（a）所示。

如图 8.4 所示，在内积计算结束时，每个任务只计算出结果向量中的一个元素。如果采用按行分解，每个进程（包含几个元任务）会得到结果向量中的几个元素。

如果 $m=n$，线性矩阵向量乘法的时间复杂度便为$\Theta(n^2)$。下面来考察并行算法的复杂度。每个进程完成它所分配到A的那部分元素与向量b的乘法。任何进程分得的部分都不会超过$\lceil n/p \rceil$行。所以并行算法的乘法部分的复杂度是$\Theta(n^2/p)$。

在第 3 章我们看到，在一个高效的全收集通信中，每个进程发送$\lceil \log p \rceil$个消息；所传送的元素总数是 $n(p-1)/p$，其中 p 是 2 的幂。所以并行算法的通信复杂度是$\Theta(\log p+n)$。

综合考虑算法的计算部分和后面的通信部分，并行矩阵向量乘法的复杂度是$\Theta(n^2/p+n+\log p)$。

下面来考察这个算法的等效率特性。串行算法的时间复杂度为$\Theta(n^2)$。并行算法惟一的开销就是全收集操作。当 n 非常大的时候，在全收集操作中消耗的数据传输延时将远远超过消息延迟。鉴于这个原因我们将通信的复杂度简化至$\Theta(n)$。所以按行分解的矩阵向量乘法算法其等效率函数是：

$$n^2 \geqslant Cpn \Rightarrow n \geqslant Cp$$

如果问题规模为 n，矩阵有 n^2 个元素，那么内存使用函数 $M(n)=n^2$。并行算法的可扩展函数为：

$$M(Cp)/p = C^2p^2/p = C^2p$$

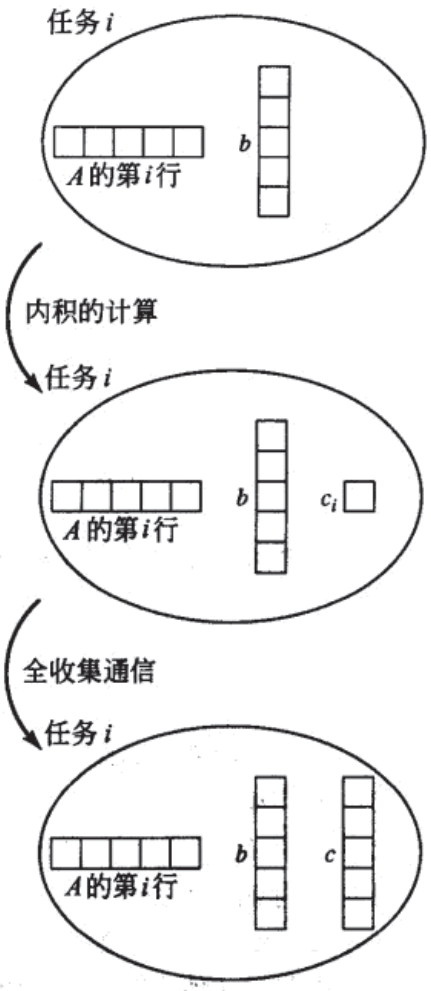

图 8.4 在我们所选择的域分解方式中，每个任务都有矩阵的一行和输入向量的完整副本。一次内积计算将得到结果向量 c 中的一个元素。复制向量 c 需要一次全收集操作

为了保持不变的效率，每个处理器所使用的内存量必须随着处理器的数目的增加线性地增大。可见，这种算法的可扩展性不好。

8.4.2 复制分块的向量

某个进程完成它所负责的那部分的矩阵向量相乘计算后便得到了结果向量 c 中的一块。我们必须把这个分块的向量传给到被复制的 c 向量中，如图 8.5 所示。

下面考虑如何才能完成这个传递。首先，每个进程必须分配一块内存来存放整个向量，而不仅仅是它的一部分。所分配内存空间的大小依赖于元素的类型，例如，字符、整数、浮点数或双精度浮点数。第二，进程必须将这些分块组合为一个完整的向量并共享组合的结果。幸好 MPI 库中有完成组合操作的函数。

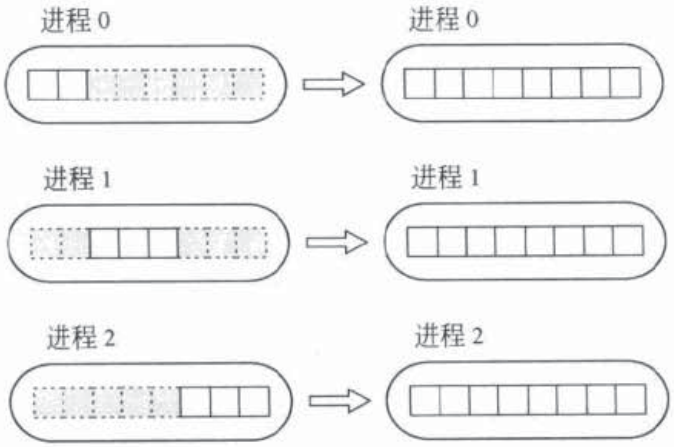

图 8.5　将向量的分块转换为被复制向量，其中该向量的分块分布在各个进程中。向量中的每个元素只存在于一个进程中。当向量被复制后情况则恰恰相反，每个进程将拥有完整的被复制向量

8.4.3　函数 MPI_Allgatherv

一个全收集通信可以连接分布在一组进程中的向量数据块，并把结果向量复制至所有的进程。这个函数就是 MPI_Allgatherv，如图 8.6 所示。

如果从每个进程收集同样数目的元素，较简单的 MPI_Allgather 函数就非常合适。但在向量块分解方式中，只有当向量中的元素总数是进程总数的整数倍时，每个进程分到的向量元素才是相等的。显然这点不可能总能得到保证，所以我们使用 MPI_Allgatherv。

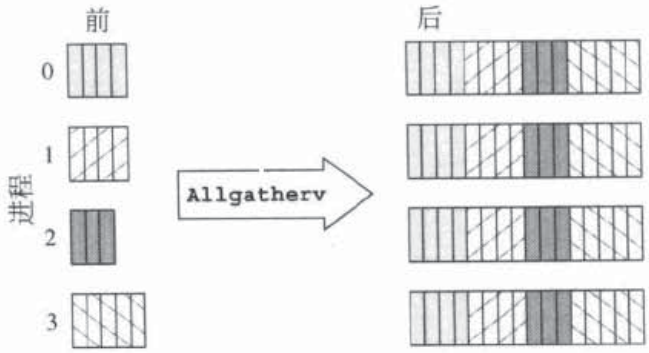

图 8.6　函数 MPI_Allgatherv 负责从一个通信域的所有进程中收集数据项，并将其连接起来。如果从每个进程收集的数据个数相同，也可以用稍微简单一点的 MPI_Allgather 函数

函数声明：

```
int MPI_Allgatherv（void* send_buffer, int send_cnt,
    MPI_Datatype send_type, void* receive_buffer,
    int* receive_cnt, int* receive_disp,
    MPI_Datatype receive_type, MPI_Comm communicator）
```

除了第四个参数外其他参数都是输入参数：

- send_buffer　此进程要发送的数据的起始地址；
- send_cnt　此进程要发送的数据的个数；
- receive_cnt　包含要从每个进程（包括自身）接收的数据个数的数组；
- receive_disp　从每个进程接收的数据项在缓冲区中的偏移量；
- receive_type　待接收数据的数据类型；
- communicator　本操作所在通信域。

第四个参数，receive_buffer，是用来存放所要收集到的元素的缓冲区的起始地址。

图 8.7 描述了 MPI_Allgatherv 的工作原理。每个进程将 send_cnt 设置为它所负责的元素的数目。数组 receive_cnt 由每个进程要接收的元素个数所组成，每个进程中该值是相同的。在这个例子中，每个进程按照进程的编号顺序对元素进行拼接，所以数组 receive_disp 中的值也是相同的。

MPI_Allgatherv 需要传递两个数组，每个都会涉及到所有的进程。第一个数组指出每个进程所贡献的元素个数，第二个数组指出这些元素在结果数组中的位置。

我们经常遇到被分块映射并顺序连接的数组。我们可以写一个函数来构造这两个数组，称之为 create_mixed_xfer_arrays，见附录 B。

此时，我们可以编写一个函数以完成分块向量到复制向量的传递，这也是矩阵向量乘法算法的最后一步。这个函数，即函数 replicate_block_vector，见附录 B。

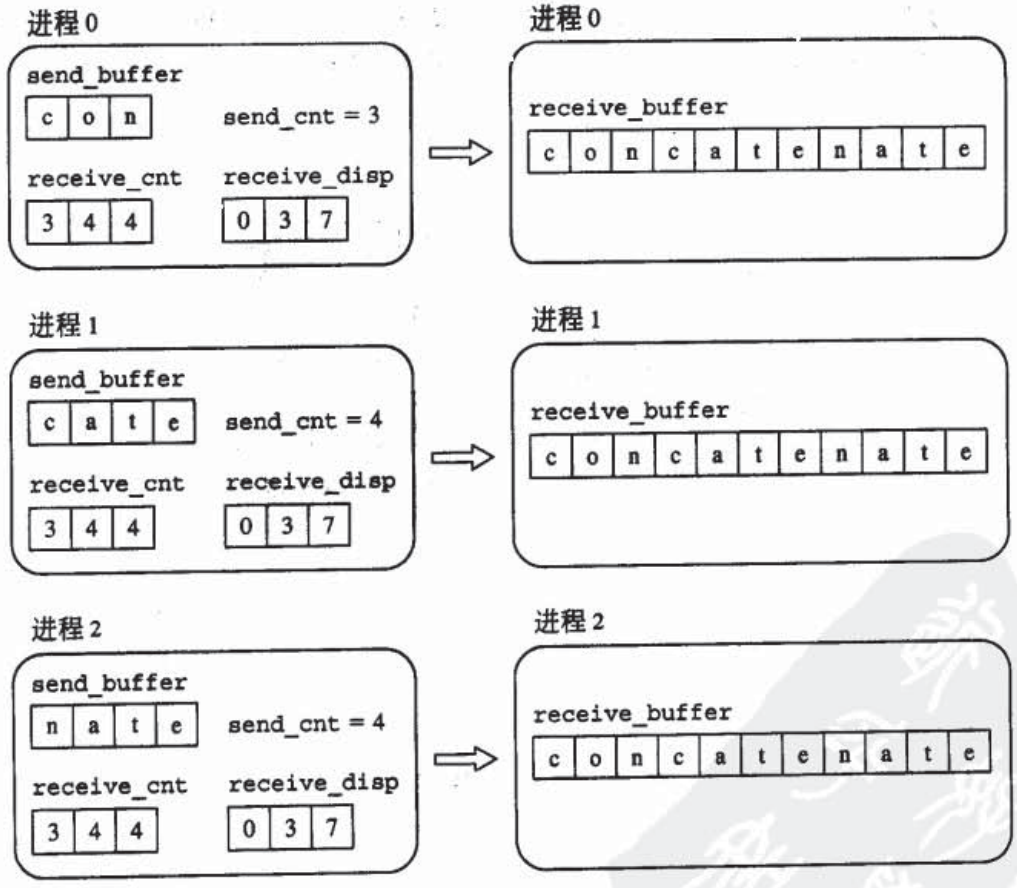

图 8.7　MPI_Allgatherv 直接进行连结时，进程初始化 send_cnt、receive_cnt 和 receive_disp 的实例

8.4.4　被复制向量的输入/输出

我们需要一个函数从文件中读取待复制的数组。假设这个文件由 fwrite 创建，并可由 fread 读取。文件开始是开头为整数 n，它表示向量中元素的个数，之后是 n 个元素。

进程 p–1 尝试打开并读取数据文件。如果可以打开，它就读取 n 并广播给其他进程。如果打开失败，就给其他进程广播 0，然后所有进程结束计算。如果打开成功，所有进程分配空间以存放此向量。之后进程 p–1 读取向量并广播给其他进程。read_replicated_vector 的源码见附录 B。

从并行编程的角度来看，打印复制向量的工作很简单。为了不让标准输出变得混乱，通常由一个进程完成所有的打印工作。因为每个进程都有此向量的一份副本，我们只需保证只有一个进程执行了 printf 即可。函数 print_replicated_vector 的源码见附录 B。

8.4.5　编写并行程序

所有的准备工作都做好了，现在我们可以编写矩阵向量乘法的并行程序了。再回顾一下图 8.4，图中总括了这个算法的主要步骤。完整的 C 代码如图 8.8 所示。

程序开头包含了标准头文件，另外还包含了我们开发的工具函数头文件 MyMPI.h。

```
/*
* Matrix-vector multiplication, Version 1
*/
#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
/* Change these two definitions when the matrix and vector
   element types change */
typedef double dtype;
#define mpitype MPI_DOUBLE
int main（int argc, char *argv[]）{
      dtype **a; /* First factor, a matrix */
      dtype *b; /* Second factor, a vector */
      dtype *c_block; /* Partial product vector */
      dtype *c; /* Replicated product vector */
      dtype *storage; /* Matrix elements stored here */
      int i, j; /* Loop indices */
      int id; /* Process ID number */
      int m; /* Rows in matrix */
      int n; /* Columns in matrix */
      int nprime; /* Elements in vector */
      int p; /* Number of processes */
      int rows; /* Number of rows on this process */
      MPI_Init（&argc, &argv）;
      MPI_Comm_rank（MPI_COMM_WORLD, &id）;
```

```
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    read_row_striped_matrix (argv[1], (void *) &a,
        (void *) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
    rows = BLOCK_SIZE (id,p,m);
    print_row_striped_matrix ((void **) a, mpitype, m, n, MPI_COMM_WORLD);
    read_replicated_vector (argv[2], (void *) &b, mpitype,
        &nprime, MPI_COMM_WORLD);
    print_replicated_vector (b, mpitype, nprime, MPI_COMM_WORLD);
    c_block = (dtype *) malloc (rows * sizeof (dtype));
    c = (dtype *) malloc (n * sizeof (dtype));
    for (i = 0; i < rows; i++) {
        c_block[i] = 0.0;
        for (j = 0; j < n; j++)
            c_block[i] += a[i][j] * b[j];
    }
    replicate_block_vector (c_block, n, (void *) c, mpitype, MPI_COMM_WORLD);
    print_replicated_vector (c, mpitype, n, MPI_COMM_WORLD);
    MPI_Finalize ();
    return 0;
}
```

图 8.8 矩阵向量乘法（第 1 版）

为了保证在最小限度内编辑代码来实现改变矩阵和向量的类型，我们使用 dtype 来表示矩阵和向量中数据的类型，用 mpitype 作为 MPI 函数的类型标志符。在程序头部我们用 typedef 和一个宏定义确定 dtype 和 mpitype 的值。

MPI 初始化完毕后，进程读取并打印矩阵 A（见第 6 章），同时读取并打印向量 b。

每个进程为它所分配到的结果向量 c 分配空间并完成内积计算。

此时，每个进程都有一部分的 c。我们将 c 转化为一个完整向量并完成复制，之后程序结束。

8.4.6 测试

下面我们推导一个适用于商用集群上并行程序执行时间的表达式。令 χ 表示内积计算循环中一次迭代所消耗的时间。我们可以将串行算法的执行时间除以 n^2 来得到 χ。所以并行算法计算部分的执行时间是 $\chi n(n/p)$。

全收集归约操作中每个发送 $(\log p)$ 个消息。每个消息的延迟为 λ。在全收集操作中总共传输的向量元素个数为 $n(2^{\lceil \log p \rceil}-1)/2^{\lceil \log p \rceil}$。每个向量元素是大小为 8 字节的双精度浮点数。所以全收集操作的执行时间为 $\lambda(\log p)+8n((2^{\lceil \log p \rceil}-1)/2^{\lceil \log p \rceil})/\beta$。

测试集群配置：450MHz PentiumII，快速以太网参数：χ=63.4ns，λ=250μs,β=106bps。

表 8.1 中比较了矩阵向量乘法的实际执行时间和预测执行时间，问题规模为 1000，分别在 1, 2,…, 8, 16 个处理器上进行测试。实际报告的时间是运行 100 次的平均时间。将总浮点操作次数（$2n^2$）除以执行时间，再除以一百万得到 MFLOPS。此程序的加速情况如本章结尾图 8.20 所示。

表 8.1　按行块分解的矩阵向量乘法的预测时间和执行时间的比较。其中矩阵大小为 1000×1000，向量中有 1000 个元素

处理器数	预期时间	实际时间	加速比	Mflops
1	0.0634	0.0634	1.00	31.6
2	0.0324	0.0327	1.94	61.2
3	0.0223	0.0227	2.79	88.1
4	0.0170	0.0178	3.56	112.4
5	0.0141	0.0152	4.16	131.6
6	0.0120	0.0133	4.76	150.4
7	0.0105	0.0122	5.19	163.9
8	0.0094	0.0111	5.70	180.2
16	0.0057	0.0072	8.79	277.8

8.5　矩阵按列分解

8.5.1　设计与分析

本节我们将设计另一种矩阵-向量相乘算法。假设任务 i 分配到矩阵 A 的第 i 列和向量 b 与 c 的第 i 个元素。图 8.9 是这个并行算法的结构。

计算过程从每个任务将它所拥有的 A 的列与 b_i 相乘开始，这将得到一个向量作为中间结果。计算任务结束时，第 i 个任务将只需保留一个元素 c_i。因而我们需要进行一次全交换通信：任务 i 所得到的部分结果元素 j 必须传递给任务 j。此时，每个任务 i 都有得到 c_i 的 n 个部分结果。

因为每个任务的计算和通信特征都是相同的，所以可以将它们组合成含有相同列数（或多一列或少一列）的更大的任务来保证负载平衡。所以我们将初始任务组合成为 p 个元任务，并把每个元任务映射为一个进程。

在前一节我们将矩阵 A 按行分解并分配给了各个进程，这叫做按行分解。现在我们使用按列分解的方法，将矩阵 A 中相邻的连续几列进行聚合，如图 8.3（b）所示。

下面分析这种算法的复杂度。假设 $m=n$，每个进程将它所分配到的部分矩阵 A 和部分向量 b 相乘。任何进程的计算量都不会超过 A 的$\lceil n/p \rceil$列和 b 的元素。因此开始的乘法部分的时间复杂度为$\Theta(n(n/p))=\Theta(n^2/p)$。全收集过程完成后，每个进程将从其他进程收集到的部分向量求和。一共有 p 个部分向量，每个的长度不会超过$\lceil n/p \rceil$。这一步的时间复杂度为$\Theta(n)$。因此，整个计算的时间复杂度为$\Theta(n^2/p)$。

利用超立方通信模式，一个全交换在（$\log p$）步之内完成。在每一步，每个进程都向它的协作者发送 $n/2$ 个数据，同样它也从其协作者处接收 $n/2$ 个数据。在全交换交换中发送和接收的元素总数为 n。所以通信复杂度为$\Theta(n\log p)$。

另一种完成全交换的方式是由每个进程向其他 $p-1$ 个进程发送消息。每个消息只包含目标进程期望从源进程获得的数据。这种方式需要发出的消息总数是 $p-1$，但由每个进程传递的元素数量均不大于 n。这个算法的时间复杂度为$\Theta(p+n)$。

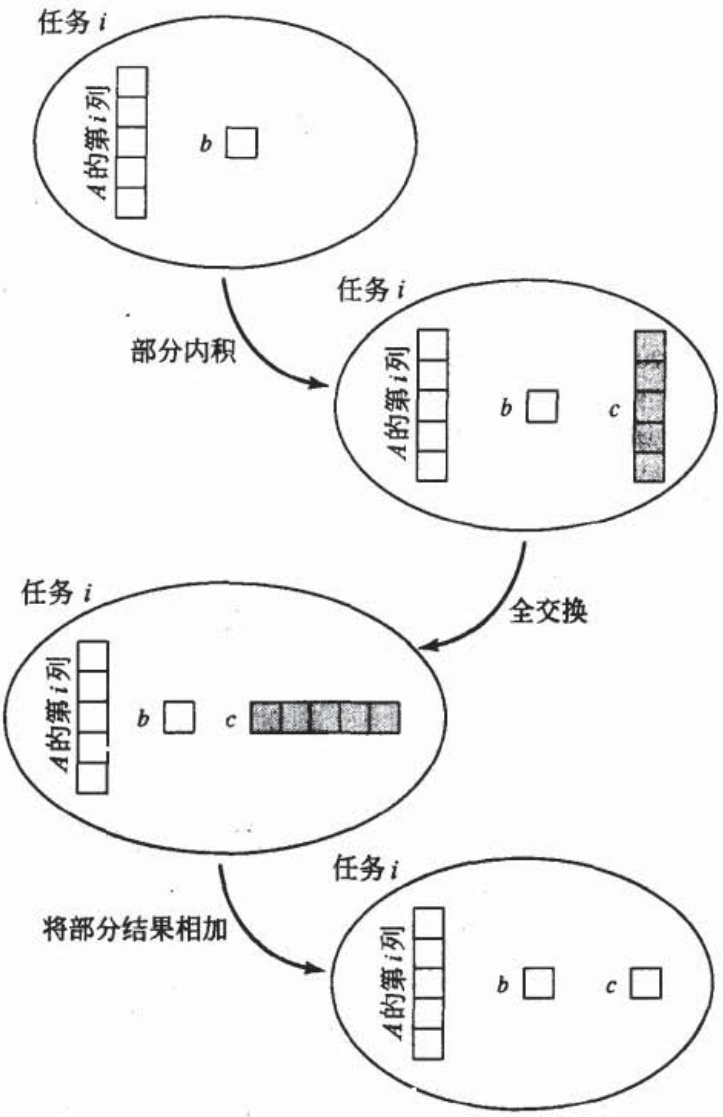

图 8.9 在本并行算法中，每个任务负责矩阵中的一列和向量中的一个元素。通过全交换通信将各个数据项移动到相应的任务，再将它们求和

综合此算法的计算阶段和全交换通信阶段，它的总的复杂度为$\Theta(n^2/p+n\log p)$或$\Theta(n^2/p+n+p)$，最终结果取决与采用哪种通信方式。

现在来考察这个并行算法的等效率特性。串行算法的时间复杂度是$\Theta(n^2)$。并行算法的开销在于全交换操作。当 n 非常大的时候，在全交换操作中消息传递的时间远远超过消息延迟。用第二种方法实现全交换，时间复杂度为$\Theta(n)$，所有进程都进行这一步。

所以基于按列分解的矩阵—向量相乘的并行算法，其等效率性函数为：

$$n^2 \geqslant Cpn \Rightarrow n \geqslant Cp$$

这于我们使用按行分解得到的等效率函数是一样的。此算法的扩展性也不好，为了保证效率不变，内存的使用须随着处理器的增加而线性增大。

8.5.2 读取按列分解的矩阵

我们需要编写一个函数来读取矩阵数据文件（在这个文件中，矩阵按照行优先的规则

存放），并将其按列分发到各个进程去。如果一个按行优先存放的矩阵有很多行，而且要按照列块分解的方式进行分解，那么分配到某个进程的数据在文件中是不连续的。实际上，矩阵的每行都会被分发到所有的进程去。

我们仍然只让一个进程完成 I/O 操作，如图 8.10 所示。第一步，一个进程读取矩阵一行至一个临时缓冲区；第二步，这个进程分发缓冲区中的元素至所有进程。此后对矩阵的剩余行重复进行这两步操作。函数 read_col_striped_matrix 的源码见附录 B。

函数 read_col_striped_matrix 使用 MPI 库函数 MPI_Scatterv 在进程间散发行数据。下面我们进一步来研究一下这个函数。

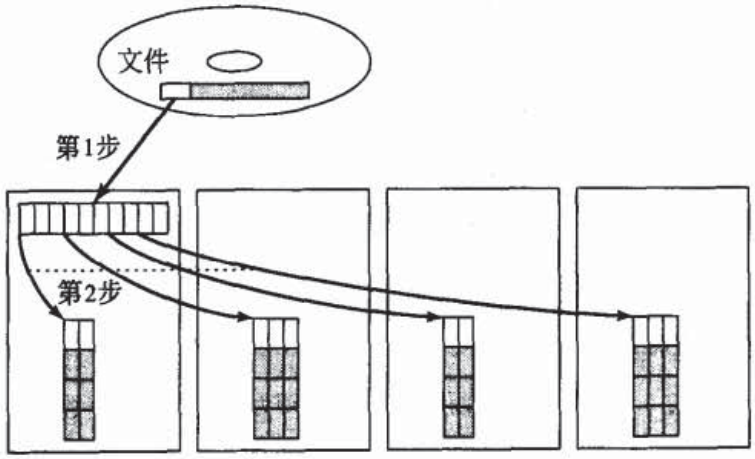

图 8.10　在按列分解算法中，矩阵的每一行都分布在进程中。一个进程输入矩阵的一行（第 1 步）之后将其散发开来（第 2 步）

8.5.3　函数 MPI_Scatterv

MPI 函数 MPI_Scatterv，如图 8.11 所示，通过一个根进程向在一个通信域内的所有进程（包括自身）分发一组连续的数据元素。

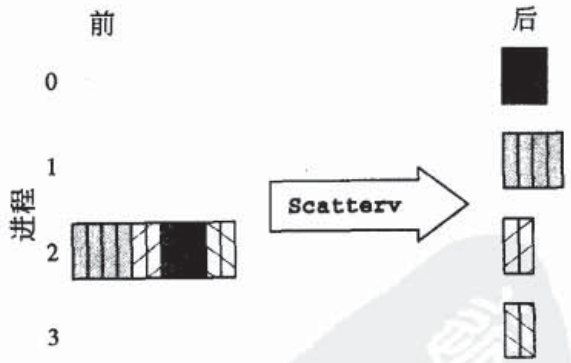

图 8.11　组通信函数 MPI_Scatterv 允许一个 MPI 进程将连续的数据项分解并把各个部分散发到通信域的其他进程去。如果向每个进程分发的数据项个数相同，则相对简单的 MPI_Scatter 较为适合

MPI_Scatterv 的声明：

```
MPI_Scatterv(void *send_buffer, int* send_cnt,
    int* send_disp, MPI_Datatype send_type,
    void *recv_buffer, int recv_cnt,
    MPI_Datatype recv_type, int root, MPI_COMM communicator)
```

本函数有 9 个参数，除第 5 个参数外全是输入变量：

- send_buffer　指向含有待分发元素缓冲区的指针。
- send_cnt　第 i 个元素是 send_buffer 中要发送到进程 i 的一连串数据的个数。
- send_disp　第 i 个元素是 send_buffer 中要发送到进程 i 的第一个元素在 send_buffer 中的偏移量。
- send_type　send_buffer 中数据的类型。
- recv_buffer　指向本进程用于接收数据的缓冲区指针。
- recv_cnt　本进程要接收的数据个数。
- recv_type　recv_buffer 的数据类型。
- root　分发数据进程的 ID；
- comunicator　散发操作所在的通信域。

MPI_Scatterv 是一个组通信函数，通信域中的所有进程都参与其执行。此函数要求所有进程初始化两个数组：一个指出根进程向每个进程发送数据的个数，一个指出散发数据在缓冲区中的偏移量。散发操作按照进程号的顺序进行：进程 0 得到第一块，进程 1 得到第二块，以此类推。在收集操作中我们已经编写了 create_mixed_xfer_arrays 函数，此处我们也可以直接使用。每个进程的元素数目和偏移量是相同的。

8.5.4 打印输出按列分块矩阵

我们需要设计一个函数完成按列分块矩阵的打印。为了保证数据以正确的顺序打印，我们只允许一个进程完成所有的打印。为了打印一行，此进程必须从其他所有进程收集该行的数据。所以此函数的数据流与 read_col_striped_matrix 正好相反。函数 print_col_striped_matrix 源码见附录 B。

函数 print_col_striped_matrix 用到 MPI 库函数 MPI_Gatherv 来收集元素至进程 0，然后进程 0 将其打印。下面我们来看看 MPI_Gatherv。

8.5.5 函数 MPI_Gatherv

MPI 组通信函数 MPI_Gatherv（如图 8.12 所示）完成数据收集功能。

MPI_Gatherv 函数的声明：

```
MPI_Gatherv(void* send_buffer, int send_cnt,
    MPI_Datatype send_type, void *recv_buffer,
    int* recv_cnt, int* recv_disp, MPI_Datatype recv_type,
    int root, MPI_Comm communicator)
```

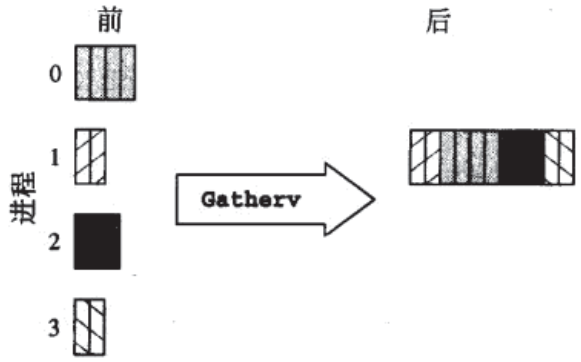

图 8.12　MPI_Gatherv 可以将存储在通信域所有进程中的数据收集到某个进程中。如果每个进程贡献的数据个数相同，可以使用较为简单的 MPI_Gather 函数

此函数有 9 个参数，除了第 5 个参数外均为输入参数：

- send_buffer　此进程被收集数据的起始地址；
- send_cnt　本进程被收集的数据个数；
- send_type　send_buffer 中数据的类型；
- recv_buffer　根进程存放收集到数据的起始地址；
- recv_cnt　这个数组的第 i 个元素表示从进程 i 收集的数据个数；
- recv_disp　第 i 个元素是从进程 i 收集来数据的存放地址相对于 recv_buffer 起始地址的偏移量；
- recv_type　recv_buffer 的数据的类型；
- root　收集数据的根进程；
- comunicator　收集操作所在的通信域。

8.5.6　分发中间结果

m 个内积操作得到了 m 个元素的结果向量 c：

$$c[0]=a[0,0]b[0]+a[0,1]b[1]+\cdots+a[0][n-1]b[n-1]$$
$$c[1]=a[1,0]b[0]+a[1,1]b[1]+\cdots+a[1][n-1]b[n-1]$$
$$\cdots$$
$$c[m-1]=a[m-1,0]b[0]+a[m-1,1]b[1]+\cdots+a[m-1][n-1]b[n-1]$$

在域分解中，每个原始任务 i 分配到矩阵 A 的第 i 列和向量 b 的第 i 个元素。将列的每个元素与 b_i 相乘分别得到 $a[0,i]b[i]$，$a[1,i]b[i]$，…，$a[n-1,i]b[i]$。$a[0,i]b[i]$是组成 $c[0]$ 的第 i 个元素，$a[1,i]b[i]$是 $c[1]$的第 i 个元素，依此类推。也就是说任务 i 进行的 n 个乘法所得到的 n 个元素不能相加得到内积。得到的第 j 个元素是用于计算 $c[j]$的一部分（$0\leqslant j<n$）。

乘法完成之后，每个进程都要把它所不需要的 $n-1$ 个结果分发到其他进程，同时向其他进程收集所需要的 $n-1$ 个结果：这叫做全交换（all-to-all exchange）。交换之后，任务 i 将它所有的 n 个元素（任务 0 生成的 $a[i,0]b[0]$，任务 1 生成的 $a[i,1]b[1]$）求和得到 $c[i]$。

在这种矩阵向量乘法的实现中，每个进程分配到 A 的某些列和 b 的对应元素。原则是

一样的：进程之间互相交换各自所不需要的和其所需要的数据。任务 i 收到 BLOCK_SIZE（i, p, n）个元素。交换之后，它有 p 个大小为 BLOCK_SIZE（i, p, n）的数组，将这些数组求和它便得到了 c 的一部分。

8.5.7 函数 MPI_Alltoallv

MPI_Alltoallv 可以完成在一个通信域的所有进程之间相互交换数据（如图 8.13 所示）。函数声明如下：

```
int MPI_Alltoallv(void *send_buffer, int *send_count,
    int *send_displacement, MPI_Datatype send_type,
    void *recv_buffer, int *recv_count,
    int *recv_displacement, MPI_Datatype recv_type,
    MPI_Comm communicator)
```

- send_buffer　待交换数组的起始地址；
- send_count　这是一个数组，它的第 i 个元素指定进程 i 的元素个数；
- send_displacement　第 i 个元素为进程 i 的数据在 send_buffer 的起始地址；
- send_type　send_buffer 元素数据类型；
- recv_buffer　接收数据（包括自己发送给自己的数据）缓冲区起始地址；
- recv_count　这是一个数组，它的第 i 个元素表示本进程将要从进程 i 接收的数据个数；
- recv_displacement　第 i 个元素表示从进程 i 接收的数据存入 recv_buffer 的起始地址；
- recv_type　在放入 recv_buffer 前数据要被转换为此类型；
- communicator　指明参加全交换的进程集合。

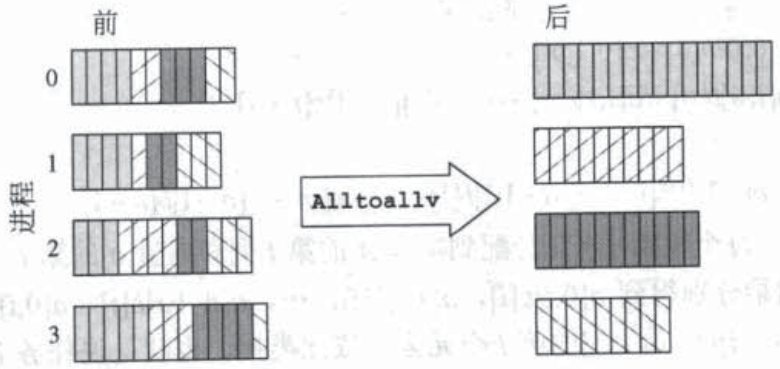

图 8.13　MPI_Alltoallv 使通信域中的所有进程向所有进程收集数据。如果从任一进程传送到任意其他进程的数据个数均相同，则可以使用较为简单的 MPI_Alltoall 函数

8.5.8 编写并行程序

现在我们已经可以编写第二个矩阵向量乘法的并行程序了。源代码如图 8.14 所示。

```
/*
* Matrix-vector multiplication, Version 2
*/
#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
/* Change these two definitions when the matrix and vector
   element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
    dtype **a; /* The first factor, a matrix */
    dtype *b; /* The second factor, a vector */
    dtype *c; /* The product, a vector */
    dtype *c_part_out; /* Partial sums, sent */
    dtype *c_part_in; /* Partial sums, received */
    int *cnt_out; /* Elements sent to each proc */
    int *cnt_in; /* Elements received per proc */
    int *disp_out; /* Indices of sent elements */
    int *disp_in; /* Indices of received elements */
    int i, j; /* Loop indices */
    int id; /* Process ID number */
    int local_els; /* Cols of 'a' and elements of 'b' held by this process */
    int m; /* Rows in the matrix */
    int n; /* Columns in the matrix */
    int nprime; /* Size of the vector */
    int p; /* Number of processes */
    dtype *storage; /* This process's portion of 'a' */
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    read_col_striped_matrix (argv[1], (void ***) &a,
        (void **) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
    print_col_striped_matrix ((void **) a, mpitype, m, n, MPI_COMM_WORLD);
    read_block_vector (argv[2], (void **) &b, mpitype,
        &nprime, MPI_COMM_WORLD);
    print_block_vector ((void *) b, mpitype, nprime, MPI_COMM_WORLD);

    /* Each process multiplies its columns of 'a' and vector
        'b', resulting in a partial sum of product 'c'. */
    c_part_out = (dtype *) my_malloc (id, n * sizeof (dtype));
    local_els = BLOCK_SIZE (id,p,n);
    for (i = 0; i < n; i++) {
```

```
            c_part_out[i] = 0.0;
            for (j = 0; j < local_els; j++)
                c_part_out[i] += a[i][j] * b[j];
        }
        create_mixed_xfer_arrays (id, p, n, &cnt_out, &disp_out);
        create_uniform_xfer_arrays (id, p, n, &cnt_in, &disp_in);
        c_part_in = (dtype*) my_malloc (id, p*local_els*sizeof (dtype));
        MPI_Alltoallv (c_part_out, cnt_out, disp_out, mpitype,
        c_part_in, cnt_in, disp_in, mpitype, MPI_COMM_WORLD);
        c = (dtype*) my_malloc (id, local_els * sizeof (dtype));
        for (i = 0; i < local_els; i++) {
            c[i] = 0.0;
            for (j = 0; j < p; j++)
                c[i] += c_part_in[i + j*local_els];
        }
        print_block_vector ((void *) c, mpitype, n, MPI_COMM_WORLD);
        MPI_Finalize ();
        return 0;
    }
```

图 8.14 矩阵向量乘法并行程序（第 2 版）

MPI 的初始化完成之后，调用函数 read_col_striped_matrix 从数据文件读取矩阵并分发到其他进程，然后将矩阵打印输出。

类似地，读取向量 b 并输出。

每个进程分配空间用来存储 c_part_out，它是中间结果，最终将被传递给其他进程。同样地，要为 c_part_in 分配空间，以接收从其他进程传递来的数据。

随后是真正的计算过程。每个进程用它拥有的部分矩阵（n×local_else）乘以它的部分向量（长度为 local_else），得到长度为 n 的中间结果向量。

用于大小各异的各个 c_part_out，我们通过调用 create_mixed_xfer_arrays 来设置各自的数量和偏移量。相反，所有用于输入的 c_part_in 大小都相同，通过调用 carete_uniform_xfer_array 来正确的初始化各自的数量和偏移量。最后由 MPI_Alltoallv 完成全交换通信，将各个部分传递至目的地。

此时，每个进程都有长度为 local_els 的 n 个数据。将其求和得到 c 的一个元素。

8.5.9 测试

下面我们来推导一个适用于计算商用集群上该并行程序执行时间的表达式。如前，令 χ 表示内积计算循环中一次迭代所需要的时间。并行算法计算部分的时间是 $\chi n\lceil n/p \rceil$。

本算法要进行一次向量 c 部分结果的全交换操作。我们有两种方法实现全交换：第一种是每个进程发送（$\log p$）个长度为 $n/2$ 的消息，总共传递的数据个数为 $n/2$。

第二种方法是每个进程向目的进程直接发送目的进程所需要的数据。每个进程需要发送 $p-1$ 个消息，总共传递得数据个数为 $n(p-1)/p$。

当 n 较大的时候，消息的时延主要由消息传递所构成，所以第二种方法更有优势。令消息的延迟为 λ，传递一个字节耗时 $1/\beta$，完成一次双精度浮点数全收集将耗时 $(p-1)(\lambda+8n/(p\beta))$。

测试集群配置：450MHz PentiumII，快速以太网参数：χ=63.4ns，λ=250μs, β=106bps。

表 8.2 比较了矩阵向量乘法执行的实测时间和预测时间，问题规模为 1000，分别在 1，2，…，8，16 个处理器上测试。实测时间是运行 100 次的平均时间。此程序的加速情况见本章结尾的图 8.20。

表 8.2　第二个并行算法在 450MHz PII 商用集群上执行的预测时间和实测时间的比较

处理器数	预测时间	实际时间	加速比	Mflops
1	0.0634	0.0638	1.00	31.4
2	0.0324	0.0329	1.92	60.8
3	0.0222	0.0226	2.80	88.5
4	0.0172	0.0175	3.62	114.3
5	0.0143	0.0145	4.37	137.9
6	0.0125	0.0126	5.02	158.7
7	0.0113	0.0112	5.65	178.6
8	0.0104	0.0100	6.33	200.0
16	0.0085	0.0075	8.23	263.2

8.6　棋盘式分解

8.6.1　设计与分析

在这种域分解方式中，每个原始任务对应一个矩阵元素。负责元素 $a_{i,j}$ 的任务将它与 b_j 相乘得到 $d_{i,j}$。结果向量的每个元素 c_i 等于 $\sum_{j=0}^{n-1} d_{i,j}$。也就是说，对第 i 行而言，将所有 $d_{i,j}$ 求和得到向量 c 的第 i 个元素，如图 8.15 所示。

$$\begin{pmatrix}4&5&3\\6&2&1\end{pmatrix}\begin{pmatrix}1\\2\\3\end{pmatrix}\Rightarrow\begin{pmatrix}4\times1+5\times2+3\times3\\6\times1+2\times2+1\times3\end{pmatrix}\Rightarrow\begin{pmatrix}23\\13\end{pmatrix}$$

$$\begin{pmatrix}a_{0,0}&a_{0,1}&a_{0,2}\\a_{1,0}&a_{1,1}&a_{1,2}\end{pmatrix}\begin{pmatrix}b_0\\b_1\\b_2\end{pmatrix}\Rightarrow\begin{pmatrix}d_{0,0}&d_{0,1}&d_{0,2}\\d_{1,0}&d_{1,1}&d_{1,2}\end{pmatrix}\Rightarrow\begin{pmatrix}c_0\\c_1\end{pmatrix}$$

图 8.15　第三种算法将矩阵 A 的每个元素与一个原始（基本）任务相关联。每个任务完成 $a_{i,j}\times b_j$，得到 $d_{i,j}$。将 $d_{i,j}$ 按行归约即可得到向量 c 的元素

我们将基本任务规则地聚合成矩形块，并在任进程与块之间建立对应关系，如图 8.3（c）。因为所有的块大小相同，每块内所要完成的工作量也基本相同，所以我们要设置块的大小以便于将一个任务映射给一个进程。可以将进程想像为一个二维的网格。向量 b 被分发到任务网格第一列的任务中，如图 8.16 所示。

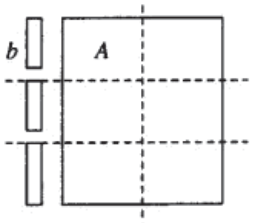

图 8.16 聚合完成后，任务形成一个 2 维网格，每个任务负责 A 的一块。本图显示了一个 3×2 的任务网格。向量 b 被分解到任务网格中与第一列的各块所对应的进程中

现在我们来规划这个并行算法的三个重要步骤，以及实现这三个步骤所用的通信模式，如图 8.17 所示。与块 $A_{i,j}$ 关联的任务完成与子向量 b_j 的矩阵向量乘法。第一步，重新分发 b，保证每个任务得到 b 中相应的部分（后面会详细介绍如何分发）。第二步，每个任务完成各自部分的矩阵向量乘法。第三步，对任务网格每一行中的任务进行一次求和归约。在这之后，结果向量 c 就可从任务网格的第一列任务所对应的块中得到。

现在来看如何重新分发向量 b。假设 p 个任务被划分成 $k \times l$ 网格。开始时，b 被分解到任务网格第一列的 k 个任务中。重新分发之后，每一个 b 分片被分发到对应行的 l 个任务中。

如果 k=1，重新分发很容易。如图 8.18（a）。位于位置（i, 0）的一个任务将自己得到的部分 b 发送至（0, i），然后第一行的所有任务将自己部分的 b 广播至同列的其他进程。

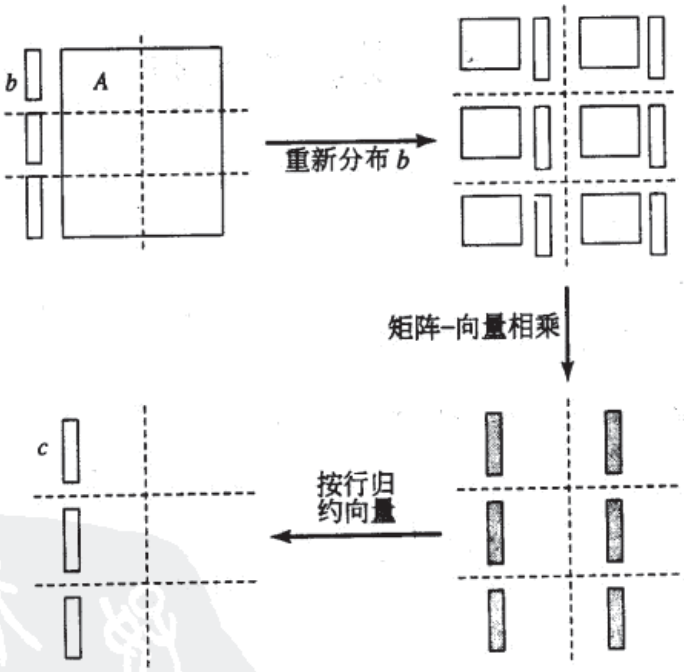

图 8.17 基于棋盘式分解的矩阵向量相乘算法的几个阶段。第一，将 b 向量散发到各任务中；第二，每个任务完成自己部分矩阵 A 和向量 b 的乘法；第三，每一行任务通过归约求和得到向量 c

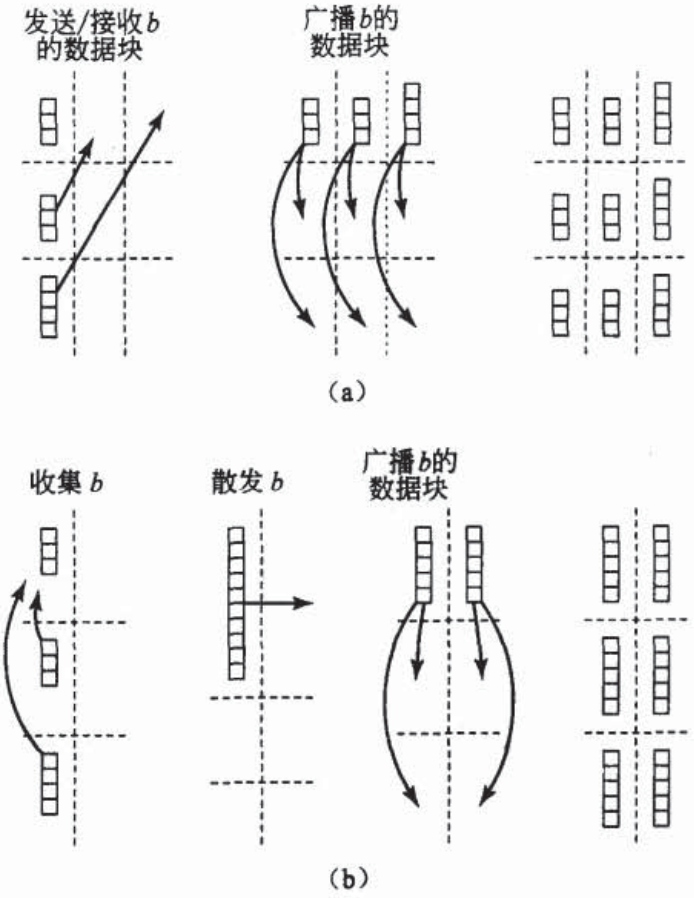

图 8.18　向量的重新分发（a）如果进程网格是方的，算法将会比较简单。第一列的进程将自己部分的 b 发送至第一行的进程。然后第一行的进程将其广播到与各自同列的进程。（b）进程网格不是方形的时候。首先，第一列的进程将向量收集 b 至位于（0, 0）的进程，然后（0, 0）出的进程再散发 b 至第一行的各个进程，最后，第一行的进程将相应部分的 b 广播至同列进程

如果 $k \neq l$，重新分发就非常复杂了，这是因为 b 分块的大小改变了，如图 8.18（b）所示。在这种情况下，我们将 b 的元素收集至任务（0, 0），然后再将它散发到第一行的各个进程。最后，第一行的进程将自己部分的 b 广播到同列的其他进程中。

下面分析此并行算法的复杂度。假设 $m=n$，p 是一个平方数，以便进程可以组织为方形网格（毫无疑问，这是最简单的假设。但如果网格是 $p \times 1$，分解就成了行块分解；如果网格是 $1 \times p$，分解就成了列块分解。这两种方法的复杂度都已经讨论过了）。

每个进程负责一块矩阵，最大为 $(n/\sqrt{p}) \times (n/\sqrt{p})$。所以矩阵向量乘法这一步的时间复杂度为 $\Theta(n^2/p)$。

如果 p 是一个平方数，b 的重新分发将在两步内完成。首先，第一列的所有进程发送相应部分的 b 至第一行，时间复杂度为 $\Theta(n/\sqrt{p})$。然后，第一行的所有进程将对应部分的 b 广播到同列的其他进程。广播的时间复杂度为 $\Theta(\log \sqrt{p}\ (n/\sqrt{p})) = \Theta(n\log p/\sqrt{p})$。

在矩阵向量乘法完成后，进程按行进行求和归约。这个通信操作的时间复杂度为 $\Theta(\log \sqrt{p}\ (n/\sqrt{p})) = \Theta(n\log p/\sqrt{p})$。

综合以上几个步骤，基于棋盘式分解的矩阵向量相乘算法的时间复杂度是 $\Theta(n^2/p + n\log p/\sqrt{p})$。

下面我们来研究此算法的等效率特性。其串行算法的时间复杂度为$\Theta(n^2)$。并行算法的开销是 p 次通信的复杂度，即 $np\log p/\sqrt{p} = n\sqrt{p}\log p$。所以等效率函数为：

$$n^2 \geqslant Cn\sqrt{p}\log p \Rightarrow n \geqslant C\sqrt{p}\log p$$

因为 $M(n)=n^2$，所以可扩展函数为：

$$M(C\sqrt{p}\log p)/p = C^2 p\log^2 p/p = C^2\log^2 p$$

这个并行算法比前两个的扩展性要好。

8.6.2 创建通信域

通信域是一个不透明的对象，它为进程提供了消息传递的环境。到目前为止我们实现的所有 MPI 程序中，组通信都会涉及到所有的进程，我们可以使用默认的通信域，即，MPI_COMM_WORLD。在棋盘式分解矩阵向量相乘算法的实现中，有 4 个组通信函数涉及到子进程组：

- 如果 p 不是平方数，虚拟进程网格中第一列的进程参与收集 b 的通信；
- 如果 p 不是平方数，虚拟进程网格中第一行的进程参与散发 b 的通信；
- 第一行的每个进程广播 b 的相应部分至与其同列的其他进程；
- 每行的所有进程完成独立的求和归约，在第一列的进程中产生向量 c。

为了在某些组通信中只涉及原进程组中的部分进程，我们需要创建新的通信域。

一个通信域由一个进程组，上下文，以及其他属性构成。进程拓扑是通信域的一个重要特征。拓扑可以为进程建立新的编址模式，而不仅仅是使用进程编号。拓扑是虚拟的，也就是说它并不依赖于处理器的实际连接方式。MPI 支持两种拓扑结构：笛卡儿拓扑（即网格）和图拓扑。我们的程序需要建立笛卡儿拓扑通信域，也就是由进程组成的二维虚拟网格。

8.6.3 函数 MPI_Dims_create

为了使算法具有最好的可扩展性，所建立的虚拟进程网格最好接近方形。将笛卡儿网格节点数和网格的维数传递给 MPI_Dims_create，它可以返回一个整数数组来指定每一维分别有多少个节点从而能最大程度地达到负载平衡。函数声明如下：

```
int MPI_Dims_create(int nodes, int dims, int *size)
```

三个参数的含义如下：

- nodes　输入参数，网格中的进程数；
- dims　输入参数，我们想要的的网格维数；
- size　输入/输出参数，网格中每一维的大小。在调用此函数之前，size 中的每个元素（size[0]，…，size[dims–1]）必须进行初始化。如果 size[i]=0，此函数就会决定这个维度的大小。如果 size[i]>0，那么就是用户所指定的该维的大小。

例如，如果要寻找含有 p 个进程的二维网格，就可以用代码完成这项工作：

```
int p;
int size[2];
...
size[0] = size[1] = 0;
MPI_Dims_create(p, 2, size);
```

函数 MPI_Dims_create 返回后，size[0]，size[1]分别是网格的行数和列数。

8.6.4　函数 MPI_Cart_create

确定了虚拟网格的每一维的大小之后，需要为这种拓扑建立通信域。组函数 MPI_Cart_create 可以完成此项任务，其声明如下：

```
int MPI_Cart_create(
MPI_Comm old_comm, int dims, int *size, int *periodic,
int reorder, MPI_Comm *cart_comm)
```

本函数有以下 5 个参数。

- old_comm：以前的通信域。这个通信域内所有的进程都必须调用此函数；
- dims：网格维数；
- size：长度为 dims 的数组，size[*j*]是第 *j* 维的进程数；
- periodic：长度为 dims 的数组，如果第 *j* 维具有周期性（通信在网格的边缘卷回），那么 periodic[*j*]为 1，否则为 0；
- reorder：标志信息，指示进程是否能被重新编号。如果 reorder 为 0，那么进程在新的通信域中将保留在旧的通信域 old_comm 中的进程号。

MPI_Cart_create 有一个输出参数。cart_comm 将指向新的笛卡儿通信域。

下面来看看如何在我们的程序中使用这个函数。先前的通信域是 MPI_COMM_WORLD。网格是二维的，MPI_Dims_create 设置 size 为每一维的大小。我们不采用循环通信，我们也不关心在新的通信域进程是否保持原来的顺序。由此我们得到下面的代码片段：

```
MPI_Comm cart_comm; /* Cartesian topology communicator */
int p;              /* Processes */
int periodic[2];    /* Message wraparound flags */
int size[2];        /* Size of each grid dimension */
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, size, periodic,1, &cart_comm);
```

8.6.5　读取棋盘式矩阵

我们仍然只允许一个进程负责打开文件，读取数据，并分发至对应的进程。分发的方式与列块分解时使用的方式很像。区别在于不是将矩阵的每行分发到各个进程，而是将每

行分发到一个进程的小集合——虚拟进程网格中处于同一行的进程。如图 8.19 所示。进程 0 负责矩阵输入。每读取一行，它就将其发送到进程网格相应行的第一个进程。接收进程收到此行的数据后就将其进一步分发到进程网格中与之位于同一行的其他进程中。

为了实现这个目标，我们还需要以下其他几个 MPI 函数。

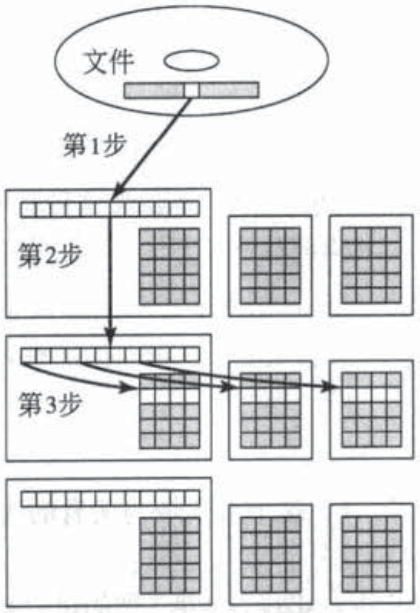

图 8.19 矩阵的棋盘式分解中，矩阵的每一行都散发在一些进程的子集合中。这个进程子集合占据着进程网格中的一行。在这个例子中，9 个进程被组织为 3×3 的网格。当前状态下，文件已经读入了大半。一个进程将下一列读入（第一步），然后将它传向虚拟网格对应行中第一列对应的进程（第二步）。最后这一列进程中的第一个进程负责把矩阵这一行的各个元素散发到同一列的其他进程中去（第三步）

8.6.6 函数 MPI_Cart_rank

为了将矩阵的一行发送到进程网格相应行的第一个进程中去，进程 0 必须知道它们的进程号。函数 MPI_Cart_rank 可以通过进程在网格中的座标得到其进程号。其声明如下：

```
MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

comm 是一个输入参数，它指定了通信操作所在的通信域。coors 也是一个输入参数，指定了一个进程在虚拟网格中的坐标。rank 是一个输出参数，指定了 comm 中位于某个坐标的进程的进程号。

例如，假设虚拟网格有 r 行，而矩阵有 m 行。矩阵的第 i 行映射到进程网格中的第 BLOCK_OWNER（i, r, m）行。下面的代码片段说明了进程 0 怎样确定接收此行的进程：

```
int dest_coord[2]; /* Coordinates of process receiving row */
int dest_id; /* Rank of process receiving row */
int grid_id; /* Rank of process in virtual grid */
int i;
```

```
...
for (i = 0; i < m; i++){
    dest_coord[0] = BLOCK_OWNER(i,r,m);
    dest_coord[1] = 0;
    dest_id = MPI_Cart_rank (grid_comm, dest_coord, dest_id)
    if (grid_id == 0){
        /* Read matrix row 'i' */
        ...
        /* Send matrix row 'i' to process 'dest_id' */
        ...
    } else if (grid_id == dest_id){
        /* Receive matrix row 'i' from process 0 */
        ...
    }
}
```

8.6.7　函数 MPI_Cart_coords

同样，进程也应该能确定自己在虚拟网格中的坐标。这项功能非常有用。例如，它可以指导进程为相应部分的矩阵和向量分配大小合适的内存空间；如果进程 0（读取矩阵的进程）正好是进程网格中某行的第一个进程，通过确定其自身的坐标可以避免它向自己发送消息。MPI_Cart_coords 可以完全胜任这个要求，其声明如下：

```
int MPI_Cart_coords (MPI_Comm comm, int rank,int dims, int *coords)
```

前三个参数是输入变量，包含调用者传入的信息。ccmm 是通信域，rank 是待确定其坐标的进程；dims 是虚拟网格的维数；最后一个参数作为输出参数，它将返回这个进程在虚拟网格中的坐标。

继续前面的例子，下面是进程号为 grid_id 的进程如何在二维网格中确定其坐标的代码：

```
int grid_id;
MPI_Comm grid_comm;
int grid_coords[2];
...
MPI_Cart_coords (grid_comm, grid_id, 2, grid_coords);
```

8.6.8　函数 MPI_Comm_split

散发（Scatter）是一个组操作。为了仅在进程网格一行的的进程中分发输入矩阵某一行中的数据，我们必须把笛卡儿通信域按行分解为一个个独立的通信域。函数 MPI_Comm_split 将某个已存在的通信域中进程进一步划分为几组，并为每组建立一个新的

通信域。其声明如下：

```
int MPI_Comm_split(MIP_Comm old_comm, int partition,int new_rank,
                   MPI_Comm *new_comm)
```

前三个变量是输入参数：

- old_comm　待划分的进程所在的通信域。由于这个函数是一个组操作，所以 old_comm 中的所有进程都要调用此函数；
- partition　分组数；
- new_rank　在新的通信域中本进程的位置。

函数通过参数 new_comm 返回指向新通信域的指针。

前面我们通过使用 MPI_Cart_create 创建了一个笛卡儿通信域，从而将进程组织到了虚拟二维网格 grid_comm 中。各个进程还可以通过调用 MPI_Cart_coords 将其坐标存入数组 grid_coords 中。grid_coords[0]是行号，grid_coords[1]是列号。

现在我们使用 MPI_Comm_split 将进程网格按行分解。因为我们期望将同一行的各个进程划为一组，所以我们将 grid_coords[0]的值作为分组数，同时以 grid_coords[1]作为进程编号变量来按列为进程编号。

```
MPI_Comm grid_comm; /* 2-D process grid */
MPI_Comm grid_coords[2]; /* Location of process in grid */
MPI_Comm row_comm; /* Processes in same row */
MPI_Comm_split (grid_comm, grid_coords[0],grid_coords[1], &row_comm);
```

同样，也可以用这个函数将笛卡儿通信域按列划分为新的通信域。

有了原先的通信域和这两个新的通信域，我们就可以完成 b 的再分发和最终求和归约中所有的通信操作了。

8.6.9 测试

我们已经编写了基于棋盘式分解的矩阵向量乘法并行程序。关键函数的编写见本章练习。

下面我们来研究本程序的性能模型，我们将仍然使用与测试前两个程序所用的相同的集群系统。同时，我们只考察 p 是平方数时的情形。

同样，令 χ 表示内积计算循环中一次进行迭代所需要的时间。每个进程负责的矩阵的最大尺寸为 $(n/\sqrt{p})\times(n/\sqrt{p})$。所以我们估计并行算法计算部分消耗的时间是 $\chi(n/\sqrt{p})\times(n/\sqrt{p})$。

在重新分发 b 过程的第一步，网格中第一列的进程要向第一行的进程发送相应部分的 A。一个进程最多处理 b 的 $(n/\sqrt{p})$ 个元素。所以发送或接收一个包含这些数据的消息耗时为 $\lambda+8(n/\sqrt{p})/\beta$。在 b 向量重新分发过程的第二步中，网格第一行的进程将相应部分的 b 广播到同列的其他进程，耗时为 $\log\sqrt{p}\ (\lambda+8(n/\sqrt{p})/\beta)$。

当所有进程完成自己部分的矩阵向量乘法运算之后，每行的进程将通过归约得到各自负责的那一部分 c。由于通信时间较长，可以忽略加法运算消耗的时间。通信时间与按列

广播相同：$\log\sqrt{p}\ (\lambda+8(n/\sqrt{p})/\beta)$。

集群配置：450MHz PentiumII，快速以太网参数：χ=63.4ns，λ=250μs, β=106bps。

表 8.3 比较了矩阵向量乘法的实际执行时间和预测的执行时间。问题规模为 1000，分别在 1, 2,…, 8, 16 个处理器上进行测试。实际时间是 100 次运行的平均时间。此程序与前两个程序的加速比如图 8.20 所示。

这种方法与前两种方法所发送的消息数目完全相同。主要的区别在于：本算法每个进程得到的 b 和 c 的元素数是$\Theta(n/\sqrt{p})$，而另两种方法则是$\Theta(n)$。所以当处理器的数目变大的时候，棋盘式分解比列块分解和行块分解的性能更好。我们的实验结果也证实了这一点。当处理器的数目是 1，4 和 9 的时候，棋盘式分解不如另两种方法，但当处理器数增加到 16 的时候它远远超过了另两种方法。

表 8.3　棋盘式分解矩阵向量乘法算法的预测执行时间和实际执行时间的比较。矩阵维 1000×1000，向量有 1000 个元素

处理器数	预期时间	实际时间	加速比	Mflops
1	0.0634	0.0634	1.00	31.6
4	0.0178	0.0174	3.64	114.9
9	0.0097	0.0097	6.53	206.2
16	0.0062	0.0062	10.21	322.6

8.7 本 章 小 结

本章我们设计、分析并测试了 3 个矩阵向量乘法的 MPI 并行程序。三者分别基于按行分解，按列分解和棋盘式分解。

我们分析了这三种算法的等效率特性。棋盘式分解算法等效率函数的性质最好，也就是说它比另两种方法更适合处理器数目较多的情况。测试结果也证明了这一点，如图 8.20 所示。

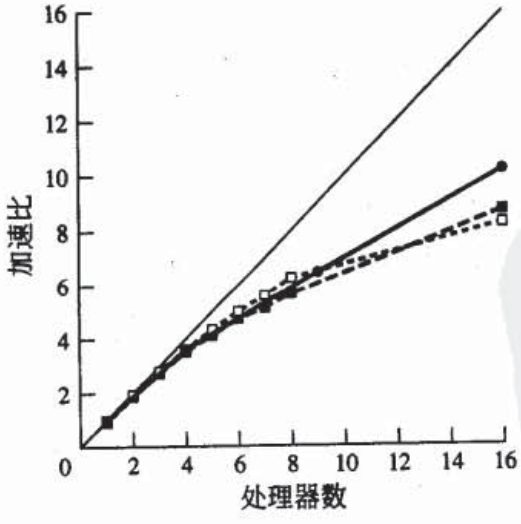

图 8.20　商用集群上三种算法处理 1000×1000 矩阵与长为 1000 的向量乘法的加速情况。点线为按行分解加速情况，短划线是按列分解的加速情况，而实线则是棋盘式分解的加速情况

由于每种算法都基于不同的矩阵向量分解方式，其产生的程序也有不同的通信方式。所以我们遇到了许多功能强大的 MPI 散发和收集函数。我们还学会了如何创建网格拓扑通信域，以及如何将一个通信域内的函数划分为小组以构建新的通信域。

与完成同样功能的 C 程序相比，并行程序和辅助函数的代码总长度要长很多。读取和分发矩阵向量数据在并行程序中要复杂的多，除此之外二者实际用于计算的代码量不相上下。开发和调试并行程序是比较麻烦的，这也就是为什么我们尽量使它们更一般化，放入库中以便重用的原因了。我们也可以使用一些免费的库。ScaLAPACK 项目有一个庞大的与 MPI 兼容的支持科学和工程计算的函数包。ScaLAPACK 的详细信息请见参考文献。

8.8 主 要 术 语

all-gather communication	全收集通信
all-to-all communication	全交换通信
attributes	属性
block-decomposed vector	分块的向量
checkerboard block decomposition	棋盘式分解
columnwise block-striped decomposition	按列块分解
communicator	通信域
context	上下文
distributed vector	分布式向量
process group	进程组
replicated vector	复制（冗余）向量
topology	拓扑

8.9 参 考 文 献

介绍矩阵向量乘法的文献还有：Pacheco【89】（按行分解），Bertsekas 和 Teitsiklis【9】（按行和按列分解），Fox 等【33】（棋盘式分解），Grama 等【44】（按行和棋盘式分解）。

20 世纪 90 年代中期，美国几个政府机构开始了 ScaLAPACK 项目。Oak Ridge 国家实验室、Rice 大学、加州大学伯克利分校、加州大学洛杉矶分校、伊利诺斯大学、田纳西大学诺克斯维尔分校都参与其中。这些组织开发了许多与 MPI 兼容的数值函数库。这些免费的库有许多功能，包括矩阵和向量的基本运算，线性方程求解，特征值和特征向量求解，以及迭代矩阵的预处理。更详细资料请参见 www.netlib.org/scalapack/。

Communications of the ACM 1994 年 3 月刊聚焦于人工智能，其中有三篇关于神经网络的综述。

8.10 练 习 题

8.1　在你的并行计算机上测试本章的并行程序，测试应包括矩阵和向量的读取时间。哪个程序的性能最好？为什么？

8.2　使用本章 8.4 节的性能模型，估算第一个程序的执行时间，加速比和 MFLOPS。假设 n=1,000，处理器为 9,10,…,15 个。

8.3　编写矩阵向量乘法程序，其中矩阵按行分解，向量分块。假设矩阵和向量从数据文件中读取，文件格式与本章例子中用到的相同。程序执行结束时，结果向量 c 分块存在于各个进程中。

8.4　用另一种方法实现 read_col_striped_matrix。在本章的函数中，由一个进程负责打开和读取文件。之后的过程要求与本章函数的实现不同，矩阵的分发将通过 $p-1$ 次简单的发送-接收消息过程来完成。对于矩阵的每一行，读取进程要调用 $p-1$ 次 MPI_Send，其他进程分别调用一次 MPI_Recv。所有进程分配空间都不大于 $n\lceil n/p \rceil$ 个矩阵元素。

8.5　用另一种方法实现 read_col_striped_matrix。不调用 MPI 函数，所有进程各自打开文件，只读取自己需要的部分。

8.6　编写一个计算矩阵向量相乘的程序，矩阵按列分解散发在各个进程中，向量被复制到各个进程中。假设矩阵和向量从数据文件中读取，其格式与本章例子中的相同。程序执行结束时，结果向量 c 需要在各个进程中被复制。

8.7　假设 grid_comm 是具有笛卡儿拓扑结构(进程被组织成一个二维网格)的通信域。编写一段代码将进程网格按列划分。每个进程的 col_comm 变量应该是一个包括有调用进程和所有与之同列进程的通信域，而任何其他进程都不包含在该通信域中。

8.8　假设 grid_comm 是具有笛卡儿拓扑结构(进程被组织成一个二维网格)的通信域。编写一段代码，以说明怎样可以通过使用 read_block_vector 函数来打开包含向量的数据文件并将其分发到 grid_comm 的各个进程中去。其文件名为“Vector”，元素为双精度浮点数。

8.9　为棋盘式分解算法的矩阵向量乘法程序编写重新分发向量 b 的函数。假设向量有 n 个元素，进程组织成 $r \times c$ 的网格。开始时，b 被分发到进程网格中第一列的各个进程中。$(i, 0)$ 处的进程负责从第 BLOCK_LOW (i, r, n) 开始的 BLOCK_SIZE (i, r, n) 个 b 的元素。重新分配后，第 j 列的每个进程负责从 BLOCK_LOW (j, c, n) 开始的 BLOCK_SIZE (j, c, n) 个 b 的元素。

(a) 假设 p 是平方数，即：$r = c$；

(b) 假设 p 不是平方数，即：$r \neq c$；

(c) 对 p 不作任何假设。

8.10　编写基于棋盘式分解的矩阵向量乘法程序。程序从文件读取矩阵及向量并将结果打印至标准输出。输入文件的名称作为命令行参数给出。

8.11　编写一个函数完成大小为 $n \times n$ 的矩阵 A 的转置。假设调用前 A 按行分解位于 p 个进程中。调用之后，A 按列分解位于 p 个进程中。

8.12　二叉搜索树通过线性结构来组织 n 个值，它可以保证 $\Theta(\log n)$ 的检索时间。如

果知道每个值的访问概率，可以建立一棵优化的二叉搜索树以最小化搜索时间，如图 8.21 所示。

brown	0.16
dog	0.13
fox	0.06
jumped	0.08
lazy	0.07
over	0.17
quick	0.05
the	0.28

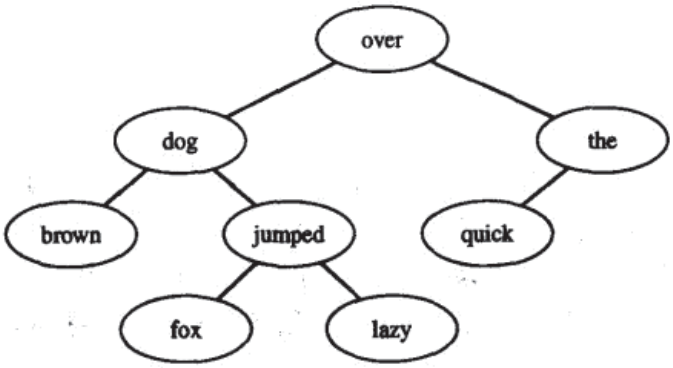

图 8.21 给定一组值和每个值的访问概率，优化的二叉搜索树可以缩短检索时间

图 8.22 是用 C 编写的一个优化二叉搜索树的查找算法，假设已经给定了一组概率值。程序的伪码来自于 Baase 和 Gelder【5】；详细情况可以查阅他们的教材。

```
/*
* Given p[0], p[1], ..., p[N-1],the probability of each key
* in an ordered list of keys being the target of a search,
* this program uses dynamic programming to compute the
* optimal binary search tree that minimizes the average
* number of comparisons needed to find a key.
*
* Last modification: 12 September 2002
*/
#include <stdio.h>
#include <values.h>

main（int argc, char *argv[]）{
    float bestcost; /* Lowest cost subtree found so far */
    int bestroot; /* Root of lowest cost subtree */
    int high; /* Highest key in subtree */
    int i, j;
    int low; /* Lowest key in subtree */
    int n; /* Number of keys */
    int r; /* Possible subtree root */
    float rcost; /* Cost of subtree rooted by r */
    int **root; /* Best subtree roots */
    float **cost; /* Best subtree costs */
    float *p; /* Probability of each key */
    void alloc_matrix（void ***, int, int, int）;
    void print_root（int **, int, int）;
    /* Input the number of keys and probabilities */
```

```
scanf（"%d", &n）;
p =（float *）malloc（n * sizeof（float））;
for（i = 0; i < n; i++）
scanf（"%f", &p[i]）;
/* Find optimal binary search tree */
alloc_matrix（(void ***) &cost, n+1, n+1, sizeof（float））;
alloc_matrix（(void ***) &root, n+1, n+1, sizeof（int））;
for（low = n; low >= 0; low--）{
    cost[low][low] = 0.0;
    root[low][low] = low;
    for（high = low+1; high <= n; high++）{
        bestcost = MAXFLOAT;
        for（r = low; r < high; r++）{
            rcost = cost[low][r] + cost[r+1][high];
            for（j = low; j < high; j++）rcost += p[j];
            if（rcost < bestcost）{
                bestcost = rcost;
                bestroot = r;
            }
        }
        cost[low][high] = bestcost;
        root[low][high] = bestroot;
    }
}
/* Print structure of binary search tree */
print_root（root, 0, n-1）;
}
/* Print the root of the subtree spanning keys
'low' through 'high' */
void print_root（int **root, int low, int high）{
printf（"Root of tree spanning %d-%d is %d\n",
    low, high, root[low][high+1]）;
if（low < root[low][high+1]-1）
print_root（root, low, root[low][high+1]-1）;
if（root[low][high+1] < high-1）
print_root（root, root[low][high+1]+1, high）;
}
/* Allocate a two-dimensional array with 'm' rows and
'n' columns, where each entry occupies 'size' bytes */
void alloc_matrix（void ***a, int m, int n, int size）
{
    int i;
    void *storage;
    storage =（void *）malloc（m * n * size）;
    *a =（void **）malloc（m * sizeof（void *））;
```

```
    for（i = 0; i < m; i++）{
        （*a）[i] = storage + i * n * size;
    }
}
```

图 8.22 优化二叉搜索树查找算法的 C 语言实现

用分解-通信-组合-映射的设计方法对此程序进行并行化（提示：寻找一种允许多个进程同时计算的组合方式）。