单元 V　数值并行算法 MPI 编程实现

第十八章　矩阵运算

第十九章　线性方程组的直接解法

第二十章　线性方程组的迭代解法

第二十一章 矩阵特征值计算

第二十二章 快速傅氏变换和离散小波变换

这一单元主要介绍典型的数值并行算法的 MPI 编程实现，包括矩阵运算（第十八章）、线性方程组的直接解法（第十九章）和迭代解法（第二十章）、矩阵特征值计算（第二十一章）以及 FFT 和 DWT 算法（第二十二章）等。

本单元的第十八章介绍矩阵运算及其 MPI 编程实现，包括矩阵转置、矩阵向量相乘、矩阵乘法（行列划分法、分块 Cannon 法）、矩阵分解（LU 分解、QR 分解、奇异值分解、Cholesky 分解）和矩阵求逆等；第十九章介绍线性方程组的直接解法及其 MPI 编程实现，包括高斯消去法和约旦消去法等；第二十章介绍线性方程组的迭代解法及其 MPI 编程实现，包括雅可比法、高斯-赛德尔法和松弛法等；第二十一章介绍矩阵特征值计算及其 MPI 编程实现，包括乘幂法、雅可比法、单侧旋转法和 QR 法等；第二十二章介绍傅氏变换和小波变换算法及其 MPI 编程实现，包括快速 FFT 变换和二维小波 DWT 变换等。

为了压缩篇幅，第十八章只给出了 Cannon 乘法的 MPI 源程序、矩阵 LU 分解并行算法的 MPI 源程序以及方阵求逆并行算法的 MPI 源程序，第十九章只给出了高斯消去法并行算法的 MPI 源程序，第二十章只给出了高斯-塞德尔迭代并行算法的 MPI 源程序，第二十一章只给出了求对称矩阵特征值的雅可比算法的 MPI 源程序，第二十二章只给出了并行 FFT 算法的 MPI 源程序，其余者均放在随书的光盘中，也可从中国科学技术大学国家高性能计算中心（合肥）的网站 http://www.nhpcc.ustc.edu.cn 下载。

第十八章 矩阵运算

矩阵运算是数值计算中最重要的一类运算,特别是在线性代数和数值分析中,它是一种最基本的运算。本章讨论的矩阵运算包括矩阵转置、矩阵向量相乘、矩阵乘法、矩阵分解以及方阵求逆等。在讨论并行矩阵算法时分三步进行：①算法描述及其串行算法；②算法的并行化及其实现算法框架以及简单的算法分析；③算法实现的 MPI 源程序，以利于读者实践操作。

1.1 矩阵转置

1.1.1 矩阵转置及其串行算法

对于一个 n 阶方阵 $A=[a_{ij}]$，将其每一下三角元素 a_{ij} $(i>j)$沿主对角线与其对称元素 a_{ji} 互换就构成了转置矩阵 A^T。假设一对数据的交换时间为一个单位时间，则下述**矩阵转置**(Matrix Transposing)算法 18.1 的运行时间为$(n^2-n)/2=O(n^2)$。

算法 18.1　单处理器上矩阵转置算法

输入：矩阵 $A_{n\times n}$

输出：矩阵 $A_{n\times n}$ 的转置 $A^T{}_{n\times n}$

```
Begin
    for i=2 to n do
        for j=1 to i-1 do
            交换 a[i,j]和 a[j,i]
        endfor
    endfor
End
```

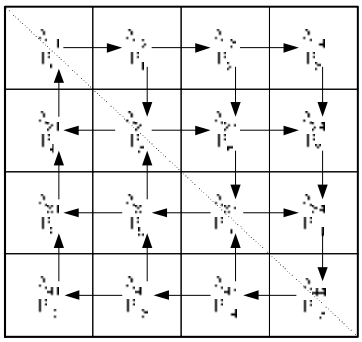

图 1.1　子块转置

1.1.2 矩阵转置并行算法

此处主要讨论网孔上的**块棋盘划分**(Block-Checker Board Partitioning，又称为块状划分)的矩阵转置算法，它对**循环棋盘划分**(Cyclic-Checker Board Partitioning)也同样适用。另外，超立方上块棋盘划分的矩阵转置算法可参见文献[1]。

实现矩阵的转置时，若处理器个数为 p，且它们的编号依次是 0,1, …,p-1，则将 n 阶矩

阵 A 分成 p 个大小为 $m\times m$ 的子块，$m=\lceil n/p\rceil$。p 个子块组成一个 $\sqrt{p}\times\sqrt{p}$ 的子块阵列。记其中第 i 行第 j 列的子块为 A_{ij}，它含有 A 的第$(i\text{-}1)m+1$ 至第 im 行中的第$(j\text{-}1)m+1$ 至第 jm 列的所有元素。对每一处理器按行主方式赋以二维下标，记编号为 i 的处理器的二维下标为(v,u)，其中 $v=\lfloor i/\sqrt{p}\rfloor$，$u=i \bmod \sqrt{p}$，将 A 的子块存入下标为(v,u)表示的对应处理器中。

这样，转置过程分两步进行：第一步，子块转置，具体过程如图 1.1 所示；第二步，处理器内部局部转置。

为了避免对应子块交换数据时处理器发生死锁，可令下三角子块先向与之对应的上三角子块发送数据，然后从上三角子块接收数据；上三角子块先将数据存放在缓冲区 buffer 中，然后从与之对应的下三角子块接收数据；最后再将缓冲区中的数据发送给下三角子块。具体并行算法框架描述如下：

算法 18.2　网孔上的矩阵转置算法

输入：矩阵 $A_{n\times n}$

输出：矩阵 $A_{n\times n}$ 的转置 $A^T{}_{n\times n}$

Begin

对所有处理器 my_rank(my_rank=0,…,p-1)同时执行如下的算法:

(1)计算子块的行号 v=my_rank/ sqrt(p), 计算子块的列号 u=my_rank mod sqrt(p)

(2)**if** (u<v) **then**　　/*对存放下三角块的处理器*/

(2.1)将所存的子块发送到其对角块所在的处理器中

(2.2)接收其对角块所在的处理器中发来的子块

else　　/*对存放上三角块的处理器*/

(2.3)将所存的子块在缓冲区 buffer 中做备份

(2.4)接收其对角块所在的处理器中发来的子块

(2.5)将 buffer 中所存的子块发送到其对角块所在的处理器中

end if

(3)**for** i=1 **to** m **do**　/*处理器内部局部转置*/

for j=1 **to** i **do**

交换 $a[i,j]$和 $a[j,i]$

end for

end for

End

若记 t_s 为发送启动时间，t_w 为单位数据传输时间，t_h 为处理器间的延迟时间，则第一步由于每个子块有 n^2/p 个元素，又由于通信过程中为了避免死锁，错开下三角子块与上三角子块的发送顺序，因此子块的交换时间为 $2(t_s+t_w n^2/p+t_h\sqrt{p})$；第二步，假定一对数据的交换时间为一个单位时间，则局部转置时间为 $n^2/2p$。因此所需的并行计算时间

$$T_p=\frac{n^2}{2p}+2t_s\sqrt{p}+2t_w\frac{n^2}{p}+t_h\sqrt{p}$$。

MPI 源程序请参见所附光盘。

1.2 矩阵-向量乘法

1.2.1 矩阵-向量乘法及其串行算法

矩阵-向量乘法(Matrix-Vector Multiplication)是将一个 $n\times n$ 阶方阵 $\boldsymbol{A}=[a_{ij}]$乘以 $n\times 1$ 的向量 $\boldsymbol{B}=[b_1,b_2,\cdots,b_n]^T$得到一个具有 n 个元素的列向量 $\boldsymbol{C}=[c_1,c_2,\cdots,c_n]^T$。假设一次乘法和加法运算时间为一个单位时间，则下述矩阵向量乘法算法 18.3 的时间复杂度为 $O(n^2)$。

算法 18.3　单处理器上矩阵-向量乘算法

输入：$A_{n\times n}$，$B_{n\times 1}$

输出：$C_{n\times 1}$

```
Begin
    for i=0 to n-1 do
        c[i]= 0
        for j=0 to n-1 do
            c[i]=c[i] + a[i,j]*b[j]
        end for
    end for
End
```

1.2.2 矩阵-向量乘法的并行算法

矩阵-向量乘法同样可以有**带状划分**(Striped Partitioning)和**棋盘划分**(Checker Board Partitioning)两种并行算法。以下仅讨论行带状划分矩阵-向量乘法,列带状划分矩阵-向量乘法是类似的。设处理器个数为 p，对矩阵 $\boldsymbol{A}$ 按行划分为 p 块，每块含有连续的 m 行向量，$m=\lceil n/p \rceil$，这些行块依次记为 $\boldsymbol{A}_0,\boldsymbol{A}_1,\cdots,\boldsymbol{A}_{p-1}$，分别存放在标号为 0,1,⋯,$p$-1 的处理器中，同时将向量 $\boldsymbol{B}$ 广播给所有处理器。各处理器并行地对存于局部数组 a 中的行块 $\boldsymbol{A}_i$ 和向量 $\boldsymbol{B}$ 做乘积操作，具体并行算法框架描述如下：

算法 18.4　行带状划分的矩阵-向量乘并行算法

输入：$A_{n\times n}$，$B_{n\times 1}$

输出：$C_{n\times 1}$

```
Begin
    对所有处理器 my_rank(my_rank=0,⋯, p-1)同时执行如下的算法:
    for i=0 to m-1 do
        c(i)=0.0
        for j=0 to n-1 do
            c[i] = c[i]+a[i,j]*b[j]
        end for
    end for
End
```

假设一次乘法和加法运算时间为一个单位时间，不难得出行带状划分的矩阵-向量乘算法 18.4 并行计算时间 $T_p=n^2/p$，若处理器个数和向量维数相当，则其时间复杂度为 $O(n)$。

MPI 源程序请参见所附光盘。

1.3 行列划分矩阵乘法

一个 $m\times n$ 阶矩阵 $\boldsymbol{A}=[a_{ij}]$乘以一个 $n\times k$ 的矩阵 $\boldsymbol{B}=[b_{ij}]$就可以得到一个 $m\times k$ 的矩阵 $\boldsymbol{C}=[c_{ij}]$，它的元素 c_{ij} 为 $\boldsymbol{A}$ 的第 i 行向量与 $\boldsymbol{B}$ 的第 j 列向量的内积。矩阵相乘的关键是相乘的两个元素的下标要满足一定的要求(即对准)。为此常采用适当旋转矩阵元素的方法(如后面将要阐述的 Cannon 乘法)，或采用适当复制矩阵元素的办法(如 DNS 算法)，或采用流水线的办法使元素下标对准，后两种方法读者可参见文献[1]。

1.3.1 矩阵相乘及其串行算法

由矩阵乘法定义容易给出其串行算法 18.5，若一次乘法和加法运算时间为一个单位时间，则显然其时间复杂度为 O(mnk) 。

算法 18.5　单处理器上矩阵相乘算法

输入：$A_{m\times n}$，$B_{n\times k}$

输出：$C_{m\times k}$

```
Begin
    for i=0 to m-1 do
        for j=0 to k-1 do
            c[i,j]=0
            for r=0 to n-1 do
                c[i,j]= c[i,j]+a[i,r]*b[r,j]
            end for
        end for
    end for
End
```

1.3.2 简单的矩阵并行分块乘法算法

矩阵乘法也可以用分块的思想实现并行，即**分块矩阵乘法**(Block Matrix Multiplication)，将矩阵 $\boldsymbol{A}$ 按行划分为 p 块(p 为处理器个数)，设 $u=\lceil m/p\rceil$，每块含有连续的 u 行向量，这些行块依次记为 $\boldsymbol{A}_0,\boldsymbol{A}_1,\cdots,\boldsymbol{A}_{p-1}$，分别存放在标号为 0,1,…, p-1 的处理器中。对矩阵 $\boldsymbol{B}$ 按列划分为 P 块，记 $v=\lceil k/p\rceil$，每块含有连续的 v 列向量，这些列块依次记为 $\boldsymbol{B}_0,\boldsymbol{B}_1,\cdots,\boldsymbol{B}_{p-1}$，分别存放在标号 0,1, …, p-1 为的处理器中。将结果矩阵 $\boldsymbol{C}$ 也相应地同时进行行、列划分，得到 $p\times p$ 个大小为 $u\times v$ 的子矩阵，记第 i 行第 j 列的子矩阵为 $\boldsymbol{C}_{ij}$，显然有 $\boldsymbol{C}_{ij}=\boldsymbol{A}_i\times\boldsymbol{B}_j$，其中，$\boldsymbol{A}_i$ 大小为 $u\times n$，$\boldsymbol{B}_j$ 大小为 $n\times v$。

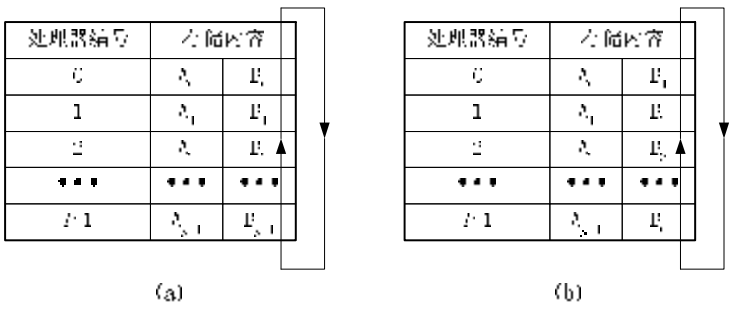

图 1.2 矩阵相乘并行算法中的数据交换

开始，各处理器的存储内容如图 1.2 (a)所示。此时各处理器并行计算 $\boldsymbol{C}_{ii}=\boldsymbol{A}_i\times\boldsymbol{B}_j$ 其中 i=0,1,⋯,p-1，此后第 i 号处理器将其所存储的 B 的列块送至第 i-1 号处理器（第 0 号处理器将 B 的列块送至第 p-1 号处理器中，形成循环传送），各处理器中的存储内容如图 1.2 (b)所示。它们再次并行计算 $\boldsymbol{C}_{ij}=\boldsymbol{A}_i\times\boldsymbol{B}_j$，这里 j=(i+1)mod$p$。B 的列块在各处理器中以这样的方式循环传送 p-1 次并做 p 次子矩阵相乘运算，就生成了矩阵 $\boldsymbol{C}$ 的所有子矩阵。编号为 i 的处理器的内部存储器存有子矩阵 $\boldsymbol{C}_{i0},\boldsymbol{C}_{i1},\ldots,\boldsymbol{C}_{i(p-1)}$。为了避免在通信过程中发生死锁，奇数号及偶数号处理器的收发顺序被错开，使偶数号处理器先发送后接收；而奇数号处理器先将 B 的列块存于缓冲区 buffer 中，然后接收编号在其后面的处理器所发送的 B 的列块，最后再将缓冲区中原矩阵 B 的列块发送给编号在其前面的处理器，具体并行算法框架描述如下：

算法 18.6　矩阵并行分块乘法算法

输入：$A_{m\times n}$, $B_{n\times k}$,

输出：$C_{m\times k}$

Begin

对所有处理器 my_rank(my_rank=0,⋯,p-1)同时执行如下的算法:

(1)目前计算 C 的子块号 l=(i+my_rank) mod p

(2)**for** z=0 **to** u-1 **do**

for j=0 **to** v-1 **do**

$c[l,z,j]$=0

for s=0 **to** n-1 **do**

$c[l,z,j]=c[l,z,j]+a[z,s]*b[s,j]$

end for

end for

end for

(3)计算左邻处理器的标号 mm1=(p+my_rank-1) mod p

计算右邻处理器的标号 mp1=(my_rank+1) mod p

(4)**if** ($i\neq p$-1) **then**

(4.1)**if** (my_rank mod 2= 0) **then**　/*编号为偶数的处理器*/

(i)将所存的 B 的子块发送到其左邻处理器中

(ii)接收其右邻处理器中发来的 B 的子块

end if

(4.2)**if** (my_rank mod 2≠ 0) **then**　/*编号为奇数的处理器*/

(i)将所存的 B 子块在缓冲区 buffer 中做备份

(ii)接收其右邻处理器中发来的 B 的子块

(iii)将 buffer 中所存的 B 的子块发送到其左邻处理器中

end if

end if

End

设一次乘法和加法运算时间为一个单位时间，由于每个处理器计算 p 个 $u\times n$ 与 $n\times v$ 阶的子矩阵相乘，因此计算时间为 $u*v*n*p$；所有处理器交换数据 p-1 次，每次的通信量为 $v*n$，通信过程中为了避免死锁，错开奇数号及偶数号处理器的收发顺序，通信时间为 $2(p-1)(t_s+nv*t_w)=O(nk)$，所以并行计算时间 $T_p=uvnp+2(p-1)(t_s+nvt_w)=mnk/p+2(p-1)(t_s+nvt_w)$。

MPI 源程序请参见所附光盘。

1.4 Cannon 乘法

1.4.1 Cannon 乘法的原理

Cannon 算法是一种存储有效的算法。为了使两矩阵下标满足相乘的要求，它和上一节的并行分块乘法不同，不是仅仅让 B 矩阵的各列块循环移动，而是有目的地让 A 的各行块以及 B 的各列块皆施行循环移位，从而实现对 C 的子块的计算。将矩阵 A 和 B 分成 p 个方块 A_{ij} 和 B_{ij}，$(0 \le i, j \le \sqrt{p}-1)$，每块大小为 $\lceil n/\sqrt{p} \rceil \times \lceil n/\sqrt{p} \rceil$，并将它们分配给 $\sqrt{p} \times \sqrt{p}$ 个处理器 $(P_{00}, P_{01}, ..., P_{\sqrt{p}-1\sqrt{p}-1})$。开始时处理器 P_{ij} 存放块 A_{ij} 和 B_{ij}，并负责计算块 C_{ij}，然后算法开始执行：

⑴将块 A_{ij} 向左循环移动 i 步；将块 B_{ij} 向上循环移动 j 步；

⑵P_{ij} 执行乘加运算后将块 A_{ij} 向左循环移动 1 步，块 B_{ij} 向上循环移动 1 步；

⑶重复第⑵步，总共执行 $\sqrt{p}$ 次乘加运算和 $\sqrt{p}$ 次块 A_{ij} 和 B_{ij} 的循环单步移位。

1.4.2 Cannon 乘法的并行算法

图 1.3 示例了在 16 个处理器上，用 Cannon 算法执行 $A_{4\times4} \times B_{4\times4}$ 的过程。其中(a)和(b)对应于上述算法的第⑴步；(c)、(d)、(e)、(f)对应于上述算法的第⑵和第⑶步。在算法第⑴步时，A 矩阵的第 0 列不移位，第 1 行循环左移 1 位，第 2 行循环左移 2 位，第 3 行循环左移 3 位；类似地，B 矩阵的第 0 行不移位，第 1 列循环上移 1 位，第 2 列循环上移 2 列，第 3 列循环上移 3 列。这样 Cannon 算法具体描述如下：

算法 18.7　Cannon 乘法算法

输入： $A_{n\times n}$，$B_{n\times n}$

输出： $C_{n\times n}$

Begin

对所有处理器 my_rank(my_rank=0,…,p-1)同时执行如下的算法:

(1)计算子块的行号 i=my_rank/sqrt(p)

计算子块的列号 j=my_rank mod sqrt(p)

(2)**for** k=0 **to** $\sqrt{p}$ -1 **do**

if (i>k) **then** Leftmoveonestep(a) **end if**　/* a 循环左移至同行相邻处理器中*/

if (j>k) **then** Upmoveonestep(b) **end if**　/* b 循环上移至同列相邻处理器中*/

end for

(3)**for** i=0 **to** m-1 **do**

for j=0 **to** m-1 **do**

c[i,j]=0

end for

end for

(4)**for** k=0 **to** $\sqrt{p}$ -1 **do**

for i=0 **to** m-1 **do**

for j=0 **to** m-1 **do**

for k1=0 **to** m-1 **do**

```
                    c[i,j]= c[i,j]+ a[i,k1]* b[k1,j]
                end for
            end for
        end for
        Leftmoveonestep(a) /*子块 a 循环左移至同行相邻的处理器中*/
        Upmoveonestep(b) /*子块 b 循环上移至同列相邻的处理器中*/
    end for
End
```

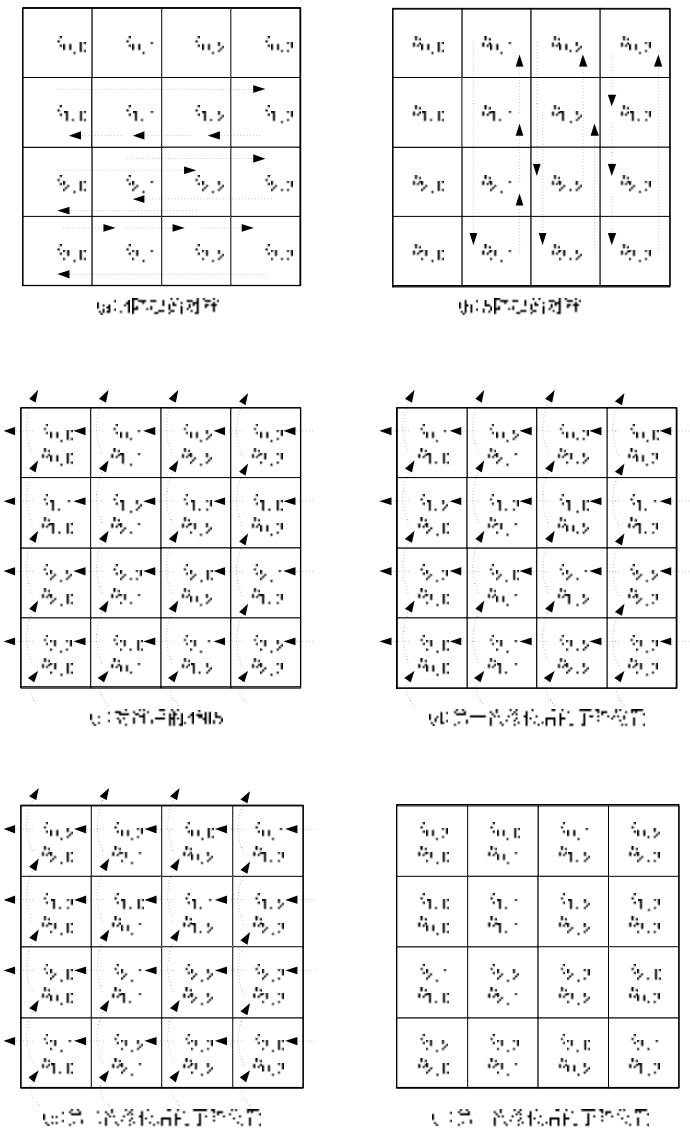

图 1.3　16 个处理器上的 Cannon 乘法过程

这里函数 Leftmoveonestep(*a*)表示子块 *a* 在编号处于同行的处理器之间以循环左移的方式移至相邻的处理器中；函数 Upmoveonestep(*b*)表示子块 *b* 在编号处于同列的处理器之间以循环上移的方式移至相邻的处理器中。这里我们以函数 Leftmoveonestep(*a*)为例，给出处理器间交换数据的过程：

算法 18.8　函数 Leftmoveonestep(*a*)的基本算法

Begin

(1)**if** (*j*=0) **then**　/*最左端的子块*/

(1.1)将所存的 A 的子块发送到同行最右端子块所在的处理器中

(1.2)接收其右邻处理器中发来的 A 的子块

end if

(2)**if** ((*j* = sqrt(*p*)-1) and (*j* mod 2 = 0)) **then**　/*最右端子块处理器且块列号为偶数*/

(2.1)将所存的 A 的子块发送到其左邻处理器中

(2.2)接收其同行最左端子块所在的处理器发来的 A 的子块

end if

(3)**if** ((*j* = sqrt(*p*)-1) and (*j* mod 2 ≠ 0)) **then**　/*最右端子块处理器且块列号为奇数*/

(3.1)将所存的 A 的子块在缓冲区 buffer 中做备份

(3.2)接收其同行最左端子块所在的处理器发来的 A 的子块

(3.3)将在缓冲区 buffer 中所存的 A 的子块发送到其左邻处理器中

end if

(4)**if** ((*j* ≠ sqrt(*p*)-1) and (*j* mod 2 = 0) and (*j* ≠ 0)) **then**　/*其余的偶数号处理器*/

(4.1)将所存的 A 的子块发送到其左邻处理器中

(4.2)接收其右邻处理器中发来的 A 的子块

end if

(5)**if** ((*j* ≠ sqrt(*p*)-1) and (*j* mod 2 = 1) and (*j* ≠ 0)) **then**　/*其余的奇数号处理器*/

(5.1)将所存的 A 的子块在缓冲区 buffer 中做备份

(5.2)接收其右邻处理器中发来的 A 的子块

(5.3)将在缓冲区 buffer 中所存的 A 的子块发送到其左邻处理器中

end if

End

当算法执行在 $\sqrt{p}\times\sqrt{p}$ 的二维网孔上时,若使用切通 CT 选路法,算法 18.7 第(2)步的循环移位时间为 $2(t_s+t_w\frac{n^2}{p})\sqrt{p}$,第(4)步的单步移位时间为 $2(t_s+t_w\frac{n^2}{p})\sqrt{p}$ 、运算时间为 n^3/p 。所以在二维网孔上 Cannon 乘法的并行运行时间为 $T_p=4(t_s+t_w\frac{n^2}{p})\sqrt{p}+n^3/p$ 。

MPI 源程序请参见章末附录。

1.5 LU 分 解

从本小节起我们将对 LU 分解等矩阵分解的并行计算做一些简单讨论。在许多应用问题的科学计算中，矩阵的 LU 分解是基本、常用的一种矩阵运算，它是求解线性方程组的基础，尤其在解多个同系数阵的线性方程组时特别有用。

1.5.1 矩阵的 LU 分解及其串行算法

对于一个 n 阶非奇异方阵 $A=[a_{ij}]$，对 A 进行 LU 分解是求一个主对角元素全为 1 的下三角方阵 $L=[l_{ij}]$与上三角方阵 $U=[u_{ij}]$，使 A=LU。设 A 的各阶主子行列式皆非零，U 和 L 的元素可由下面的递推式求出：

$a_{ij}^{(1)}=a_{ij}$

$a_{ij}^{(k+1)}=a_{ij}^{(k)}-l_{ik}u_{kj}$

$$l_{ik}=\begin{cases}0 & i<k\\ 1 & i=k\\ a_{ik}^{(k)}u_{kk}^{-1} & i>k\end{cases}$$

$$u_{kj}=\begin{cases}0 & k>j\\ a_{kj}^{(k)} & k\le j\end{cases}$$

在计算过程中，首先计算出 U 的第一行元素，然后算出 L 的第一列元素，修改相应 A 的元素；再算出 U 的第二行，L 的第二列…，直至算出 u_{nn} 为止。若一次乘法和加法运算或一次除法运算时间为一个单位时间，则下述 LU 分解的串行算法 18.9 时间复杂度为 $\sum_{i=1}^{n}(i-1)i=(n^3-n)/3=O(n^3)$。

算法 18.9　单处理器上矩阵 LU 分解串行算法

输入：矩阵 $\boldsymbol{A}_{n\times n}$

输出：下三角矩阵 $\boldsymbol{L}_{n\times n}$,上三角矩阵 $\boldsymbol{U}_{n\times n}$

```
Begin
    (1)for k=1 to n do
        (1.1)for i=k+1 to n do
                a[i,k]=a[i,k]/a[k,k]
            end for
        (1.2)for i=k+1 to n do
                for j=k+1 to n do
                    a[i,j]=a[i,j]-a[i,k]*a[k,j]
                end for
            end for
      end for
    (2)for i=1 to n do
        (2.1)for j=1 to n do
                if  (j<i)  then
                    l[i,j]=a[i,j]
                else
                    u[i,j]=a[i,j]
                end if
            end for
        (2.2)l[i,i]=1
      end for
End
```

1.5.2 矩阵 LU 分解的并行算法

在 LU 分解的过程中，主要的计算是利用主行 i 对其余各行 j，($j>i$)作初等行变换，各行计算之间没有数据相关关系，因此可以对矩阵 A 按行划分来实现并行计算。考虑到在计算过程中处理器之间的负载均衡，对 A 采用行交叉划分：设处理器个数为 p，矩阵 A 的阶数为

n， $m=\lceil n/p \rceil$，对矩阵 A 行交叉划分后，编号为 $i(i=0,1,\cdots,p-1)$的处理器存有 A 的第 $i, i+p,\cdots, i+(m-1)p$ 行。然后依次以第 0,1,⋯,n-1 行作为主行，将其广播给所有处理器，各处理器利用主行对其部分行向量做行变换，这实际上是各处理器轮流选出主行并广播。若以编号为 my_rank 的处理器的第 i 行元素作为主行，并将它广播给所有处理器，则编号大于等于 my_rank 的处理器利用主行元素对其第 i+1,⋯,m-1 行数据做行变换，其它处理器利用主行元素对其第 i,⋯,m-1 行数据做行变换。具体并行算法框架描述如下：

算法 18.10　矩阵 LU 分解的并行算法

输入：矩阵 $A_{n\times n}$

输出：下三角矩阵 $\boldsymbol{L}_{n\times n}$,上三角矩阵 $\boldsymbol{U}_{n\times n}$

```
Begin
    对所有处理器 my_rank(my_rank=0,⋯,p-1)同时执行如下的算法:
    for i=0 to m-1 do
        for j=0 to p-1 do
            (1)if (my_rank=j) then   /*当前处理的主行在本处理器*/
                (1.1) v=i*p+j        /* v 为当前处理的主行号*/
                (1.2)for k=v to n do
                        f[k]=a[i,k]   /*  主行元素存到数组 f 中*/
                    end for
                (1.3)向其它所有处理器广播主行元素
              else   /*当前处理的主行不在本处理器*/
                (1.4)v=i*p+j
                (1.5)接收主行所在处理器广播来的主行元素
              end if
            (2)if (my_rank≤j) then
                (2.1)for k=i+1 to m-1 do
                        (i)a[k,v]=a[k,v]/f[v]
                        (ii)for w=v+1 to n-1 do
                            a[k,w]=a[k,w]-f[w]*a[k,v]
                          end for
                    end for
              else
                (2.2)for k=i to m-1 do
                    (i)a[k,v]=a[k,v]/f[v];
                    (ii)for w=v+1 to n-1 do
                        a[k,w]=a[k,w]-f[w]*a[k,v]
                      end for
                    end for
              end if
        end for
    end for
End
```

计算完成后，编号为 0 的处理器收集各处理器中的计算结果，并从经过初等行变换的矩阵 A 中分离出下三角矩阵 L 和上三角矩阵 U。若一次乘法和加法运算或一次除法运算时间为一个单位时间，则计算时间为$(n^3-n)/3p$；又 n-1 行数据依次作为主行被广播，通信时间为

$(n\text{-}1)(t_s+nt_w)\log p=O(n\log p)$，因此并行计算时间 $T_p=(n^3\text{-}n)/3p+(n\text{-}1)(t_s+nt_w)\log p$。

MPI 源程序请参见章末附录。

1.6 QR 分 解

$A=[a_{ij}]$ 为一个 n 阶实矩阵，对 A 进行 QR 分解，就是求一个**非奇异**(Nonsingular)方阵 Q 与上三角方阵 R，使得 $A=QR$。其中方阵 Q 满足：$Q^{\mathrm{T}}=Q^{-1}$，称为**正交矩阵**(Orthogonal Matrix)，因此 QR 分解又称为正交三角分解。

1.6.1 矩阵 QR 分解的串行算法

对 A 进行正交三角分解，可用一系列平面旋转矩阵左乘 A，以使 A 的下三角元素逐个地被消为 0。设要消去的下三角元素为 $a_{ij}(i>j)$，则所用的平面旋转方阵为：

$$Q_{ij}=\begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & c & s & & \\ & & -s & c & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

其中，除了$(i,i)(j,j)$元素等于 c，(i,j)元素与(j,i)元素分别等于$-s$ 与 s 以外，其余元素皆与单位方阵 I 的对应元素相等，而 c,s 由下式计算：

$$c=\cos q=a_{jj}/\sqrt{a_{jj}^2+a_{ij}^2}$$

$$s=\sin q=a_{ij}/\sqrt{a_{jj}^2+a_{ij}^2}$$

其中，θ 为旋转角度。这样，在旋转后所得到的方阵 A' 中，元素 a_{ij}' 为 0，A' 与 A 相比仅在第 i 行、第 j 行上的元素不同：

$$a_{jk}^{'}=c\times a_{jk}+s\times a_{ik}$$

$$a_{ik}^{'}=-s\times a_{jk}+c\times a_{ik} \qquad (k=1,2,...,n)$$

消去 A 的 $r=n(n\text{-}1)/2$ 个下三角元素后得到上三角阵 R，实际上是用 r 个旋转方阵 $Q_1,Q_2,\cdots,Q_r$ 左乘 A，即：

$$\mathrm{R}=Q_r\,Q_{r\text{-}1}\cdots Q_1A$$

设 $Q_0=Q_r\,Q_{r\text{-}1}\cdots Q_1$，易知 Q_0 为一正交方阵，则有：

$$\mathrm{R}=Q_0A$$

即

$$A=Q_0^{-1}\mathrm{R}=Q_0^{\mathrm{T}}\mathrm{R}=Q\mathrm{R}$$

其中 $Q=Q_0^{-1}=Q_0^{T}$ 为一正交方阵，而 Q_0 可以通过对单位阵 I 进行同样的变换而得到, 这样可得到 A 的正交三角分解。QR 分解串行算法如下：

算法 18.11　单处理器上矩阵的 QR 分解串行算法算法

输入：矩阵 $A_{n\times n}$，单位矩阵 Q

输出：矩阵 $Q_{n\times n}$，矩阵 $R_{n\times n}$

```
Begin
    (1)for j=1 to n do
        for i=j+1 to n do
            (i)sq=sqrt(a[j,j]*a[j,j]+a[i,j]*a[i,j])
                c= a[j,j]/sq
                s= a[i,j]/sq
            (ii)for k=1 to n do
                aj[k]= c*a[j,k] + s*a[i,k]
                qj[k]=c*q[j,k] + s*q[i,k]
                ai[k]= −s*a[j,k] + c*a[i,k]
                qi[k]= −s*q[j,k] + c*q[i,k]
              end for
            (iii)for k=1 to n do
                a[j,k]=aj[k]
                q[j,k]=qj[k]
                a[i,k]=ai[k]
                q[i,k]=qi[k]
              end for
        end for
     end for
    (2)R=A
    (3)MATRIX_TRANSPOSITION(Q)     /* 对 Q 实施算法 18.1 的矩阵转置*/
End
```

算法 18.11 要对 $n(n-1)/2$ 个下三角元素进行消去，每消去一个元素要对两行元素进行旋转变换，而对一个元素进行旋转变换又需要 4 次乘法和 2 次加法，所以若取一次乘法或一次加法运算时间为一个单位时间，则该算法的计算时间为 $n(n-1)/2*2n*6=6n^2(n-1)=O(n^3)$。

1.6.2 矩阵 QR 分解的并行算法

由于 QR 分解中消去 a_{ij} 时，同时要改变第 i 行及第 j 行两行的元素，而在 LU 分解中，仅利用主行 $i(i<j)$变更第 j 行的元素。因此 QR 分解并行计算中对数据的划分与分布与 LU 分解就不一样。设处理器个数为 p，对矩阵 A 按行划分为 p 块，每块含有连续的 m 行向量，$m=\lceil n/p \rceil$，这些行块依次记为 $A_0,A_1,\cdots,A_{p-1}$，分别存放在标号为 0,1,⋯,p-1 的处理器中。

在 0 号处理器中，计算开始时，以第 0 行数据作为主行，依次和第 0,1,⋯,m-1 行数据做旋转变换，计算完毕将第 0 行数据发送给 1 号处理器，以使 1 号处理器将收到的第 0 行数据作为主行和自己的 m 行数据依次做旋转变换；在此同时，0 号处理器进一步以第 1 行数据作为主行，依次和第 2,3,⋯,m-1 行数据做旋转变换，计算完成后将第 1 行数据发送给 1 号处理器；如此循环下去。一直到以第 m-1 行数据作为主行参与旋转变换为止。

在 1 号处理器中，首先以收到的 0 号处理器的第 0 行数据作为主行和自己的 m 行数据做旋转变换，计算完将该主行数据发送给 2 号处理器；然后以收到的 0 号处理器的第 1 行数据作为主行和自己的 m 行数据做旋转变换，再将该主行数据发送给 2 号处理器；如此循环下去，一直到以收到的 0 号处理器的第 m-1 行数据作为主行和自己的 m 行数据做旋转变换并将第 m-1 行数据发送给 2 号处理器为止。然后，1 号处理器以自己的第 0 行数据作为主行，依次和第 1, 2,⋯,m-1 行数据做旋转变换，计算完毕将第 0 行数据发送给 2 号处理器；接着以

第 1 行数据作为主行，依次和第 2,3,…,m-1 行数据做旋转变换，计算完毕将第 1 行数据发送给 2 号处理器；如此循环下去。一直到以第 m-1 行数据作为主行参与旋转变换为止。除了 p-1 号处理器以外的所有处理器都按此规律先顺序接收前一个处理器发来的数据，并作为主行和自己的 m 行数据作旋转变换，计算完毕将该主行数据发送给后一个处理器。然后依次以自己的 m 行数据作主行对主行后面的数据做旋转变换，计算完毕将主行数据发给后一个处理器。

在 p-1 号处理器中，先以接收到的数据作为主行参与旋转变换，然后依次以自己的 m 行数据作为主行参与旋转变换。所不同的是，p-1 号处理器作为最后一个处理器，负责对经过计算的全部数据做汇总。具体并行算法框架描述如下：

算法 18.12　矩阵 QR 分解并行算法

输入：矩阵 $A_{n\times n}$，单位矩阵 Q

输出：矩阵 $Q_{n\times n}$，矩阵 $R_{n\times n}$

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

(1)**if** (my_rank=0) **then**　/*0 号处理器*/

(1.1)**for** j=0 **to** m-2 **do**

(i)**for** i=j+1 **to** m-1 **do**

Turnningtransform( )　/*旋转变换*/

end for

(ii)将旋转变换后的 A 和 Q 的第 j 行传送到第 1 号处理器

end for

(1.2)将旋转变换后的 A 和 Q 的第 m-1 行传送到第 1 号处理器

end if

(2)**if** ((my_rank>0) and (my_rank<(p-1))) **then**　/*中间处理器*/

(2.1) **for** j=0 **to** my_rank***m**-1 **do**

(i)接收左邻处理器传送来的 A 和 Q 子块中的第 j 行

(ii)**for** i=0 **to** m-1 **do**

Turnningtransform( )　/*旋转变换*/

end for

(iii)将旋转变换后的 A 和 Q 子块中的第 j 行传送到右邻处理器

end for

(2.2) **for** j=0 **to** m-2 **do**

(i)z=my_rank***m**

(ii)**for** i=j+1 **to** m-1 **do**

Turnningtransform( )　/*旋转变换*/

end for

(iii)将旋转变换后的 A 和 Q 子块中的第 j 行传送到右邻处理器

end for

(2.3)将旋转变换后的 A 和 Q 子块中的第 m-1 行传送到右邻处理器

end if

(3)**if** (my_rank= (p-1)) **then**　/*p-1 号处理器*/

(3.1) **for** j=0 **to** my_rank***m**-1 **do**

(i)接收左邻处理器传送来的 A 和 Q 子块中的第 j 行

(ii)**for** i=0 **to** m-1 **do**

```
                Turnningtransform( )   /*旋转变换*/
            end for
        end for
    (3.2)for j=0 to m-2 do
            for i=j+1 to m-1 do f
                Turnningtransform( )   /*旋转变换*/
            end for
        end for
    end if
End
```

当所有的计算完成后，编号为 p-1 的处理器负责收集各处理器中的计算结果，R 矩阵为经旋转变换的 A 矩阵，Q 矩阵为经旋转变换的 Q 的转置。QR 分解并行计算的时间复杂度分析因每个处理器的开始计算时间不同而不同于其它算法，每个处理器都要顺序对其局部存储器中的 m 行向量做自身的旋转变换，其时间用 $T_{computer}$ 表示。以 16 阶矩阵为例，4 个处理器对其进行 QR 分解的时间分布图如图 1.4 所示，这里 $i^{(j)}$ 表示以接收到的第 j 号处理器的第 i 行数据作为主行和自己的 m 行数据做旋转变换。Δt 表示第 p-1 号处理器与第 0 号处理器开始计算时间的间隔。

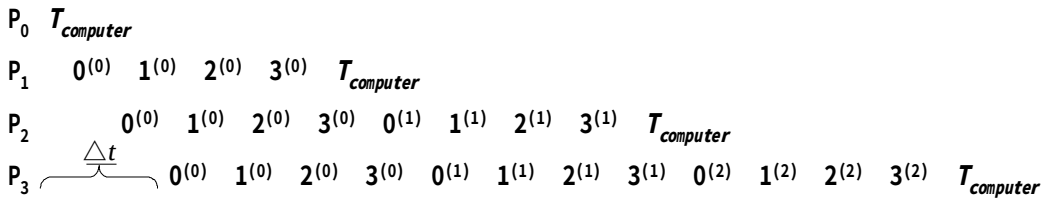

图 1.4　QR 分解并行计算的时间分布图

此处不妨以第 p-1 号处理器为对象分析并行计算时间，若取一次乘法或一次加法运算时间为一个单位时间，由于对一个元素进行旋转变换需要 4 次乘法和 2 次加法，因而需要六个单位时间。要对 $m(m-1)/2$ 个下三角元素进行消去，每消去一个元素要对两行元素进行旋转变换，故自身的旋转变换时间 $T_{computer}=6mn(m-1)$；第 p-1 号处理器依次接收前 n-m 行数据作为主行参与旋转变换，其计算时间为$(n-m)*m*2n*6=12nm(n-m)$，通信时间为 $2(n-m)(t_s+nt_w)$；时间间隔Δt 即第 p-1 号处理器等待第 0 号处理器第 0 行数据到达的时间，第 0 行数据在第 0 号处理器中参与$(m-1)$行旋转变换，然后被发送给第 1 号处理器，这段计算时间为$(m-1)*2n*6$，通信时间为 $2(t_s+nt_w)$；接着第 0 行数据经过中间$(p-2)$个处理器的旋转变换被发送给第 p-1 号处理器，这段计算时间为 $(p-2)m*2n*6$，通信时间为 $4(p-2)(t_s+nt_w)$，所以 $\Delta t=12n(mp-m-1)+(4p-6)(t_s+nt_w)$，并行计算时间为 $T_p=12n^2-18mn-6nm^2+12n^2m-12n+(2n-2m+4p-6)(t_s+nt_w)$。

MPI 源程序请参见所附光盘。

1.7 奇异值分解

$A=[a_{ij}]$ 为一个 $m\times n$ 阶矩阵，若存在各列两两正交的 $m\times n$ 的矩阵 U、n 阶正交方阵 V 与主对角元素全为非负的对角阵 $D=diag\ (d_0, d_1,\cdots, d_{(n-1)})$使 $A=UDV^T$，则称 $d_0, d_1,\cdots, d_{(n-1)}$为 A 的奇异值。将 A 写成上式右端的形式，称为对 A 进行**奇异值分解**(Singular Value

Decomposition)。其中 U 的各列向量 $u_0, u_1, \cdots, u_{(n-1)}$ 为 A 的左奇异向量，V 的各列向量 $v_0, v_1, \cdots, v_{(n-1)}$ 为 A 的右奇异向量。

1.7.1 矩阵奇异值分解的串行算法

这里我们将介绍对矩阵进行奇异值分解的 Henstenes 方法。它的基本思想是产生一个正交方阵 V，使 $AV=Q$，Q 的各列之间也两两正交，若将 Q 的各列标准化，即使各列向量 q_i 的长度 $(q_i,q_i)^{1/2}$ 等于 1，这里 (s,t) 表示向量 s, t 的内积。这样，得到：

$$Q = UD$$

此处，U 满足：$U^TU = I$，D 为对角阵：$D=diag\,(d_0, d_1, \cdots, d_{(n-1)})$，非负数 d_i 为 Q 第 i 列的长度。

由 $Q=UD$ 可得到：$A= QV^T =UDV^T$，即 A 的奇异值分解。

Henstenes 方法应用逐次平面旋转来求 Q 与 V。设 $A=A_0$，逐次选择旋转方阵 R_k，使 A_k 的某两列正交化，反复进行旋转变换：

$$A_{k+1} = A_kR_k \quad (k=0,1,\cdots)$$

可得矩阵序列 $\{A_k\}$，适当选择列的正交化顺序，A_k 的各列将趋向于两两正交，即 A_k 趋向于 Q，而 $V=R_0,R_1,\cdots, R_s$，这里 s 为旋转的总次数。

设 R_k 使 A_k 的第 p,q 两列正交，$(p<q)$，且 $R_k=[r_{ij}]$，取：

$$r_{pp} = r_{qq} = cos\theta \qquad r_{pq} = - r_{qp} = sin\theta$$

$$r_{ii} = 1\ (i \neq p, i \neq q) \qquad r_{ij} = 0\ (i \neq p, q, j \neq p,q, i \neq j\ )$$

显然 R_k 为正交方阵，我们称 R_k 为 Givens 旋转矩阵，这样的变换为 **Givens 旋转变换**（Givens Rotation）。A_{k+1} 仅在第 p,q 两列上与 A_k 不同：

$$\begin{cases} a_p^{(k+1)} = a_p^{(k)} \cos q - a_q^{(k)} \sin q \\ a_q^{(k+1)} = a_p^{(k)} \sin q + a_q^{(k)} \cos q \end{cases}$$

即

$$(a_p^{(k+1)}, a_q^{(k+1)}) = (a_p^{(k)}, a_q^{(k)}) \begin{bmatrix} +\cos q & \sin q \\ -\sin q & \cos q \end{bmatrix}$$

为了使 A_{k+1} 的 p, q 列能够正交，旋转角度 θ 必须按如下公式选取：假设 $e = (a_p^{(k)}, a_q^{(k)}), d = (a_p^{(k)}, a_p^{(k)}), b = (a_q^{(k)}, a_q^{(k)})$，若 e=0 则说明此两列已经正交，取 θ=0；

否则记 $l = \dfrac{b-d}{2e}, t = \dfrac{sign(l)}{|l| + \sqrt{1+l^2}}, \cos q = \dfrac{1}{\sqrt{1+t^2}}, \sin q = t\cos q$，由此可求得 θ，满足 $|q| \le \dfrac{p}{4}$。

被正交化的两列 $a_p^{(k)}$ 及 $a_q^{(k)}$ 为一个列对，简记为 $[p,q]$，用上述旋转方法对所有列对 $[p,q]$($p<q$)按照 i=0,1,⋯,n-1, j=i+1,⋯,n-1 的顺序正交化一次，称为一轮，一轮计算以后，再做第二轮计算直至 A 的各列两两正交为止。实际计算中，可给定一个足够小的正数 ε，当所有列对的内积 e 都满足 $|e|$<ε 时，计算即可结束。奇异值分解串行算法如下：

算法 18.13　矩阵奇异值分解串行算法

输入：矩阵 A 和矩阵 E

输出：矩阵 U, 矩阵 D 和矩阵 V^T

Begin

(1)**while** (p>ε)

p=0

for i=1 **to** n **do**

for j=i+1 **to** n **do**

(1.1)*sum*[0]=0 , *sum*[1]=0 , *sum*[2]=0
(1.2)**for** *k*=1 **to** *m* **do**
sum[0]= *sum*[0]+*a*[*k,i*]**a*[*k, j*]
sum[1]= *sum*[1]+ *a*[*k,i*]* *a*[*k,i*]
sum[2]= *sum*[2]+ *a*[*k, j*]* *a*[*k, j*]
end for
(1.3)**if** (| *sum*[0] | >ε) **then**
(i)*aa*=2**sum*[0]
bb=*sum*[1]-*sum*[2]
rr=sqrt(*aa***aa*+*bb***bb*)
(ii)**if** (*bb* ≥ 0) **then**
c=sqrt((*bb*+*rr*)/(2**rr*))
s=*aa*/(2**rr***c*)
else
s=sqrt((*rr*-*bb*)/(2**rr*))
c=*aa*/(2**rr***s*)
end if
(iii)**for** *k*=1 **to** *m* **do**
temp[*k*]=*c***a*[*k,i*]+*s***a*[*k,j*]
a[*k,j*] =(-*s*)**a*[*k,i*]+*c***a*[*k,j*]
end for
(iv)**for** *k*=1 **to** *n* **do**
*temp*1[*k*]= *c***e*[*k,i*]+*s***e*[*k,j*]
e[*k,j*]=(-*s*)**e*[*k,i*]+*c***e*[*k,j*]
end for
(v) **for** *k*=1 **to** *m* **do**
a[*k,i*]= *temp*[*k*]
end for
(vi)**for** *k*=1 **to** *n* **do**
e[*k,i*]= *temp*1[*k*]
end for
(vii)**if** (| *sum*[0] | >*p*) **then** *p*= | *sum*[0] | **end if**
end if
end for
end for
end while
(2)**for** *i*=1 **to** *n* **do**
(2.1)*sum*=0
(2.2)**for** *j*=1 **to** *m* **do**
sum=*sum*+*a*[*j,i*]* *a*[*j,i*]
end for
(2.3) *D*[*i,i*]=sqrt[*sum*]
end for
(3)**for** *j*=1 **to** *n* **do**

for i=1 **to** m **do**

$U[i,j]=A[i,j]/D[j,j]$

end for

end for

(4)V^T=MATRIX_TRANSPOSITION(E)　/*对 E 实施算法 18.1 的矩阵转置*/

End

上述算法的一轮迭代需进行 $n(n\text{-}1)/2$ 次旋转变换，若取一次乘法或一次加法运算时间为一个单位时间，则一次旋转变换要做 3 次内积运算而消耗 6m 个单位时间；与此同时，两列元素进行正交计算还需要 6m+6n 个单位时间，所以奇异值分解的一轮计算时间复杂度为 $n(n\text{-}1)/2*(12m+6n)= n(n\text{-}1)\ (6m+3n)=O(n^2m)$。

1.7.2 矩阵奇异值分解的并行算法

在并行计算矩阵奇异值分解时，对 $m\times n$ 的矩阵 A 按行划分为 p 块(p 为处理器数)，每块含有连续的 q 行向量，这里 $q=\lceil m/p \rceil$，第 i 块包含 A 的第 $i\times q$, …,$(i+1)\times q$-1 行向量，其数据元素被分配到第 i 号处理器上(i=0,1,…,p-1)。E 矩阵取阶数为 n 的单位矩阵 I，按同样方式进行数据划分，记 $z=\lceil n/p \rceil$，每块含有连续的 z 行向量。对矩阵 A 的每一个列向量而言，它被分成 p 个长度为 q 的子向量，分布于 p 个处理器之中，它们协同地对各列向量做正交计算。在对第 i 列与第 j 列进行正交计算时，各个处理器首先求其局部存储器中的 q 维子向量[i,j]、[i,i]、[j,j]的内积，然后通过归约操作的求和运算求得整个 m 维向量对[i,j]、[i,i]、[j,j]的内积并广播给所有处理器，最后各处理器利用这些内积对其局部存储器中的第 i 列及第 j 列 q 维子向量的元素做正交计算。下述算法 18.14 按这样的方式对所有列对正交化一次以完成一轮运算，重复进行若干轮运算，直到迭代收敛为止。具体算法框架描述如下：

算法 18.14　矩阵奇异值分解并行算法

输入：矩阵 A 和矩阵 E

输出：矩阵 U, 矩阵 D 和矩阵 V^{T}

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

while (*not convergence*) **do**

(1)k=0

(2)**for** i=0 **to** n-1 **do**

for j=i+1 **to** n-1 **do**

(2.1)sum[0]=0,　sum[1]=0,　sum[2]=0

(2.1)计算本处理器所存的列子向量 a[*,i]、a[*,j]的内积赋给 ss[0]

(2.3)计算本处理器所存的列子向量 a[*,i]、a[*,j]的内积赋给 ss[1]

(2.4)计算本处理器所存的列子向量 a[*,i]、a[*,j]的内积赋给 ss[2]

(2.5)通过规约操作实现:

(i)计算所有处理器中 ss[0]的和赋给 sum[0]

(ii)计算所有处理器中 ss[1]的和赋给 sum[1]

(iii)计算所有处理器中 ss[2]的和赋给 sum[2]

(iv)将 sum [0]、sum [1]、sum [2] 的值广播到所有处理器中

(2.6)**if** ($|sum(0)|>\varepsilon$) **then** /*各个处理器并行地对 i 和 j 列的正交化*/

(i)aa=2*sum[0]

(ii)bb=sum[1]-sum[2]

(iii)*rr*=sqrt(*aa***aa*+*bb***bb*)
(iv)**if** (*bb*>=0) **then**
c=sqrt((*bb*+*rr*)/(*2***rr*))
s=*aa*/(2**rr***c*)
else
s=sqrt((*rr*-*bb*)/(*2***rr*))
c=*aa*/(*2***rr***s*)
end if
(v)**for** *v*=0 **to** *q*-1 **do**
temp[*v*]=*c***a*[*v*, *i*]+*s***a*[*v*, *j*]
a[*v*, *j*] =(-*s*)**a*[*v*, *i*]+*c***a*[*v*, *j*]
end for
(vi)**for** *v*=0 **to** *z*-1 **do**
*temp*1[*v*]=*c***e*[*v*, *i*]+*s***e*[*v*, *j*]
e[*v*,*j*] =(-*s*)**e*[*v*, *i*]+*c***e*[*v*, *j*]
end for
(vii)**for** *v*=0 **to** *q*-1 **do**
a[*v*,*i*]=*temp*[*v*]
end for
(viii)**for** *v*=0 **to** *z*-1 **do**
e[*v*,*i*]=*temp*1[*v*]
end for
(ix) *k*=*k*+1
end if
end for
end for
end while
End

计算完成后，编号为 0 的处理器收集各处理器中的计算结果，并进一步计算出矩阵 *U*，*D* 和 V^T。算法 18.14 一轮迭代要进行 *n*(*n*-1)/2 次旋转变换,一次旋转变换需要做 3 次内积运算，若取一次乘法或一次加法运算时间为一个单位时间，则需要 6*q* 个单位时间，另外，还要对四列子向量中元素进行正交计算花费 6*q*+6*z* 个单位时间，所以一轮迭代需要的计算时间为 *n*(*n*-1)(6*q*+3*z*)；内积计算需要通信，一轮迭代共做归约操作 *n*(*n*-1)/2 次，每次通信量为 3，因而通信时间为 $[2t_s(\sqrt{p}-1)+3t_w(p-1)]*n(n-1)/2$。由此得出一轮迭代的并行计算时间为 $T_p=[2t_s(\sqrt{p}-1)+3t_w(p-1)]*n(n-1)/2+n(n-1)(6q+3z)$。

MPI 源程序请参见所附光盘。

1.8 Cholesky 分解

对于 n 阶方阵 *A*=[a_{ij}]，若满足 *A*=A^T，则称**方阵** *A* 为**对称矩阵**(Symmetric Matrix)。对非奇异对称方阵的 *LU* 分解有其特殊性，由线性代数知识可知：对于一个对称方阵 *A*，存在一

个下三角方阵 L，使得 $A=LL^T$。由 A 求得 L 的过程，称为对 A 的 Cholesky 分解。

1.8.1 矩阵 Cholesky 分解的串行算法

在对矩阵 A 进行 Cholesky 分解时,记 $L=[l_{ij}]$，当 $i<j$ 时，$l_{ij}=0$，当 $i>j$ 时，l_{ij} 可由下列递推式求得：

$$a_{ij}^{(1)}=a_{ij}$$
$$a_{ij}^{(k+1)}= a_{ij}^{(k)}+l_{ik}(-l_{jk})$$
$$l_{ij}=a_{ij}^{(j)}/l_{jj}$$

当 $i=j$ 时，l_{ii} 则可由下列递推式求得：

$$a_{ii}^{(1)}=a_{ii}$$
$$a_{ii}^{(k+1)}= a_{ii}^{(k)}-l_{ik}^2$$
$$l_{ii}=\sqrt{a_{ii}^{(i)}}$$

在串行计算时，可按 L 的行顺序逐行计算其各个元素，如果取一次乘法和加法运算时间或一次除法运算时间为一个单位时间，则下述算法 18.15 的计算时间为 $\sum_{i=1}^{n} i(i+1)/2=\frac{n^3+3n^2+2n}{6}=\mathrm{O}(n^3)$。

算法 18.15　单处理器上矩阵的 Cholesky 分解的串行算法

输入：矩阵 $A_{n\times n}$

输出：下三角矩阵 $L_{n\times n}$

```
Begin
  (1)for k=1 to n do
      (1.1)a[k, k]=sqrt(a[k, k])
      (1.2)for i=k+1 to n do
              (i)a[i, k]=a[i, k]/a[k, k]
              (ii)for j=k+1 to i do
                    a[i, j]=a[i, j]-a[i, k]*a[j, k]
                  end for
           end for
    end for
  (2)for i= 1 to n do
      for j= 1 to n do
          if (j ≤ i) then l[i,j]= a[i, j] end if
      end for
    end for
End
```

1.8.2 矩阵 Cholesky 分解的并行算法

设 $A=[a_{ij}]$，$G=[g_{ij}]$均为 $n\times n$ 阶矩阵，A 为实对称正定矩阵，G 为上三角矩阵且 $A=G^{\mathrm{T}}G$，

设 $A=\begin{pmatrix} a_{11} & \mathrm{M} & a \\ \Lambda & \Lambda & \Lambda \\ a^T & \mathrm{M} & A_1 \end{pmatrix}=\begin{pmatrix} g_{11} & \mathrm{M} & 0 \\ \Lambda & \Lambda & \Lambda \\ j^T & \mathrm{M} & G_1^T \end{pmatrix}\begin{pmatrix} g_{11} & \mathrm{M} & j \\ \Lambda & \Lambda & \Lambda \\ 0 & \mathrm{M} & G_1 \end{pmatrix}=G^TG$ 则有：

$$\begin{cases} a_{11} = g_{11}^2 \\ \alpha = g_{11} j \\ A_1 = j^T j + G^T G \end{cases} \quad \text{即} \quad \begin{cases} g_{11} = \sqrt{a_{11}} \\ j = \dfrac{\alpha}{g_{11}} = \dfrac{\alpha}{\sqrt{a_{11}}} \\ G^T G = A_1 - \dfrac{\alpha^T \alpha}{g_{11}^2} = A_1 - \dfrac{\alpha^T \alpha}{a_{11}} \end{cases}$$

令 $G' = G, A' = A_1 - \dfrac{a^T a}{a_{11}}$ 即 $A' = G'^T G'$。A' 和 G' 的阶数均减小了 1，这样进行 n-1 次则可以使问题化简到 1 阶的情况。又注意到 A、G 都是对称矩阵所以可以考虑节省空间的压缩存储法将 A 的第 k 次操作矩阵记作 $A^{(k)} = (a_{ij}^{(k)})_{n\times n}$ 则 Cholersky 分解的变换公式推导如下：

$$A^{(1)} = A, A^{(k)} = \begin{pmatrix} a_{11}^{(k)} & \Lambda & a_{1k}^{(k)} & \Lambda & a_{1n}^{(k)} \\ \mathrm{M} & \mathrm{O} & \mathrm{M} & \mathrm{O} & \mathrm{M} \\ a_{k1}^{(k)} & \Lambda & a_{kk}^{(k)} & \Lambda & a_{kn}^{(k)} \\ \mathrm{M} & \mathrm{O} & \mathrm{M} & \mathrm{O} & \mathrm{M} \\ a_{n1}^{(k)} & \Lambda & a_{n2}^{(k)} & \Lambda & a_{nn}^{(k)} \end{pmatrix}, a_{ij}^{(k+1)} = \begin{cases} a_{ij}^{(k)}, i<k \text{或} j<k \\ a_{ij}^{(k)} - \dfrac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}}, i>k \text{且} j>k \\ \dfrac{a_{ij}^{(k)}}{\sqrt{a_{kk}^{(k)}}}, otherwise \end{cases}$$

算法 18.16　矩阵 Cholesky 分解的并行算法

输入：矩阵 $A_{n\times n}$,

输出：对应的上对角阵

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

while (k<n) **do**

(1)将 a_{ij}(i=k,j=k,…, n)分配给 p 个处理器，让每个处理器计算 $a_{ij}' = \dfrac{a_{ij}}{\sqrt{a_{kk}}}$

(2)聚集相应数据，且将第 k 列对应于第 k 行更新，将它们广播到各处理器

(3)将 a_{ij}(i=k+1,…, n, j=i+1,…, n)分配给 p 个处理器同时计算 $a_{ij}' = a_{ij} - a_{ik}' a_{kj}'$

(4)聚集(3)中各处理器的数据并广播到各处理器中

(5)k=k+1

(6) $a_{ij} = a_{ij}'(i, j = 1, \Lambda, n)$

end while

End

很显然，本算法中开平方的次数可减少到 n，而乘除次数为 $\sum_{k=1}^{n}[\binom{2}{n+2k}-1] = \binom{3}{n+2} - n$，所以时间复杂度为 $O\left(\dfrac{n^3}{p}\right)$。

MPI 源程序请参见所附光盘。

1.9 方阵求逆

矩阵求逆(Matrix Inversion)是一常用的矩阵运算。对于一个 $n\times n$ 阶的非奇异方阵 $A=[a_{ij}]$，其逆矩阵是指满足 $A^{-1}A=AA^{-1}=I$ 的 $n\times n$ 阶方阵，其中 I 为单位方阵。

1.9.1 求方阵的逆的串行算法

这里，我们不妨记 $A^{(0)}=A$，用 Gauss-Jordan 消去法求 A^{-1} 的算法是由 $A^{(0)}$出发通过变换得到方阵序列$\{A^{(k)}\}$，$A^{(k)} = [a_{ij}^{(k)}]$，(k=0,1,…,n)，每次由 $A^{(k-1)}$到 $A^{(k)}$的变换执行下述的计算：

$$a_{kk}^{(k)} = 1/a_{kk}^{(k-1)}$$

对于 k 行的其它诸元素：$a_{kj}^{(k)} = a_{kk}^{(k)} * a_{kj}^{(k-1)} \quad j=1,2,\cdots,n; j\neq k$

除 k 行、k 列外的其它元素：$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} * a_{kj}^{(k)} \quad (1\le i,j\le n, i\neq k, j\neq k)$

对于 k 列的其它诸元素：$a_{ik}^{(k)} = -a_{ik}^{(k-1)} * a_{kk}^{(k)} \quad i=1,2,\cdots,n; i\neq k$

这样计算得到的 $A^{(n)}$就是 A^{-1}，矩阵求逆串行算法如算法 18.17 所示。假定一次乘法和加法运算时间或一次除法运算时间为一个单位时间，则下述矩阵求逆算法 18.17 的时间复杂度为 O(n^3)。

算法 18.17　单处理器上的矩阵求逆算法

输入：矩阵 $\boldsymbol{A_{n\times n}}$

输出：矩阵 $\boldsymbol{A^{-1}}_{n\times n}$

```
Begin
   for i=1 to n do
      (1) a[i,i]=1/a[i,i]
      (2)for j=1 to n do
            if (j≠i) then a[i,j]=a[i,j]*a[i,i] end if
         end for
      (3)for k=1 to n do
            for j=1 to n do
               if ((k≠i and  j≠i)) then a[k,j]=a[k,j]-a[k,i]*a[i,j]  end if
            end for
         end for
      (4)for k=1 to n do
            if (k≠i) then a[k,i]= -a[k,i]*a[i,i]  end if
         end for
   end for
End
```

1.9.2 方阵求逆的并行算法

矩阵求逆的过程中，依次利用主行 $i(i=0,1,\cdots,n-1)$对其余各行 $j(j\neq i)$作初等行变换，由于各行计算之间没有数据相关关系，因此我们对矩阵 A 按行划分来实现并行计算。考虑到在计算过程中处理器之间的负载均衡，对 A 采用行交叉划分：设处理器个数为 p，矩阵 A 的阶数为 n，$m=\lceil n/p \rceil$，对矩阵 A 行交叉划分后，编号为 $i(i=0,1,\cdots,p-1)$的处理器存有 A 的第 $i, i+p,\cdots, i+(m-1)p$ 行。在计算中，依次将第 $0,1,\cdots,n-1$ 行作为主行，将其广播给所有处理器，这实际上是各处理器轮流选出主行并广播。发送主行数据的处理器利用主行对其主行之外的 $m-1$ 行行向量做行变换，其余处理器则利用主行对其 m 行行向量做行变换。具体算法框架描述如下：

算法 18.18　矩阵求逆的并行算法

输入：矩阵 $\boldsymbol{A_{n\times n}}$,

输出：矩阵 $\boldsymbol{A^{-1}}_{n\times n}$

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

for i=0 **to** m-1 **do**

for j=0 **to** p-1 **do**

(1)**if** (my_rank=j) **then**　/* 主元素在本处理器中*/

(1.1) $v=i*p+j$　/* v 为主元素的行号*/

$a[i,v]=1/a[i,v]$

(1.2)**for** k=0 **to** n-1 **do**

if ($k\neq v$) **then** $a[i,k]=a[i,k]*a[i,v]$ **end if**

end for

(1.3)**for** k= 0 **to** n-1 **do**

$f[k]=a[i,k]$

end for

(1.4)将变换后的主行元素(存于数组 f 中)广播到所有处理器中

else　/* 主元素不在本处理器中*/

(1.5)　$v=i*p+j$　/* v 为主元素的行号*/

(1.6) 接收广播来的主行元素存于数组 f 中

end if

(2)**if** (my_rank≠j) **then**　/* 主元素不在本处理器中*/

(2.1)**for** k= 0 **to** m-1 **do**　/*处理非主行、非主列元素*/

for w= 0 **to** n-1 **do**

if ($w\neq v$) **then** $a[k,w]=a[k,w]-f[w]*a[k,v]$ **end if**

end for

end for

(2.2)**for** k= 0 **to** m-1 **do**　/*处理主列元素*/

$a[k,v]=-f[v]*a[k,v]$

end for

else　/*处理主行所在的处理器中的其它元素*/

(2.3)**for** k= 0 **to** m-1 **do**

if ($k\neq i$) **then**

for w= 0 **to** n-1 **do**

if ($w\neq v$) **then** $a[k,w]=a[k,w]-f[w]*a[k,v]$ **end if**

```
                    end for
                end if
            end for
        (2.4)for k= 0 to m-1 do
                if ( k ≠ i) then a[k,v]=-f[v]*a[k,v] end if
            end for
        end if
    end for
end for
End
```

若一次乘法和加法运算或一次除法运算时间为一个单位时间，则算法 18.18 所需的计算时间为 mn^2；又由于共有 n 行数据依次作为主行被广播，其通信时间为 $n(t_s+nt_w)\log p$，所以该算法并行计算时间为 $T_p=mn^2+n(t_s+nt_w)\log p$。

MPI 源程序请参见章末附录。

1.10 小结

本章讨论了矩阵转置、矩阵向量乘、矩阵乘法、矩阵的分解及其它一些有关矩阵的数值计算问题，这方面的更详尽的讲解可参见[1]、[2]。关于并行与分布式数值算法，[3]是一本很好的参考书。有关矩阵运算，[4]是一本比较经典的教本。本章所讨论的 Cannon 乘法原始论文来源于[5]，习题中的 Fox 乘法来源于[6]，这些算法的改进版本可参见[7]和[8]。

参考文献

[1]. 陈国良 编著．并行计算——结构·算法·编程．高等教育出版社,1999.10

[2]. 陈国良 编著．并行算法的设计与分析（修订版）．高等教育出版社，2002.11

[3]. Bertsekas D P and Tsitsilklis J N.Parallel and Distributed Computation:Numerical Methods. Prentice-Hall,1989

[4]. Golub G H,Loan C V.Matrix Computations.(2nd Ed).The Johns Hopkings Univ.Press,1989

[5]. Cannon L E.A Cellular Computer to Implement the Kalman Filter Algorithm: Ph.D.thesis. Montana State Univ.,1969

[6]. Fox G C,Otto S W,Hey A J G.Matrix Algorithms on Hypercube I:Matrix Multiplication.Parallel Computing,1987,4:17~31

[7]. Berntsen J.Communication Effcient Matrix Multiplication on Hypercubes.Parallel Computing,1989,12:335~342

[8]. Ho C T,Johnsson S L,Edelman A.Matrix Multiplication on Hypercubes using Full Bandwidth and Constant Storage.Proc.Int'1 91 Conference on Parallel Processing,1997,447~451

附录一. Cannon 乘法并行算法的 MPI 源程序

1. **源程序** cannon.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#define MAX_PROCESSOR_NUM 12
void MatrixMultiply(int n, double *a, double *b,
double *c);

main (int argc, char *argv[])
{
        int i,j,k,m,p;
        int n, nlocal;
        double *a, *b, *c;
        int npes, dims[2], periods[2];
        int myrank, my2drank, mycoords[2];
        int shiftsource, shiftdest, rightrank;
        int leftrank, downrank, uprank;
        MPI_Status status;
        MPI_Comm comm_2d;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD,
                &npes);
        MPI_Comm_rank(MPI_COMM_WORLD,
                &myrank);

        if (argc !=2)
        {
                if (myrank ==0)
                printf("Usage: %s <the dimension of
                        thematrix>\n", argv[0]);
                MPI_Finalize();
                exit(0);
        }

        n=atoi(argv[1]);
        dims[0] = sqrt(npes);
        dims[1] = npes/dims[0];

        if (dims[0] != dims[1])
        {
                if (myrank ==0 )
                printf("The number of processes must be a
                        perfect square.\n");
                MPI_Finalize();
                exit(0);
        }

        periods[0]=periods[1]=1;
        MPI_Cart_create(MPI_COMM_WORLD,2,
                dims, periods, 0, &comm_2d);
        MPI_Comm_rank(comm_2d, &my2drank);
        MPI_Cart_coords(comm_2d,my2drank,2,
                mycoords);
        nlocal = n/dims[0];
        a=(double*)malloc(nlocal*nlocal*
                sizeof (double));
        b= (double *)malloc(nlocal*nlocal*
                sizeof (double));
        c= (double *)malloc(nlocal*nlocal*
                sizeof (double));
        srand48((long)myrank);

        for ( i=0; i<nlocal*nlocal; i++)
        {
                a[i] = b[i] =(double)i;
                c[i] = 0.0;
        }

        MPI_Cart_shift(comm_2d, 0, -mycoords[0],
                &shiftsource, &shiftdest);
        MPI_Sendrecv_replace(a,nlocal*nlocal,
                MPI_DOUBLE, shiftdest,1,shiftsource, 1,
                comm_2d, &status);
        MPI_Cart_shift(comm_2d, 1, -mycoords[1],
                &shiftsource, &shiftdest);
        MPI_Sendrecv_replace(b, nlocal*nlocal,
                MPI_DOUBLE,shiftdest, 1, shiftsource, 1,
                comm_2d, &status);
        MPI_Cart_shift(comm_2d, 0,-1, &rightrank,
                &leftrank);
        MPI_Cart_shift(comm_2d, 1, -1, &downrank,
                &uprank);

        for (i=0; i<dims[0]; i++)
        {
                MatrixMultiply(nlocal, a, b, c);
```

```
        MPI_Sendrecv_replace(a, nlocal*nlocal,
                MPI_DOUBLE,leftrank,1,rightrank,
                1, comm_2d, &status);
        MPI_Sendrecv_replace(b, nlocal*nlocal,
                MPI_DOUBLE,uprank,1, downrank,
                1, comm_2d, &status);
}

MPI_Cart_shift(comm_2d, 0, +mycoords[0],
        &shiftsource, &shiftdest);
MPI_Sendrecv_replace(a, nlocal*nlocal,
        MPI_DOUBLE, shiftdest, 1, shiftsource,
        1, comm_2d, &status);
MPI_Cart_shift(comm_2d, 1, +mycoords[1],
        &shiftsource, &shiftdest);
MPI_Sendrecv_replace(b, nlocal*nlocal,
        MPI_DOUBLE, shiftdest, 1, shiftsource,
        1,comm_2d, &status);
MPI_Comm_free(&comm_2d);

if (myrank ==0)
{
        puts("Random Matrix A");
        for(i = 0; i < nlocal; i ++)
        {
                for(j = 0; j < nlocal; j ++)
                        printf("%9.7f ", a[i*nlocal+j]);
                printf("\n");
        }
        puts("Random Matrix B");
        for(i = 0; i < nlocal; i ++)
        {
                for(j = 0; j < nlocal; j ++)
                        printf("%9.7f ", b[i*nlocal+j]);
                printf("\n");
        }
        puts("Matrix C = A*B");
        for(i = 0; i < nlocal; i ++)
        {
                for(j = 0; j < nlocal; j ++)
                        printf("%9.7f ", c[i*nlocal+j]);
                printf("\n");
        }
}
```

```
        free(a);
        free(b);
        free(c);

        MPI_Finalize();

}


void MatrixMultiply(int n, double *a, double *b,
double *c)
{
        int i, j, k;
        for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                        for (k = 0; k < n; k++)
                                c[i*n+j]+=a[i*n+k]*b[k*n+j];

}
```

2. 运行实例

编译：mpicc cannon.cc –o cannon

运行：可以使用命令 mpirun –np ProcessSize cannon ArraySize 来运行该 Cannon 乘法程序，其中 ProcessSize 是所使用的处理器个数,ArraySize 是矩阵的维数,这里分别取为 3 和 4。本实例中使用了

mpirun –np 3 cannon 4

运行结果：

```
Random Matrix A
0.1708280    0.7499020
0.0963717    0.8704652
Random Matrix B
0.1708280    0.7499020
0.0963717    0.8704652
Matrix C = A*B
0.3999800    1.1517694
1.0546739    1.2320793
```

说明：该运行实例中，A 为 2×2 的矩阵，B 为 2×2 的矩阵，两者相乘的结果存放于矩阵 C 中输出。

附录二. 矩阵 LU 分解并行算法的 MPI 源程序

1. 源程序 ludep.c

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#define a(x,y) a(x*M+y)
/*A 为 M*M 矩阵*/
#define A(x,y) A(x*M+y)
#define l(x,y) l(x*M+y)
#define u(x,y) u(x*M+y)
#define floatsize sizeof(float)
#define intsize sizeof(int)

int M,N;
int m;
float *A;
int my_rank;
int p;
MPI_Status status;

void fatal(char *message)
{
    printf("%s\n",message);
    exit(1);
}

void Environment_Finalize(float *a,float *f)
{
    free(a);
    free(f);
}
int main(int argc, char **argv)
{
    int i,j,k,my_rank,group_size;
    int i1,i2;
    int v,w;
    float *a,*f,*l,*u;
    FILE *fdA;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,
        &group_size);
```

```
MPI_Comm_rank(MPI_COMM_WORLD,
        &my_rank);

p=group_size;

if (my_rank==0)
{
        fdA=fopen("dataIn.txt","r");
        fscanf(fdA,"%d %d", &M, &N);
        if(M != N)
        {
                puts("The input is error!");
                exit(0);
        }
        A=(float *)malloc(floatsize*M*M);
        for(i = 0; i < M; i ++)
              for(j = 0; j < M; j ++)
                      fscanf(fdA, "%f", A+i*M+j);
        fclose(fdA);
}

/*0 号处理器将 M 广播给所有处理器*/
MPI_Bcast(&M,1,MPI_INT,0,
        MPI_COMM_WORLD);
m=M/p;
if (M%p!=0) m++;
/*分配至各处理器的子矩阵大小为 m*M*/
a=(float*)malloc(floatsize*m*M);
/*各处理器为主行元素建立发送和接收缓冲
        区*/
f=(float*)malloc(floatsize*M);

/*0 号处理器为 l 和 u 矩阵分配内存，以分离
        出经过变换后的 A 矩阵中的 l 和 u 矩阵
        */
if (my_rank==0)
{
        l=(float*)malloc(floatsize*M*M);
        u=(float*)malloc(floatsize*M*M);
}

/*0 号处理器采用行交叉划分将矩阵 A 划分为
        大小 m*M 的 p 块子矩阵，依次发送给 1
        至 p-1 号处理器*/
if (a==NULL) fatal("allocate error\n");

if (my_rank==0)
{
        for(i=0;i<m;i++)
                for(j=0;j<M;j++)
                        a(i,j)=A((i*p),j);
        for(i=0;i<M;i++)
        if ((i%p)!=0)
        {
                i1=i%p;
                i2=i/p+1;
                MPI_Send(&A(i,0),M,MPI_FLOAT,
                        i1,i2,MPI_COMM_WORLD);
        }
}
else
{
        for(i=0;i<m;i++)
                MPI_Recv(&a(i,0),M,MPI_FLOAT,
                        0,i+1,MPI_COMM_WORLD,
                        &status);
}

for(i=0;i<m;i++)
        for(j=0;j<p;j++)
        {
                /*j 号处理器负责广播主行元素*/
                if (my_rank==j)
                {
                        v=i*p+j;
                        for (k=v;k<M;k++)
                                f(k)=a(i,k);

                        MPI_Bcast(f,M,MPI_FLOAT,
                                my_rank,
                                MPI_COMM_WORLD)
                                ;
                }
                else
                {
                        v=i*p+j;
                        MPI_Bcast(f,M,MPI_FLOAT,
                                j,
```

```
                    MPI_COMM_WORLD)
                ;
            }

            /*编号小于 my_rank 的处理器（包
                括 my_rank 本身）利用主行对
                其第 i+1,…,m-1 行数据做行
                变换*/
            if (my_rank<=j)
                for(k=i+1;k<m;k++)
                {
                    a(k,v)=a(k,v)/f(v);
                    for(w=v+1;w<M;w++)
                        a(k,w)=a(k,w)
                        -f(w)*a(k,v);
                }

            /*编号大于 my_rank 的处理器利用
                主行对其第 i,…,m-1 行数据
                做行变换*/
            if (my_rank>j)
                for(k=i;k<m;k++)
                {
                    a(k,v)=a(k,v)/f(v);
                    for(w=v+1;w<M;w++)
                        a(k,w)=a(k,w)-f(w)
                            *a(k,v);
                }
        }

    /*0 号处理器从其余各处理器中接收子矩阵 a，
        得到经过变换的矩阵 A*/
    if (my_rank==0)
    {
        for(i=0;i<m;i++)
            for(j=0;j<M;j++)
                A(i*p,j)=a(i,j);
    }
    if (my_rank!=0)
    {
        for(i=0;i<m;i++)
            MPI_Send(&a(i,0),M,MPI_FLOAT,
                0,i,
                MPI_COMM_WORLD);
    }
    else
    {
        for(i=1;i<p;i++)
            for(j=0;j<m;j++)
            {
                MPI_Recv(&a(j,0),M,
                    MPI_FLOAT,i,j,
                    MPI_COMM_WORLD,
                    &status);
                for(k=0;k<M;k++)
                    A((j*p+i),k)=a(j,k);
            }
    }

    if (my_rank==0)
    {
        for(i=0;i<M;i++)
            for(j=0;j<M;j++)
                u(i,j)=0.0;
        for(i=0;i<M;i++)
            for(j=0;j<M;j++)
                if (i==j)
                    l(i,j)=1.0;
                else
                    l(i,j)=0.0;
         for(i=0;i<M;i++)
            for(j=0;j<M;j++)
                if (i>j)
                    l(i,j)=A(i,j);
                else
                    u(i,j)=A(i,j);
        printf("Input of file \"dataIn.txt\"\n");
        printf("%d\t %d\n",M, N);
        for(i=0;i<M;i++)
        {
            for(j=0;j<N;j++)
                printf("%f\t",A(i,j));
            printf("\n");
        }
        printf("\nOutput of LU operation\n");
        printf("Matrix L:\n");
        for(i=0;i<M;i++)
        {
```

```
            for(j=0;j<M;j++)
                printf("%f\t",l(i,j));
                printf("\n");
        }
        printf("Matrix U:\n");
        for(i=0;i<M;i++)
        {
            for(j=0;j<M;j++)
                printf("%f\t",u(i,j));
            printf("\n");
        }
    }
    MPI_Finalize();
    Environment_Finalize(a,f);
    return(0);
}
```

2. 运行实例

编译：mpicc ludep.cc –o ludep

运行：可以使用命令 mpirun –np ProcessSize ludep 来运行该矩阵 LU 分解程序，其中 ProcessSize 是所使用的处理器个数,这里取为 4。本实例中使用了

mpirun –np 4 ludep

运行结果：

```
Input of file "dataIn.txt"
3       3
2.000000   1.000000   2.000000
0.500000   1.500000   2.000000
2.000000   -0.666667  -0.666667

Output of LU operation
Matrix L:
1.000000   0.000000   0.000000
0.500000   1.000000   0.000000
2.000000   -0.666667  1.000000
Matrix U:
2.000000   1.000000   2.000000
0.000000   1.500000   2.000000
0.000000   0.000000   -0.666667
```

说明：该运行实例中， A 为 3×3 的矩阵，其元素值存放于文档“dataIn.txt”中，LU 分解后得到矩阵 L、U 作为结果输出。

附录三. 方阵求逆并行算法的 MPI 源程序

1. 源程序 invert.c

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#define a(x,y) a[x*M+y]
/*A 为 M*M 矩阵*/
#define A(x,y) A[x*M+y]
#define floatsize sizeof(float)
#define intsize sizeof(int)

int M,N;
float *A;
int my_rank;
int p;
MPI_Status status;
```

```
void fatal(char *message)
{
        printf("%s\n",message);
        exit(1);
}

void Environment_Finalize(float *a,float *f)
{
        free(a);
        free(f);
}
int main(int argc, char **argv)
{
        int i,j,k,my_rank,group_size;
        int i1,i2;
        int v,w;
        int m;
        float *f;
        float *a;
        FILE *fdA;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,
                &group_size);
        MPI_Comm_rank(MPI_COMM_WORLD,
                &my_rank);
        p=group_size;

        if (my_rank==0)
        {
                fdA=fopen("dataIn.txt","r");
                fscanf(fdA,"%d %d", &M, &N);
                if(M != N)
                {
                        puts("The input is error!");
                        exit(0);
                }
                A=(float *)malloc(floatsize*M*M);
                for(i = 0; i < M; i ++)
                {
                        for(j = 0; j < M; j ++)
                                fscanf(fdA, "%f", A+i*M+j);
                }
        fclose(fdA);
        printf("Input of file \"dataIn.txt\"\n");
        printf("%d\t%d\n", M, M);
        for(i=0;i<M;i++)
        {
                for(j=0;j<M;j++)
                        printf("%f\t",A(i,j));

                printf("\n");
        }
}

/*0 号处理器将 M 广播给所有处理器*/
MPI_Bcast(&M,1,MPI_INT,0,
        MPI_COMM_WORLD);
m=M/p;
if (M%p!=0) m++;
/*各处理器为主行元素建立发送和接收缓冲
        区*/
f=(float*)malloc(sizeof(float)*M);
/*分配至各处理器的子矩阵大小为 m*M*/
a=(float*)malloc(sizeof(float)*m*M);
if (a==NULL||f==NULL)
        fatal("allocate error\n");

/*0 号处理器采用行交叉划分将矩阵 A 划分为
        m*M 的 p 块子矩阵，依次发送给 1 至 p-1
        号处理器*/
if (my_rank==0)
{
        for(i=0;i<m;i++)
                for(j=0;j<M;j++)
                        a(i,j)=A(i*p,j);
        for(i=0;i<M;i++)
                if ((i%p)!=0)
                {
                        i1=i%p;
                        i2=i/p+1;
                        MPI_Send(&A(i,0),M,
                                MPI_FLOAT,i1,i2,
                                MPI_COMM_WORLD);
                }
}
else
```

```
{
    for(i=0;i<m;i++)
        MPI_Recv(&a(i,0),M,MPI_FLOAT,
            0,i+1,MPI_COMM_WORLD,
            &status);
}

for(i=0;i<m;i++)
    for(j=0;j<p;j++)
    {
        /*j 号处理器负责广播主行元素*/
        if (my_rank==j)
        {
            v=i*p+j;
            a(i,v)=1/a(i,v);
            for(k=0;k<M;k++)
                if (k!=v)
                    a(i,k)=a(i,k)*a(i,v);
            for(k=0;k<M;k++)
                f[k]=a(i,k);
            MPI_Bcast(f,M,MPI_FLOAT,
                my_rank,
                MPI_COMM_WORLD);
        }
        else
        {
            v=i*p+j;
            MPI_Bcast(f,M,MPI_FLOAT,j
            , MPI_COMM_WORLD);
        }

        /*其余处理器则利用主行对其 m 行
        行向量做行变换*/
        if (my_rank!=j)
        {
            for(k=0;k<m;k++)
                for(w=0;w<M;w++)
                if (w!=v)
                    a(k,w)=a(k,w)
                        -f[w]*a(k,v);
                for(k=0;k<m;k++)
                    a(k,v)=-f[v]*
                        a(k,v);
        }
        /*发送主行数据的处理器利用主行
            对其主行之外的 m-1 行行向
            量做行变换*/
        if (my_rank==j)
        {
            for(k=0;k<m;k++)
            if (k!=i)
            {
                for(w=0;w<M;w++)
                    if (w!=v)
                        a(k,w)=a(k,w)-
                            f[w]*a(k,v);
            }
            for(k=0;k<m;k++)
                if (k!=i)
                    a(k,v)=-f[v]*
                        a(k,v);
        }
}

/*0 号处理器从其余各处理器中接收子矩阵 a，
    得到经过变换的逆矩阵 A*/
if (my_rank==0)
{
    for(i=0;i<m;i++)
        for(j=0;j<M;j++)
            A(i*p,j)=a(i,j);
}

if (my_rank!=0)
 {
    for(i=0;i<m;i++)
        for(j=0;j<M;j++)
            MPI_Send(&a(i,j),1,
                MPI_FLOAT,0,my_rank,
                MPI_COMM_WORLD)
                ;
 }
else
 {
    for(i=1;i<p;i++)
        for(j=0;j<m;j++)
            for(k=0;k<M;k++)
```

```
            {
                MPI_Recv(&a(j,k),1,
                    MPI_FLOAT,i,i,
                    MPI_COMM_WORLD,
                    &status);
                A((j*p+i),k)=a(j,k);
            }
        }

        if(my_rank==0)
        {
            printf("\nOutput of Matrix
                A'sinversion\n");
            for(i=0;i<M;i++)
            {
                for(j=0;j<M;j++)
                    printf("%f\t",A(i,j));
                printf("\n");
            }
        }

        MPI_Finalize();
        Environment_Finalize(a,f);
        return(0);
        free(A);
}
```

2. 运行实例

编译：mpicc invert.cc –o invert

运行：可以使用命令 mpirun –np ProcessSize invert 来运行该矩阵求逆分解程序，其中 ProcessSize 是所使用的处理器个数,这里取为 3。本实例中使用了

mpirun –np 3 invert

运行结果：

```
Input of file "dataIn.txt"
3  3
1.000000      -1.000000  1.000000
5.000000      -4.000000  3.000000
2.000000      1.000000   1.000000

Output of Matrix A's inversion
-1.400000     0.400000   0.200000
0.200000      -0.200000  0.400000
2.600000      -0.600000  0.200000
```

说明：该运行实例中， A 为 3×3 的矩阵，其元素值存放于文档“dataIn.txt”中，求得的逆矩阵 A^{-1} 作为结果输出。

第十九章 线性方程组的直接解法

在求解**线性方程组**(System of Linear Equations)的算法中，有两类最基本的算法，一类是直接法，即以消去为基础的解法。如果不考虑误差的影响，从理论上讲，它可以在固定步数内求得方程组的准确解。另一类是迭代解法，它是一个逐步求得近似解的过程，这种方法便于编制解题程序，但存在着迭代是否收敛及收敛速度快慢的问题。在迭代过程中，由于极限过程一般不可能进行到底，因此只能得到满足一定精度要求的近似解。本章我们主要介绍几种直接法，迭代法将在下一章讨论。

2.1 高斯消去法解线性方程组

在直接方法中最主要的是**高斯消去法**(Gaussian Elimination)。它分为消元与回代两个过程，消元过程将方程组化为一个等价的三角方程组，而回代过程则是解这个三角方程组。

19.1.1 高斯消去及其串行算法

对于方程组 $\boldsymbol{Ax=b}$，其中 A 为 n 阶非奇异阵，其各阶主子行列式不为零，x，b 为 n 维向量。将向量 b 看成是 A 的最后一列，此时 A 就成了一个 $n\times(n+1)$的方程组的**增广矩阵**(Augmented Matrix)，消去过程实质上是对增广矩阵 A 进行线性变换，使之三角化。高斯消去法按 k=1,2,…,n 的顺序，逐次以第 k 行作为主行进行消去变换，以消去第 k 列的主元素以下的元素 $a_{k+1k}, a_{k+2k}, \Lambda, a_{nk}$。消去过程分为两步，首先计算：

$a_{kj}=a_{kj}/a_{kk}$ ，j=k+1, …,n

$b_k=b_k/a_{kk}$

这一步称为**归一化**(Normalization)。它的作用是将主对角线上的元素变成 1，同时第 k 行上的所有元素与常数向量中的 b_k 都要除以 a_{kk}。由于 A 的各阶主子式非零，可以保证在消去过程中所有主元素 a_{kk} 皆为非零。然后计算：

$a_{ij}=a_{ij}-a_{ik}a_{kj}$ ，$i,j=k+1, \cdots,n$

$b_i=b_i-a_{ik}b_k$ ，$i=k+1, \cdots,n$

这一步称为消元，它的作用是将主对角线 a_{kk} 以下的元素消成 0，其它元素与向量 B 中的元素也应作相应的变换。

在回代过程中，按下式依次解出 $x_n, x_{n-1}, \cdots, x_1$：

① 直接解出 x_n，即 $x_n=b_n/a_{nn}$；

② 进行回代求 $x_i = b_i - \sum_{j=i+1}^{n} a_{ij}x_j, \quad i = n-1, \Lambda, 2,1$

在归−化的过程中，要用 a_{kk} 作除数，当｜a_{kk}｜很小时，会损失精度，并且可能会导致商太大而使计算产生溢出。如果系数 A 虽为非奇异，但不能满足各阶主子式全不为零的条件，就会出现主元素 a_{ii} 为零的情况，导致消去过程无法继续进行。为了避免这种情形，在每次归一化之前，可增加一个选**主元**(Pivot)的过程，将绝对值较大的元素交换到主对角线的位置上。根据选取主元的范围不同，选主元的方法主要有列主元与全主元两种。

列主元(Column Pivot)方法的基本思想是当变换到第 k 步时，从第 k 列的 a_{kk} 以下（包括

a_{kk}）的各元素中选出绝对值最大者。然后通过行交换将它交换到 a_{kk} 的位置上。但列主元不能保证所选的 a_{kk} 是同一行中的绝对值最大者，因此采用了列主元虽然变换过程不会中断，但计算还是不稳定的。

全主元方法的基本思想是当变换到第 k 步时，从右下角（$n-k+1$）阶子阵中选取绝对值最大的元素，然后通过行变换和列变换将它交换到 a_{kk} 的位置上。对系数矩阵做行交换不影响计算结果，但列交换会导致最后结果中未知数的次序混乱。即在交换第 i 列与第 j 列后，最后结果中 x_i 与 x_j 的次序也被交换了。因此在使用全主元的高斯消去法时，必须在选主元的过程中记住所进行的一切列变换，以便在最后结果中恢复。全主元的高斯消去串行算法如下，其中 a_{ij} 我们用 $a[i,j]$来表示：

算法 19.1　单处理器采用全主元高斯消去法的消去过程算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

(1)**for** i=1 **to** n **do**

$shift[i]=i$

end for

(2)**for** k=1 **to** n **do**

(2.1) d=0

(2.2)**for** $i=k$ **to** n **do**

for $j=k$ **to** n **do**

if ($|a[i,j]|>d$) **then**　$d=|a[i,j]|$，　$js=j$，　$l=i$　**end if**

end for

end for

(2.3) **if** ($js\neq$ k) **then**

(i)**for** i=1 **to** n **do**

交换 $a[i,k]$和 $a[i,js]$

end for

(ii)交换 $shift[k]$和 $shift[js]$

end if

(2.4) **if** ($l\neq$ k) **then**

(i)**for**$j=k$ **to** n **do**

交换 $a[k,j]$和 $a[l,j]$

end for

(ii)交换 $b[k]$和 $b[l]$

end if

(2.5) **for** $j=k$+1 **to** n **do**

$a[k,j]= a[k,j]/a[k,k]$

end for

(2.6) $b[k]=b[k]/a[k,k]$，$a[k,k]$=1

(2.7) **for**$i=k$+1 **to** n **do**

(i)**for**$j=k$+1 **to** n **do**

$a[i,j]=a[i,j]- a[i,k]*a[k,j]$

end for

(ii)$b[i]=b[i]-a[i,k]*b[k]$

(iii)$a[i,k]=0$

end for

end for

(3)**for** $i=n$ **downto** 1 **do** /*采用全主元高斯消去法的回代过程*/

for$j=i+1$ **to** n **do**

$b[i]=a[i,j]*x[i]$

end for

end for

(4)**for** $k=1$ **to** n **do**

for $i=1$ **to** n **do**

if ($shift[i]=k$) **then** 输出 $x[k]$的值 $x[i]$ **end if**

end for

end for

End

在全主元高斯消去法中，由于每次选择主元素的数据交换情况无法预计，因此我们不考虑选主元的时间而仅考虑一般高斯消去法的计算时间复杂度。若取一次乘法和加法运算时间或一次除法运算时间为一个单位时间，则消去过程的时间复杂度为 $\sum_{i=1}^{n} i^2$ ，回代过程的时间复杂度为 $\sum_{i=1}^{n} i$，算法 19.1 的时间复杂度为$(n^3+3n^2+2n)/3=O(n^3)$。

19.1.2 并行高斯消去算法

高斯消去法是利用主行 i 对其余各行 j，($j>i$)作初等行变换，各行计算之间没有数据相关关系，因此可以对矩阵 A 按行划分。考虑到在计算过程中处理器之间的负载均衡，对 A 采用行交叉划分。设处理器个数为 p,矩阵 A 的阶数为 n，$m=\lceil n/p \rceil$，对矩阵 A 行交叉划分后，编号为 i(i=0,1,…, p -1)的处理器含有 A 的第 i, $i+p$,…,$i+(m-1)p$ 行和向量 B 的第 i, $i+p$,…,$i+(m-1)p$ 一共 m 个元素。

消去过程的并行是依次以第 0,1,…,n-1 行作为主行进行消去计算，由于对行的交叉划分与分布，这实际上是由各处理器轮流选出主行。在每次消去计算前，各处理器并行求其局部存储器中右下角阶子阵的最大元。若以编号为 my_rank 的处理器的第 i 行作为主行，则编号在 my_rank 后面的处理器（包括 my_rank 本身）求其局部存储器中第 i 行至第 m-1 行元素的最大元，并记录其行号、列号及所在处理器编号；编号在 my_rank 前面的处理器求其局部存储器中第 i+1 行至第 m-1 行元素的最大元，并记录其行号、列号及所在处理器编号。然后通过扩展收集操作将局部存储器中的最大元按处理器编号连接起来并广播给所有处理器，各处理器以此求得整个矩阵右下角阶子阵的最大元 $maxvalue$ 及其所在行号、列号和处理器编号。若 $maxvalue$ 的列号不是原主元素 a_{kk} 的列号，则交换第 k 列与 $maxvalue$ 所在列的两列数据；若 $maxvalue$ 的处理器编号不是原主元素 a_{kk} 的处理器编号，则在处理器间的进行行交换；若 $maxvalue$ 的处理器编号是原主元素 a_{kk} 的处理器编号，但行号不是原主元素 a_{kk} 的行号，则在处理器内部进行行交换。在消去计算中，首先对主行元素作归一化操作 $a_{kj}=a_{kj}/a_{kk}$ ，$b_k=b_k/a_{kk}$，然后将主行广播给所有处理器，各处理器利用接收到的主行元素对其部分行向量做行变换。若以编号为 my_rank 的处理器的第 i 行作为主行，并将它播送给所有的处理器。则编号在 my_rank 前面的处理器（包括 my_rank 本身）利用主行对其第 i+1,…, m-1 行数据和子向量 B 做行变换。编号在 my_rank 后面的处理器利用主行对其第 i,…,m-1 行数据和子向量 B 做行变换。

回代过程的并行是按 $x_n,x_{n-1},\cdots,x_1$ 的顺序由各处理器依次计算 $x(i*p+\text{my\_rank})$，一旦 x($i*p$+my_rank)被计算出来就立即广播给所有处理器，用于与对应项相乘并做求和计算。具体算法框架描述如下：

算法 19.2　全主元高斯消去法过程的并行算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

对所有处理器 my_rank(my_rank=0,⋯,p-1)同时执行如下的算法:

/*消去过程*/

(1)**for** *i*=0 **to** *m*-1 **do**

for *j*=0 **to** *p*-1 **do**

(1.1)**if** (my_rank<*j*) **then**　/*对于主行前面的块*/

(i)*v*=*i***p*+*j*　　/* *v* 为主元素的行号*/

/*确定本处理器所存未消元部分的最大元及其位置存于数组 *lmax* 中*/

(ii)*lmax*(0)=*a*[*i*+1,*v*]

(iii)**for** *k*=*i*+1 **to** *m*-1 **do**

for *t*=*v* **to** *n*-1 **do**

if (|*a*[*k*,*t*]|>*lmax*[0]) **then**

lmax[0]= *a*[*k*,*t*],*lmax*[1]=*k*,

lmax[2]=*t*,*lmax*[3]=my_rank

end if

end for

end for

end if

(1.2)**if** (my_rank≥*j*) **then**　/*对于主行前面的块*/

(i)*v*=*i***p*+*j*　/* *v* 为主元素的行号*/

/*确定本处理器所存未消元部分的最大元及其位置存于数组 *lmax* 中*/

(ii)*lmax*[0]=*a*[*i*,*v*]

(iii)**for** *k*=*i* **to** *m*-1 **do**

for *t*=*v* **to** *n*-1 **do**

if (|*a*[*k*,*t*]|>*lmax*[0]) **then**

lmax[0]= *a*[*k*,*t*],*lmax*[1]=*k* ,

lmax[2]=*t*,*lmax*[3]=my_rank

end if

end for

end for

end if

(1.3)用 Allgather 操作将每一个处理器中的 *lmax* 元素广播到其它所有处理器中

(1.4) /*确定最大元及其位置*/

maxvalue=getpivort(*max*),*maxrow*=getrow(*max*)

maxcolumn=getcolumn(*max*),*maxrank*=getrank(*max*)

(1.5) /*列交换*/

if (*maxcolumn*≠*v*) **then**

(i)**for** *t*=0 **to** *m* **do**

交换 *a*[*t*,*v*]与 *a*[*t*,*maxcolumn*]

end for

(ii)交换 *shift*[*v*]与 *shift*[*maxcolumn*]

end if

(1.6) /*行交换*/

if (my_rank=*j*) **then**

if (*maxcolumn* ≠ *a*[*i*,*v*]) **then**

(i)**if** ([*maxrank*=*j*] and [*i*≠*maxrow*]) **then**

innerexchangerow( ) /*处理器内部换行*/

end if

(ii)**if** (*maxrank* ≠ *j*) **then**

outerexchangerow( ) /*处理器之间换行*/

end if

end if

end if

(1.7)**if** (my_rank=*j*) **then** /*主行所在的处理器*/

/*对主行作归一化运算*/

(i)**for** *k*=*v*+1 **to** *n*-1 **do**

a[*i*,*k*]= *a*[*i*,*k*]/*a*[*i*,*v*]

end for

(ii)*b*[*i*]=*b*[*i*]/ *a*[*i*,*v*],*a*[*i*,*v*]=1

/*将主行元素赋予数组 *f* */

(iii)**for** *k*=*v*+1 **to** *n*-1 **do**

f[*k*]= *a*[*i*,*k*]

end for

(iv)*f*[*n*]=*b*[*i*]

(v)广播主行到所有处理器

else /*非主行所在的处理器*/

接收广播来的主行元素存于数组 *f* 中

end if

(1.8)**if** (my_rank ≤ *j*) **then**

for *k*=*i*+1 **to** *m*-1 **do**

(i)**for** *w*=*v*+1 **to** *n*-1 **do**

a[*k*,*w*]= *a*[*k*,*w*]-*f*[*w*]* *a*[*k*,*v*]

end for

(ii)*b*[*k*]=*b*[*k*]-*f*[*n*]* *a*[*k*,*v*]

end for

end if

(1.9)**if** (my_rank>*j*) **then**

for *k*=*i* **to** *m*-1 **do**

(i)**for** *w*=*v*+1 **to** *n*-1 **do**

a[*k*,*w*]= *a*[*k*,*w*]-*f*[*w*]* *a*[*k*,*v*]

end for

(ii)*b*[*k*]=*b*[*k*]-*f*[*n*]* *a*[*k*,*v*]

```
                end for
            end if
        end for
    end for
    /*回代过程*/
    (2)for i=0 to m-1 do
        sum[i]=0.0
      end for
    (3)for i= m-1 downto 0 do
        for j= p-1 downto 0 do
            if (my_rank=j) then     /*主行所在的处理器*/
                (i)x[i*p+j]=(b[i]-sum[i])/a[i,i*p+j]
                (ii)将 x[i*p+j]广播到所有处理器中
                (iii)for k =0 to i-1 do
                    /*计算有关 x[i*p+j]的内积项并累加*/
                    sum[k]=sum[k]+ a[k,i*p+j]* x[i*p+j]
                  end for
            else     /*非主行所在的处理器*/
                (iv)接收广播来的 x[i*p+j]的值
                (v)if (my_rank>j) then
                    for k =0 to i-1 do
                      /*计算有关 x[i*p+j]的内积项并累加*/
                      sum[k]=sum[k]+ a[k,i*p+j]* x[i*p+j]
                    end for
                  end if
                (vi)if (my_rank<j) then
                    for k =0 to i do
                      /*计算有关 x[i*p+j]的内积项并累加*/
                      sum[k]=sum[k]+ a[k,i*p+j]* x[i*p+j]
                    end for
                  end if
            end if
        end for
    end for
End
```

消去过程中，参与计算的行向量数在减少，同时参与计算的行向量长度也在减少。设第i次消去参与变换的行向量数为$(m-i)$，行向量长度为$(n-v)$，其中$v=ip+p/2$若取一次乘法和加法运算时间或一次除法运算时间为一个单位时间，则上述算法所需的计算时间为$T_1=\sum_{i=0}^{m-1}(m-i)*p*(n-v)=(3n^2-pn+4mn^2)/12$，其间共有$n$行数据作为主行被广播，通信时间为$n[(t_s+(n+1)t_w)\log p$；回代过程中，由于0号处理器对对应项进行乘积求和的计算量最大，因此以0号处理器为对象分析。由于0号处理器计算解向量$x(0),x(p),\cdots,x((m-1)*p)$的时间分别为$mp,(m-1)p,\cdots,p$，因此其回代过程的计算时间为$T_2=mp(m+1)/2$，解向量$x$的$n$个元素被播送给所有处理器的通信时间为$n(t_s+t_w)\log p$。若不考虑选主元的时间而仅考虑一般高斯消去法的计

算时间，则此并行计算时间为T_p= $(3n^2-pn+4mn^2)/12+mp\ (m+1)/2+n\log p[2t_s+\ (n+2)t_w]$。

MPI源程序请参见章末附录。

2.2 约当消去法解线性方程组

2.2.1 约当消去及其串行算法

约当消去法(Jordan Elimination)是一种无回代过程的直接解法，它直接将系数矩阵 $\boldsymbol{A}$ 变换为单位矩阵，经变换后的常数向量 $\boldsymbol{b}$ 即是方程组的解。这种消去法的基本过程与高斯消去法相同，只是在消去过程中，不但将主对角线以下的元素消成 0，而且将主对角线以上的元素也同时消成 0。一般约当消去法的计算过程是按 k=1,2, …,n 的顺序，逐次以第 k 行作为主行进行消去，以消去第 k 列除主元素以外的所有元素 a_{1k},a_{2k}, …,a_{nk}。记 $A^{(0)}=A$，用约当消去法由 $A^{(0)}$出发通过变换得到方阵序列{ $A^{(k)}$}，$A^{(k)}=[a_{ij}^{(k)}]$，（k=0,1,2,…,n），每次由 $A^{(k-1)}$到 $A^{(k)}$ 的过程分为三步：

⑴ $a_{kj}^{(k)} = a_{kj}^{(k-1)} / a_{kk}^{(k-1)} \quad (j = k+1,...,n)$

$b_k^{(k)} = b_k^{(k-1)} / a_{kk}^{(k-1)}$

⑵ $a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k)} \quad (1 \le i \le n, i \ne k, k < j \le n)$

$b_i^{(k)} = b_i^{(k-1)} - b_k^{(k-1)} a_{kj}^{(k)}$

⑶ $a_{kk}^{(k)} = 1, \quad a_{ik}^{(k)} = 0 \quad (1 \le i \le n, i \ne k)$

若取一次乘法和加法运算时间或一次除法运算时间为一个单位时间，则由于在第 i 次消去中，参与变换的行向量数为 n，行向量长度为 i，所以下述算法 19.3 的约当消去法的时间复杂度为 $\sum_{i=1}^{n} ni = n^2(n+1)/2 = O(n^3)$。

算法 19.3　单处理器上约当消去法求解线性方程组的算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$

输出：解向量 $x_{n\times 1}$

```
Begin
   (1)for i=1 to n do
        shift(i)=i
      end for
   (2)for k=1 to n do
        (2.1) d=0
        (2.2)for i=k to n do
               for j=k to n do
                 if (| a[i,j] |>d) then d =| a[i,j] |, js=j, l=i    end if
               end for
             endfor
```

(2.3)**if** ($js \neq k$) **then**
(i)**for** i=1 **to** n **do**
交换 $a[i,k]$和 $a[i,js]$
end for
(ii)交换 $shift[k]$与 $shift[js]$
end if
(2.4) **if** ($l \neq k$) **then**
(i)**for**j=k **to** n **do**
交换 $a[k,j]$与 $a[l,j]$
end for
(ii)交换 $b[k]$与 $b[l]$
end if
(2.5) **for**j=k+1 **to** n **do**
$a[k,j]$= $a[k,j]$/$a[k,k]$
end for
(2.6)$b[k]$= $b[k]$/$a[k,k]$,$a[k,k]$=1
(2.7)**for** i=1 **to** n **do**
if ($I \neq k$) **then**
(i)**for**j=k+1 **to** n **do**
$a[i,j]$= $a[i,j]$- $a[i,k]$* $a[k,j]$
end for
(ii)$b[i]$= $b[i]$- $a[i,k]$* $b[k]$
(iii)$a[i,k]$=0
end if
end for
end for
(3)**for** k=1 **to** n **do**
for i=1 **to** n **do**
if ($shift[i]$=k) **then** 输出 $x[k]$的值 $b[i]$ **end if**
end for
end for
End

2.2.2 约当消去法的并行算法

约当消去法采用与高斯消去法相同的数据划分和选主元的方法。在并行消去过程中，首先对主行元素作归一化操作 $a_{kj}=a_{kj}/a_{kk}$ ($j=k+1, \cdots,n$)，$b_k= b_k/a_{kk}$ ，然后将主行广播给所有处理器，各处理器利用接收到的主行元素对其部分行向量做行变换。若以编号为 my_rank 的处理器的第 i 行作为主行，在归一化操作之后，将它广播给所有处理器，则编号不为 my_rank 的处理器利用主行对其第 0,1, …, m-1 行数据和子向量做变换，编号为 my_rank 的处理器利用主行对其除 i 行以外的数据和子向量做变换（第 i 个子向量除外）。具体算法框架描述如下：

算法 19.4　约当消去法求解线性方程组的并行算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

对所有处理器 my_rank(my_rank=0,…, *p*-1)同时执行如下的算法:

/*消去过程*/

for *i*=0 **to** *m*-1 **do**

for *j*=0 **to** *p*-1 **do**

(1)**if** (my_rank<*j*) **then** /*对于主行前面的块*/

(1.1)*v*=*i***p*+*j* /* *v* 为主元素的行号*/

/*确定本处理器所存的未消元部分的最大元及其位置存于数组 *lmax* 中*/

(1.2)*lmax*(0)=*a*[*i*+1,*v*]

(1.3)**for** *k*=*i*+1 **to** *m*-1 **do**

for *t*=*v* **to** *n*-1 **do**

if (|*a*[*k*,*t*]|>*lmax*[0])

lmax[0]= *a*[*k*,*t*], *lmax*[1]=*k* ,

lmax[2]=*t* ,*lmax*[3]=my_rank

end if

end for

end for

end if

(2)**if** (my_rank≥*j*) **then**

(2.1)*v*=*i***p*+*j*

(2.2)*lmax*[0]=*a*[*i*,*v*]

(2.3)**for** *k*=*i* **to** *m*-1 **do**

for *t*=*v* **to** *n*-1 **do**

if (|*a*[*k*,*t*]|>*lmax*[0])

lmax[0]= *a*[*k*,*t*], *lmax*[1]=*k*,

lmax[2]=*t* , *lmax*[3]=my_rank

end if

end for

end for

end if

(3)用 Allgather 操作将每个处理器中的 *lmax* 元素广播到其它所有处理器中

(4)/*确定最大元及其位置*/

maxvalue=getpivort(*max*),*maxrow*=getrow(*max*)

maxcolumn=getcolumn(*max*),*maxrank*= getrank(*max*)

(5)/*列交换*/

if (*maxcolumn*≠*v*) **then**

(5.1)**for** *t*=0 **to** *m*-1 **do**

交换 *a*[*t*,*v*]和 *a*[*t*,*maxcolumn*]

end for

(5.2)交换 *shift*[*v*]和 *shift*[*maxcolumn*]

end if

(6)/*行交换*/

if (my_rank=*j*) **then**

```
                    if (maxcolumn ≠ a[i,v]) then
                        (6.1)if ((maxrank=j) and (i≠maxrow)) then
                                innerexchangerow( )   /*处理器内部换行*/
                            end if
                        (6.2)if (maxrank≠j) then
                                outerexchangerow( )   /*处理器之间换行*/
                            end if
                    end if
                end if
            (7)if (my_rank=j) then   /*主行所在的处理器*/
                (7.1)for k=v+1 to n-1 do
                        a[i,k]= a[i,k]/a[i,v]
                    end for
                (7.2) b[i]=b[i]/a[i,v]，a[i,v]=1
                (7.3)for k=v+1 to n-1 do   /*将主行元素赋予数组 f */
                        f[k]= a[i,k]
                    end for
                (7.4) f[n]=b[i]
                (7.5)广播主行到所有处理器
                (7.6)for k=0 to m-1 do   /*处理存于该处理器中的非主行元素*/
                        if (k ≠ i) then
                            (i)for w=v+1 to n-1 do
                                a[k,w]= a[k,w]-f[w]* a[k,v]
                               end for
                            (ii)b[k]=b[k]-f[n]* a[k,v]
                        end if
                    end for
                else   /*非主行所在的处理器*/
                (7.7)接收广播来的主行元素存于数组 f 中
                (7.8)for k=0 to m do   /*处理非主行元素*/
                        (i)for w=v+1 to n do
                            a[k,w]= a[k,w]-f[w]* a[k,v]
                           end for
                        (ii)b[k]=b[k]-f[n]* a[k,v]
                    end for
                end if
            end for
        end for
    End
```

上述算法的计算过程中，参与计算的行向量数为 n，行向量长度为$(n-v)$，其中 $v=ip+p/2$。若取一次乘法和加法运算时间或一次除法运算时间为一个单位时间，则其所需的计算时间为

$T_1= \sum_{i=0}^{m-1} n*(n-v)= \sum_{i=0}^{m-1} n(n-ip-p/2)= mn^2-n^2/2- n^2 (m-1)/2$；另外，由于其间共有 n 行数据依次

作为主行被广播，通信时间为 $n[t_s+ (n+1)t_w] \log p$ ，所以该算法总共需要的并行计算时间为 $T_p= mn^2-n^2/2- n^2 (m -1)/2+n[t_s+(n+1)t_w] \log$。

MPI 源程序请参见所附光盘。

2.3 小结

线性代数方程组的求解在科学和工程计算中应用非常广泛，这是因为很多科学和工程问题大多可以化为线性代数方程组来求解。本章主要讨论线性方程组的直接解法，使读者能够了解与掌握利用矩阵变换技巧逐步消元从而求解方程组的基本方法。这种方法可以预先估计运算量，并可以得到问题的准确解，但由于实际计算过程中总存在舍入误差，因此得到的结果并非绝对精确，并且还存在着计算过程的稳定性问题。另外，文献[1]展示了三角形方程组求解器可在多计算机上有效地实现，[2]讨论了共享存储和分布存储结构的并行机上稠密线性方程组的并行算法，[3]是一本很好的综述性专著，它全面地讨论了向量和并行机上线性方程组的直接法和迭代法的并行求解方法，[4]中对各类线性方程组的直接解法有更详尽的讲解与分析，读者可以追踪进一步阅读。

参考文献

[1]. Romine C H, Ortega J M. Parallel Solution of Triangular Systems of Equations. Parallel Computing, 1988, 6:109-114

[2]. Gallivan K A, Plemmons R J, Sameh A H. Parallel Algorithms for Dense Linear Algebra Computations. SIAM Rev., 1990,32(1):54-135

[3]. Ortega J M. Introduction to Parallel and Vector Solution of Linear Systems. Plenum,1988

[4]. 陈国良 编著．并行算法的设计与分析（修订版）．高等教育出版社，2002.11

附录　全主元高斯消去法并行算法的 MPI 源程序

1. 源程序 gauss.c

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "math.h"
#define a(x,y) a[x*M+y]
#define b(x) b[x]
/*A 为 M*M 的系数矩阵*/
#define A(x,y) A[x*M+y]
#define B(x) B[x]
#define floatsize sizeof(float)
#define intsize sizeof(int)

int M;
int N;
int m;
float *A;
float *B;
int my_rank;
int p;
int l;
MPI_Status status;

void fatal(char *message)
{
    printf("%s\n",message);
    exit(1);
```

```
}


void Environment_Finalize(float *a,float *b,float
        *x,float *f)
{
        free(a);
        free(b);
        free(x);
        free(f);
}


int main(int argc, char **argv)
{
        int i,j,t,k,my_rank,group_size;
        int i1,i2;
        int v,w;
        float temp;
        int tem;
        float *sum;
        float *f;
        float lmax;
        float *a;
        float *b;
        float *x;
        int *shift;
        FILE *fdA,*fdB;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,
                &group_size);
        MPI_Comm_rank(MPI_COMM_WORLD,
                &my_rank);
        p=group_size;

        if (my_rank==0)
        {
                fdA=fopen("dataIn.txt","r");
                fscanf(fdA,"%d %d", &M, &N);
                if (M != N-1)
                {
                        printf("the input is wrong\n");
                        exit(1);
                }
                A=(float *)malloc(floatsize*M*M);
                B=(float *)malloc(floatsize*M);
                for(i = 0; i < M; i++)
                {
                        for(j = 0; j < M; j++)
                                fscanf(fdA,"%f", A+i*M+j);
                        fscanf(fdA,"%f", B+i);
                }
                fclose(fdA);
}

/*0 号处理器将 M 广播给所有处理器*/
MPI_Bcast(&M,1,MPI_INT,0,
        MPI_COMM_WORLD);
m=M/p;
if (M%p!=0) m++;
/*各处理器为主行元素建立发送和接收缓冲
区(M+1) */
f=(float*)malloc(sizeof(float)*(M+1));
/*分配至各处理器的子矩阵大小为 m*M*/
a=(float*)malloc(sizeof(float)*m*M);
/*分配至各处理器的子向量大小为 m*/
b=(float*)malloc(sizeof(float)*m);
sum=(float*)malloc(sizeof(float)*m);
x=(float*)malloc(sizeof(float)*M);
shift=(int*)malloc(sizeof(int)*M);

if (a==NULL||b==NULL||f==NULL||
        sum==NULL||x==NULL||shift==NULL)
                fatal("allocate error\n");

for(i=0;i<M;i++)
        shift[i]=i;

/*0 号处理器采用行交叉划分将矩阵 A 划分为
        大小为 m*M 的 p 块子矩阵,将 B 划分为
        大小为 m 的 p 块子向量，依次发送给 1
        至 p-1 号处理器*/
if (my_rank==0)
{
        for(i=0;i<m;i++)
                for(j=0;j<M;j++)
                        a(i,j)=A(i*p,j);
```

```
        for(i=0;i<m;i++)
            b(i)=B(i*p);
        for(i=0;i<M;i++)
            if ((i%p)!=0)
            {
                i1=i%p;
                i2=i/p+1;
                MPI_Send(&A(i,0),M,
                    MPI_FLOAT,i1,i2,
                    MPI_COMM_WORLD)
                    ;
                MPI_Send(&B(i),1,
                    MPI_FLOAT,i1,i2,
                    MPI_COMM_WORLD)
                    ;
            }
}
else
{
        for(i=0;i<m;i++)
        {
            MPI_Recv(&a(i,0),M,MPI_FLOAT,
                0,i+1,MPI_COMM_WORLD,
                &status);
            MPI_Recv(&b(i),1,MPI_FLOAT,
                0,i+1,MPI_COMM_WORLD,
                &status);
         }
}

/*消去*/
for(i=0;i<m;i++)
        for(j=0;j<p;j++)
        {
            /*j 号处理器负责广播主行元素*/
            if (my_rank==j)
            {
                /*主元素在原系数矩阵 A 中
                    的行号和列号为 v*/
                v=i*p+j;
                lmax=a(i,v);
                l=v;

                /*在同行的元素中找最大元，
                    并确定最大元所在的列
                    l*/
                for(k=v+1;k<M;k++)
                    if (fabs(a(i,k))>lmax)
                    {
                        lmax=a(i,k);
                        l=k;
                    }

                /*列交换*/
                if (l!=v)
                {
                    for(t=0;t<m;t++)
                    {
                        temp=a(t,v);
                        a(t,v)=a(t,l);
                        a(t,l)=temp;
                    }
                    tem=shift[v];
                    shift[v]=shift[l];
                    shift[l]=tem;
                }
                /*归一化*/
                for(k=v+1;k<M;k++)
                    a(i,k)=a(i,k)/a(i,v);
                b(i)=b(i)/a(i,v);
                a(i,v)=1;
                for(k=v+1;k<M;k++)
                    f[k]=a(i,k);
                f[M]=b(i);
                /*发送归一化后的主行*/
                MPI_Bcast(&f[0],M+1,
                    MPI_FLOAT,my_rank,
                    MPI_COMM_WORLD)
                    ;
                /*发送主行中主元素所在的
                    列号*/
                MPI_Bcast(&l,1,MPI_INT,
                    my_rank,
                    MPI_COMM_WORLD)
                    ;
            }
            else
            {
```

```
        v=i*p+j;
        /*接收归一化后的主行*/
        MPI_Bcast(&f[0],M+1,
                MPI_FLOAT,
                MPI_COMM_WORLD)
                ;
        /*接收主行中主元素所在的
                列号*/
        MPI_Bcast(&l,1,MPI_INT, j,
                MPI_COMM_WORLD)
                ;
        /*列交换*/
        if (l!=v)
        {
                for(t=0;t<m;t++)
                {
                        temp=a(t,v);
                        a(t,v)=a(t,l);
                        a(t,l)=temp;
                }
                tem=shift[v];
                shift[v]=shift[l];
                shift[l]=tem;
        }
}

/*编号小于j的处理器(包括j本身)
        利用主行对其第 i+1,…,m-1
        行数据做行变换*/
if (my_rank<=j)
        for(k=i+1;k<m;k++)
        {
                for(w=v+1;w<M;w++)
                        a(k,w)=a(k,w)
                                -f[w]* a(k,v);
                b(k)=b(k)-f[M]*a(k,v);
        }

/*编号大于 j 的处理器利用主行对
        其第 i,…,m-1 行数据做行变
        换*/
if (my_rank>j)
        for(k=i;k<m;k++)
        {
                for(w=v+1;w<M;w++)
                        a(k,w)=a(k,w)
                                -f[w]*a(k,v);

                b(k)=b(k)-f[M]*a(k,v);
        }
}

for(i=0;i<m;i++)
        sum[i]=0.0;

/*回代过程*/
for(i=m-1;i>=0;i--)
        for(j=p-1;j>=0;j--)
                if (my_rank==j)
                {
                        x[i*p+j]=(b(i)-sum[i])/
                                a(i,i*p+j);
                        MPI_Bcast(&x[i*p+j],1,
                                MPI_FLOAT,my_rank,
                                MPI_COMM_WORLD)
                                ;

                        for(k=0;k<i;k++)
                                sum[k]=sum[k]+
                                        a(k,i*p+j)*
                                        x[i*p+j];
                }
                else
                {
                        MPI_Bcast(&x[i*p+j],1,
                                MPI_FLOAT,j,
                                MPI_COMM_WORLD)
                                ;
                        if (my_rank>j)
                                for(k=0;k<i;k++)
                                        sum[k]=sum[k]
                                                +a(k,i*p+j)
                                                *x[i*p+j];
                        if (my_rank<j)
                                for(k=0;k<=i;k++)
                                        sum[k]=sum[k]
                                                +a(k,i*p+j)
                                                *x[i*p+j];
```

```
            }

    if (my_rank!=0)
        for(i=0;i<m;i++)
            MPI_Send(&x[i*p+my_rank],1,
                MPI_FLOAT,0,i,
                MPI_COMM_WORLD);
    else
        /*0 号处理器从其余各处理器中接收子
            解向量 x*/
        for(i=1;i<p;i++)
            for(j=0;j<m;j++)
                MPI_Recv(&x[j*p+i],1,
                    MPI_FLOAT, i,j,
                    MPI_COMM_WORLD,
                    &status);

    if (my_rank==0)
    {
        printf("Input of file \"dataIn.txt\"\n");
        printf("%d\t%d\n", M, N);
        for(i=0;i<M;i++)
        {
            for(j=0;j<M;j++)
                printf("%f\t",A(i,j));
            printf("%f\n",B(i));
        }
        printf("\nOutput of solution\n");
        for(k=0;k<M;k++)
        {
            for(i=0;i<M;i++)
            {
                if (shift[i]==k)
                    printf("x[%d]=%f\n",
                        k,x[i]);
            }
        }
    }

    MPI_Finalize();
    Environment_Finalize(a,b,x,f);
    return(0);
}
```

2. 运行实例

编译：mpicc –o gauss gauss.cc

运行：可以使用命令 mpirun －np ProcessSize gauss 来运行该程序，其中 ProcessSize 是所使用的处理器个数,这里取为 5。本实例中使用了

mpirun –np 5 gauss

运行结果：

```
Input of file "dataIn.txt"
4     5
1.000000   4.000000   -2.000000  3.000000   6.000000
2.000000   2.000000   0.000000   4.000000   2.000000
3.000000   0.000000   -1.000000  2.000000   1.000000
1.000000   2.000000   2.000000   -3.000000  8.000000

Output of solution
x[0]=1.000000
x[1]=2.000000
x[2]=0.000000
x[3]=-1.000000
```

说明：该运行实例中，A 为 4×4 的矩阵，B 是长度为 4 的向量，它们的值存放于文档“dataIn.txt”中，其中前 4 列为矩阵 A，最后一列为向量 B，最后输出线性方程组 AX=B 的解向量 X。

第二十章 线性方程组的迭代解法

在阶数较大、系数阵为稀疏阵的情况下，可以采用迭代法求解线性方程组。用**迭代法**(Iterative Method)求解线性方程组的优点是方法简单，便于编制计算机程序，但必须选取合适的迭代格式及初始向量，以使迭代过程尽快地收敛。迭代法根据迭代格式的不同分成**雅可比**(Jacobi)迭代、**高斯-塞德尔**(Gauss-Seidel)迭代和**松弛**(Relaxation)法等几种。在本节中，我们假设系数矩阵 A 的主对角线元素 $a_{ii} \neq 0$，且按行严格**对角占优**(Diagonal Donimant)，即：

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \quad (i = 1,2,...,n)$$

3.1 雅可比迭代

3.1.1 雅可比迭代及其串行算法

雅可比迭代的原理是：对于求解 n 阶线性方程组 $\boldsymbol{Ax=b}$，将原方程组的每一个方程 $a_{i1}x_1+a_{i2}x_2+\cdots+a_{in}x_n=b_i$ 改写为未知向量 $\boldsymbol{x}$ 的分量的形式：

$$x_i = (b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j) / a_{ii} \qquad (1 \le i \le n)$$

然后使用第 k-1 步所计算的变量 $x_i^{(k-1)}$来计算第 k 步的 $x_i^{(k)}$的值：

$$x_i^{(k)} = (b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k-1)}) / a_{ii} \qquad (1 \le i,k \le n)$$

这里，$x_i^{(k)}$为第 k 次迭代得到的近似解向量 $x^{(k)}=(x_1^{(k)}, x_2^{(k)}, \cdots, x_n^{(k)})^T$ 的第 i 个分量。取适当初始解向量 $x^{(0)}$代入上述迭代格式中，可得到 $x^{(1)}$，再由 $x^{(1)}$得到 $x^{(2)}$，依次迭代下去得到近似解向量序列$\{x^{(k)}\}$。若原方程组的系数矩阵按行严格对角占优，则$\{x^{(k)}\}$收敛于原方程组的解 $\boldsymbol{x}$。实际计算中，一般认为，当相邻两次的迭代值 $x_i^{(k+1)}$与 $x_i^{(k)}$ i=(1,2, $\cdots$,n)很接近时，$x_i^{(k+1)}$与准确解 $\boldsymbol{x}$ 中的分量 x_i 也很接近。因此，一般用 $\max\limits_{1\le i\le n}\left|x_i^{(k+1)}-x_i^{(k)}\right|$ 判断迭代是否收敛。

如果取一次乘法和加法运算时间或一次比较运算时间为一个单位时间，则下述雅可比迭代算法 20.1 的一轮计算时间为 $n^2+n=O(n^2)$。

算法 20.1　单处理器上求解线性方程组雅可比迭代算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，ε，初始解向量 $x_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

(1)**for** i=1 **to** n **do**

$x_i=b_i/a_{ii}$

end for

(2)diff=ε
(3)**while** (diff ≥ ε) **do**
(3.1)diff=0
(3.2)**for** *i*=1 **to** *n* **do**
(i)newx_i= b_i
(ii)**for** *j*= 1 **to** *n* **do**
if (*j* ≠ *i*) **then**
newx_i= newx_i- a_{ij} x_j
end if
end for
(iii)newx_i= newx_i/ a_{ii}
end for
(3.3)**for** *i*=1 **to** *n* **do**
(i)diff=max {diff, |newx_i- x_i|}
(ii)x_i=newx_i
end for
end while
End

3.1.2 雅可比迭代并行算法

A 是一个 *n* 阶系数矩阵，***b*** 是 *n* 维向量，在求解线性方程组 ***Ax=b*** 时，若处理器个数为 *p*，则可对矩阵 *A* 按行划分以实现雅可比迭代的并行计算。设矩阵被划分为 *p* 块，每块含有连续的 *m* 行向量，这里 $m=\lceil n/p \rceil$，编号为 *i* 的处理器含有 *A* 的第 *im* 至第(*i*+1)*m*-1 行数据，同时向量 ***b*** 中下标为 *im* 至(*i*+1)*m*-1 的元素也被分配至编号为 *i* 的处理器(*i*=0,1, …,*p*-1)，初始解向量 ***x*** 被广播给所有处理器。

在迭代计算中，各处理器并行计算解向量 *x* 的各分量值，编号为 *i* 的处理器计算分量 x_{im} 至 $x_{(i+1)m-1}$ 的新值。并求其分量中前后两次值的最大差 *localmax*，然后通过归约操作的求最大值运算求得整个 *n* 维解向量中前后两次值的最大差 *max* 并广播给所有处理器。最后通过扩展收集操作将各处理器中的解向量按处理器编号连接起来并广播给所有处理器，以进行下一次迭代计算，直至收敛。具体算法框架描述如下：

算法 20.2　求解线性方程组的雅可比迭代并行算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，ε，初始解向量 $x_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

对所有处理器 my_rank(my_rank=0,…, *p*-1)同时执行如下的算法:

while (*max*>*ε*) **do**

(1)**for** *i*=0 **to** *m*-1 **do**　/*各个处理器由所存的行计算出解 *x* 的相应的分量*/

(1.1)*sum*=0.0

(1.2)**for** *j*=0 **to** *n*-1 **do**

if (*j* ≠ (my_rank**m*+*i*)) **then**

sum=*sum*+*a*[*i*,*j*]**x*[*j*]

end if

end for

(1.3)*x*1[*i*]=(*b*[*i*] - *sum*)/*a*[*i*,my_rank**m*+*i*]

end for

(2)/*求出本处理器计算的 *x* 的相应的分量的新值与原值的差的最大值 *localmax* */

localmax=|*x*1[0]-*x*[0]|

(3)**for** *i*=1 **to** *m*-1 **do**

if (|*x*1[*i*]-*x*[*i*] |>*localmax*) **then**

localmax =|*x*1[*i*]-*x*[*i*] |

end if

end for

(4)用 Allgather 操作将 *x* 的所有分量的新值广播到所有处理器中

(5)用 Allreduce 操作求出所有处理器中 *localmax* 值的最大值 *max* 并广播到所有处理器中

end while

End

若取一次乘法和加法运算时间或一次比较运算时间为一个单位时间，则一轮迭代的计算时间为 $mn+m$；另外，各处理器在迭代中做一次归约操作，通信量为 1，一次扩展收集操作，通信量为 m，需要的通信时间为 $4t_s(\sqrt{p}-1)+(m+1)t_w(p-1)$，因此算法 20.2 的一轮并行计算时间为 $T_p=4t_s(\sqrt{p}-1)+(m+1)t_w(p-1)+mn+m$。

MPI 源程序请参见所附光盘。

3.2 高斯-塞德尔迭代

3.2.1 高斯-塞德尔迭代及其串行算法

高斯-塞德尔迭代的基本思想与雅可比迭代相似。它们的区别在于，在雅可比迭代中，每次迭代时只用到前一次的迭代值，而在高斯-塞德尔迭代中，每次迭代时充分利用最新的迭代值。一旦一个分量的新值被求出，就立即用于后续分量的迭代计算，而不必等到所有分量的新值被求出以后。设方程组 $Ax=b$ 的第 i 个方程为：

$$\sum_{j=1}^{n} a_{ij}\ x_j = b_i \qquad (i=1,2,\Lambda\ ,n)$$

高斯-塞德尔迭代公式为：

$$x_i^{(k+1)}=\frac{1}{a_{ii}}(b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{(k+1)}-\sum_{j=i+1}^{n}a_{ij}x_j^{(k)}) \qquad (i=1,2,\Lambda\ ,n)$$

取适当的 $x^{(0)}$作为初始向量，由上述迭代格式可得出近似解向量$\{x^{(k)}\}$。若原方程组的系数矩阵是按行严格对角占优的，则$\{x^{(k)}\}$收敛于方程组的解 $\boldsymbol{x}$，若取一次乘法和加法运算时间或一次比较运算时间为一个单位时间，则下述高斯-塞德尔迭代算法 20.3 的一轮计算时间为 $n^2+n=O(n^2)$。

算法 20.3　单处理器上求解线性方程组的高斯-塞德尔迭代算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，ε，初始解向量 $x_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

(1)**for** i=1 **to** n **do**

x_i=0

end for

(2)p=ε+1

(3)**while** ($p \geq \varepsilon$) **do**

for i=1 **to** n **do**

(i) $t = x_i$

(ii) s=0

(iii)**for** j= 1 **to** n **do**

if ($j \neq i$) **then**

s= s+ a_{ij} x_j

end if

end for

(iv) x_i=(b_i-s)/ a_{ii}

(v) **if** ($|x_i-t|>p$) **then** p= $|x_i-t|$ **end if**

end for

end while

End

3.2.2 高斯-塞德尔迭代并行算法

在并行计算中，高斯-塞德尔迭代采用与雅可比迭代相同的数据划分。对于高斯-塞德尔迭代，计算 x_i 的新值时，使用 x_{i+1}, …,x_{n-1} 的旧值和 x_0, …,x_{i-1} 的新值。计算过程中 x_i 与 x_0, …,x_{i-1} 及 x_{i+1}, …,x_{n-1} 的新值会在不同的处理器中产生，因此可以考虑采用时间偏移的方法，使各个处理器对新值计算的开始和结束时间产生一定的偏差。编号为 my_rank 的处理器一旦计算出 x_i(my_rank×$m \leq i <$ (my_rank+1)×m)的新值，就立即广播给其余处理器，以供各处理器对 x 的其它分量计算有关 x_i 的乘积项并求和。当它计算完 x 的所有分量后，它还要接收其它处理器发送的新的 x 分量，并对这些分量进行求和计算，为计算下一轮的 x_i 作准备。计算开始时，所有处理器并行地对主对角元素右边的数据项进行求和，此时编号为 0 的处理器（简称为 P_0）计算出 x_0,然后广播给其余处理器，其余所有的处理器用 x_0 的新值和其对应项进行求和计算,接着 P_0 计算出 x_1,x_2, …,当 P_0 完成对 x_{m-1} 的计算和广播后，P_1 计算出 x_m，并广播给其余处理器，其余所有的处理器用 x_m 的新值求其对应项的乘积并作求和计算。然后 P_1 计算出 x_{m+1},x_{m+2}, …,当 P_1 完成对 x_{2*m-1} 的计算和广播后，P_2 计算出 x_{2*m} …，如此重复下去，直至 x_{n-1} 在 P_{p-1} 中被计算出并广播至其余的处理器之后，P_0 计算出下一轮的新的 x_0，这样逐次迭代下去，直至收敛为止。具体算法框架描述如下：

算法 20.4　求解线性方程组的高斯-塞德尔迭代并行算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，ε，初始解向量 $x_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

(1)**for** i=my-rank* m **to** (my-rank+1)*m-1 **do**

/*所有处理器并行地对主对角元素右边的数据求和*/

(1.1)sum[i]=0.0

(1.2)**for** j=i+1 **to** n-1 **do**

sum[i]=sum[i]+a[i,j]*x[j]

end for

end for

(2)**while** (*total*<*n*) **do** /**total* 为新旧值之差小于 ε 的 *x* 的分量个数*/

(2.1) *iteration*=0

/**iteration* 为本处理器中新旧值之差小于 ε 的 *x* 的分量个数*/

(2.2)**for** *j*=0 **to** *n*-1 **do** /*依次以第 0,1, …, *n*-1 行为主行*/

(i) *q*=*j*/*m*

(ii)**if** (my_rank=*q*) **then** /*主行所在的处理器*/

temp= *x*[*j*], *x*[*j*]= (*b*[*j*]- *sum*[*j*])/*a*[*j*,*j*] /* 产生 *x*(*j*)的新的值*/

if (| *x*[*j*]-*temp* | <ε) **then** *iteration*= *iteration* +1 **end if**

将 *x*[*j*]的新值广播到其它所有处理器中

/*对其余行计算 *x*[*j*]所对于的内积项并累加*/

sum[*j*]=0

for *i*=my-rank* *m* **to** (my-rank+1)**m*-1 **do**

if ($j \neq i$) **then**

sum[*i*]=*sum*[*i*]+*a*[*i*,*j*]**x*[*j*]

end if

end for

else /*其它处理器*/

接收广播来的 *x*[*j*]的新值

/*对所存各行计算 *x*[*j*]所对于的内积项并累加*/

for *i*=my-rank* *m* **to** (my-rank+1)* *m*-1 **do**

sum[*i*]=*sum*[*i*]+*a*[*i*,*j*]**x*[*j*]

end for

end if

end for

(2.3)用 Allreduce 操作求出所有处理器中 *iteration* 值的和 *total* 并广播到所有处理器中

end while

End

若取一次乘法和加法运算时间或一次比较运算时间为一个单位时间。在算法开始阶段，各处理器并行计算主对角线右边元素相应的乘积项并求和,所需时间 *mn*-(1+*m*)*m*/2，进入迭代计算后，虽然各个处理器所负责的 *x* 的分量在每一轮计算中的开始时间是不一样的，但一轮迭代中的计算量都是相等的，因此不妨取 0 号处理器为对象分析一轮迭代计算的时间，容易得出 0 号处理器计算时间为 *mn*+*m*；另外它在一轮迭代中做广播操作 *n* 次，通信量为 1，归约操作 1 次，通信量为 1，所有的通信时间为 $n(t_s + t_w)\log p + 2t_s(\sqrt{p} - 1) + t_w(p - 1)$，因此高斯-塞德尔迭代的一轮并行计算时间为 $T_p = mn + m + n(t_s + t_w)\log p + 2t_s(\sqrt{p} - 1) + t_w(p - 1)$。

MPI 源程序请参见章末附录。

3.3 松弛法

3.3.1 松弛法及其串行算法

为了加快雅可比迭代与高斯-塞德尔迭代的收敛速度，可采用松弛法。以高斯-塞德尔迭代为例，首先，由高斯-塞德尔迭代格式求得第 k+1 次迭代值 $x_i^{(k+1)}$，即：

$$\bar{x}_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)})$$

然后计算第 k+1 次迭代值与第 k 次迭代值之差；即：

$$\bar{x}_i^{(k+1)} - x_i^{(k)}$$

最后在第 k 次迭代值的基础上，加上这个差的一个倍数作为实际的第 k+1 次迭代值，即：

$$x_i^{(k+1)} = x_i^{(k)} + w(\bar{x}_i^{(k+1)} - x_i^{(k)})$$

其中 w 为一个常数。综合以上过程，可以得到如下迭代格式：

$$x_i^{(k+1)} = (1-w)x_i^{(k)} + w(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}) \frac{1}{a_{ii}} \qquad (i = 1,2,\Lambda\ ,n)$$

其中， w 称为松弛因子，为保证收敛，要求松弛因子 w 满足：0<w<2。当 w>1 时，称之为超松弛法，此时，$\bar{x}_i^{(k+1)}$的比重被加大；当 w=1 时，它就成了一般的高斯-塞德尔迭代。实际应用时，可根据系数矩阵的性质及对其反复计算的经验来决定适合的松弛因子 w。若取一次乘法和加法运算时间或一次比较运算时间为一个单位时间，则下述算法 20.5 一轮计算时间计算为 $n^2+n=O(n^2)$。

算法 20.5　单处理器上松弛法求解线性方程组的算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，ε，初始解向量 $x_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

(1) **for** i=1 **to** n **do**

x_i=0

end for

(2) p=ε+1

(3) **while** ($p \geq \varepsilon$) **do**

(3.1) **for** i=1 **to** n **do**

(i) s=0

(ii) **for** j= 1 **to** n **do**

if ($j \neq i$) **then**

$s = s + a_{ij} x_j$

end if

end for

(iii) $y_i = (1-w)\, x_i + w(b_i - s)/a_{ii}$

(iv) **if** ($|y_i - x_i| > p$) **then** $p = |y_i - x_i|$ **end if**

(v) $x_i = y_i$

end for

end while

End

3.3.2 松弛法并行算法

松弛法并行算法和高斯-塞德尔迭代并行算法基本相同，只是在计算分量的时候，将对 x 分量的新值的计算改为“$x[j]=(1-w)*temp+w*(b[j]-sum[j])/a[j,j]$”，其中 temp 为 $x[j]$的原有值。具体并行算法描述如下:

算法 20.6　求解线性方程组的松弛迭代并行算法

输入：系数矩阵 $A_{n\times n}$，常数向量 $b_{n\times 1}$，ε，初始解向量 $x_{n\times 1}$

输出：解向量 $x_{n\times 1}$

Begin

对所有处理器 my_rank(my_rank=0,…, *p*-1)同时执行如下的算法:

/*所有处理器并行地对主对角元素右边的数据求和*/

(1) **for** *i*=my-rank* *m* **to** (my-rank+1)**m*-1 **do**

(1.1)*sum*[*i*]=0.0

(1.2)**for** *j*=*i*+1 **to** *n*-1 **do**

sum[*i*]=*sum*[*i*]+*a*[*i,j*]**x*[*j*]

end for

end for

(2) **while** (*total*<*n*) **do**　/**total* 为新旧值之差小于 ε 的 *x* 的分量个数*/

(2.1) *iteration*=0　/* *iteration* 为本处理器新旧值之差小于 ε 的 *x* 的分量个数*/

(2.2) **for** *j*=0 **to** *n*-1 **do**　/*依次以第 0,1, …, *n*-1 行为主行*/

(i) *q*=*j*/*m*

(ii) **if** (my_rank=*q*) **then**　/*主行所在的处理器*/

temp= *x*[*j*]

x[*j*]=(1-*w*)**temp*+*w** (*b*[*j*]- *sum*[*j*])/*a*[*j,j*]　/* 产生 *x*[*j*]的新的值*/

if (| *x*[*j*]-*temp* | <ε) **then** *iteration*= *iteration* +1 **end if**

将 *x*(*j*)的新值广播到其它所有处理器中

/*对其余行计算 *x*[*j*]所对于的内积项并累加*/

sum[*j*]=0

for *i*=my-rank* *m* **to** (my-rank+1)**m*-1 **do**

if (*j* ≠ *i*) **then**

sum[*i*]=*sum*[*i*]+*a*[*i,j*]**x*[*j*]

end if

end for

(iii) **else**　/*其它处理器*/

接收广播来的 *x*[*j*]的新值

/*对所存各行计算 *x*[*j*]所对于的内积项并累加*/

for *i*=my-rank* *m* **to** (my-rank+1)* *m*-1 **do**

sum[*I*]=*sum*[*i*]+*a*[*i,j*]**x*[*j*]

end for

end if

end for

(2.3) 用 Allreduce 操作求出所有处理器中 *iteration* 值的和 *total* 并广播到所有

处理器中

end while

End

与并行高斯-塞德尔迭代相似,并行松弛迭代法的一轮并行计算时间为: $T_p = mn + m + n(t_s + t_w)\log p + 2t_s(\sqrt{p} - 1) + t_w(p - 1)$。

MPI 源程序请参见所附光盘。

3.4 小结

本章主要讨论线性方程组的迭代解法，这种方法是一种逐步求精的近似求解过程，其优点是简单，易于计算机编程，但它存在着迭代是否收敛以及收敛速度快慢的问题。一般迭代过程由预先给定的精度要求来控制，但由于方程组的准确解一般是不知道的，因此判断某次迭代是否满足精度要求也是比较困难的，需要根据具体情况而定。文献[1]给出了稀疏线性方程组迭代解法的详尽描述，还包含了**多重网格**(Multigrid)法、**共轭梯度**(Conjugate Gradient)法，[2]综述了稀疏线性方程组的并行求解算法，[3]综述了在向量机和并行机上偏微分方程的求解方法，[4]讨论了超立方多处理机上的多重网格算法，[5]讨论了并行共轭梯度算法，[6]深入而全面地论述了 SIMD 和 MIMD 模型上的数值代数、离散变换和卷积、微分方程、计算数论和最优化计算的并行算法，对并行排序算法也作了介绍。此外，在[7]中第九章的参考文献注释中还列举了大量有关参考文献，进一步深入研究的读者可在这些文献中获得更多的资料。

参考文献

[1]. 陈国良 编著．并行计算——结构·算法·编程．高等教育出版社,1999.10

[2]. Heath M T, Ng E and Peyton B W. Parallel Algorithm for Space Linear Systems. SIAM Review,1991,33:420-460

[3]. Ortega J M, Voigt R G. Solution of Partial Differential Equations on Vector and Parallel Computers. SIAM Review,1985,27(2):149-240

[4]. Chan T F,Saad Y. Multigrid Algorithms on the Hypercube Multiprocessor. IEEE-TC,1986,C-35(11):969-977

[5]. Chronopoulos A T, Gear C W. On the Efficient Implementation of Pre-condition S-step Conjugate Gradient Methods on Multiprocessors with Memory Hierarchy. Parallel Computing,1989,11:37-53

[6]. 李晓梅，蒋增荣 等编著．并行算法(第五章) ．湖南科技出版社, 1992

[7]. Quinn M J. Parallel Computing-Theory and Practic(second edition)McGraw-Hill, Inc., 1994

附录 高斯-塞德尔迭代并行算法的 MPI 源程序

1. 源程序 seidel.c

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "math.h"
```

```
#define E 0.0001
#define a(x,y) a[x*size+y]
#define b(x) b[x]
#define v(x) v[x]
/*A 为 size*size 的系数矩阵*/
#define A(x,y) A[x*size+y]
#define B(x) B[x]
#define V(x) V[x]
#define intsize sizeof(int)
#define floatsize sizeof(float)
#define charsize sizeof(char)

int size,N;
int m;
float *B;
float *A;
float *V;
int my_rank;
int p;
MPI_Status status;
FILE *fdA,*fdB,*fdB1;

void Environment_Finalize(float *a,float *b,float *v)
{
        free(a);
        free(b);
        free(v);
}


int main(int argc, char **argv)
{
        int i,j,my_rank,group_size;
        int k;
        float *sum;
        float *b;
        float *v;
        float *a;
        float *differ;
        float temp;
        int iteration,total,loop;
        int r,q;

        loop=0;
total=0;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,
        &group_size);
MPI_Comm_rank(MPI_COMM_WORLD,
        &my_rank);
p=group_size;

if(my_rank==0)
{
        fdA=fopen("dataIn.txt","r");
        fscanf(fdA,"%d %d", &size, &N);
        if (size != N-1)
        {
                printf("the input is wrong\n");
                exit(1);
        }
        A=(float *)malloc(floatsize*size*size);
        B=(float *)malloc(floatsize*size);
        V=(float *)malloc(floatsize*size);

        for(i = 0; i < size; i++)
        {
                for(j = 0; j < size; j++)
                        fscanf(fdA,"%f", A+i*size+j);
                fscanf(fdA,"%f", B+i);
        }

        for(i = 0; i < size; i ++)
                fscanf(fdA,"%f", V+i);
        fclose(fdA);
        printf("Input of file \"dataIn.txt\"\n");
        printf("%d\t%d\n", size, N);

        for(i = 0; i < size; i ++)
        {
                for(j = 0; j < size; j ++)
                        printf("%f\t",A(i,j));
                printf("%f\n",B(i));
        }

        printf("\n");
        for(i = 0; i < size; i ++)
                printf("%f\t", V(i));
```

```
        printf("\n\n");
        printf("\nOutput of result\n");
}

/*0 号处理器将 size 广播给所有处理器*/
MPI_Bcast(&size,1,MPI_INT,0,
        MPI_COMM_WORLD);
m=size/p;if (size%p!=0) m++;
v=(float *)malloc(floatsize*size);
a=(float *)malloc(floatsize*m*size);
b=(float *)malloc(floatsize*m);
sum=(float *)malloc(floatsize*m);
if (a==NULL||b==NULL||v==NULL)
        printf("allocate space    fail!");
if (my_rank==0)
{
        for(i=0;i<size;i++)
                v(i)=V(i);
}

/*初始解向量 v 被广播给所有处理器*/
MPI_Bcast(v,size,MPI_FLOAT,0,
        MPI_COMM_WORLD);
/*0 号处理器采用行块划分将矩阵 A 划分为大
        小为 m*size 的 p 块子矩阵,将 B 划分为
        大小为 m 的 p 块子向量，依次发送给 1
        至 p-1 号处理器*/
if (my_rank==0)
{
        for(i=0;i<m;i++)
                for(j=0;j<size;j++)
                        a(i,j)=A(i,j);
        for(i=0;i<m;i++)
                b(i)=B(i);
        for(i=1;i<p;i++)
        {
                MPI_Send(&(A(m*i,0)), m*size,
                        MPI_FLOAT,i,i,
                        MPI_COMM_WORLD);
                MPI_Send(&(B(m*i)),m,
                        MPI_FLOAT,i,i,
                        MPI_COMM_WORLD);
        }
        free(A); free(B); free(V);
}
else
{
        MPI_Recv(a,m*size,MPI_FLOAT,
                0,my_rank,MPI_COMM_WORLD,
                &status);
        MPI_Recv(b,m,MPI_FLOAT,0,my_rank,
                MPI_COMM_WORLD,&status);
}

/*所有处理器并行地对主对角元素右边的数
        据求和*/
for(i=0;i<m;i++)
{
        sum[i]=0.0;
        for(j=0;j<size;j++)
                if (j>(my_rank*m+i))
                        sum[i]=sum[i]+a(i,j)*v(j);
}

while (total<size)
{
        iteration=0;
        total=0;
        for(j=0;j<size;j++)
        {
                r=j%m; q=j/m;
                /*编号为 q 的处理器负责计算解向
                        量并广播给所有处理器*/
                if (my_rank==q)
                {
                        temp=v(my_rank*m+r);
                        for(i=0;i<r;i++)
                                sum[r]=sum[r]+
                                        a(r,my_rank*m+i)*
                                        v(my_rank*m+i);
                        /*计算出的解向量*/
                        v(my_rank*m+r)=
                                (b(r)-sum[r])/
                                a(r,my_rank*m +r);
                        if (fabs(v(my_rank*m+r)
                                -temp)<E)
                                        iteration++;
                        /*广播解向量*/
```

```
            MPI_Bcast(&v(my_rank*m+
                r), 1,MPI_FLOAT,
                my_rank,
                MPI_COMM_WORLD)
                ;
            sum[r]=0.0;
            for(i=0;i<r;i++)
                sum[i]=sum[i]+a(i,
                    my_rank*m+r)*
                    v(my_rank*m+r);
        }
        else
        /*各处理器对解向量的其它分量计
            算有关乘积项并求和*/
        {
            MPI_Bcast(&v(q*m+r),1,
                MPI_FLOAT,q,
                MPI_COMM_WORLD)
                ;
            for(i=0;i<m;i++)
                sum[i]=sum[i]+ a(i,q*m
                    +r)* v(q*m+r);
        }
    }
```

```
        /*通过归约操作的求和运算以决定是否
            进行下一次迭代*/
        MPI_Allreduce(&iteration,&total,1,
            MPI_FLOAT,MPI_SUM,
            MPI_COMM_WORLD);
        loop++;
        if (my_rank==0)
            printf("in the %d times total vaule =
                %d\n",loop,total);
    }

    if (my_rank==0)
    {
        for(i = 0; i < size; i ++)
            printf("x[%d] = %f\n",i,v(i));
        printf("\n");
    }

    printf("Iteration num = %d\n",loop);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    Environment_Finalize(a,b,v);
    return (0);
}
```

2. 运行实例

编译：mpicc –o seidel seidel.cc

运行：可以使用命令 mpirun －np ProcessSize seidel 来运行该程序，其中 ProcessSize 是所使用的处理器个数,这里取为 4。本实例中使用了

mpirun –np 4 seidel

运行结果：

Input of file "dataIn.txt"

3 4

9.000000 -1.000000 -1.000000 7.000000

-1.000000 8.000000 0.000000 7.000000

-1.000000 0.000000 9.000000 8.000000

0.000000 0.000000 1.000000

Output of result

in the 1 times total vaule = 0

in the 2 times total vaule = 0

in the 3 times total vaule = 0

in the 4 times total vaule = 3
x[0] = 0.999998
x[1] = 1.000000
x[2] = 1.000000
Iteration num = 4

说明：该运行实例中，A 为 3×4 的矩阵，B 是长度为 3 的向量，它们的值存放于文档“dataIn.txt”中，其中前 3 行为矩阵 A，最后一行为向量 B，最后输出线性方程组 AX=B 的解向量 X。

第二十一章 矩阵特征值计算

在实际的工程计算中，经常会遇到求 n 阶方阵 A 的**特征值**(Eigenvalue)与**特征向量**(Eigenvector)的问题。对于一个方阵 A，如果数值 λ 使方程组

$$Ax=\lambda x$$

即 $(A-\lambda I_n)x=0$ 有非零**解向量**(Solution Vector)x，则称 λ 为方阵 A 的特征值，而非零向量 x 为特征值 λ 所对应的特征向量，其中 I_n 为 n 阶单位矩阵。

由于根据定义直接求矩阵特征值的过程比较复杂，因此在实际计算中，往往采取一些数值方法。本章主要介绍求一般方阵绝对值最大的特征值的**乘幂**(Power)法、求对称方阵特征值的雅可比法和**单侧旋转**(One-side Rotation)法以及求一般矩阵全部特征值的 QR 方法及一些相关的并行算法。

4.1 求解矩阵最大特征值的乘幂法

4.1.1 乘幂法及其串行算法

在许多实际问题中，只需要计算绝对值最大的特征值，而并不需要求矩阵的全部特征值。乘幂法是一种求矩阵绝对值最大的特征值的方法。记实方阵 A 的 n 个特征值为 λ_i $i=(1,2,\cdots,n)$，且满足：

$$|\lambda_1|\geq|\lambda_2|\geq|\lambda_3|\geq\cdots\geq|\lambda_n|$$

特征值 λ_i 对应的特征向量为 x_i。乘幂法的做法是：①取 n 维非零向量 v_0 作为初始向量；②对于 $k=1,2,\cdots$,做如下迭代：

$$u_k=Av_{k-1} \quad v_k=u_k/\|u_k\|_\infty$$

直至 $\|u_{k+1}\|_\infty-\|u_k\|_\infty<\varepsilon$ 为止，这时 v_{k+1} 就是 A 的绝对值最大的特征值 λ_1 所对应的特征向量 x_1。若 v_{k-1} 与 v_k 的各个分量同号且成比例，则 $\lambda_1=\|u_k\|_\infty$；若 v_{k-1} 与 v_k 的各个分量异号且成比例，则 $\lambda_1=-\|u_k\|_\infty$。若各取一次乘法和加法运算时间、一次除法运算时间、一次比较运算时间为一个单位时间，则因为一轮计算要做一次矩阵向量相乘、一次求最大元操作和一次规格化操作，所以下述乘幂法串行算法 21.1 的一轮计算时间为 $n^2+2n=O(n^2)$。

算法 21.1　单处理器上乘幂法求解矩阵最大特征值的算法

输入：系数矩阵 $A_{n\times n}$，初始向量 $v_{n\times 1}$，ε

输出：最大的特征值 *max*

```
Begin
    while (|diff|>ε) do
      (1)for i=1 to n do
        (1.1)sum=0
        (1.2)for j= 1 to n do
              sum=sum+a[i,j]*x[j]
             end for
```

(1.3)$b[i]= sum$

end for

(2)$max=|b[1]|$

(3)**for** i=2 **to** n **do**

if ($|b[i]|>max$) **then** $max=|b[i]|$ **end if**

end for

(4)**for** i=1 **to** n **do**

$x[i] =b[i]/max$

end for

(5)diff=$max-oldmax$, $oldmax=max$

end while

End

4.1.2 乘幂法并行算法

乘幂法求矩阵特征值由反复进行矩阵向量相乘来实现，因而可以采用矩阵向量相乘的数据划分方法。设处理器个数为 p，对矩阵 A 按行划分为 p 块，每块含有连续的 m 行向量，这里 $m=\lceil n/p \rceil$，编号为 i 的处理器含有 A 的第 im 至第$(i+1)m-1$ 行数据，(i=0,1, …,p-1)，初始向量 v 被广播给所有处理器。

各处理器并行地对存于局部存储器中 a 的行块和向量 v 做乘积操作，并求其 m 维结果向量中的最大值 *localmax*，然后通过归约操作的求最大值运算得到整个 n 维结果向量中的最大值 *max* 并广播给所有处理器，各处理器利用 *max* 进行规格化操作。最后通过扩展收集操作将各处理器中的 m 维结果向量按处理器编号连接起来并广播给所有处理器，以进行下一次迭代计算，直至收敛。具体算法框架描述如下：

算法 21.2　乘幂法求解矩阵最大特征值的并行算法

输入：系数矩阵 $A_{n\times n}$，初始向量 $v_{n\times 1}$，ε

输出：最大的特征值 *max*

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

while (|diff|>ε) **do**　/* diff 为特征向量的各个分量新旧值之差的最大值*/

(1)**for** i=0 **to** m-1 **do**　/*对所存的行计算特征向量的相应分量*/

(1.1)sum=0

(1.2)**for** j= 0 **to** n-1 **do**

$sum=sum+a[i,j]*x[j]$

end for

(1.3)$b[i]= sum$

end for

(2)$localmax=|b[0]|$　/*对所计算的特征向量的相应分量
求新旧值之差的最大值 *localmax* */

(3)**for** i=1 **to** m-1 **do**

if ($|b[i]|>localmax$) **then** $localmax=|b[i]|$ **end if**

end for

(4)用 Allreduce 操作求出所有处理器中 *localmax* 值的最大值 *max*
并广播到所有处理器中

(5)**for** i=0**to** m-1 **do**　/*对所计算的特征向量归一化 */

$b[i] = b[i]/max$

end for

(6)用 Allgather 操作将各个处理器中计算出的特征向量的分量的新值组合并广播到所有处理器中

(7)diff=*max-oldmax*, *oldmax=max*

end while

End

若各取一次乘法和加法运算时间、一次除法运算时间、一次比较运算时间为一个单位时间，一轮迭代的计算时间为 $mn+2m$；一轮迭代中，各处理器做一次归约操作，通信量为 1，一次扩展收集操作，通信量为 m，则通信时间为 $4t_s(\sqrt{p}-1)+(m+1)t_w(p-1)$。因此乘幂法的一轮并行计算时间为 $T_p = mn+2m+4t_s(\sqrt{p}-1)+(m+1)t_w(p-1)$。

MPI 源程序请参见所附光盘。

4.2 求对称矩阵特征值的雅可比法

4.2.1 雅可比法求对称矩阵特征值的串行算法

若矩阵 $A=[a_{ij}]$是 n 阶实对称矩阵，则存在一个正交矩阵 U，使得

$$U^TAU=D$$

其中 D 是一个对角矩阵，即

$$D=\begin{bmatrix} l_1 & 0 & \Lambda & 0 \\ 0 & l_2 & \Lambda & 0 \\ \mathrm{M} & \mathrm{M} & \mathrm{O} & \mathrm{M} \\ 0 & 0 & \Lambda & l_n \end{bmatrix}$$

这里 λ_i (i=1,2,⋯,n)是 A 的特征值，U 的第 i 列是与 λ_i 对应的特征向量。雅可比算法主要是通过正交相似变换将一个实对称矩阵对角化，从而求出该矩阵的全部特征值和对应的特征向量。因此可以用一系列的初等正交变换逐步消去 A 的非对角线元素，从而使矩阵 A 对角化。设初等正交矩阵为 $R(p,q,\theta)$，其(p,p)(q,q)位置的数据是 $\cos\theta$，(p, q)(q, p)位置的数据分别是 $-\sin\theta$ 和 $\sin\theta(p \neq q)$，其它位置的数据和一个同阶数的单位矩阵相同。显然可以得到：

$$R(p,q,\theta)\ ^TR(p,q,\theta)=I_n$$

不妨记 $B= R(p,q,\theta)^TAR(p,q,\theta)$，如果将右端展开，则可知矩阵 B 的元素与矩阵 A 的元素之间有如下关系：

$b_{pp} = a_{pp}\cos^2\theta+a_{qq}\sin^2\theta+a_{pq}\sin2\theta$　　$b_{qq} = a_{pp}\sin^2\theta+a_{qq}\cos^2\theta-a_{pq}\sin2\theta$

$b_{pq} = ((a_{qq}-a_{pp})\sin2\theta)/2+a_{pq}\cos2\theta$　　$b_{qp} = b_{pq}$

$b_{pj}= a_{pj}\cos\theta+a_{qj}\sin\theta$　　$b_{qj} = -a_{pj}\sin\theta+a_{qj}\cos\theta$

$b_{ip}= a_{ip}\cos\theta+a_{iq}\sin\theta$　　$b_{iq}= -a_{ip}\sin\theta+a_{iq}\cos\theta$

$b_{ij}= a_{ij}$

其中 $1 \le i, j \le n$ 且 $i,j \neq p,q$。因为 A 为对称矩阵，R 为正交矩阵，所以 B 亦为对称矩阵。若要求矩阵 B 的元素 b_{pq} =0，则只需令 $((a_{qq}-a_{pp})\sin2\theta)/2+a_{pq}\cos2\theta=0$，即：

$$tg2q = \frac{-a_{pq}}{(a_{qq} - a_{pp})/2}$$

在实际应用时，考虑到并不需要解出 θ，而只需要求出 sin2θ，sinθ 和 cosθ 就可以了。如果限制 θ 值在$-\pi/2 < 2\theta \leqslant \pi/2$，则可令

$$m = -a_{pq}, \quad n = \frac{1}{2}(a_{qq} - a_{pp}), \quad w = \mathrm{sgn}(n)\frac{m}{\sqrt{m^2 + n^2}}$$

容易推出：

$$\sin 2q = w, \quad \sin q = \frac{w}{\sqrt{2(1+\sqrt{1-w^2})}}, \quad \cos q = \sqrt{1-\sin^2 q}$$

利用 sin2θ，sinθ 和 cosθ 的值，即得矩阵 B 的各元素。矩阵 A 经过旋转变换，选定的非主对角元素 a_{pq} 及 a_{qp}（一般是绝对值最大的）就被消去，且其主对角元素的平方和增加了 $2a_{pq}^2$，而非主对角元素的平方和减少了 $2a_{pq}^2$，矩阵元素总的平方和不变。通过反复选取主元素 a_{pq}，并作旋转变换，就逐步将矩阵 A 变为对角矩阵。在对称矩阵中共有$(n^2-n)/2$ 个非主对角元素要被消去，而每消去一个非主对角元素需要对 $2n$ 个元素进行旋转变换，对一个元素进行旋转变换需要 2 次乘法和 1 次加法，若各取一次乘法运算时间或一次加法运算时间为一个单位时间，则消去一个非主对角元素需要 3 个单位运算时间，所以下述算法 21.3 的一轮计算时间复杂度为 $(n^2-n)/2*2n*3=3n^2(n-1)=O(n^3)$。

算法 21.3　单处理器上雅可比法求对称矩阵特征值的算法

输入：矩阵 $A_{n\times n}$，ε

输出：矩阵特征值 *Eigenvalue*

Begin

(1)**while** (max >ε) **do**

(1.1) *max*=a[1,2]

(1.2)**for** *i*=1 **to** *n* **do**

for *j*= *i*+1 **to** *n* **do**

if (| *a*[*i*,*j*]) | >*max*) **then** *max* = | *a*[*i*,*j*] |，*p*=*i*，*q*=*j*　**end if**

end for

end for

(1.3)Compute:

m=- *a*[*p*,*q*]，n=(*a*[*q*,*q*]- *a*[*p*,*p*])/2，*w*=sgn(*n*)**m*/sqrt(*m***m*+*n***n*)，

*si*2=*w*，*si*1=*w*/sqrt(2(1+ sqrt(1-*w***w*)))，co1=sqrt(1-*si*1**si*1)，

b[*p*,*p*]= *a*[*p*,*p*]*co1*co1+ *a*[*q*,*q*]**si*1**si*1+ *a*[*p*,*q*]**si*2，

b[*q*,*q*]= *a*[*p*,*p*]**si*1**si*1+ *a*[*q*,*q*]*co1*co1- *a*[*p*,*q*]**si*2，

b[*q*, *p*]=0，*b*[*p*,*q*]=0

(1.4)**for** *j*=1 **to** *n* **do**

if (($j \neq p$) and ($j \neq q$)) **then**

(i)*b*[*p*,*j*]= *a*[*p*,*j*]*co1+ *a*[*q*,*j*]**si*1

(ii)*b*[*q*,*j*]= -*a*[*p*,*j*]**si*1 + *a*[*q*,*j*]*co1

end if

end for

(1.5)**for** *i*=1 **to** *n* **do**

```
                if((i ≠ p) and (i ≠ q)) then
                    (i)b[i, p]= a[i,p]*co1+ a[i,q]*si1
                    (ii)b[i, q]= - a[i,p]*si1+ a[i,q]*co1
                end if
              end for
          (1.6)for i=1 to n do
                 for j=1 to n do
                    a[i,j]=b[i,j]
                 end for
                end for
        end while
     (2)for i=1 to n do
              Eigenvalue[i]= a[i, i]
        end for
End
```

4.2.2 雅可比法求对称矩阵特征值的并行算法

串行雅可比算法逐次寻找非主对角元绝对值最大的元素的方法并不适合于并行计算。因此，在并行处理中，我们每次并不寻找绝对值最大的非主对角元消去，而是按一定的顺序将 A 中的所有上三角元素全部消去一遍，这样的过程称为一轮。由于对称性，在一轮中，A 的下三角元素也同时被消去一遍。经过若干轮，可使 A 的非主对角元的绝对值减少，收敛于一个对角方阵。具体算法如下：

设处理器个数为 p，对矩阵 A 按行划分为 $2p$ 块，每块含有连续的 $m/2$ 行向量，记 $m=\lceil n/p \rceil$，这些行块依次记为 $A_0,A_1,\cdots,A_{2p-1}$，并将 A_{2i} 与 A_{2i+1} 存放与标号为 i 的处理器中。

每轮计算开始，各处理器首先对其局部存储器中所存的两个行块的所有行两两配对进行旋转变换，消去相应的非对角线元素。然后按图 21.1 所示规律将数据块在不同处理器之间传送，以消去其它非主对角元素。

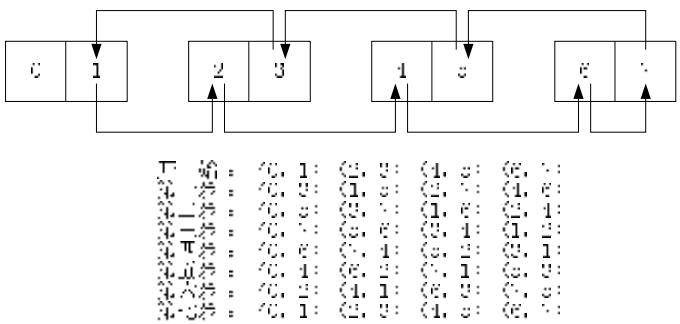

图 4.1 p=4 时的雅可比算法求对称矩阵特征值的数据交换图

这里长方形框中两个方格内的整数被看成是所移动的行块的编号。在要构成新的行块配对时，只要将每个处理器所对应的行块按箭头方向移至相邻的处理器即可，这样的传送可以在行块之间实现完全配对。当编号为 *i* 和 *j* 的两个行块被送至同一处理器时，令编号为 *i* 的行块中的每行元素和编号为 *j* 的行块中的每行元素配对，以消去相应的非主对角元素，这样在所有的行块都两两配对之后，可以将所有的非主对角元素消去一遍，从而完成一轮计算。由图 4.1 可以看出，在一轮计算中，处理器之间要 2*p*-1 次交换行块。为保证不同行块配对时可以将原矩阵 *A* 的非主对角元素消去，引入变量 *b* 记录每个处理器中的行块数据在原矩阵 *A* 中的实际行号。在行块交换时，变量 *b* 也跟着相应的行块在各处理器之间传送。

处理器之间的数据块交换存在如下规律：

0 号处理器前一个行块(简称前数据块，后一个行块简称后数据块)始终保持不变，将后数据块发送给 1 号处理器，作为 1 号处理器的前数据块。同时接收 1 号处理器发送的后数据块作为自己的后数据块。

p-1 号处理器首先将后数据块发送给编号为 *p*-2 的处理器，作为 *p*-2 号处理器的后数据块。然后将前数据块移至后数据块的位置上，最后接收 *p*-2 号处理器发送的前数据块作为自己的前数据块。

编号为 my_rank 的其余处理器将前数据块发送给编号为 my_rank+1 的处理器，作为 my_rank+1 号处理器的前数据块。将后数据块发送给编号为 my_rank-1 的处理器，作为 my_rank-1 号处理器的后数据块。

为了避免在通信过程中发生死锁，奇数号处理器和偶数号处理器的收发顺序被错开。使偶数号处理器先发送后接收；而奇数号处理器先将数据保存在缓冲区中，然后接收偶数号处理器发送的数据，最后再将缓冲区中的数据发送给偶数号处理器。数据块传送的具体过程描述如下：

算法 21.4　雅可比法求对称矩阵特征值的并行过程中处理器之间的数据块交换算法

输入：矩阵 *A* 的行数据块和向量 *b* 的数据块分布于各个处理器中

输出：在处理器阵列中传送后的矩阵 *A* 的行数据块和向量 *b* 的数据块

Begin

对所有处理器 my_rank(my_rank=0,…, *p*-1)同时执行如下的算法:

/*矩阵 *A* 和向量 *b* 为要传送的数据块*/

(1)**if** (my-rank=0) **then**　/*0 号处理器*/

(1.1)将后数据块发送给 1 号处理器

(1.2)接收 1 号处理器发送来的后数据块作为自己的后数据块

end if

(2)**if** ((my-rank=*p*-1) and (my-rank mod 2 ≠ 0)) **then**　/*处理器 *p*-1 且其为奇数*/

(2.1)**for** *i*=*m*/2 **to** *m*-1 **do**　/* 将后数据块保存在缓冲区 *buffer* 中*/

for *j*=0 **to** *n*-1 **do**

buffer[*i*-*m*/2,*j*]=*a*[*i*,*j*]

end for

end for

(2.2)**for** *i*=*m*/2 **to** *m*-1 **do**

buf [*i*-*m*/2] =*b*[*i*]

end for

(2.3)**for** *i*=0 **to** *m*/2-1 **do** /*将前数据块移至后数据块的位置上*/

for *j*=0 **to** *n*-1 **do**

a[*i*+*m*/2,*j*]=*a*[*i*,*j*]

end for

end for

(2.4)**for** *i*=0 **to** *m*/2-1 **do**

b[*i*+*m*/2] =*b*[*i*]

end for

(2.5)接收 *p*-2 号处理器发送的数据块作为自己的前数据块

(2.6)将 *buffer* 中的后数据块发送给编号为 *p*-2 的处理器

end if

(3)**if** ((my-rank=*p*-1) and (my-rank mod 2=0)) **then** /*处理器 *p*-1 且其为偶数*/

(3.1)将后数据块发送给编号为 *p*-2 的处理器

(3.2)**for** *i*=0 **to** *m*/2-1 **do** /*将前数据块移至后数据块的位置上*/

for *j*=0 **to** *n*-1 **do**

a[*i*+*m*/2,*j*]=*a*[*i*,*j*]

end for

end for

(3.3)**for** *i*=0 **to** *m*/2-1 **do**

b[*i*+*m*/2] =*b*[*i*]

end for

(3.4)接收 *p*-2 号处理器发送的数据块作为自己的前数据块

end if

(4)**if** ((my-rank ≠ *p*-1) and (my-rank ≠ 0)) **then** /*其它的处理器*/

(4.1)**if** (my-rank mod 2=0) **then** **/***偶数号处理器*/

(i)将前数据块发送给编号为 my_rank+1 的处理器

(ii)将后数据块发送给编号为 my_rank-1 的处理器

(ii)接收编号为 my_rank-1 的处理器发送的数据块作为自己的前数据块

(iv)接收编号为 my_rank+1 的处理器发送的数据块作为自己的后数据块

else /*奇数号处理器*/

(v)**for** *i*=0 **to** *m*-1 **do** /* 将前后数据块保存在缓冲区 *buffer* 中*/

for *j*=0 **to** *n*-1 **do**

buffer[*i*,*j*]=*a*[*i*,*j*]

end for

end for

(vi)**for** *i*=0 **to** *m*-1 **do**

buf[*i*] =*b*[*i*]

end for

(vii)接收编号为 my_rank-1 的处理器发送的数据块作为自己的前数据块

(viii)接收编号为 my_rank+1 的处理器发送的数据块作为自己的后数据块

(ix)将存于 *buffer* 中的前数据块发送给编号为 my_rank+1 的处理器

(x)将存于 *buffer* 中的后数据块发送给编号为 my_rank-1 的处理器

end if

end if

End

各处理器并行地对其局部存储器中的非主对角元素 a_{ij} 进行消去，首先计算旋转参数并对第 i 行和第 j 行两行元素进行旋转行变换。然后通过扩展收集操作将相应的旋转参数及第 i 列和第 j 列的列号按处理器编号连接起来并广播给所有处理器。各处理器在收到这些旋转参数和列号之后，按 0,1,⋯,p-1 的顺序依次读取旋转参数及列号并对其 m 行中的第 i 列和第 j 列元素进行旋转列变换。

经过一轮计算的 2p-1 次数据交换之后，原矩阵 A 的所有非主对角元素都被消去一次。此时，各处理器求其局部存储器中的非主对角元素的最大元 *localmax*，然后通过归约操作的求最大值运算求得将整个 n 阶矩阵非主对角元素的最大元 *max*，并广播给所有处理器以决定是否进行下一轮迭代。具体算法框架描述如下：

算法 21.5　雅可比法求对称矩阵特征值的并行算法

输入：矩阵 $A_{n\times n}$，ε

输出：矩阵 A 的主对角元素即为 A 的特征值

Begin

对所有处理器 my_rank(my_rank=0,⋯, p-1)同时执行如下的算法:

(a)**for** i=0 **to** m-1 **do**

$b[i]$ =myrank*m+i　/* b 记录处理器中的行块数据在原矩阵 A 中的实际行号*/

end for

(b)**while** (| *max* | >ε) **do**　/* *max* 为 A 中所有非对角元最大的绝对值*/

(1)**for** i=my_rank*m **to** (my_rank+1)*m-2 **do**

/*对本处理器内部所有行两两配对进行旋转变换*/

for j=i+1 **to** (my_rank+1)*m-1 **do**

(1.1)r=i mod m ,　t=j mod m　/*i, j 为进行旋转变换行(称为主行)的实际行号, r, t 为它们在块内的相对行号*/

(1.2)**if** ($a[r,j] \neq 0$) **then**　/*对四个主元素的旋转变换*/

(i)Compute:

f=-$a[r,j]$, g=($a[t,j]$- $a[r,i]$)/2, h=*sgn*(g)*f/*sqrt*(f*f+g*g) ,

*sin*2=h , *sin*1=h/*sqrt*(2*(1+*sqrt*(1-h*h))) , *cos*1=*sqrt*(1-*sin*1*sin1)，

bpp= $a[r,i]$*cos1*cos1+$a[t,j]$*sin1*sin1+$a[r,j]$*sin2，

bqq= $a[r,i]$* sin1*sin1+$a[t,j]$* cos1*cos1-$a[r,j]$*sin2，

bpq=0 , bqp=0

(ii)**for** v=0 **to** n-1 **do**　/*对两个主行其余元素的旋转变换*/

if (($v \neq i$) and ($v \neq j$)) **then**

$br[v]$ = $a[r,v]$*cos1 + $a[t,v]$*sin1

$a[t,v]$ = -$a[r,v]$* sin1 + $a[t,v]$* cos1

end if

end for

(iii)**for** *v*=0 **to** *n*-1 **do**

if ((*v* ≠ *i*) and (*v* ≠ *j*)) **then**

a[*r*,*v*]=*br*[*v*]

end if

end for

(iv)**for** *v*=0 **to** *m*-1 **do**

/*对两个主列在本处理器内的其余元素的旋转变换*/

if ((*v* ≠ *r*) and (*v* ≠ *t*)) **then**

bi[*v*] = *a*[*v*,*i*]**cos*1 + *a*[*v*,*j*]**sin*1

a[*v*,*j*]= - *a*[*v*,*i*]* *sin*1 + *a*[*v*,*j*]* *cos*1

end if

end for

(v)**for** *v*=0 **to** *m*-1**do**

if ((*v* ≠ *r*) and (*v* ≠ *t*)) **then**

a[*v*,*i*]= *bi*[*v*]

end if

end for

(vi)Compute:

a[*r*,*i*]=*bpp* , *a*[*t*,*j*]=*bqq* , *a*[*r*,*j*]=*bpq* , *a*[*t*,*i*]=*bqp*，

/*用 *temp*1 保存本处理器主行的行号和旋转参数*/

*temp*1[0]=*sin*1, *temp*1[1]=*cos*1，

*temp*1[2]=(**float**)*i* ,*temp*1[3]= (**float**)*j*

else

(vii)Compute:

*temp*1[0]=0, *temp*1[1]= 0，

*temp*1[2]= 0 , *temp*1[3]= 0

end if

(1.3)将所有处理器 *temp*1 中的旋转参数及主行的行号

按处理器编号连接起来并广播给所有处理器,存于 *temp*2 中

(1.4)*current*=0

(1.5)**for** *v*=1 **to** *p* **do**

/*根据 *temp*2 中的其它处理器的旋转参数及主行的行号对相关的列在本处理器的部分进行旋转变换*/

(i)Compute:

*s*1=*temp*2[(*v*-1)*4+0] , *c*1=*temp*2[(*v*-1)*4+1]，

*i*1=(**int**)*temp*2[(*v*-1)*4+2], *j*1=(**int**)*temp*2[(*v*-1)*4+3]

(ii)**if** (*s*1、*c*1、 *i*1、 *j*1 中有一不为 0) **then**

if (my-rank ≠ *current*) **then**

for *z*=0 **to** *m*-1 **do**

zi[*z*]=*a*[*z*,*i*1]**c*1 + *a*[*z*,*j*1]**s*1

a[*z*,*j*1]=- *a*[*z*,*i*1]**s*1 + *a*[*z*,*j*1]**c*1

end for

for *z*=0 **to** *m*-1 **do**

a[*z*,*i*1]= *zi*[*z*]

end for

end if

end if

(iii)*current*=*current*+1

end for

end for

end for

(2)**for** *counter*=1 **to** 2*p*-2 **do**

/*进行 2*p*-2 次处理器间的数据交换, 并对交换后处理器中所有行两两配对进行旋转变换*/

(2.1)Data_exchange( ) /*处理器间的数据交换*/

(2.2)**for** *i*=0 **to** *m*/2-1 **do**

for *j*=*m*/2 **to** *m*-1 **do**

(i) **if** (*a*[*i*,*b*[*j*]] ≠ 0) **then** /*对四个主元素的旋转变换*/

①Compute:

f= -*a*[*i*,*b*[*j*]]，*g*=(*a*[*j*,*b*[*j*]]- *a*[*i*,*b*[*i*]])/2，

h=*sgn*(*g*)**f*/*sqrt*(*f***f*+*g***g*)，

*sin*2=*h*, *sin*1=*h*/*sqrt*(2*(1+*sqrt*(1-*h***h*)))，

*cos*1=*sqrt*(1-*sin*1**sin*1)，

bpp= *a*[*i*,*b*[*i*]]**cos*1**cos*1+ *a*[*j*,*b*[*j*]]**sin*1*s*in*1+*a*[*i*,*b*[*j*]]**sin*2，

bqq= *a*[*i*,*b*[*i*]]* *sin*1**sin*1+*a*[*j*,*b*[*j*]]* *cos*1**cos*1-*a*[*i*,*b*[*j*]]**sin*2，

bpq=0, *bqp*=0

②**for** *v*=0 **to** *n*-1 **do** /*对两个主行其余元素的旋转变换*/

if ((*v* ≠ *b*[*i*]) and (*v* ≠ *b*[*j*])) **then**

br[*v*] = *a*[*i*,*v*]**cos*1 + *a*[*j*,*v*]**sin*1

a[*j*,*v*] = -*a*[*i*,*v*]* *sin*1 + *a*[*j*,*v*]* *cos*1

end if

end for

③**for** *v*=0 **to** *n*-1 **do**

if ((*v* ≠ *b*[*i*]) and (*v* ≠ *b*[*j*])) **then**

a[*i*,*v*]=*br*[*v*]

end if

end for

④**for** *v*=0 **to** *m*-1 **do**

/*对本处理器内两个主列的其余元素旋转变换*/

if ((*v* ≠ *i*) and (*v* ≠ *j*)) **then**

bi[*v*] = *a*[*v*, *b*[*i*]]**cos*1 + *a*[*v*, *b*[*j*]]**sin*1

a[*v*, *b*[*j*]] = - *a*[*v*, *b*[*i*]]* *sin*1 + *a*[*v*, *b*[*j*]]* *cos*1

end if

end for

⑤**for** *v*=0 **to** *m*-1 **do**

if ((*v* ≠ *i*) and (*v* ≠ *j*)) **then**

a[*v*, *b*[*i*]]=*bi*[*v*]

end if

end for

⑥Compute:

a[*i*, *b*[*i*]]=*bpp* , *a*[*j*, *b*[*j*]]=*bqq* ,

a[*i*, *b*[*j*]]=*bpq* , *a*[*j*, *b*[*i*]]=*bqp*

/*用 *temp*1 保存本处理器主行的行号和旋转参数*/

*temp*1[0]=*sin*1 , *temp*1[1]=*cos*1,

*temp*1[2]=(**float**)*b*[*i*] , *temp*1[3]= (**float**)*b*[*j*]

else

⑦Compute:

*temp*1[0]=0,*temp*1[1]= 0,

*temp*1[2]= 0,*temp*1[3]= 0

end if

(ii)将所有处理器 *temp*1 中的旋转参数及主行的行号按处理器编号连接起来并广播给所有处理器,存于 *temp*2 中

(iii)*current*=0

(iv)**for** *v*=1 **to** *p* **do**

/*根据 *temp*2 中的其它处理器的旋转参数及主行的行号对相关的列在本处理器的部分进行旋转变换*/

①Compute:

*s*1=*temp*2[(*v*-1)*4+0], *c*1=*temp*2[(*v*-1)*4+1],

*i*1=(**int**)*temp*2[(*v*-1)*4+2], *j*1=(**int**)*temp*2[(*v*-1)*4+3]

②**if** (*s*1、*c*1、 *i*1、 *j*1 中有一不为 0) **then**

if (my-rank ≠ *current*) **then**

for *z*=0 **to** *m*-1 **do**

zi[*z*]=*a*[*z*,*i*1]**c*1 + *a*[*z*,*j*1]**s*1

a[*z*,*j*1]=- *a*[*z*,*i*1]*s*1 + *a*[*z*,*j*1]**c*1

end for

for *z*=0 **to** *m*-1 **do**

a[*z*,*i*1]= *zi*[*z*]

end for

end if

end if

③*current*=*current*+1

end for

end for

end for

end for

(3)Data-exchange( )

/*进行一轮中的最后一次处理器间的数据交换,使数据回到原来的位置*/

(4)*localmax*=*max* /**localmax* 为本处理器中非对角元最大的绝对值*/

(5)**for** *i*=0 **to** *m*-1 **do**

for *j*=0 **to** *n*-1 **do**

if ((*m**my-rank+*i*) ≠ *j*) **then**

if (| *a*[*i*,*j*] | > *localmax*) **then** *localmax*= | *a*[*i*,*j*] | **end if**

end if

end for

end for

(6)通过 Allreduce 操作求出所有处理器中 *localmax* 的最大值为新的 *max* 值

end while

End

在上述算法中，每个处理器在一轮迭代中要处理 2 p 个行块对，由于每一个行块对含有 $m/2$ 行，因而对每一个行块对的处理要有$(m/2)^2=m^2/4$ 个行的配对，即消去 $m^2/4$ 个非主对角元素．每消去一个非主对角元素要对同行 n 个元素和同列 n 个元素进行旋转变换．由于按行划分数据块，对同行 n 个元素进行旋转变换的过程是在本处理器中进行的．而对同列 n 个元素进行旋转变换的过程则分布在所有处理器中进行．但由于所有处理器同时在为其它处理器的元素消去过程进行列的旋转变换，对每个处理器而言，对列元素进行旋转变换的处理总量仍然是 n 个元素。 因此，一个处理器每消去一个非主对角元素共要对 $2n$ 个元素进行旋转变换。而对一个元素进行旋转变换需要 2 次乘法和 1 次加法，若取一次乘法运算时间或一次加法运算时间为一个单位时间，则其需要 3 个单位运算时间，即一轮迭代的计算时间为 $T_1=3\times 2p\times 2n\times m^2/4=3n^3/p$；在每轮迭代中，各个处理器之间以点对点的通信方式相互错开交换数据 $2p$-1 次，通信量为 $mn+m$，扩展收集操作 $n(n-1)/(2p)$次，通信量为 4，另外有归约操作 1 次，通信量为 1，从而不难得出上述算法求解过程中的总通信时间为：

$$T_2=[4t_s+m(n+1)t_w](4p-2)+[n(n-1)/p+2]t_s(\sqrt{p}-1)+[2n(n-1)/p+1]t_w(p-1)$$

因此雅可比算法求对矩阵特征值的一轮并行计算时间为$T_p=T_1+T_2$。我们的大量实验结果说明，对于 n 阶方阵，用上述算法进行并行计算，一般需要不超过 $O(logn)$轮就可以收敛。

MPI 源程序请参见章末附录。

4.3 求对称矩阵特征值的单侧旋转法

4.3.1 单侧旋转法的算法描述

求解对称方阵特征值及特征向量的雅可比算法在每次消去计算前都要对非主对角元素选择最大元，导致非实际计算开销很大。在消去计算时，必须对方阵同时进行行、列旋转变换，这称之为**双侧旋转**(Two-side Rotation)变换。在双侧旋转变换中，方阵的行、列方向都有数据相关关系，使得整个计算中的相关关系复杂，特别是在对高阶矩阵进行特征值求解时，增加了处理器间的通信开销。针对传统雅可比算法的缺点，可以将双侧旋转改为单侧旋转，得出一种求对称矩阵特征值的快速算法。这一算法对矩阵仅实施列变换，使得数据相关关系仅在同列之间，因此方便数据划分，适合并行计算，称为**单侧旋转法**（One-side Rotation）。若 A 为一对称方阵， λ 是 A 的特征值，x 是 A 的特征向量，则有：$Ax=\lambda x$。又 $A=A^T$，所以 $A^TAx=A\lambda x=\lambda\lambda x$，这说明了 λ^2 是 A^TA 的特征值，x 是 A^TA 的特征向量 ，即 A^TA 的特征值是 A 的特征值的平方，并且它们的特征向量相同。

我们使用 18.7.1 节中所介绍的 Givens 旋转变换对 A 进行一系列的列变换，得到方阵 Q 使其各列两两正交，即 $AV=Q$，这里 V 为表示正交化过程的变换方阵。 因 $Q^TQ=(AV)^TAV=V^TA^TAV=diag(\delta_1,\delta_2,\cdots,\delta_n)$ 为 n 阶对角方阵，可见这里 $\delta_1,\delta_2,\cdots,\delta_n$ 是矩阵 A^TA 的特征

值，V 的各列是 A^TA 的特征向量。由此可得：$d_i = l_i^2$ (i=1,2, …,n)。设 Q 的第 i 列为 q_i, 则 $q_i^Tq_i = \delta_i$, $\|q_i\|_2 = \sqrt{q_i^Tq_i} = \sqrt{d_i} = |l_i|$。因此在将 A 进行列变换得到各列两两正交的方阵 Q 后，其各列的谱范数$\|q_i\|_2$即为 A 的特征值的绝对值。记 V 的第 i 列为 v_i， $AV=Q$，所以 $Av_i = q_i$。又因为 v_i 为 A 的特征向量，所以 $Av_i = \lambda_i v_i$，即推得 $\lambda_i v_i = q_i$。因此 λ_i 的符号可由向量 q_i 及 v_i 的对应分量是否同号判别，实际上在具体算法中我们只要判断它们的第一个分量是否同号即可。若相同，则 λ_i 取正值，否则取负值。

求对称矩阵特征值的单侧旋转法的串行算法如下：

算法 21.6　求对称矩阵特征值的单侧旋转法

输入：对称矩阵 A，精度 e

输出：矩阵 A 的特征值存于向量 B 中

```
Begin
  (1)while (p> ε )
        p=0
        for i=1 to n do
          for j=i+1 to n do
            (1.1 )sum[0]=0,   sum[1]=0,   sum[2]=0
            (1.2 )for k=1 to n do
                    sum[0]= sum[0]+a[k,i]*a[k, j]
                    sum[1]= sum[1]+ a[k,i]* a[k,i]
                    sum[2]= sum[2]+ a[k, j]* a[k, j]
                  end for
            (1.3 )if ( |sum[0]| > ε ) then
                    (i) aa=2*sum[0]
                       bb=sum[1]-sum[2]
                       rr=sqrt(aa*aa+bb*bb)
                    (ii) if (bb≥0) then
                              c=sqrt((bb+rr)/(2*rr))
                              s=aa/(2*rr*c)
                       else
                              s=sqrt((rr-bb)/(2*rr))
                              c=aa/(2*rr*s)
                       end if
                    (iii) for k=1 to n do
                              temp[k]=c*a[k,i]+s*a[k,j]
                              a[k,j]=[-s]*a[k,i]+c*a[k,j]
                          end for
                    (iv) for k=1 to n do
                              temp1[k]= c*e[k,i]+s*e[k,j]
                              e[k,j]=[-s]*e[k,i]+c*e[k,j]
                          end for
                    (v) for k=1 to n do
```

```
                    a[k,i]= temp[k]
                end for
            (vi) for k=1 to n do
                    e[k,i]= temp1[k]
                end for
            (vii) if (|sum[0]|>p) then p=|sum[0]| end if
        end if
    end for
  end for
end while
(2)for i=1 to n do
    su=0
    for j=1 to n do
      su=su+a[j,i]* a[j,i]
    end for
    B[j]=sqrt[su]
      if (e[0,j]*a[0,j]<0) then B[j]= - B[j]   endif
end for
End
```

上述算法的一轮迭代需进行 $n(n-1)/2$ 次旋转变换，若取一次乘法或一次加法运算时间为一个单位时间，则一次旋转变换要做 3 次内积运算而消耗 $6n$ 个单位时间；与此同时，两列元素进行正交计算还需要 $12n$ 个单位时间，所以奇异值分解的一轮计算时间复杂度为 $n(n-1)/2*(12\ n+6n)= 9n(n-1)\ n=O(n^3)$。

4.3.2 求对称矩阵特征值的单侧旋转法的并行计算

在求对称矩阵特征值的单侧旋转法的计算中，主要的计算是矩阵的各列正交化过程.为了进行并行计算，我们对 n 阶对称矩阵 A 按行划分为 p 块(p 为处理器数)，每块含有连续的 q 行向量，这里 $q=n/p$，第 i 块包含 A 的第 $i\times q$, …,$(i+1)\times q-1$ 行向量，其数据元素被分配到第 i 号处理器上($i=0,1,\cdots,p-1$)。E 矩阵取阶数为 n 的单位矩阵 I，按同样方式进行数据划分，每块含有连续的 q 行向量。对矩阵 A 的每一个列向量而言，它被分成 p 个长度为 q 的子向量，分布于 p 个处理器之中，它们协同地对各列向量做正交计算。在对第 i 列与第 j 列进行正交计算时，各个处理器首先求其局部存储器中的 q 维子向量[i,j]、[i,i]、[j,j]的内积，然后通过归约操作的求和运算求得整个 n 维向量对[i,j]、[i,i]、[j,j]的内积并广播给所有处理器，最后各处理器利用这些内积对其局部存储器中的第 i 列及第 j 列 q 维子向量的元素做正交计算。算法 21.7 按这样的方式对所有列对正交化一次以完成一轮运算，重复进行若干轮运算，直到迭代收敛为止。在各列正交后，编号为 0 的处理器收集各处理器中的计算结果，由 0 号处理器计算矩阵的特征值. 具体算法框架描述如下：

算法 21.7　求对称矩阵特征值的并行单侧旋转算法

输入：对称矩阵 A，精度 e

输出：对称矩阵 A 的特征值存于向量 B 中

```
Begin
  对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法：
  (a) while (not convergence) do
      (1)k=0
```

(2)**for** *i*=0 **to** *n*-1 **do**

(2.1)**for** *j*=*i*+1 **to** *n*-1 **do**

(i)*sum*[0]=0, *sum*[1]=0, *sum*[2]=0

(ii)计算本处理器所存的列子向量 *a*[*, *i*]、*a*[*, *j*]的内积赋值给 *ss*[0]

(iii)计算本处理器所存的列子向量 *a*[*, *i*]、*a*[*, *i*]的内积赋值给 *ss*[1]

(iv)计算本处理器所存的列子向量 *a*[*, *j*]、*a*[*, *j*]的内积赋值给 *ss*[2]

(v)通过规约操作实现:

①计算所有处理器中 *ss*[0]的和赋给 *sum* [0]

②计算所有处理器中 *ss*[1]的和赋给 *sum* [1]

③计算所有处理器中 *ss*[2]的和赋给 *sum*[2]

④将 *sum* [0]、*sum* [1]、*sum* [2] 的值广播到所有处理器中

(vi)**if** (| *sum*(0) | >*e*) **then** /*各个处理器并行进行对 *i* 和 *j* 列正交化*/

①*aa*=2**sum*[0]

②*bb*=*sum*[1]-*sum*[2]

③*rr*=sqrt(*aa***aa*+*bb***bb*)

④**if** (*bb*>=0) **then**

c=sqrt((*bb*+*rr*)/(*2***rr*))

s=*aa*/(2**rr***c*)

else

s=sqrt((*rr*-*bb*)/(*2***rr*))

c=*aa*/(*2***rr***s*)

end if

⑤**for** *v*=0 **to** *q*-1 **do**

temp[*v*]=*c***a*[*v, i*]+*s***a*[*v, j*]

a[*v, j*]=(-*s*)**a*[*v, i*]+*c***a*[*v, j*]

end for

⑥**for** *v*=0 **to** *z*-1 **do**

*temp*1[*v*]=*c***e*[*v, i*]+*s***e*[*v, j*]

e[*v,j*]=(-*s*)**e*[*v, i*]+*c***e*[*v, j*]

end for

⑦**for** *v*=0 **to** *q*-1 **do**

a[*v,i*]=*temp*[*v*]

end for

⑧**for** *v*=0 **to** *z*-1 **do**

e[*v,i*]=*temp*1[*v*]

end for

⑨*k*＝ *k+1*

end if

end for

end for

end while

0 号处理器从其余各处理器中接收子矩阵 a 和 e，得到经过变换的矩阵 A 和 I：

/*0 号处理器负责计算特征值*/

(b) **for** *i*=1 **to** *n* **do**

(1)*su*=0
(2)**for** *j*=1 **to** *n* **do**
su=*su*+*A*[*j*,*i*]* *A*[*j*,*i*]
end for
(3)*B*[*j*]=sqrt(*su*)
(4)**if** (I[0,j]*a[0,j]<0) **then** *B*[*j*]= - *B*[*j*] **endif**
end for
End

上述算法的一轮迭代要进行 $n(n-1)/2$ 次旋转变换，而一次旋转变换需要做 3 次内积运算，若取一次乘法或一次加法运算时间为一个单位时间，则需要 $6q$ 个单位时间，另外，还要对四列子向量中元素进行正交计算花费 $12q$ 个单位时间，所以一轮迭代需要的计算时间 $T_1=n(n-1)6q$；内积计算需要通信，一轮迭代共做归约操作 $n(n-1)/2$ 次，每次通信量为 3，因而通信时间 $T_2=[2t_s(\sqrt{p}-1)+3t_w(p-1)]*n(n-1)/2$。由此得出一轮迭代的并行计算时间 $T_p=T_1+T_2$。

MPI 源程序请参见所附光盘。

4.4 求一般矩阵全部特征值的 QR 方法

4.4.1 QR 方法求一般矩阵全部特征值的串行算法

在 18.6 节中，我们介绍了对一个 n 阶方阵 A 进行 QR 分解的并行算法，它可将 A 分解成 A=QR 的形式，其中 R 是上三角矩阵，Q 是正交矩阵，即满足 $Q^TQ=I$。利用 QR 方法也可求方阵特征值，它的基本思想是：

首先令 $A_1=A$，并对 A_1 进行 QR 分解，即： $A_1=Q_1R_1$；然后对 A_1 作如下相似变换：$A_2=Q_1^{-1}A_1Q_1=Q_1^{-1}Q_1R_1Q_1=R_1Q_1$，即将 R_1Q_1 相乘得到 A_2；再对 A_2 进行 QR 分解得 $A_2=Q_2R_2$，然后将 R_2Q_2 相乘得 A_3。反复进行这一过程，即得到矩阵序列：$A_1, A_2, \cdots, A_m, A_{m+1}, \cdots$，它们满足如下递推关系：$A_i=Q_iR_i$ ； $A_{i+1}=R_iQ_i$ (i=1,2, …, m,…)其中 Q_i 均为正交阵，R_i 均为上三角方阵。这样得到的矩阵序列 $\{A_i\}$ 或者将收敛于一个以 A 的特征值为对角线元素的上三角矩阵，形如：

$$\begin{pmatrix} l_1 & * & * & * & \Lambda & * \\ & l_2 & * & * & \Lambda & * \\ & & l_3 & * & \Lambda & * \\ & & & \Lambda & \Lambda & \Lambda \\ & & & & & l_n \end{pmatrix}$$

或者将收敛于一个特征值容易计算的块上三角矩阵。

算法 21.8　单处理器上 QR 方法求一般矩阵全部特征值的算法

输入：矩阵 $A_{n\times n}$,单位矩阵 Q，ε

输出：矩阵特征值 *Eigenvalue*

Begin

(1) **while** ((p>ε) and (*count*<1000)) **do**
(1.1)p=0 , *count* = *count* +1
(1.2)QR_DECOMPOSITION(*A*,*Q*,*R*)　　/*算法 18.11 对矩阵 *A* 进行 QR 分解

*/

(1.3)MATRIX_TRANSPOSITION(*Q*)　/*由于算法 18.11 对 A 进行 QR 分解只得到 Q^{-1}，因而要使用算法 18.1 对矩阵 Q^{-1} 进行转置，得到$(Q^{-1})^{T}=(Q^{-1})^{-1}$=Q*/

(1.4)*A*=MATRIX_PRODUCT(*R*,*Q*)　/*算法 18.5 将矩阵 *RQ* 相乘，得到新的 *A* 矩阵 */

(1.5)**for** *i*=1 to *n* **do**

for *j*=1 to *i*-1**do**

if ($|a[i,j]| > p$) **then**　$p=|a[i,j]|$ **end if**

end for

end for

end while

(2) **for** *i*=1 to *n* **do**

Eigenvalue[*i*]=*a*[*i*,*i*]

end for

End

显然，QR 分解求矩阵特征值算法的一轮时间复杂度为 QR 分解、矩阵转置和矩阵相乘算法的时间复杂度之和。

4.4.2 QR 方法求一般矩阵全部特征值的并行算法

并行 QR 分解求矩阵特征值的思想就是反复运用并行 QR 分解和并行矩阵相乘算法进行迭代，直到矩阵序列$\{A_i\}$收敛于一个上三角矩阵或块上三角矩阵为止。具体的并行算法描述如下：

算法 21.9　QR 方法求一般矩阵全部特征值的并行算法

输入：矩阵 $A_{n\times n}$,单位矩阵 *Q*，ε

输出：矩阵特征值 *Eigenvalue*

Begin

对所有处理器 my_rank(my_rank=0,…, *p*-1)同时执行如下的算法:

(a) **while** ((*p*>ε) and (*count*<1000)) **do**

(1)*p*=0 , *count* = *count* +1

/* 对矩阵 *A* 进行 QR 分解*/

(2)**if**(my_rank=0) **then**/*0 号处理器*/

(2.1)**for** *j*=0 **to** *m*-2 **do**

(i)**for** *i*=*j*+1 **to** *m*-1 **do**

Turnningtransform( )　/*旋转变换*/

end for

(ii)将旋转变换后的 A 和 Q 的第 *j* 行传送到第 1 号处理器

end for

(2.2)将旋转变换后的 A 和 Q 的第 *m*-1 行传送到第 1 号处理器

end if

(3)**if** ((my_rank>0) and (my_rank<(*p*-1))) **then**　/*中间处理器*/

(3.1)**for** *j*=0 **to** my_rank***m*-1 **do**

(i)接收左邻处理器传送来的 A 和 Q 子块中的第 *j* 行

(ii)**for** *i*=0 **to** *m*-1 **do**

Turnningtransform( ) /*旋转变换*/

end for

(iii)将旋转变换后的 A 和 Q 子块中的第 *j* 行传送到右邻处理器

end for

(3.2)**for** *j*=0 **to** *m*-2 **do**

(i)*z*=my_rank**m*

(ii)**for** *i*=*j*+1 **to** *m*-1 **do**

Turnningtransform( ) /*旋转变换*/

end for

(iii)将旋转变换后的 A 和 Q 子块中的第 *j* 行传送到右邻处理器

end for

(3.3)将旋转变换后的 A 和 Q 子块中的第 *m*-1 行传送到右邻处理器

end if

(4)**if** (my_rank= (*p*-1)) **then** /*p-1 号处理器*/

(4.1)**for** *j*=0 **to** my_rank**m*-1 **do**

(i)接收左邻处理器传送来的 A 和 Q 子块中的第 *j* 行

(ii)**for** *i*=0 **to** *m*-1 **do**

Turnningtransform( ) /*旋转变换*/

end for

end for

(4.2)**for** *j*=0 **to** *m*-2 **do**

for *i*=*j*+1 **to** *m*-1 **do** *f*

Turnningtransform( ) /*旋转变换*/

end for

end for

end if

(5)p-1 号处理器对矩阵 Q 进行转置

(6)p-1 号处理器分别将矩阵 R 和矩阵 Q 划分为大小为 m*M 和 M*m 的 p 块子阵，依次发送给 0 至 p-2 号处理器

(7)使用算法 18.6 对矩阵 *RQ* 进行并行相乘得到新的 *A* 矩阵

(8)**for** *i*=1 to *n* **do** /* 求出 A 中元素的最大绝对值赋予 *p*/

for *j*=1 to *i*-1**do**

if ($|a[i,j]| > p$) **then** $p=|a[i,j]|$ **end if**

end for

end for

end while

(b)**for** *i*=1 to *n* **do**

Eigenvalue[*i*] =*a*[*i*,*i*]

end for

End

在上述算法的一轮迭代中，实际上是使用算法 18.12、18.1、18.6 并行地进行矩阵的 QR 分解、矩阵的转置、矩阵的相乘各一次，因此，一轮迭代的计算时间为上述算法的时间复杂度之和。

MPI 源程序请参见所附光盘。

4.5 小结

矩阵的特征值与特征向量在工程技术上应用十分广泛，在诸如系统稳定性问题和数字信号处理的 *K-L* 变换中，它都是最基本、常用的算法之一。本章主要讨论了矩阵特征值的几种基本算法，关于该方面的其它讲解与分析读者可参考文献[1]、[2]。

参考文献

[1]. 陈国良 编著．并行算法的设计与分析（修订版）．高等教育出版社，2002.11
[2]. 陈国良,陈峻 编著．VLSI 计算理论与并行算法．中国科学技术大学出版社，1991

附录 求对称矩阵特征值的雅可比并行算法 MPI 源程序

源程序 cjacobi.c

```
#include "stdio.h"
#include "stdlib.h"
#include "mpi.h"
#include "math.h"
#include "string.h"
#define E 0.00001
#define min -1
#define intsize sizeof(int)
#define charsize sizeof(char)
#define floatsize sizeof(float)
/*A 是一个 N*N 的对称矩阵*/
#define A(x,y) A[x*N+y]
/*I 是一个 N*N 的单位矩阵*/
#define I(x,y) I[x*N+y]
#define a(x,y) a[x*N+y]
#define e(x,y) e[x*N+y]
#define b(x) b[x]
#define buffer(x,y) buffer[x*N+y]
#define buffee(x,y) buffee[x*N+y]

int M,N;
int *b;
int m,p;
int myid,group_size;
float *A,*I;
float *a,*e;
float max;
float sum;
MPI_Status status;

float sgn(float v)
{
    float f;
    if (v>=0)    f=1;
    else   f=-1;
    return f;
}

void read_fileA()
{
    int i,j;
    FILE *fdA;

    fdA=fopen("dataIn.txt","r");
    fscanf(fdA,"%d %d", &M, &N);
    if(M != N)
    {
        puts("The input is error!");
        exit(0);
    }
    m=N/p; if (N%p!=0) m++;
    A=(float*)malloc(floatsize*N*m*p);
    I=(float*)malloc(floatsize*N*m*p);
    for(i = 0; i < M; i ++)
```

```
            for(j = 0; j < M; j ++)
                fscanf(fdA, "%f", A+i*M+j);
        fclose(fdA);
        printf("Input of file \"dataIn.txt\"\n");
        printf("%d\t %d\n",M, N);
        for(i=0;i<M;i++)
        {
            for(j=0;j<N;j++)
                printf("%f\t",A(i,j));
            printf("\n");
        }
        for(i=0;i<N;i++)
        {
            for(j=0;j<N;j++)
                if (i==j) I(i,j)=1;
                else I(i,j)=0;
        }
}

void send_a()
{
        int i;

        for(i=1;i<p;i++)
        {
            MPI_Send(&(A(m*i,0)),m*N, MPI
                _FLOAT,i,i,MPI_COMM_WORLD);
            MPI_Send(&(I(m*i,0)),m*N,MPI_
                FLOAT, i,i,MPI_COMM_WORLD);
        }
        free(A);
        free(I);
}

void get_a()
{
        int i,j;

        if (myid==0)
        {
            for(i=0;i<m;i++)
                for(j=0;j<N;j++)
                {
                    a(i,j)=A(i,j);
                    e(i,j)=I(i,j);
                }
        }
        else
        {
            MPI_Recv(a,m*N,MPI_FLOAT,0,myid,
                MPI_COMM_WORLD,&status);
            MPI_Recv(e,m*N,MPI_FLOAT,0,myid,
                MPI_COMM_WORLD,&status);
        }
}

int main(int argc,char **argv)
{
        float *c;
        int k;
        int loop;
        int i,j,v,z,r,t,y;
        int i1,j1;
        float f,g,h;
        float sin1,sin2,cos1;
        float s1,c1;
        float *br,*bt,*bi,*bj,*zi,*zj;
        float *temp1,*temp2,*buffer,*buffee;
        int counter,current;
        int *buf;
        int mml,mpl;
        float bpp,bqq,bpq,bqp;
        float lmax;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,
            &group_size);
        MPI_Comm_rank(MPI_COMM_WORLD,
            &myid);
        p=group_size;
        max=10.0;
        loop=0;
        if (myid==0) read_fileA();
        /*0 号处理器将 N 广播给所有处理器*/
        MPI_Bcast(&N,1,MPI_INT,0,
        MPI_COMM_WORLD);
        m=N/p;
        if (N%p!=0) m++;
```

```
a=(float*)malloc(floatsize*m*N);
e=(float*)malloc(floatsize*m*N);
b=(int*)malloc(intsize*m);
br=(float*)malloc(floatsize*N);
bt=(float*)malloc(floatsize*N);
bi=(float*)malloc(floatsize*m);
bj=(float*)malloc(floatsize*m);
zi=(float*)malloc(floatsize*m);
zj=(float*)malloc(floatsize*m);
temp1=(float*)malloc(floatsize*4);
temp2=(float*)malloc(floatsize*4*p);

if ((myid==p-1)&&(myid%2!=0))
{
    buffer=(float*)malloc(floatsize*m/2*N);
    buffee=(float*)malloc(floatsize*m/2*N);
    buf=(int*)malloc(intsize*m/2);
}

if ((myid%2!=0)&&(myid!=p-1))
{
    buffer=(float*)malloc(floatsize*m*N);
    buffee=(float*)malloc(floatsize*m*N);
    buf=(int*)malloc(intsize*m);
}

/*0 号处理器采用行块划分将矩阵 A 和 I 划分
    为 m*N 的 p 块子矩阵，依次发送给 1 至
    p-1 号处理器*/
if (myid==0)
{
    get_a();
    send_a();
}
else
    get_a();

for(i=0;i<m;i++)
    b(i)=myid*m+i;

while (fabs(max)>E)
{
    loop++;
    /*每轮计算开始，各处理器对其局部存
    储器中离主对角线最近的行块元
    素进行消去*/
for(i=myid*m;i<(myid+1)*m-1;i++)
    for(j=i+1;j<=(myid+1)*m-1;j++)
    {
        r=i%m; t=j%m;
        /*消去 a(r,j)元素*/
        if(a(r,j)!=0)
        {
            f=(-a(r,j));
            g=(a(t,j)-a(r,i))/2;
            h=sgn(g)*f/
                sqrt(f*f+g*g);
            sin2=h;
            sin1=h/sqrt(2*(1+
                sqrt(1-h*h)));
            cos1=sqrt(1-sin1*sin1);

            bpp=a(r,i)*cos1*cos1
                +a(t,j)*sin1*sin1
                +a(r,j)*sin2;

            bqq=a(r,i)*sin1*sin1
                +a(t,j)*cos1*cos1
                -a(r,j)*sin2;

            bpq=0; bqp=0;

            /*对子块 a 的第 t，r 两
                行元素进行行变
                换*/
            for(v=0;v<N;v++)
            if ((v!=i)&&(v!=j))
            {
                br[v]=a(r,v)*cos1
                    +a(t,v)*sin1;
                a(t,v)=-a(r,v)
                    *sin1
                    +a(t,v)*cos1;
            }

            for(v=0;v<N;v++)
                if ((v!=i)&&(v!=j))
                    a(r,v)=br[v];
```

```
        /*对子块 e 的第 i, j 两列
              元素进行列变换*/
        for(v=0;v<m;v++)
              br[v]=e(v,i)*cos1
              +e(v,j)*sin1;
        for(v=0;v<m;v++)
              e(v,j)=e(v,i)*(-sin1)
              +e(v,j)*cos1;
        for(v=0;v<m;v++)
              e(v,i)=br[v];

        /*对子块 a 的第 i，j 两
              列元素进行列变
              换*/
        for(v=0;v<m;v++)
        if ((v!=r)&&(v!=t))
        {
               bi[v]=a(v,i)*cos1
                    +a(v,j)*sin1;
               a(v,j)=-a(v,i) *sin1
                    +a(v,j)*cos1;
        }

        for(v=0;v<m;v++)
              if ((v!=r)&&(v!=t))
                    a(v,i)=bi[v];
        a(r,i)=bpp;
        a(t,j)=bqq;
        a(r,j)=bpq;
        a(t,i)=bqp;
        temp1[0]=sin1;
        temp1[1]=cos1;
        temp1[2]=(float)i;
        temp1[3]=(float)j;
}
else
{
        temp1[0]=0.0;
        temp1[1]=0.0;
        temp1[2]=0.0;
        temp1[3]=0.0;
}
```

```
/*各处理器通过扩展收集操
      作将自己的旋转参数及
      第 i 列和第 j 列的列号按
      处理器编号连接起来并
      广播给所有处理器*/
MPI_Allgather(temp1,4,
      MPI_FLOAT,temp2,4,
      MPI_FLOAT,
      MPI_COMM_WORLD);
current=0;
/*各处理器在收到这些旋转
      参数和列号之后，按
      0,1,…,p-1 的顺序依次读
      取旋转参数及列号并对
      其 m 行中的第 i 列和第
      j 列元素进行旋转列变
      换*/
for(v=1;v<=p;v++)
{
      s1=temp2[(v-1)*4+0];
      c1=temp2[(v-1)*4+1];
      i1=temp2[(v-1)*4+2];
      j1=temp2[(v-1)*4+3];
      if ((s1!=0.0)||(c1!=0.0)
            ||(i1!=0)||(j1!=0))
      {
            if (myid!=current)
            {
            for(z=0;z<m;z++)
            {
                  zi[z]=a(z,i1)
                    *c1+a(z,j1)
                    *s1;
                  a(z,j1)= s1*
                    (-a(z,i1))
                    +a(z,j1)
                    *c1;
            }
            for(z=0;z<m;z++)
                  a(z,i1)=zi[z];
            for(z=0;z<m;z++)
                  zi[z]=e(z,i1)
                  *c1+e(z,j1)
                  *s1;
```

```
                    for(z=0;z<m;z++)
                        e(z,j1)= c1*
                        e(z,j1)-e(z,i1)
                        *s1;
                    for(z=0;z<m;z++)
                        e(z,i1)=zi[z];
                }
                }
                current=current+1;
            } /* for v */
        }   /* for i,j */

/*前 2p-2 次数据交换*/
for(counter=1;counter<=2*p-2;counter++)
{
    if (myid==0)
    {
        MPI_Send(&(a(m/2,0)),
            m/2*N,
            MPI_FLOAT,myid+1,
            myid+1,
            MPI_COMM_WORLD);

    MPI_Send(&(e(m/2,0)),m/2*N,
        MPI_FLOAT,myid+1,myid+1,
        MPI_COMM_WORLD);

    MPI_Send(&b(m/2),m/2,MPI_INT,
        myid+1,myid+1,
        MPI_COMM_WORLD);

    MPI_Recv(&(a(m/2,0)),m/2*N,
        MPI_FLOAT,myid+1,myid,
        MPI_COMM_WORLD,
        &status);

    MPI_Recv(&(e(m/2,0)),m/2*N,
        MPI_FLOAT,myid+1,myid,
        MPI_COMM_WORLD,
        &status);

    MPI_Recv(&b(m/2),m/2, MPI_
        INT,myid+1,myid,
        MPI_COMM_WORLD,
        &status);
    }

if ((myid==p-1)&&(myid%2!=0))
{
for(i=m/2;i<m;i++)
    for(j=0;j<N;j++)
        buffer((i-m/2),j)=a(i,j);
for(i=m/2;i<m;i++)
    for(j=0;j<N;j++)
        buffee((i-m/2),j)=e(i,j);
for(i=m/2;i<m;i++)
    buf[i-m/2]=b(i);
for(i=0;i<m/2;i++)
    for(j=0;j<N;j++)
        a((i+m/2),j)=a(i,j);
for(i=0;i<m/2;i++)
    for(j=0;j<N;j++)
        e((i+m/2),j)=e(i,j);
for(i=0;i<m/2;i++)
    b(m/2+i)=b(i);
      MPI_Recv(&(a(0,0)),m/2*N,
        MPI_FLOAT,myid-1,
        myid,
        MPI_COMM_WORLD,
        &status);

MPI_Recv(&(e(0,0)),m/2*N,
    MPI_FLOAT,myid-1,myid,
    MPI_COMM_WORLD,
    &status);

MPI_Recv(&b(0),m/2,
    MPI_INT, myid-1,myid,
    MPI_COMM_WORLD,
    &status);

MPI_Send(buffer,m/2*N,)
     MPI_FLOAT,myid-1,myid-1,
     MPI_COMM_WORLD);

MPI_Send(buffee,m/2*N,
    MPI_FLOAT,myid-1,myid-1,
    MPI_COMM_WORLD);
```

```
MPI_Send(buf,m/2,MPI_INT,
        myid-1, myid-1,
        MPI_COMM_WORLD);
}

if ((myid==p-1)&&(myid%2==0))
{

MPI_Send(&(a(m/2,0)),m/2*N,
        MPI_FLOAT,myid-1,myid-1,
        MPI_COMM_WORLD);

MPI_Send(&(e(m/2,0)),m/2*N,
        MPI_FLOAT,myid-1,myid-1,
        MPI_COMM_WORLD);

MPI_Send(&b(m/2),m/2,
        MPI_INT,myid-1,myid-1,
        MPI_COMM_WORLD);

for(i=0;i<m/2;i++)
for(j=0;j<N;j++)
        a((i+m/2),j)=a(i,j);
for(i=0;i<m/2;i++)
        for(j=0;j<N;j++)
                e((i+m/2),j)=e(i,j);
for(i=0;i<m/2;i++)
        b(i+m/2)=b(i);

MPI_Recv(&(a(0,0)),m/2*N,
        MPI_FLOAT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);

MPI_Recv(&(e(0,0)),m/2*N,
        MPI_FLOAT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);

MPI_Recv(&b(0),m/2,
        MPI_INT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);
}

if ((myid!=0)&&(myid!=p-1))
{
if(myid%2==0)
{

        MPI_Send(&(a(0,0)),m/2*N,
                MPI_FLOAT,myid+1,
                myid+1,
                MPI_COMM_WORLD);

        MPI_Send(&(e(0,0)),m/2*N,
                MPI_FLOAT,myid+1,
                myid+1,
                MPI_COMM_WORLD);

        MPI_Send(&b(0),m/2,
                MPI_INT,myid+1,
                myid+1,
                MPI_COMM_WORLD);

        MPI_Send(&(a(m/2,0)),m/2*
                N, MPI_FLOAT,myid-1,
                myid-1,
                MPI_COMM_WORLD);

        MPI_Send(&(e(m/2,0)),m/2*
                N, MPI_FLOAT,myid-1,
                myid-1,
                MPI_COMM_ ORLD);

        MPI_Send(&b(m/2),m/2,
                MPI_INT,myid-1,
                myid-1,
                MPI_COMM_WORLD);

        MPI_Recv(&(a(0,0)),m/2*N,
                MPI_FLOAT,myid-1,
                myid,
                MPI_COMM_WORLD,
                &status);

        MPI_Recv(&(e(0,0)),m/2*N,
```

```
            MPI_FLOAT,myid-1,
            myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&b(0),m/2,
            MPI_INT,myid-1,myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&(a(m/2,0)),m/2*
            N, MPI_FLOAT,myid+1,
            myid, MPI_COMM_
            WORLD,&status);

        MPI_Recv(&(e(m/2,0)),m/2*
            N, MPI_FLOAT,myid+1
            ,myid, MPI_COMM_
            WORLD, &status);

        MPI_Recv(&b(m/2),m/2,
            MPI_INT,myid+1,myid,
            MPI_COMM_WORLD,
            &status);
}

if(myid%2!=0)
{
        for(i=0;i<m;i++)
            for(j=0;j<N;j++)
                buffer(i,j)=a(i,j);
        for(i=0;i<m;i++)
        for(j=0;j<N;j++)
            buffee(i,j)=e(i,j);
        for(i=0;i<m;i++)
            buf[i]=b(i);

        MPI_Recv(&(a(0,0)),m/2*N,
            MPI_FLOAT,myid-1,
            myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&(e(0,0)),m/2*N,
            MPI_FLOAT,myid-1,
            myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&b(0),m/2,
            MPI_INT,myid-1,myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&(a(m/2,0)),m/2*
            N, MPI_FLOAT,myid+1,
            myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&(e(m/2,0)),m/2*
            N, MPI_FLOAT,myid+1,
            myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&b(m/2),m/2,
            MPI_INT,myid+1,myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Send(&(buffer(0,0)),m/2
            *N,MPI_FLOAT,
            myid+1,myid+1,
            MPI_COMM_WORLD);

        MPI_Send(&(buffee(0,0)),m/2
            *N,MPI_FLOAT,myid+1
            ,myid+1,
            MPI_COMM_WORLD);

        MPI_Send(&buf[0],m/2,MPI_
            INT,myid+1,myid+1,
            MPI_COMM_WORLD);

        MPI_Send(&(buffer(m/2,0)),m
            /2*N,MPI_FLOAT,
            myid-1,myid-1,
```

```
        MPI_COMM_WORLD);

MPI_Send(&(buffee(m/2,0)),
        m/2*N,MPI_FLOAT,
        myid-1,myid-1,
        MPI_COMM_WORLD);

MPI_Send(&buf[m/2],m/2,
        MPI_INT,myid-1,
        myid-1,
        MPI_COMM_WORLD);
}
}

/*每经过一次交换，消去相应
        的非主对角元素*/
for(i=0;i<m/2;i++)
for(j=m/2;j<m;j++)
{
        if (a(i,b(j))!=0)
        {
        f=-a(i,b(j));
        g=(a(j,b(j))-a(i,b(i)))/2;
        h=sgn(g)*f/
                sqrt(f*f+g*g);
        sin2=h;
        sin1=h/sqrt(2*(1+
                sqrt(1-h*h)));
        cos1=sqrt(1-sin1*sin1);
        bpp=a(i,b(i))*cos1*cos1
                +a(j,b(j))*sin1*
                sin1+a(i,b(j))
                *sin2;
        bqq=a(i,b(i))*sin1*sin1
                +a(j,b(j))*cos1*
                cos1-a(i,b(j))
                *sin2;
        bpq=0; bqp=0;

        for(v=0;v<N;v++)
        if ((v!=b(i))&&(v!=b(j)))
        {
                br[v]=a(i,v)*cos1
                        +a(j,v)*sin1;
                a(j,v)=-a(i,v)*sin1
                        +a(j,v)*cos1;
        }
        for(v=0;v<N;v++)
        if ((v!=b(i))&&(v!=b(j)))
                a(i,v)=br[v];
        for(v=0;v<m;v++)
                br[v]=e(v,b(i))
                        *cos1+
                        e(v,b(j))
                        *sin1;
        for(v=0;v<m;v++)
                e(v,b(j))=e(v,b(i))
                        *(-sin1)+
                        e(v,b(j))
                        *cos1;
        for(v=0;v<m;v++)
                e(v,b(i))=br[v];

        for(v=0;v<m;v++)
        if ((v!=i)&&(v!=j))
        {
                bi[v]=a(v,b(i))
                        *cos1+
                        a(v,b(j))*
                        sin1;
                a(v,b(j))=-a(v,b(i))
                        *sin1+
                        a(v,b(j))*
                        cos1;
        }

        for(v=0;v<m;v++)
        if ((v!=i)&&(v!=j))
                a(v,b(i))=bi[v];
        a(i,b(i))=bpp;
        a(j,b(j))=bqq;
        a(i,b(j))=bpq;
        a(j,b(i))=bqp;
        temp1[0]=sin1;
        temp1[1]=cos1;
        temp1[2]=(float)b(i);
        temp1[3]=(float)b(j);
}
```

```
else
{
        temp1[0]=0.0;
        temp1[1]=0.0;
        temp1[2]=0.0;
        temp1[3]=0.0;
}

MPI_Allgather(temp1,4,
        MPI_FLOAT,temp2,4,
        MPI_FLOAT,
        MPI_COMM_WORLD);
current=0;
for(v=1;v<=p;v++)
{
        s1=temp2[(v-1)*4+0];
        c1=temp2[(v-1)*4+1];
        i1=temp2[(v-1)*4+2];
        j1=temp2[(v-1)*4+3];
        if ((s1!=0.0)||(c1!=0.0)
                ||(i1!=0)||(j1!=0))
        {
        if (myid!=current)
        {
                for(z=0;z<m;z++)
                {
                        zi[z]=a(z,i1)
                                *c1 +
                                a(z, j1)
                                *s1;
                        a(z,j1)= c1*
                                a(z,j1)-
                                s1 *
                                a(z,i1);
                }
                for(z=0;z<m;z++)
                        a(z,i1)=zi[z];
                for(z=0;z<m;z++)
                        zi[z]=e(z,i1)
                                *c1+
                                e(z,j1)
                                *s1;
                for(z=0;z<m;z++)
                        e(z,j1)= c1*
                                e(z,j1)- s1*
                                e(z,i1);
                        for(z=0;z<m;z++)
                                e(z,i1)=zi[z];
                } /* if myid!=current */
                }      /* if */
                current=current+1;
        } /* for v */
    }    /* for i,j */
}    /* counter */

/*第 2p-1 次数据交换*/
if (myid==0)
{

    MPI_Send(&(a(m/2,0)),m/2*N,
            MPI_FLOAT,myid+1,myid+1,
            MPI_COMM_WORLD);

    MPI_Send(&(e(m/2,0)),m/2*N,
            MPI_FLOAT,myid+1,myid+1,
            MPI_COMM_WORLD);

    MPI_Send(&b(m/2),m/2,MPI_INT,
            myid+1,myid+1,
            MPI_COMM_WORLD);

    MPI_Recv(&(a(m/2,0)),m/2*N,
            MPI_FLOAT,myid+1,myid,
            MPI_COMM_WORLD,
            &status);

    MPI_Recv(&(e(m/2,0)),m/2*N,
            MPI_FLOAT,myid+1,myid,
            MPI_COMM_WORLD,
            &status);

    MPI_Recv(&b(m/2),m/2,MPI_INT,
            myid+1,myid,
            MPI_COMM_WORLD,
            &status);
}

if ((myid==p-1)&&(myid%2!=0))
```

```
{
    for(i=m/2;i<m;i++)
        for(j=0;j<N;j++)
            buffer((i-m/2),j)=a(i,j);
    for(i=m/2;i<m;i++)
        for(j=0;j<N;j++)
            buffee((i-m/2),j)=e(i,j);
    for(i=m/2;i<m;i++)
        buf[i-m/2]=b(i);
    for(i=0;i<m/2;i++)
        for(j=0;j<N;j++)
            a((i+m/2),j)=a(i,j);
    for(i=0;i<m/2;i++)
        for(j=0;j<N;j++)
            e((i+m/2),j)=e(i,j);
    for(i=0;i<m/2;i++)
        b(m/2+i)=b(i);
    MPI_Recv(&(a(0,0)),m/2*N,
        MPI_FLOAT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);
          MPI_Recv(&(e(0,0)),m/2*N,
        MPI_FLOAT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);

    MPI_Recv(&b(0),m/2,MPI_INT,
        myid-1,myid,
        MPI_COMM_WORLD,
        &status);
    MPI_Send(buffer,m/2*N,
        MPI_FLOAT,myid-1,myid-1,
        MPI_COMM_WORLD);
    MPI_Send(buffee,m/2*N,
        MPI_FLOAT,myid-1,myid-1,
        MPI_COMM_WORLD);

    MPI_Send(buf,m/2,MPI_INT,myid-
        1,myid-1,
        MPI_COMM_WORLD);
}

if ((myid==p-1)&&(myid%2==0))
{
    MPI_Send(&(a(m/2,0)),m/2*N,
        MPI_FLOAT,myid-1,myid-1,
        MPI_COMM_WORLD);

    MPI_Send(&(e(m/2,0)),m/2*N,
        MPI_FLOAT,myid-1,myid-1,
        MPI_COMM_WORLD);

    MPI_Send(&b(m/2),m/2,MPI_INT,
        myid-1,myid-1,
        MPI_COMM_WORLD);

    for(i=0;i<m/2;i++)
        for(j=0;j<N;j++)
            a((i+m/2),j)=a(i,j);
    for(i=0;i<m/2;i++)
        for(j=0;j<N;j++)
            e((i+m/2),j)=e(i,j);
    for(i=0;i<m/2;i++)
        b(i+m/2)=b(i);
    MPI_Recv(&(a(0,0)),m/2*N,
        MPI_FLOAT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);
    MPI_Recv(&(e(0,0)),m/2*N,
        MPI_FLOAT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);

    MPI_Recv(&b(0),m/2,MPI_INT,
        myid-1,myid,
        MPI_COMM_WORLD,
        &status);
}

if ((myid!=0)&&(myid!=p-1))
{
    if(myid%2==0)
    {

        MPI_Send(&(a(0,0)),m/2*N,
            MPI_FLOAT,myid+1,
            myid+1,
```

```
        MPI_COMM_WORLD);

MPI_Send(&(e(0,0)),m/2*N,
        MPI_FLOAT,myid+1,
        myid+1,
        MPI_COMM_WORLD);

MPI_Send(&b(0),m/2,
        MPI_INT,myid+1,
        myid+1,
        MPI_COMM_WORLD);

MPI_Send(&(a(m/2,0)),m/2*
        N,MPI_FLOAT,myid-1,
        myid-1,
        MPI_COMM_WORLD);

MPI_Send(&(e(m/2,0)),m/2*
        N,MPI_FLOAT,myid-1,
        myid-1,
        MPI_COMM_WORLD);

MPI_Send(&b(m/2),m/2,
        MPI_INT,myid-1,myid-1
        ,MPI_COMM_WORLD)
        ;

MPI_Recv(&(a(0,0)),m/2*N,
        MPI_FLOAT,myid-1,
        myid,
        MPI_COMM_WORLD,
        &status);

MPI_Recv(&(e(0,0)),m/2*N,
        MPI_FLOAT,myid-1,
        myid,
        MPI_COMM_WORLD,
        &status);

MPI_Recv(&b(0),m/2,
        MPI_INT,myid-1,myid,
        MPI_COMM_WORLD,
        &status);
        MPI_Recv(&(a(m/2,0)),m/2*
                N,MPI_FLOAT,myid+1,
                myid,
                MPI_COMM_WORLD,
                &status);

        MPI_Recv(&(e(m/2,0)),m/2*
                N,MPI_FLOAT,myid+1,
                myid,
                MPI_COMM_WORLD,
                &status);

        MPI_Recv(&b(m/2),m/2,
                MPI_INT,myid+1,myid,
                MPI_COMM_WORLD,
                &status);
    }

    if(myid%2!=0)
    {
        for(i=0;i<m;i++)
            for(j=0;j<N;j++)
                buffer(i,j)=a(i,j);

        for(i=0;i<m;i++)
            for(j=0;j<N;j++)
                buffee(i,j)=e(i,j);

        for(i=0;i<m;i++)
            buf[i]=b(i);

        MPI_Recv(&(a(0,0)),m/2*N,
                MPI_FLOAT,myid-1,
                myid,
                MPI_COMM_WORLD,
                &status);

        MPI_Recv(&(e(0,0)),m/2*N,
                MPI_FLOAT,myid-1,
                myid,
                MPI_COMM_WORLD,
                &status);

        MPI_Recv(&b(0),m/2,
```

```
            MPI_INT,myid-1,myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&(a(m/2,0)),m/2*
            N,MPI_FLOAT,myid+1,
            myid,
            MPI_COMM_WORLD,
            &status);

        MPI_Recv(&(e(m/2,0)),m/2*
            N,MPI_FLOAT,myid+1,
            myid,
            MPI_COMM_WORLD,
            &status);
        MPI_Recv(&b(m/2),m/2,
            MPI_INT,myid+1,myid,
            MPI_COMM_WORLD,
            &status);
        MPI_Send(&(buffer(0,0)),
            m/2*N,MPI_FLOAT,
            myid+1,myid+1,
            MPI_COMM_WORLD);

        MPI_Send(&(buffee(0,0)),
            m/2*N,MPI_FLOAT,
            myid+1,myid+1,
            MPI_COMM_WORLD);

        MPI_Send(&buf[0],m/2,
            MPI_INT,myid+1,
            myid+1,
            MPI_COMM_WORLD);

        MPI_Send(&(buffer(m/2,0)),
            m/2*N,MPI_FLOAT,
            myid-1,myid-1,
            MPI_COMM_WORLD);

        MPI_Send(&(buffee(m/2,0)),
            m/2*N,MPI_FLOAT,
            myid-1,myid-1,
            MPI_COMM_WORLD);
                MPI_Send(&buf[m/2],m/2,
                    MPI_INT,myid-1,myid-1
                    ,MPI_COMM_WORLD);
            }
        }

        lmax=min;
        for(i=0;i<m;i++)
            for(j=0;j<N;j++)
                if ((m*myid+i)!=j)
                    if (fabs(a(i,j))>lmax)
                        lmax=fabs(a(i,j));
        }

        /*通过归约操作的求最大值运算求得将
            整个 n 阶矩阵非主对角元素的最大
            元 max，并广播给所有处理器以决
            定是否进行下一轮迭代*/
        MPI_Allreduce(&lmax,&max,1,
            MPI_FLOAT,MPI_MAX,
            MPI_COMM_WORLD);
    } /* while */

    /*0 号处理器收集经过旋转变换的各子矩阵 a，
        得到非主对角元素全为 0 的结果矩阵
        A*/
    if (myid==0)
    {
        A=(float*)malloc(floatsize*N*m*p);
        I=(float*)malloc(floatsize*N*m*p);
        for(i=0;i<m;i++)
            for(j=0;j<N;j++)
            {
                A(i,j)=a(i,j);
                I(i,j)=e(i,j);
            }
    }

    if (myid!=0)
    {
        MPI_Send(a,m*N,MPI_FLOAT,0,myid,
            MPI_COMM_WORLD);

        MPI_Send(e,m*N,MPI_FLOAT,0,myid,
```

```
                MPI_COMM_WORLD);
        }
        else
                for(i=1;i<p;i++)
                {
                        MPI_Recv(a,m*N,MPI_FLOAT,i,
                                i, MPI_COMM_WORLD,
                                &status);

                        MPI_Recv(e,m*N,MPI_FLOAT,i,
                                i,MPI_COMM_WORLD,
                                &status);

                        for(j=0;j<m;j++)
                                for(k=0;k<N;k++)
                                        A((i*m+j),k)=a(j,k);

                        for(j=0;j<m;j++)
                                for(k=0;k<N;k++)
                                        I((i*m+j),k)=e(j,k);
                }

        /*矩阵 A 的主对角元素就是特征值*/
        if (myid==0)
        {
                for(i=0;i<N;i++)
                        printf("the %dst envalue:%f\n",i,
                        A(i,i));
                printf("\n");
                printf("Iteration num = %d\n",loop);
        }

        MPI_Finalize();
        free(a);
        free(b);
        free(c);
        free(br);
        free(bt);
        free(bi);
        free(bj);
        free(zi);
        free(zj);
        free(buffer);
        free(buf);
        free(buffee);
        free(A);
        free(I);
        free(temp1);
        free(temp2);
        return(0);
}
```

2. 运行实例

编译：mpicc –o cjacobi cjacobi.cc

运行：可以使用命令 mpirun －np ProcessSize cjacobi 来运行该程序，其中 ProcessSize 是所使用的处理器个数,这里取为 4。本实例中使用了

mpirun –np 4 cjacobi

运行结果：

Input of file "dataIn.txt"

8 8

1.000000	3.000000	4.000000	4.000000	5.000000	6.000000	7.000000	8.000000
3.000000	1.000000	2.000000	2.000000	3.000000	4.000000	5.000000	6.000000
4.000000	2.000000	1.000000	2.000000	3.000000	3.000000	2.000000	1.000000
1.000000	2.000000	3.000000	4.000000	4.000000	6.000000	7.000000	8.000000
4.000000	5.000000	5.000000	6.000000	7.000000	8.000000	1.000000	1.000000
2.000000	2.000000	2.000000	3.000000	4.000000	5.000000	7.000000	3.000000
2.000000	4.000000	5.000000	7.000000	9.000000	0.000000	2.000000	3.000000
1.000000	2.000000	4.000000	6.000000	2.000000	7.000000	8.000000	1.000000

the 0st envalue:-5.539342

the 1st envalue:9.369179

the 2st envalue:2.678424

the 3st envalue:0.378010

the 4st envalue:30.210718

the 5st envalue:-1.180327

the 6st envalue:-10.858351

the 7st envalue:-3.058303

Iteration num = 4

说明：该运行实例中，A 为 8×8 的对称矩阵，它的元素值存放于文档“dataIn.txt”中，最后输出矩阵 A 的 7 个特征值。

第二十二章 快速傅氏变换和离散小波变换

长期以来，**快速傅氏变换**(Fast Fourier Transform)和**离散小波变换**(Discrete Wavelet Transform)在数字信号处理、石油勘探、地震预报、医学断层诊断、编码理论、量子物理及概率论等领域中都得到了广泛的应用。各种快速傅氏变换(FFT)和离散小波变换(DWT)算法不断出现，成为数值代数方面最活跃的一个研究领域，而其意义远远超过了算法研究的范围，进而为诸多科技领域的研究打开了一个崭新的局面。本章分别对 FFT 和 DWT 的基本算法作了简单介绍，若需在此方面做进一步研究，可参考文献[2]。

5.1 快速傅里叶变换 FFT

离散傅里叶变换是 20 世纪 60 年代是计算复杂性研究的主要里程碑之一，1965 年 Cooley 和 Tukey 所研究的计算**离散傅里叶变换**(Discrete Fourier Test)的快速傅氏变换(FFT)将计算量从 O(n^2)下降至 O($n\log n$)，推进了 FFT 更深层、更广法的研究与应用。FFT 算法有很多版本，但大体上可分为两类：迭代法和递归法，本节仅讨论迭代法，递归法可参见文献[1]、[2]。

5.1.1 串行 FFT 迭代算法

假定 $a[0],a[1],\cdots,a[n-1]$ 为一个有限长的输入序列，$b[0],b[1],\cdots,b[n-1]$为离散傅里叶变换的结果序列，则有：$b[k]=\sum_{m=0}^{n-1}a[k]W_n^{km}(k=0,1,2,...,n-1)$，其中 $W_n=e^{\frac{2pi}{n}}$，实际上，上式可写成矩阵 W 和向量 a 的乘积形式：

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \mathrm{M} \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} w^0 & w^0 & w^0 & \Lambda & w^0 \\ w^0 & w^1 & w^2 & \Lambda & w^{n-1} \\ w^0 & w^2 & w^4 & \Lambda & w^{2(n-1)} \\ \mathrm{M} & \mathrm{M} & \mathrm{M} & \mathrm{O} & \mathrm{M} \\ w^0 & w^{(n-1)} & w^{2(n-1)} & \Lambda & w^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \mathrm{M} \\ a_{n-1} \end{bmatrix}$$

一般的 n 阶矩阵和 n 维向量相乘，计算时间复杂度为 n^2，但由于 W 是一种特殊矩阵，故可以降低计算量。FFT 的计算流图如图 5.1 所示，其串行算法如下：

算法 22.1　单处理器上的 FFT 迭代算法

输入：$a=(a_0,a_1,\cdots,a_{n-1})$

输出：$b=(b_0,b_1,\cdots,b_{n-1})$

Begin

(1)**for** k=0 **to** n-1 **do**

$c_k=a_k$

end for

(2)**for** h=log$n$-1 **downto** 0 **do**

(2.1) $p=2^h$

(2.2) $q=n/p$

(2.3) $z=w^{q/2}$

(2.4) **for** k=0 **to** n-1 **do**

if (k mod p=k mod2p) **then**

(i)$c_k = c_k + c_{k+p}$

(ii)$c_{k+p}=(c_k - c_{k+p})z^{k \bmod p}$ /* c_k 不用(i)计算的新值 */

end if

end for

end for

(3)**for** k=1 **to** n-1 **do**

$b_{r(k)} = c_k$ /* r(k)为 k 的位反 */

end for

End

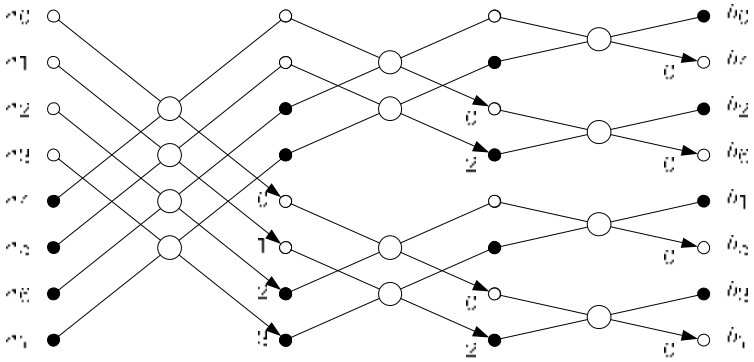

图 5.1 n=4 时的 FFT 蝶式变换图

显然, FFT 算法的计算复杂度为 O($n$log$n$)。

5.1.2 并行 FFT 算法

设 P 为处理器的个数，一种并行 FFT 实现时是将 n 维向量 a 划分成 p 个连续的 m 维子向量，这里 $m=\lceil n/p \rceil$，第 i 个子向量中下标为 $i\times m$, …, $(i+1)\times m$-1，其元素被分配至第 i 号处理器（i=0,1, …, p-1）。由图 5.1 可以看到，FFT 算法由 log$n$ 步构成，依次以 $2^{\log n-1}$、$2^{\log n-2}$、…、2、1 为下标跨度做蝶式计算，我们称下标跨度为 2^h 的计算为第 h 步（h=log$n$-1, log$n$-2, …, 1, 0）。并行计算可分两阶段执行：第一阶段，第 log$n$-1 步至第 log$m$ 步，由于下标跨度 $h \geqslant m$，各处理器之间需要通信；第二阶段，第 log$m$-1 步至第 0 步各处理器之间不需要通信。具体并行算法框架描述如下：

算法 22.2 FFT 并行算法

输入： $a=(a_0,a_1, \cdots,a_{n-1})$

输出： $b=(b_0,b_1, \cdots,b_{n-1})$

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

(1)**for** h=log$p$-1 **downto** 0 **do**

/* 第一阶段，第 log$n$-1 步至第 log$m$ 步各处理器之间需要通信*/

(1.1) $t=2^i$, $l=2^{(i+\log m)}$,$q=n/l$, $z=w^{q/2}$, $j=j+1$,$v=2^j$ /*开始 j=0*/

(1.2)**if** ((my_rank mod t)=(my_rank mod 2t)) **then**

/*本处理器的数据作为变换的前项数据*/

(i) tt= my_rank+p/v

(ii)接收由 tt 号处理器发来的数据块，并将接收的数据块存于 c[my_rank*m+n/v]到 c[my_rank*m+n/v+m]之中

(iii)**for** k=0 **to** m-1 **do**

$c[k]=c[k]+c[k+n/v]$

$c[k+n/v]=(c[k]-c[k+n/v])*z^{(\text{my\_rank}*m+k) \bmod l}$

end for

(iv)将存于 c[my_rank*m+n/v]到 c[my_rank*m+n/v+m]之中的数据块发送到 tt 号处理器

else /*本处理器的数据作为变换的后项数据*/

(v)将存于之中的数据块发送到 my_rank-p/v 号处理器

(vi)接收由 my_rank-p/v 号处理器发来的数据块存于 c 中

end if

end for

(2)**for** i=log$m$-1 **downto** 0 **do** /*第二阶段，第 log$m$-1 步至第 0 步各处理器之间不需要通信*/

(2.1) $l=2^i$,$q=n/l$, $z=w^{q/2}$

(2.2)**for** k=0 **to** m-1 **do**

if ((k mod l)=(k mod 2l)) **then**

$c[k]=c[k]+c[k+l]$

$c[k+l]=(c[k]-c[k+l])*z^{(\text{my\_rank}*m+k) \bmod l}$

end if

end for

end for

End

由于各处理器对其局部存储器中的 m 维子向量做变换计算，计算时间为 $m\log n$；点对点通信 $2\log p$ 次，每次通信量为 m，通信时间为 $2(t_s+mt_w)\log p$，因此快速傅里叶变换的并行计算时间为 $T_p=m\log n+2(t_s+mt_w)\log p$。

MPI 源程序请参见章末附录。

5.2 离散小波变换 DWT

5.2.1 离散小波变换 DWT 及其串行算法

先对一维小波变换作一简单介绍。设 $f(x)$为一维输入信号，记 $f_{jk}(x)=2^{-j/2}f(2^{-j}x-k)$，$y_{jk}(x)=2^{-j/2}y(2^{-j}x-k)$，这里 $f(x)$ 与 $y(x)$ 分别称为定标函数与子波函数，$\{f_{jk}(x)\}$ 与 $\{y_{jk}(x)\}$ 为二个正交基函数的集合。记 $P_0f=f$，在第 j 级上的一维**离散小波变换** DWT(Discrete Wavelet Transform)通过正交投影 P_jf 与 Q_jf 将 $P_{j-1}f$ 分解为：

$$P_{j-1}f=P_jf+Q_jf=\sum_k c_k^j f_{jk}+\sum_k d_k^j y_{jk}$$

其中：$c_k^j=\sum_{n=0}^{p-1}h(n)c_{2k+n}^{j-1}$，$d_k^j=\sum_{n=0}^{p-1}g(n)c_{2k+n}^{j-1}$ $(j=1,2,\ldots,L,k=0,1,\ldots,N/2^j-1)$，这里，$\{h(n)\}$ 与 $\{g(n)\}$ 分别为低通与高通权系数，它们由基函数 $\{f_{jk}(x)\}$ 与 $\{y_{jk}(x)\}$ 来确定，p 为权系数的长度。$\{C_n^0\}$ 为信号的输入数据，N 为输入信号的长度，L 为所需的级数。由上式可见，每级一维 DWT 与一维卷积计算很相似。所不同的是：在 DWT 中，输出数据下标增加 1 时，权系数在输入数据的对应点下标增加 2，这称为“间隔取样”。

算法 22.3　一维离散小波变换串行算法

输入： $c^0=d^0(c_0^0, c_1^0,\ldots, c_{N-1}^0)$

$h=(h_0, h_1,\ldots, h_{L-1})$

$g=(g_0, g_1,\ldots, g_{L-1})$

输出： $c_i^j, d_i^j\ (i=0, 1,\ldots, N/2^{j-1}, j\geqslant 0)$

Begin

(1)$j=0, n=N$

(2)**While** ($n\geqslant 1$) **do**

(2.1)**for** $i=0$ **to** $n-1$ **do**

(2.1.1)$c_i^{j+1}=0, d_i^{j+1}=0$

(2.1.2)**for** $k=0$ **to** $L-1$ **do**

$$c_i^{j+1}=c_i^{j+1}+h_k\,c_{(k+2i)\bmod n}^j,\qquad d_i^{j+1}=d_i^{j+1}+g_k d_{(k+2i)\bmod n}^j$$

end for

end for

(2.2)$j=j+1, n=n/2$

end while

End

显然，算法 22.3 的时间复杂度为 $O(N*L)$。

在实际应用中，很多情况下采用**紧支集小波**（Compactly Supported Wavelets），这时相

应的尺度系数和小波系数都是有限长度的，不失一般性设尺度系数只有有限个非零值：$h_1,\cdots,h_N$，N 为偶数，同样取小波使其只有有限个非零值：$g_1,\cdots,g_N$。为简单起见，设尺度系数与小波函数都是实数。对有限长度的输入数据序列：$c_n^0 = x_n, n = 1,2,\Lambda\ ,M$ (其余点的值都看成 0)，它的离散小波变换为:

$$c_k^{j+1} = \sum_{n\in Z} c_n^j h_{n-2k}$$

$$d_k^{j+1} = \sum_{n\in Z} c_n^j g_{n-2k}$$

$$j = 0,1,\Lambda\ ,J-1$$

其中 J 为实际中要求分解的步数，最多不超过 $\log_2 M$，其逆变换为

$$c_n^{j-1} = \sum_{k\in Z} c_k^j h_{n-2k} + \sum_{k\in Z} c_k^j h_{n-2k}$$

$$j = J,\Lambda\ ,1$$

注意到尺度系数和输入系列都是有限长度的序列，上述和实际上都只有有限项。若完全按照上述公式计算，在经过 J 步分解后，所得到的 J+1 个序列 $d_k^j, j = 0,1,\Lambda\ ,J-1$ 和 c_k^j 的非零项的个数之和一般要大于 M，究竟这个项目增加到了多少？下面来分析一下上述计算过程。

j=0 时计算过程为

$$c_k^1 = \sum_{n=1}^{M} x_n h_{n-2k}$$

$$d_k^1 = \sum_{n=1}^{M} x_n g_{n-2k}$$

不难看出，c_k^1 的非零值范围为：$k = -\frac{N}{2}+1,\Lambda\ ,-1,0,\Lambda\ ,\left\lceil\frac{M}{2}\right\rceil-1$, 即有 $k = -\frac{N}{2}-1+\left\lceil\frac{M}{2}\right\rceil=\left\lceil\frac{M+N-1}{2}\right\rceil$ 个非零值。d_k^1 的非零值范围相同。继续往下分解时，非零项出现的规律相似。分解多步后非零项的个数可能比输入序列的长度增加较多。例如，若输入序列长度为 100，N=4，则 d_k^1 有 51 项非零，d_k^2 有 27 项非零，d_k^3 有 15 项非零，d_k^4 有 9 项非零，d_k^5 有 6 项非零，d_k^6 有 4 项非零，c_k^6 有 4 项非零。这样分解到 6 步后得到的序列的非零项个数的总和为 116，超过了输入序列的长度。在数据压缩等应用中，希望总的长度基本不增加，这样可以提高压缩比、减少存储量并减少实现的难度。

可以采用稍微改变计算公式的方法，使输出序列的非零项总和基本上和输入序列的非零项数相等，并且可以完全重构。这种方法也相当于把输入序列进行延长（增加非零项），因而称为延拓法。

只需考虑一步分解的情形，下面考虑第一步分解(j=1)。将输入序列作延拓，若 M 为偶数，直接将其按 M 为周期延拓；若 M 为奇数，首先令 $x_{M+1} = 0$。然后按 M+1 为周期延拓。作了这种延拓后再按前述公式计算，相应的变换矩阵已不再是 H 和 G，事实上这时的变换

矩阵类似于循环矩阵。例如，当 M=8，N=4 时矩阵 H 变为：

$$\begin{matrix} h_3 & h_4 & 0 & 0 & 0 & 0 & h_1 & h_2 \\ h_1 & h_2 & h_3 & h_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_1 & h_2 & h_3 & h_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_1 & h_2 & h_3 & h_4 \\ h_3 & h_4 & 0 & 0 & 0 & 0 & h_1 & h_2 \end{matrix}$$

当 M=7，N=4 时矩阵 H 变为：

$$\begin{matrix} h_3 & h_4 & 0 & 0 & 0 & 0 & h_1 \\ h_1 & h_2 & h_3 & h_4 & 0 & 0 & 0 \\ 0 & 0 & h_1 & h_2 & h_3 & h_4 & 0 \\ 0 & 0 & 0 & 0 & h_1 & h_2 & h_3 \\ h_3 & h_4 & 0 & 0 & 0 & 0 & h_1 \end{matrix}$$

从上述的矩阵表示可以看出，两种情况下的矩阵内都有完全相同的行，这说明作了重复计算，因而从矩阵中去掉重复的那一行不会减少任何信息量，也就是说，这时我们可以对矩阵进行截短（即去掉一行），使得所得计算结果仍然可以完全恢复原输入信号。当 M=8，N=4 时截短后的矩阵为：

$$H=\begin{bmatrix} h_3 & h_4 & 0 & 0 & 0 & 0 & h_1 & h_2 \\ h_1 & h_2 & h_3 & h_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_1 & h_2 & h_3 & h_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_1 & h_2 & h_3 & h_4 \end{bmatrix}$$

当 M=7，N=4 时截短后的矩阵为：

$$H=\begin{bmatrix} h_3 & h_4 & 0 & 0 & 0 & 0 & h_1 \\ h_1 & h_2 & h_3 & h_4 & 0 & 0 & 0 \\ 0 & 0 & h_1 & h_2 & h_3 & h_4 & 0 \\ 0 & 0 & 0 & 0 & h_1 & h_2 & h_3 \end{bmatrix}$$

这时的矩阵都只有$\left\lceil \frac{M}{2} \right\rceil$行。分解过程成为：

$$C^1 = HC^0$$

$$D^1 = GC^0$$

向量 C^1 和 D^1 都只有$\left\lceil \frac{M}{2} \right\rceil$个元素。重构过程为：

$$C^0 = H * C^1 + G * D^1$$

可以完全重构。矩阵 H，G 有等式

$$H*H+G*G=I$$

一般情况下，按上述方式保留矩阵的$\left\lceil \frac{M}{2} \right\rceil$行，可以完全恢复原信号。

这种方法的优点是最后的序列的非 0 元素的个数基本上和输入序列的非 0 元素个数相同，特别是若输入序列长度为 2 的幂，则完全相同，而且可以完全重构输入信号。其代价是得到的变换系数 D^j 中的一些元素已不再是输入序列的离散小波变换系数，对某些应用可能是不适合的，但在数据压缩等应用领域，这种方法是可行的。

5.2.2 离散小波变换并行算法

下设输入序列长度 $N=2^t$，不失一般性设尺度系数只有有限个非零值：h_0，…，h_{L-1}，L 为偶数，同样取小波使其只有有限个非零值：g_0，…，g_{L-1}。为简单起见，我们采用的延拓方法计算。即将有限尺度的序列 $c_n^0 = x_n, (n = 0,1,\Lambda\ , N-1)$ 按周期 N 延长，使他成为无限长度的序列。这时变换公式也称为周期小波变换。变换公式为：

$$c_k^{j+1} = \sum_{n\in Z} c_n^j h_{n-2k} = \sum_{n=0}^{L-1} h_n c^{j}{}_{\langle n+2k\rangle}$$

$$d_k^{j+1} = \sum_{n=0}^{L-1} g_n d^{j}{}_{\langle n+2k\rangle}$$

$$j = 0,1,\Lambda\ , J-1$$

其中<n+2k>表示 n+2k 对于模 $N/2^j$ 的最小非负剩余。注意这时 c_k^j 和 d_k^j 是周期为 $N/2^j$ 的周期序列。其逆变换为

$$c_n^{j-1} = \sum_{k\in Z} c_k^j h_{n-2k} + \sum_{k\in Z} c_k^j g_{n-2k}$$

$$j = J,\Lambda\ ,1$$

从变换公式中可以看出，计算输出点 c_k^{j+1} 和 d_k^{j+1}，需要输入序列 c_n^j 在 n=2k 附近的值（一般而言，L 远远小于输入序列的长度）。设处理器台数为 p，将输入数据 $c_n^j (n = 0,1,\Lambda\ , N/2^j - 1)$ 按块分配给 p 台处理器，处理器 i 得到数据 $c_n^j (n = i\frac{N}{P2^j}, \Lambda\ , (i+1)\frac{N}{P2^j} - 1)$，让处理器 i 负责 c_n^{j+1} 和 $d_n^{j+1} (n = i\frac{N}{P2^{j+1}}, \Lambda\ , (i+1)\frac{N}{P2^{j+1}} - 1)$ 的计算，则不难看出，处理器 i 基本上只要用到局部数据，只有 L/2 个点的计算要用到处理器 i+1 中的数据，这时做一步并行数据发送：将处理器 i+1 中前 L-1 个数据发送给处理器 i，则各处理器的计算不再需要数据交换，关于本算法其它描述可参见文献[1]。

算法 22.4　离散小波变换并行算法

输入： h_i(i=0,…, L-1), g_i(i=0,…, L-1), c_i^0(i=0,…, N-1)

输出： c_i^k (i=0,…, $N/2^k$-1,k>0)

Begin

对所有处理器 my_rank(my_rank=0,…, p-1)同时执行如下的算法:

(1)j=0;

(2)**while** (j<r) **do**

(2.1)将数据 $c_n^j (n = 0,1,\Lambda\ , N/2^j - 1)$ 按块分配给 p 台处理器

(2.2)将处理器 i+1 中前 L-1 个数据发送给处理器 i

(2.3)处理器 i 负责 c_n^{j+1} 和 $d_n^{j+1} (n = i\frac{N}{p2^{j+1}}, \Lambda\ , (i+1)\frac{N}{p2^{j+1}} - 1)$ 的计算

(2.4)$j=j+1$

end while

End

这里每一步分解后数据 c_n^{j+1} 和 d_n^{j+1} 已经是按块存储在 P 台处理器上，因此算法第一步中的数据分配除了 $j=0$ 时需要数据传送外，其余各步不需要数据传送（数据已经到位）。因此，按 LogP 模型，算法的总的通信时间为：$2(L\max(o,g)+l)$，远小于计算时间 $O(N)$。

MPI 源程序请参见所附光盘。

5.3 小结

本章主要讨论一维 FFT 和 DWT 的简单串、并行算法，二维 FFT 和 DWT 在光学、地震以及图象信号处理等方面起着重要的作用。限于篇幅，此处不再予以讨论。同样，FFT 和 DWT 的并行算法的更为详尽描述可参见文献[2]和文献[3],专门介绍快速傅氏变换和卷积算法的著作可参见[4]。另外，二维小波变换的并行计算及相关算法可参见文献[5]，LogP 模型可参考文献[3]。

参考文献

[1]. 王能超 著．数值算法设计．华中理工大学出版社,1988.9

[2]. 陈国良 编著．并行计算——结构·算法·编程．高等教育出版社,1999.10

[3]. 陈国良 编著．并行算法设计与分析（修订版）．高等教育出版社，2002.11

[4]. Nussbaumer H J. Fast Fourier Transform and Convolution Algorithms.2nded. Springer-Verlag,1982

[5]. 陈崚．二维正交子波变换的 VLSI 并行计算．电子学报,1995,23(02):95-97

附录 FFT 并行算法的 MPI 源程序

1. 源程序 fft.c

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <math.h>
#include "mpi.h"

#define   MAX_PROCESSOR_NUM   12
#define   MAX_N               50
#define   PI                  3.141592653
#define   EPS                 10E-8
#define   V_TAG               99
#define   P_TAG               100
#define   Q_TAG               101
#define   R_TAG               102
#define   S_TAG               103
#define   S_TAG2              104

void evaluate(complex<double>* f, int beginPos,
int endPos, const complex<double>* x,
      complex<double> *y, int leftPos,
      int rightPos, int totalLength);

void shuffle(complex<double>* f, int beginPos,
     int endPos);
```

```
void print(const complex<double>* f,
        int fLength);

void readDoubleComplex(FILE *f,
        complex<double> &z);

int main(int argc, char *argv[])
{
        complex<double> p[MAX_N], q[MAX_N],
        s[2*MAX_N], r[2*MAX_N];
        complex<double> w[2*MAX_N];
        complex<double> temp;
        int variableNum;
        MPI_Status status;
        int rank, size;
        int i, j, k, n;
        int wLength;
        int everageLength;
        int moreLength;
        int startPos;
        int stopPos;
        FILE *fin;

       MPI_Init(&argc, &argv);
       MPI_Get_rank(MPI_COMM_WORLD,
              &rank);

        MPI_Get_size(MPI_COMM_WORLD,
               &size);

        if(rank == 0)
        {
              fin = fopen("dataIn.txt", "r");
              if (fin == NULL)
              {
                    puts("Not find input data file");
                    puts("Please create a file
                          \"dataIn.txt\"");
                    puts("<example for dataIn.txt> ");
                    puts("2");
                    puts("1.0    2");
                    puts("2.0    -1");
                    exit(-1);
              }

             readDoubleComplex(fin, variableNum);

             if ((variableNum < 1)||(variableNum >
                    MAX_N))
             {
                    puts("variableNum out of range!");
                    exit(-1);
             }

             for(i = 0; i < variableNum; i ++)
                    readDoubleComplex(fin, p[i]);
             for(i = 0; i < variableNum; i ++)
                    readDoubleComplex(fin, q[i]);
             fclose(fin);

             puts("Read from data file \"dataIn.txt\"");
             printf("p(t) = ");
             print(p, variableNum);
             printf("q(t) = ");
             print(q, variableNum);

             for(i = 1; i < size; i ++)
             {
                    MPI_Send(&variableNum,1,
                          MPI_INT,i, V_TAG,
                          MPI_COMM_WORLD);
                    MPI_Send(p,variableNum,
                          MPI_DOUBLE_COMPLEX,i,
                          P_TAG,
                          PI_COMM_WORLD);
                    MPI_Send(q,variableNum,
                          MPI_DOUBLE_COMPLEX,i,
                          Q_TAG,
                          MPI_COMM_WORLD);
             }
       }
       else
       {
             MPI_Recv(&variableNum,1,MPI_INT,0,
                    V_TAG,MPI_COMM_WORLD,
                    &status);
             MPI_Recv(p,variableNum,
```

```
                MPI_DOUBLE_COMPLEX,0,
                P_TAG, PI_COMM_WORLD,
                &status);
        MPI_Recv(q,variableNum,
                MPI_DOUBLE_COMPLEX,0,
                Q_TAG,MPI_COMM_WORLD,
                &status);
 }

wLength = 2*variableNum;
for(i = 0; i < wLength; i ++)
{
        w[i]= complex<double>
                (cos(i*2*PI/wLength),
                sin(i*2*PI/wLength));
}

everageLength = wLength / size;
moreLength = wLength % size;
startPos = moreLength + rank * everageLength;
stopPos = startPos + everageLength - 1;

if(rank == 0)
{
         startPos = 0;
         stopPos  =  moreLength+everageLength  -
                1;
}
evaluate(p, 0, variableNum - 1, w, s,
        startPos, stopPos, wLength);
evaluate(q, 0, variableNum - 1, w, r,
        startPos, stopPos, wLength);
for(i = startPos; i <= stopPos ; i ++)
        s[i] = s[i]*r[i]/(wLength*1.0);

if (rank > 0)
{
        MPI_Send((s+startPos),
                everageLength,
                MPI_DOUBLE_COMPLEX, 0,
                S_TAG, MPI_COMM_WORLD);
        MPI_Recv(s,wLength,
                MPI_DOUBLE_COMPLEX,0,
                S_ TAG2,MPI_ COMM_WORLD,
                &status);
}
else
{
        for(i = 1; i < size; i ++)
        {
                MPI_Recv((s+moreLength+i*
                        everageLength),everageLength,
                        MPI_DOUBLE_COMPLEX,
                        i,S_TAG,
                        MPI_COMM_WORLD,
                        &status);
        }

        for(i = 1; i < size; i ++)
        {
                MPI_Send(s,wLength,
                        MPI_DOUBLE_COMPLEX,i,
                        S_TAG2,
                        MPI_COM M_WORLD);
        }
}

for(int i = 1; i < wLength/2; i ++)
{
         temp = w[i];
        w[i] = w[wLength - i];
        w[wLength - i] = temp;
}
evaluate(s, 0, wLength - 1, w, r, startPos,
         stopPos, wLength);

if (rank > 0)
{
        MPI_Send((r+startPos), everageLength,
                MPI_DOUBLE_COMPLEX,0,
                R_TAG,
                MPI_COMM_WORLD);
}
else
{
        for(i = 1; i < size; i ++)
        {
                MPI_Recv((r+moreLength+i*
```

```
                    everageLength),
                    everageLength,
                    MPI_DOUBLE_COMPLEX,
                    i, R_TAG,
                    MPI_COMM_WORLD,
                    &status);
            }

            puts("\nAfter FFT r(t)=p(t)q(t)");
            printf("r(t) = ");
            print(r, wLength - 1);
            puts("");
            printf("Use prossor size = %d\n",size);
        }

        MPI_Finalize();
}

void evaluate(complex<double>* f, int beginPos, int
        endPos, const complex<double>* x,
        complex<double>* y, int leftPos, int
        rightPos, int totalLength)
{
        int i;
        complex<double>
            tempX[2*MAX_N],tempY1[2*MAX_N],
            tempY2[2*MAX_N];
        int midPos = (beginPos + endPos)/2;

        if ((beginPos > endPos)||(leftPos > rightPos))
        {
            puts("Error in use Polynomial!");
            exit(-1);
        }
        else if(beginPos == endPos)
        {
            for(i = leftPos; i <= rightPos; i ++)
                y[i] = f[beginPos];
        }
        else if(beginPos + 1 == endPos)
        {
            for(i = leftPos; i <= rightPos; i ++)
                y[i] = f[beginPos] + f[endPos]*x[i];
        }
        else
        {
            shuffle(f, beginPos, endPos);
            for(i = leftPos; i <= rightPos; i ++)
                tempX[i] = x[i]*x[i];
            evaluate(f, beginPos, midPos, tempX,
                tempY1, leftPos, rightPos,totalLength);
            evaluate(f, midPos+1, endPos, tempX,
                tempY2, leftPos, rightPos,
                totalLength);
            for(i = leftPos; i <= rightPos; i ++)
                y[i] = tempY1[i] + x[i]*tempY2[i];
        }
}

void shuffle(complex<double>* f, int beginPos, int
        endPos)
{
        complex<double> temp[2*MAX_N];
        int i, j;

        for(i = beginPos; i <= endPos; i ++)
            temp[i] = f[i];

        j = beginPos;
        for(i = beginPos; i <= endPos; i +=2)
        {
            f[j] = temp[i];
            j ++;
        }

        for(i = beginPos +1; i <= endPos; i += 2)
        {
            f[j] = temp[i];
            j ++;
        }
}

void print(const complex<double>* f, int fLength)
{
        bool isPrint = false;
        int i;

        if (abs(f[0].real()) > EPS)
```

```
    {
        printf(“%lf”, f[0].real());
        isPrint = true;
     }

     for(i = 1; i < fLength; i ++)
     {
          if (f[i].real() > EPS)
          {
               if (isPrint) printf(" + ");
               else isPrint = true;
               printf("%lft^%d", f[i].real(),i);
          }
          else if (f[i].real() < - EPS)
          {
               if(isPrint) printf(" - ");
               else isPrint = true;
               printf("%lft^%d", -f[i].real(),i);
          }
    }

    if (isPrint == false) printf("0");
    printf("\n");
}
```

2. 运行实例

编译：mpicc –o fft fft.cc

运行：使用如下命令运行程序

mpirun –np 1 fft
mpirun –np 2 fft
mpirun –np 3 fft
mpirun –np 4 fft
mpirun –np 5 fft

运行结果：

```
Input of file "dataIn.txt"
4
1   3   3   1
0   1   2   1

Output of solution
    Read from data file "dataIn.txt"
    p(t) = 1 + 3t^1 + 3t^2 + 1t^3
    q(t) = 1t^1 + 2t^2 + 1t^3

    After FFT r(t)=p(t)q(t)
    r(t) = 1t^1 + 5t^2 + 10t^3 + 10t^4 + 5t^5 + 1t^6

    Use prossor size = 1
    End of this running

    Read from data file "dataIn.txt"
    p(t) = 1 + 3t^1 + 3t^2 + 1t^3
    q(t) = 1t^1 + 2t^2 + 1t^3

    After FFT r(t)=p(t)q(t)
    r(t) = 1t^1 + 5t^2 + 10t^3 + 10t^4 + 5t^5 + 1t^6

    Use prossor size = 2
    End of this running

    Read from data file "dataIn.txt"
    p(t) = 1 + 3t^1 + 3t^2 + 1t^3
    q(t) = 1t^1 + 2t^2 + 1t^3

    After FFT r(t)=p(t)q(t)
    r(t) = 1t^1 + 5t^2 + 10t^3 + 10t^4 + 5t^5 + 1t^6

    Use prossor size = 3
```

```
End of this running

Read from data file "dataIn.txt"
p(t) = 1 + 3t^1 + 3t^2 + 1t^3
q(t) = 1t^1 + 2t^2 + 1t^3

After FFT r(t)=p(t)q(t)
r(t) = 1t^1 + 5t^2 + 10t^3 + 10t^4 + 5t^5 + 1t^6

Use prossor size = 4
End of this running

Read from data file "dataIn.txt"
p(t) = 1 + 3t^1 + 3t^2 + 1t^3
q(t) = 1t^1 + 2t^2 + 1t^3

After FFT r(t)=p(t)q(t)
r(t) = 1t^1 + 5t^2 + 10t^3 + 10t^4 + 5t^5 + 1t^6

Use prossor size = 5
End of this running
```

说明：运行中可以使用参数 ProcessSize，如 mpirun －np ProcessSize fft 来运行该程序，其中 ProcessSize 是所使用的处理器个数，本实例中依次取 1、2、3、4、5 个处理器分别进行计算。