1 排序

排序是数据处理中经常使用的一种重要运算，如何进行排序，特别是如何进行高效的排序，是计算机应用中的重要课题。排序的对象一般是一组记录组成的文件，而记录则是由若干数据项组成，其中的一项可用来标志一个记录，称为关键字项，该数据项的值称为关键字。所谓排序，就是要整理文件中的记录，使得它按关键字递增（或递减）的次序排列起来。若给定的文件含有n个记录｛R_1，R_2，…，R_n｝，它们的关键字分别为｛K_1，K_2，…，K_n｝，要把这n个记录重新排列成为｛R_{i1}，R_{i2}，…，R_{in}｝，使得｛$K_{i1}\geqslant K_{i2}\geqslant\cdots\geqslant K_{in}$｝（或｛$K_{i1}\leqslant K_{i2}\leqslant\cdots\leqslant K_{in}$｝）。

本章主要介绍了枚举排序、快速排序、PSRS 排序算法以及它们的 MPI 编程实现。

1.1 枚举排序

1.1.1 枚举排序及其串行算法

枚举排序（Enumeration Sort）是一种最简单的排序算法，通常也称为**秩排序**（Rank Sort）。该算法的具体思想是（假设按关键字递增排序），对每一个待排序的元素统计小于它的所有元素的个数，从而得到该元素最终处于序列中的位置。假定待排序的n个数存在a[1]…a[n]中。首先将a[1]与a[2]…a[n]比较，记录比其小的数的个数，令其为k，a[1]就被存入有序的数组b[1]…b[n]的b[k+1]位置上；然后将a[2]与a[1]，a[3]…a[n]比较，记录比其小的数的个数，依此类推。这样的比较操作共n(n-1)次，所以串行秩排序的时间复杂度为O（n^2）。

算法 13.1　枚举排序串行算法

输入：a[1]…a[n]

输出：b[1]…b[n]

```
Begin
    for i=1 to n do
    (1)  k=1
    (2)  for j=1 to n do
                if a[i]>a[j] then
                    k=k+1
                end if
         end for
    (3)  b[k]= a[i]
    end for
End
```

1.1.2 枚举排序的并行算法

对该算法的并行化是很简单的，假设对一个长为n的输入序列使用n个处理器进行排序，只需是每个处理器负责完成对其中一个元素的定位，然后将所有的定位信息集中到主进程中，由主进程负责完成所有元素的最终排位。该并行算法描述如下：

算法 13.2　枚举排序并行算法

输入：无序数组 a[1]…a[n]
输出：有序数组 b[1]…b[n]
Begin
(1) P_0**播送**a[1]…a[n]给所有P_i
(2) **for all** P_i **where** 1≤i≤n **para-do**
(2.1)k=1
(2.2)**for** j = 1 **to** n **do**
if (a[i] > a[j]) **or** (a[i] = a[j] **and** i>j) **then**
k = k+1
end if
end for
(3) P_0**收集**k并按序定位
End

在该并行算法中，使用了n个处理器，由于每个处理器定位一个元素，所以步骤⑵的时间复杂度为O（n）；步骤⑶中主进程完成的数组元素重定位操作的时间复杂度为O（n），通信复杂度分别为O（n）；同时⑴中的通信复杂度为O（n^2）；所以总的计算复杂度为O（n），总的通信复杂度为O（n^2）。

MPI 源程序请参见所附光盘。

1.2 快速排序

1.2.1 快速排序及其串行算法

快速排序（Quick Sort）是一种最基本的排序算法，它的基本思想是：在当前无序区R[1，n]中取一个记录作为比较的“基准”（一般取第一个、最后一个或中间位置的元素），用此基准将当前的无序区 R[1，n]划分成左右两个无序的子区 R[1，i-1]和 R[i，n]（1≤i≤n），且左边的无序子区中记录的所有关键字均小于等于基准的关键字，右边的无序子区中记录的所有关键字均大于等于基准的关键字；当 R[1，i-1]和 R[i，n]非空时，分别对它们重复上述的划分过程，直到所有的无序子区中的记录均排好序为止。

算法 13.3　单处理机上快速排序算法
输入：无序数组 data[1,n]
输出：有序数组 data[1,n]
Begin
call procedure quicksort(data,1,n)
End

procedure quicksort(data,i,j)
Begin
(1) **if** (i<j) **then**
(1.1)r = **partition**(data,i,j)
(1.2)**quicksort**(data,i,r-1);
(1.3)**quicksort**(data,r+1,j);
end if
End

procedure partition(data,k,*l*)
Begin
(1) pivo=data[*l*]
(2) i=k-1
(3) **for** j=k **to** *l*-1 **do**
if data[j]≤pivo **then**
i=i+1
exchange data[i] and data[j]
end if
end for
(4) **exchange** data[i+1] and data[*l*]
(5) **return** i+1
End

快速排序算法的性能主要决定于输入数组的划分是否均衡，而这与基准元素的选择密切相关。在最坏的情况下，划分的结果是一边有n-1 个元素，而另一边有 0 个元素（除去被选中的基准元素）。如果每次递归排序中的划分都产生这种极度的不平衡，那么整个算法的复杂度将是 Θ（n^2）。在最好的情况下，每次划分都使得输入数组平均分为两半，那么算法的复杂度为O（n*log*n）。在一般的情况下该算法仍能保持O（n*log*n）的复杂度，只不过其具有更高的常数因子。

1.2.2 快速排序的并行算法

快速排序算法并行化的一个简单思想是，对每次划分过后所得到的两个序列分别使用两个处理器完成递归排序。例如对一个长为n的序列，首先划分得到两个长为n/2 的序列，将其交给两个处理器分别处理；而后进一步划分得到四个长为n/4 的序列，再分别交给四个处理器处理；如此递归下去最终得到排序好的序列。当然这里举的是理想的划分情况，如果划分步骤不能达到平均分配的目的，那么排序的效率会相对较差。算法 13.4 中描述了使用 2^m个处理器完成对n个输入数据排序的并行算法。

算法 13.4　快速排序并行算法
输入：无序数组data[1,n]，使用的处理器个数 2^m
输出：有序数组 data[1,n]
Begin
para_quicksort(data,1,n,m,0)
End

procedure para_quicksort(data,i,j,m,id)
Begin
(1) **if** (j-i)≤k **or** m=0 **then**
(1.1) P_{id} **call quicksort**(data,i,j)
else
(1.2) $P_{id:}$ r=**patrition(**data,i,j)
(1.3) P_{id} **send** data[r+1,m-1] to $P_{id+2^{m-1}}$
(1.4) para_quicksort(data,i,r-1,m-1,id)
(1.5) para_quicksort(data,r+1,j,m-1,id+2^{m-1})

(1.6)P_{id+2}^{m-1} **send** data[r+1,m-1] back to P_{id}

end if

End

在最优的情况下该并行算法形成一个高度为*log*n的排序树，其计算复杂度为O（n），通信复杂度也为O（n）。同串行算法一样，在最坏情况下其计算复杂度降为O（n^2）。正常情况下该算法的计算复杂度也为O（n），只不过具有更高的常数因子。

MPI 源程序请参见所附光盘。

1.3 PSRS 排序

1.3.1 PSRS 算法原理

并行正则采样排序，简称 PSRS(Parallel Sorting by Regular Sampling)它是一种基于**均匀划分**（Uniform Partition）原理的负载均衡的并行排序算法。假定待排序的元素有 n 个，系统中有 p 个处理器，那么系统首先将 n 个元素均匀地分割成 p 段，每段含有 n/p 个元素，每段指派一个处理器，然后各个处理器同时施行局部排序。为了使各段中诸局部有序的元素在整个序列中也能占据正确的位置，那么就首先从各段中抽取几个代表元素，再从他们产生出 p-1 个主元，然后按这些主元与原各局部有序中的元素之间的偏序关系，将各个局部有序段划分成 p 段，接着通过全局交换将各个段中的对应部分集合在一起，最后将这些集合在一起的各部分采用多路归并的方法进行排序，这些有序段汇合起来就自然成为全局有序序列了。

1.3.2 PSRS 算法形式化描述

设有 n 个数据，P 个处理器，以及均匀分布在 P 个处理器上的 n 个数据。 则 PSRS 算法可描述如下：

算法 13.5　PSRS 排序算法

输入：n 个待排序的数据，均匀地分布在 P 个处理器上

输出：分布在各个处理器上，得到全局有序的数据序列

Begin

(1) 每个处理器将自己的 n/P 个数据用串行快速排序(Quicksort)，得到一个排好序的序列；

(2) 每个处理器从排好序的序列中选取第 w，2w，3w，…，(P-1)w 个共 P-1 个数据作为代表元素，其中$w=n/P^2$；

(3) 每个处理器将选好的代表元素送到处理器P_0中，并将送来的P段有序的数据序列做 P 路归并，再选择排序后的第 P-1，2(P-1)，…，(P-1)(P-1)个共 P-1 个主元；

(4) 处理器P_0将这P-1 个主元播送到所有处理器中；

(5) 每个处理器根据上步送来的 P-1 个主元把自己的 n/P 个数据分成 P 段，记 w_i^j 为处理器P_i的第j+1 段，其中i=0，…，P-1，j=0，…，P-1；

(6) 每个处理器送它的第i+1 段给处理器P_i，从而使得第i个处理器含有所有处理器的第 i 段数据(i=0，…，P-1)；

(7) 每个处理器再通过 P 路归并排序将上一步的到的数据排序;从而这 n 个数据便是有序的。

End

在该算法中，针对其中的计算部分⑴中的快速排序的代价为$O(k\log k)$，其中 k=n/p；第⑵步中各个处理器选择P个主元的代价为$O(P)$；⑶中 对P^2个主元进行归并并选取新的主元所需代价为$O(P^2\log P)$；⑸中对数据划分的代价为$O(P+n/p)$；最后第⑺步中各个处理器进行并行归并的代价为$O(n/p\log P)$。针对通信部分⑶中处理器P_0收集P^2个主元的代价为$O(P^2)$；⑷中播送新选择的P个主元的代价为$O(P)$；最后 第⑹步的多对多播送的通信复杂度与具体的算法实现相关，其最大不会超过$O(n)$。

MPI 源程序请参见章末附录。

1.4 小结

本章主要讨论了几个典型的并行排序算法，关于枚举匹配算法的具体讨论可参考［1］；快速排序算法可以参考文献［2］和［3］中的介绍；文献［4］和［5］讨论了 PSRS 排序算法。

参考文献

[1]. Chabbar E, Controle, gestion du parallelisme: tris synchrones et asynchrones. Thesis Universite de Franche-comte, France:1980

[2]. Lorin H. Sorting and sort systems. Don Mills, Ontario: Addison-Wesley, 1975, 347~365

[3]. 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

[4]. Shi H, Schaeffer J. Parallel Sorting by Regular Sampling. Journal of Parallel and Distributed Computing, 14(4), 1992

[5]. Li X, Lu P, Schaeffer J, Shillington J, Wong P S, Shi H. On the Versatility of Parallel Sorting by Regular Sampling. Parallel Computing, 19, 1993

附录 PSRS 算法 MPI 源程序

1. 源程序 psrs_sort.c

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

#define INIT_TYPE 10
#define ALLTOONE_TYPE 100
#define ONETOALL_TYPE 200
#define MULTI_TYPE 300
#define RESULT_TYPE 400
#define RESULT_LEN 500
#define MULTI_LEN 600

int Spt;
long DataSize;
int *arr,*arr1;
int mylength;
int *index;
int *temp1;

main(int argc,char* argv[])
{
    long BaseNum = 1;
    int  PlusNum;
    int  MyID;

    MPI_Init(&argc,&argv);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD,
                &MyID);
        PlusNum=60;
        DataSize = BaseNum*PlusNum;

        if (MyID==0)
                printf("The DataSize is :
                        %lu\n",PlusNum);
        Psrs_Main();

        if (MyID==0)
                printf("\n");

            MPI_Finalize();
}

Psrs_Main( )
{
        int i,j;
        int MyID,SumID;
        int n,c1,c2,c3,c4,k,l;
        FILE * fp;
        int ready;
        MPI_Status status[32*32*2];
        MPI_Request request[32*32*2];

        MPI_Comm_rank(MPI_COMM_WORLD,
                &MyID);
        MPI_Comm_size(MPI_COMM_WORLD,
                &SumID);
        Spt = SumID-1;

        /*初始化参数*/
        arr = (int *)malloc(2*DataSize*sizeof(int));
        if (arr==0)
                merror("malloc memory for arr error!");
        arr1 = &arr[DataSize];

        if (SumID>1)
        {
                temp1 = (int *)malloc(sizeof(int)*
                        SumID*Spt);
                if (temp1==0) merror("malloc memory for
                        temp1 error!");
                index = (int *)malloc(sizeof(int)*
                        2*SumID);
                if (index==0) merror("malloc memory for
                        index error!");
        }

        MPI_Barrier( MPI_COMM_WORLD);

        mylength = DataSize / SumID;
        srand(MyID);

        printf("This is node %d \n",MyID);
        printf("On node %d the input data is:\n",MyID);
        for (i=0;i<mylength;i++)
        {
                arr[i] = (int)rand();
                printf("%d : ",arr[i]);
        }
        printf("\n");

        /*每个处理器将自己的 n/P 个数据用串行快速
                排序(Quicksort)，得到一个排好序的序
                列，对应于算法 13.5 步骤（1）*/
        MPI_Barrier( MPI_COMM_WORLD);
        quicksort(arr,0,mylength - 1);
        MPI_Barrier( MPI_COMM_WORLD);

        /*每个处理器从排好序的序列中选取第 w，2w，
                3w，…，(P-1)w 个共 P-1 个数据作为代
                表元素，其中 w=n/P*P，对应于算法 13.5
                步骤（2）*/
        if (SumID>1)
        {
                MPI_Barrier(MPI_COMM_WORLD);
                n = (int)(mylength/(Spt+1));
                for (i=0;i<Spt;i++)
                        temp1[i] = arr[(i+1)*n-1];

                MPI_Barrier(MPI_COMM_WORLD);

                if (MyID==0)
                {
                        /*每个处理器将选好的代表元素送
                                到处理器 P0 中，对应于算法
```

```
            13.5 步骤（3） */
        j = 0;
        for (i=1;i<SumID;i++)
            MPI_Irecv(&temp1[i*Spt],
                sizeof(int)*Spt,
                MPI_CHAR,
                i,ALLTOONE_TYPE+i,
                MPI_COMM_WORLD,
                &request[j++]);
        MPI_Waitall(SumID-1, request,
            status);

        /* 处理器 P0 将上一步送来的 P 段
            有序的数据序列做 P 路归并，
            再选择排序后的第 P-1，
            2(P-1)，…，(P-1)(P-1)个共 P-1
            个主元，，对应于算法 13.5 步
            骤（3）*/
        MPI_Barrier(
            MPI_COMM_WORLD);
        quicksort(temp1,0,SumID*Spt-1);
        MPI_Barrier(
            MPI_COMM_WORLD);
        for (i=1;i<Spt+1;i++)
            temp1[i] = temp1[i*Spt-1];
        /*处理器 P0 将这 P-1 个主元播送到
            所有处理器中，对应于算法
            13.5 步骤（4）*/
        MPI_Bcast(temp1,
            sizeof(int)*(1+Spt),
            MPI_CHAR, 0,
            MPI_COMM_WORLD);
        MPI_Barrier(
            MPI_COMM_WORLD);
    }
    else
    {
        MPI_Send(temp1,sizeof(int)*Spt,
            MPI_CHAR,0,
            ALLTOONE_TYPE+MyID,
            MPI_COMM_WORLD);
        MPI_Barrier(
            MPI_COMM_WORLD);
        MPI_Barrier(
            MPI_COMM_WORLD);
        MPI_Bcast(temp1,
            sizeof(int)*(1+Spt),
            MPI_CHAR, 0,
            MPI_COMM_WORLD);
        MPI_Barrier(
            MPI_COMM_WORLD);
    }
    /*每个处理器根据上步送来的 P-1 个主
        元把自己的 n/P 个数据分成 P 段，
        记为处理器 Pi 的第 j+1 段，其中
        i=0,…,P-1，j=0,…,P-1，对应于算
        法 13.5 步骤（5）*/
    n = mylength;
    index[0] = 0;
    i = 1;
    while ((arr[0]>=temp1[i])&&(i<SumID))
    {
        index[2*i-1] = 0;
        index[2*i] = 0;
        i++;
    }
    if (i==SumID)
        index[2*i-1] = n;
    c1 = 0;
    while (i<SumID)
    {
        c4 = temp1[i];
        c3 = n;
        c2 = (int)((c1+c3)/2);
        while ((arr[c2]!=c4)&&(c1<c3))
        {
            if (arr[c2]>c4)
            {
                c3 = c2-1;
                c2 = (int)((c1+c3)/2);
            }
            else
            {
                c1 = c2+1;
                c2 = (int)((c1+c3)/2);
            }
        }
        while ((arr[c2]<=c4)&&(c2<n))
```

```
            c2++;
        if (c2==n)
        {
            index[2*i-1] = n;
            for (k=i;k<SumID;k++)
            {
                index[2*k] = 0;
                index[2*k+1] = 0;
            }
            i = SumID;
        }
        else
        {
            index[2*i] = c2;
            index[2*i-1] = c2;
        }
        c1 = c2;
        c2 = (int)((c1+c3)/2);
        i++;
}
if (i==SumID)
        index[2*i-1] = n;
MPI_Barrier( MPI_COMM_WORLD);

/*每个处理器送它的第 i+1 段给处理器
        Pi，从而使得第 i 个处理器含有所
        有处理器的第 i 段数据
        (i=0,…,P-1)，，对应于算法 13.5 步
        骤（6）*/
j = 0;
for (i=0;i<SumID;i++)
{
        if (i==MyID)
        {
            temp1[i] = index[2*i+1]-
                index[2*i];
            for (n=0;n<SumID;n++)
                if (n!=MyID)
                {
                    k = index[2*n+1]-
                        index[2*n];
                    MPI_Send(&k,
                        sizeof(int),
                        MPI_CHAR,
                        n,
                        MULTI_LEN
                        +MyID,
                        MPI_COMM
                        _WORLD);
                }
        }
        else
        {
            MPI_Recv(&temp1[i],
                sizeof(int),  MPI_CHAR,
                i,MULTI_LEN+i,
                MPI_COMM_WORLD,
                &status[j++]);
        }
}

MPI_Barrier(MPI_COMM_WORLD);

j = 0;
k = 0;
l = 0;
for (i=0;i<SumID;i++)
{
        MPI_Barrier(
            MPI_COMM_WORLD);
        if (i==MyID)
        {
            for (n=index[2*i];
                n<index[2*i+1];
                n++)
                arr1[k++] = arr[n];
        }
        MPI_Barrier(
            MPI_COMM_WORLD);
        if (i==MyID)
        {
            for (n=0;n<SumID;n++)
                if (n!=MyID)
                {
                    MPI_Send(&arr[
                        index[2*n]],
                        sizeof(int)*
                        (index[2*n+1
```

```
                                    ]-index[2*n])
                                    ,MPI_CHAR,
                                    n,
                                    MULTI_TYP
                                    E+MyID,
                                    MPI_COMM
                                    _WORLD);
                            }

                    }
                    else
                    {
                            l = temp1[i];
                            MPI_Recv(&arr1[k],
                                    l*sizeof(int),
                                    MPI_CHAR,
                                    i ,MULTI_TYPE+i,
                                    MPI_COMM_WORLD,
                                    &status[j++]);
                            k=k+l;
                    }
                    MPI_Barrier(
                            MPI_COMM_WORLD);
            }
            mylength = k;
            MPI_Barrier(MPI_COMM_WORLD);

            /*每个处理器再通过 P 路归并排序将上
                一步的到的数据排序；从而这 n 个
                数据便是有序的，，对应于算法 13.5
                步骤（7） */
            k = 0;
            multimerge(arr1,temp1,arr,&k,SumID);
            MPI_Barrier(MPI_COMM_WORLD);
    }

    printf("On node %d the sorted data is :
            \n",MyID);
    for (i=0;i<mylength;i++)
            printf("%d : ",arr[i]);
    printf("\n");
}

/*输出错误信息*/
merror(char* ch)
{
        printf("%s\n",ch);
       exit(1);
}

/*串行快速排序算法*/
quicksort(int *datas,int bb,int ee)
{
        int tt,i,j;
        t = datas[bb];
          = bb;
        j = ee;

        if (i<j)
        {
                while(i<j)
                {
                        while ((i<j)&&(tt<=datas[j]))
                                j--;
                                if (i<j)
                                {
                                        datas[i] = datas[j];
                                        i++;
                                        while ((i<j)&&
                                                (tt>datas[i]))
                                                 i++;
                                        if (i<j)
                                        {
                                                datas[j] = datas[i];
                                                j--;
                                                if (i==j)
                                                        datas[i] = tt;
                                        }
                                        else
                                                datas[j] = tt;
                                }
                                else
                                        datas[i] = tt;
                }

                quicksort(datas,bb,i-1);
                quicksort(datas,i+1,ee);
        }
```

```
}

/*串行多路归并算法*/
multimerge(int *data1,int *ind,int *data,int *iter,int
        SumID)
{
        int i,j,n;

        j = 0;
        for (i=0;i<SumID;i++)
                if (ind[i]>0)
                {
                        ind[j++] = ind[i];
                        if (j<i+1) ind[i] = 0;
                }
                if ( j>1 )
                {
                        n = 0;
                        for (i =0;i<j,i+1<j;i=i+2)
                        {
                                merge(&(data1[n]),ind[i],
                                        ind[i+1],&(data[n]));
                                ind[i] += ind[i+1];
                                ind[i+1] = 0;
                                n += ind[i];
                        }
                        if (j%2==1 )
                                for (i=0;i<ind[j-1];i++)
                                        data[n++]=data1[n];
                        (*iter)++;
                        multimerge(data,ind,data1,iter,
                                SumID);
                }
}

merge(int *data1,int s1,int s2,int *data2)
{
        int i,l,m;

        l = 0;
        m = s1;
        for (i=0;i<s1+s2;i++)
        {
                if (l==s1)
                        data2[i]=data1[m++];
                else
                        if (m==s2+s1)
                                data2[i]=data1[l++];
                        else
                                if (data1[l]>data1[m])
                                        data2[i]=data1[m++];
                                else
                                        data2[i]=data1[l++];
        }
}
```

2. 运行实例

编译：mpicc psrs_sort.c –o psrs

运行：可以使用命令 mpirun –np SIZE psrs 来运行该串匹配程序，其中 SIZE 是所使用的处理器个数，本实例中使用了 SIZE=3 个处理器。

mpirun –np 3 psrs

运行结果：

The DataSize is : 60

This is node 0

On node 0 the input data is:

0 : 21468 : 9988 : 22117 : 3498 : 16927 : 16045 : 19741 : 12122 : 8410 : 12261 : 27052 : 5659 : 9758 : 21087 : 25875 : 32368 : 26233 : 15212 : 17661 :

On node 0 the sorted data is :

0 : 2749 : 3498 : 4086 : 5627 : 5659 : 5758 : 7419 : 8410 : 9084 : 9758 : 9988 : 703 : 908 : 2294 : 9212 :

This is node 1

On node 1 the input data is:

16838 : 5758 : 10113 : 17515 : 31051 : 5627 : 23010 : 7419 : 16212 : 4086 : 2749 : 12767 : 9084 : 12060 : 32225 : 17543 : 25089 : 21183 : 25137 : 25566 :

On node 1 the sorted data is :

10113 : 12060 : 12122 : 12261 : 12767 : 15212 : 16045 : 16212 : 16838 : 16927 :17515 : 10239 : 10595 : 12508 : 12914 : 14363 : 16134 :

This is node 2

On node 2 the input data is:

908 : 22817 : 10239 : 12914 : 25837 : 27095 : 29976 : 27865 : 20302 : 32531 : 26005 : 31251 : 12508 : 14363 : 10595 : 9212 : 17811 : 16134 : 2294 : 703 :

On node 2 the sorted data is :

17543 : 17661 : 19741 : 21087 : 21183 : 21468 : 22117 : 23010 : 25089 : 25137 : 25566 : 25875 : 26233 : 27052 : 31051 : 32225 : 32368 : 17811 : 20302 : 22817 : 25837 : 26005 : 27095 : 27865 : 29976 : 31251 : 32531 :

说明：本程序中通过变量 DataSize 指定了待排序序列的长度为 60，顺序输出各个处理器的局部数据就可以得到全局有序的序列。

2 串匹配

串匹配（String Matching）问题是计算机科学中的一个基本问题，也是复杂性理论中研究的最广泛的问题之一。它在文字编辑处理、图像处理、文献检索、自然语言识别、生物学等领域有着广泛的应用。而且，串匹配是这些应用中最耗时的核心问题，好的串匹配算法能显著地提高应用的效率。因此，研究并设计快速的串匹配算法具有重要的理论价值和实际意义。

串匹配问题实际上就是一种模式匹配问题，即在给定的文本串中找出与模式串匹配的子串的起始位置。最基本的串匹配问题是**关键词匹配**（Keyword Matching）。所谓关键词匹配，是指给定一个长为 n 的**文本串** *T*[1，n]和长为 m 的**模式串** *P*[1，m]，找出文本串 *T* 中与模式串所有精确匹配的子串的起始位置。串匹配问题包括**精确串匹配**（Perfect String Matching）、**随机串匹配**（Randomized String Matching）和**近似串匹配**（Approximate String Matching）。另外还有**多维串匹配**（Multidimensional String Matching）和**硬件串匹配**（Hardware String Matching）等。

本章中分别介绍改进的 KMP 串匹配算法，采用散列技术的随机串匹配算法，基于过滤算法的近似串匹配算法，以及它们的 MPI 编程实现。

1.1 KMP 串匹配算法

1.1.1 KMP 串匹配及其串行算法

KMP 算法首先是由 D.E. Knuth、J.H. Morris 以及 V.R. Pratt 分别设计出来的，所以该算法被命名为 KMP 算法。KMP 串匹配算的基本思想是：对给出的的文本串 *T*［1，n］与模式串 *P*[1，m]，假设在模式匹配的进程中，执行 *T*[i]和 *P*[j]的匹配检查。 若 *T*[i]=*P*[j]，则继续检查 *T*[i+1]和 *P*[j+1]是否匹配。若 *T*[i]≠*P*[j]，则分成两种情况：若 j=1，则模式串右移一位，检查 *T*[i+1]和 *P*[1]是否匹配；若 1<j≤m，则模式串右移 j－next(j)位，检查 *T*[i]和 *P*[next(j)]是否匹配（其中 next 是根据模式串 *P*[1，m]的本身局部匹配的信息构造而成的）。重复此过程直到 j=m 或 i=n 结束。

1. 修改的 KMP 算法

在原算法基础上很多学者作了一些改进工作，其中之一就是重新定义了 KMP 算法中的 next 函数，即求 next 函数时不但要求 *P*[1，next(j) －1]=*P*[j－(next(j) －1)，j-1]，而且要求 *P*[next(j)] ≠*P*[j]，记修改后的 next 函数为 newnext。在模式串字符重复高的情况下修改的 KMP 算法比传统 KMP 算法更加有效。

算法 14.1　修改的 KMP 串匹配算法

输入：文本串 *T*[1，n]和模式串 *P*[1，m]

输出：是否存在匹配位置

procedure modeifed_KMP

Begin

（1）i=1,j=1

（2）**while** i≤n **do**

(2.1)**while** j≠0 **and** P[j]≠T[i] **do**

```
            j=newnext[j]
        end while
    (2.2)if j=m then
            return true
        else
            j=j+1,i=i+1
        end if
    end while
(3) return false
End
```

算法 14.2　next 函数和 newnext 函数的计算算法

输入：模式串 *P*[1，m]

输出：next[1，m+1]和 newnext[1，m]

```
procedure next
Begin
(1) next[1]=0
(2) j=2
(3) while j≤m do
    (3.1)i=next[j-1]
    (3.2)while i≠0 and P[i]≠P[j-1] do
            i=next[i]
        end while
    (3.3)next[j]=i+1
    (3.4)j=j+1
    end while
End

procedure newnext
Begin
(1) newnext(1)=0
(2) j=2
(3) while j≤m do
    (3.1)i=next(j)
    (3.2)if i=0 or P[j]≠P[i+1] then
            newnext[j]=i
        else
            newnext[j]=newnext[i]
        end if
    (3.3)j=j+1
    end while
End
```

2. 改进的 KMP 算法

易知算法 14.1 的时间复杂度为 O（n），算法 14.2 的时间复杂度为 O（m）。算 法 14.1 中所给出的 KMP 算法只能找到第一个匹配位置，实际应用中往往需要找出所有的匹配位置。下面给出改进后的算法 14.3 便解决了这一问题。

算法 14.3 改进 KMP 串匹配算法

输入：文本串 T[1，n]和模式串 P[1，m]

输出：匹配结果 match[1，n]

```
procedure improved_KMP
Begin
(1) i=1，j=1
(2) while i≤n do
    (2.1)while j≠0 and P[j]≠T[i] do
            j=newnext[j]
        end while
    (2.2)if j=m then
            match[i-(m-1)]=1
            j=next[m+1]
            i=i+1
        else
            i=i+1
            j=j+1
        end if
    end while
(3) max_prefix_len=j-1
End
```

算法 14.4 next 函数和 newnext 函数的计算算法

输入：模式串 P[1，m]

输出：next[1，m+1]和 newnext[1，m]

```
procedure NEXT
Begin
(1) next[1]=newnext[1]=0
(2) j=2
(3) while j≤m+1 do
    (3.1)i=next[j-1]
    (3.2)while i≠0 and W[i]≠W[j-1]) do
            i=next[i]
        end while
    (3.3)next[j]=i+1
    (3.4)if j≠m+1 then
            if W[j] ≠W[i+1] then
                newnext[j]=i+1
            else
                newnext[j]=newnext[i+1]
```

end if

end if

(3.5)j=j+1

end while

End

在算法 14.3 中，内层 while 循环遇到成功比较时和找到文本串中模式串的一个匹配位置时文本串指针 i 均加 1，所以至多做 n 次比较；内 while 循环每次不成功比较时文本串指针 i 保持不变，但是模式串指针 j 减小，所以 i－j 的值增加且上一次出内循环时的 i－j 值等于下一次进入时的值，因此不成功的比较次数不大于 i－j 的值的增加值，即小于 n，所以总的比较次数小于 2n，所以算法 14.3 的时间复杂度为 *O*（n）。算法 14.4 只比的算法 14.2 多计算了 next(m+1)，至多多做 m-1 次比较，所以算法 14.4 的时间复杂度同样为 *O*（m）。

1.1.2 KMP 串匹配的并行算法

1. 算法原理

在介绍了改进的 KMP 串行算法基础上，这一节主要介绍如何在分布存储环境中对它进行实现。设计的思路为：将长为 n 的文本串 *T* 均匀划分成互不重叠的 p 段，分布于处理器 0 到 p-1 中，且使得相邻的文本段分布在相邻的处理器中，显然每个处理器中局部文本段的长度为$\lceil n/p \rceil$（最后一个处理器可在其段尾补上其它特殊字符使其长度与其它相同）。再将长为 m 的模式串 *P* 和模式串的 newnext 函数播送到各处理器中。各处理器使用改进的 KMP 算法并行地对局部文本段进行匹配，找到所有段内匹配位置。

但是每个局部段（第p-1 段除外）段尾m-1 字符中的匹配位置必须跨段才能找到。一个简单易行的办法就是每个处理器（处理器p-1 除外）将本局部段的段尾m-1 个字符传送给下一处理器，下一处理器接收到前一处理器传来的字符串后，再接合本段的段首m-1 个字符构成一长为 2(m-1)的段间字符串，对此字符串做匹配，就能找到所有段间匹配位置。但是算法的通信量很大，采用下述两种改进通信的方法可以大大地降低通信复杂度：①降低播送模式串和newnext函数的通信复杂度。利用串的周期性质，先对模式串*P*作预处理，获得其最小周期长度|U|、最小周期个数s及后缀长度|V|（$P=U^sV$），只需播送U，s，|V|和部分newnext函数就可以了，从而大大减少了播送模式串和newnext函数的通信量。而且串的最小周期和next函数之间的关系存在着下面定理 1 所示的简单规律，使得能够设计出常数时间复杂度的串周期分析算法。②降低每个处理器（处理器p-1 除外）将本局部文本段的段尾m-1 个字符传送给下一处理器的通信复杂度。每个处理器在其段尾m-1 个字符中找到模式串*P*的最长前缀串，因为每个处理器上都有模式串信息，所以只需传送该最长前缀串的长度就行了。这样就把通信量从传送模式串*P*的最长前缀串降低到传送一个整数，从而大大地降低了通信复杂度。而且KMP算法结束时模式串指针j-1 的值就是文本串尾模式串最大前缀串的长度，这就可以在不增加时间复杂度的情况下找到此最大前缀串的长度。

2. 串的周期性分析

定义 14.1：给定串*P*，如果存在字符串*U*以及正整数K≥2，使得串*P*是串U^k的**前缀**（Prefix），则称字符串*U*为串*P*的**周期**（Period）。字符 串*P*的所有周期中长度最短的周期称为串*P*的**最小周期**（Miximal Period）。

串的周期分析对最终并行算法设计非常关键，串的最小周期和 next 函数之间的关系存在着如下定理 14.1 所示的简单规律，基于该规律可以设计出常数时间复杂度的串周期分析

算法。

定理 14.1：已知串 *P* 长为 m，则 u=m+1－next(m+1)为串 *P* 的最小周期长度。

算法 14.5　串周期分析算法

输入：next[m+1]

输出：最小周期长度 period_len
　　最小周期个数 period_num
　　模式串的后缀长度 pattern-suffixlen

procedure PERIOD_ANALYSIS

Begin

period_len=m+1-next(m+1)

period_num=(int)m/period_len

pattern_suffixlen=m mod period_len

End

3. 并行算法描述

在前述的串行算法以及对其并行实现计的分析的基础上，我们可以设计如下的并行 KMP 串匹配算法。

算法 14.6　并行 KMP 串匹配算法

输入：分布存储的文本串T_i[1，$\lceil n/p \rceil$]（i=0，1，2，…，p-1）

　　存储于P_0的模式串*P*[1，m]

输出：所有的匹配位置

Begin

(1) P_0 **call procedure** NEXT　　/*调用算法 14.4，求模式串P的
　　next 函数和 newnext 函数*/
　　P_0 **call procedure** PERIOD_ANALYSIS　/*调用算法 14.5 分析P的周期*/

(2) P_0 **broadcast** period_len，period_num，period_suffixlen to other processors　/*播送
　　P 之最小周期长度、最小周期个数和后缀长度*/
　　P_0 **broadcast** P[1，period_len]　/*不播送P之最小周期*/
　　if period_num=1 **then**　/*播送 P 之部分 newnext 函数，如周期为 1，则播送整个
　　newnext 函数；否则播送 2 倍周期长的 newnext 函数*/
　　broadcast newnext [1，m]
　　else
　　broadcast newnext[1，2*period_len]
　　end if

(3) **for** i=1 **to** p-1 **par-do**　/*由传送来的 P 之周期和部分 newnext 函数重构整个模式串
　　和 newnext 函数*/
　　P_i **rebuild** newnext
　　end for
　　for i=1 **to** p-1 **par-do**　/*调用算法 14.7 做局部段匹配，并获得局部段尾最大前缀串
　　之长度*/
　　P_i **call procedure** KMP(T，P ，n ，0，match)
　　end for

(4) **for** i =0 **to** p-2 **par-do**

P_i **send** maxprefixlen to P_{i+1}

end for

for i=1 **to** p-1 **par-do**

P_i **receive** maxprefixlen from P_{i-1}

P_i **call procedure** KMP(Ti，P，m-1， maxprefixlen，match+m-1) /*调用算法 14.7 做段间匹配*/

end for

End

该算法中调用的 KMP 算法必须重新修改如下，因为做段间匹配时已经匹配了 maxprefixlen 长度的字符串，所以模式串指针只需从 maxprefixlen+1 处开始。

算法 14.7 重新修改的 KMP 算法

输入：分布存储的文本串和存储于P_0的模式串P[1，m]

输出：所有的匹配位置

procedure KMP(T，P，textlen， matched_num，match)

Begin

（1）i=1

（2）j=matched_num+1

（3）**while** i≤textlen **do**

(3.1)**while** j≠0 **and** P[j]≠T[i] **do**

j=newnext[j]

end while

(3.2)**if** j=m **then**

match[i-(m-1)]=1

j=next[m+1]

i=i+1

else

j=j+1

i=i+1

end if

end while

（4）maxprefixlen=j-1

End

下面从计算时间复杂度和通信时间复杂度两个方面来分析该算法的时间复杂度。在分析计算时间复杂度时，假定文本串初始已经分布在各个处理器上，这是合理的，因为文本串一般很大，不会有大的变化，只需要分布一次就可以，同时也假定结果分布在各处理器上。 本算法的计算时间由算法步（1）中所调用的算法 14.4 的时间复杂度 O（m）和算法 14.5 的时间复杂度 O（1）；算法步（3）和算法步（4）所调用的改进的 KMP 算法 14.7 的时间复杂度 O（n/p）和 O（m）构成。所以本算法总的计算时间复杂度为 O（n/p+m）。通信复杂度由算法步（2）播送模式串信息（最小周期串 U 及最小周期长度、周期个数和后缀长度三个整数）和 newnext 函数（长度为 2u 的整数数组，u 为串 U 的长度）以及算法步（4）传送最大前缀串长度组成，所以通信复杂度与具体采用的播送算法有关。若采用二分树通信技术，则总的通信复杂度为 O（$ulogp$）。

MPI 源程序请参见章末附录。

1.2 随机串匹配算法

1.2.1 随机串匹配及其串行算法

采用上一节所述的 KMP 算法虽然能够找到所有的匹配位置，但是算法的复杂度十分高，在某些领域并不实用。本节给出了随机串匹配算法，该算法的主要采用了**散列**（Hash）技术的思想，它能提供对数的时间复杂度。其基本思想是：为了处理模式长度为 m 的串匹配问题，可以将任意长为 m 的串映射到 $O(\log m)$ 整数位上，映射方法须得保证两个不同的串映射到同一整数的概率非常小。所得到的整数之被视为该串的**指纹**（Fingerprint），如果两个串的指纹相同则可以判断两个串相匹配。

1. 指纹函数

本节中假定文本串和模式串取字符集 $\Sigma=\{0, 1\}$ 中的字母。

如上所述，随机串匹配算法的基本策略是将串映射到某些小的整数上。令 T 是长度为 n 的文本串，P 是长度为 $m \leqslant n$ 的模式串，匹配的目的就是识别 P 在 T 中出现的所有位置。考虑长度为 m 的 T 的所有子串集合 B。这样的起始在位置 $i(1 \leqslant i \leqslant n-m+1)$ 的子串共有 n-m+1 个。于是问题可重新阐述为去识别与 P 相同的 B 中元素的问题。该算法中最重要的是如何选择一个合适的**映射函数**（Mapping Function），下面将对此进行简单的讨论。

令 $F=\{f_p\}_{p\in s}$ 是函数集，使得 f_p 将长度为 m 的串映射到域 D，且要求集合 F 满足下述三个性质：①对于任意串 $X\in B$ 以及每一个 $p\in S$（S 为模式串的取值域），$f_p(X)$ 由 $O(\log m)$ 位组成；②随机选择 $f_p \in F$，它将两个不等的串 X 和 Y 映射到 D 中同一元素的概率是很小的；③对于每个 $p\in S$ 和 B 中所有串，应该能够很容易的并行计算 $f_p(X)$。

上述三个性质中，性质①隐含着 $f_p(X)$ 和 $f_p(P)$ 可以在 $O(1)$ 时间内比较，其中 $f_p(X)$ 被称为串 X 的指纹；性质②是说，如果两个串 X 和 Y 的指纹相等，则 X=Y 的概率很高；性质③主要是针对该算法的并行实现的需求对集合 F 加以限制。对于串行算法函数 f_p 只需要满足前两个性质即可。

本节中我们采用了这样一类指纹函数：将取值 $\{0, 1\}$ 上的串 X 集合映射到取值整数环 Z 上的 2×2 矩阵。令 $f(0)$ 和 $f(1)$ 定义如下：

$$f(0)=\begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix},\qquad f(1)=\begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix}$$

该函数目前只满足性质 2，为了使其同时满足性质 1 需要对该函数作如下更改：

令 p 是区间 $[1, 2, \cdots, M]$ 中的一个素数，其中 M 是一个待指定的整数；令 Z_p 是取模 p 的整数环。对于每个这样的 p，我们定义 $f_p(X)$ 为 $f(X) \bmod p$，即 $f_p(X)$ 是一个 2×2 的矩阵，使得 $f_p(X)$ 的（i，j）项等于 $f(X)$ 的相应项取模 p。

由此构造的函数 f_p 同时满足前述性质 1 和 2。

2. 串行随机串匹配算法

用上面定义的指纹函数可以构造一个随机串匹配算法。先计算模式串 P（1，m）和子串 T（i，i+m-1）的指纹函数（其中 1≤i≤n-m+1，m≤n），然后每当 P 的指纹和 T（i，i+m-1）的指纹相等时，则标记在文本 T 的位置 i 与 P 出现匹配。算法描述如下：

算法 14.8　串行随机串匹配算法

输入：数组 T(1，n)和 P(1，m)，整数 M。

输出：数组 MATCH，其分量指明 P 在 T 中出现的匹配位置。

```
Begin
(1) for i=1 to n-m+1 do
        MATCH(i)=0
    end for
(2) 在区间[1，2，…，M]中随机的选择一素数，并计算 f_p(P)
(3) for i=1 to n-m+1 do
        Li= f_p(T(i,i+m-1))
    end for
(4) for i=1 to n-m+1 do
        if Li= f_p(P) then
            MATCH(i)=1
        end if
    end for
End
```

很显然在该算法中步骤（1）和（4）的时间复杂度均为 O（n-m)；步骤（2）和（3）中求模式串和文本串各个子串的指纹的时间复杂度分别 O（m）和 O（n-m)。

1.2.2 随机串匹配的并行算法

对上述串行算法的并行化主要是针对算法 14.7 中步骤（3）的并行处理，也就是说需要选择一个合适的函数 f_p 使其同时满足上述三个性质。前面一节给出了同时满足前两个性质的函数，这里我们主要针对性质 3 来讨论已得指纹函数类 F。

函数类 $F=\{f_p\}$ 的一个关键性质就是每个 f_p 都是**同态**（Homomorphic）的，即对于任意两个串 X 和 Y，$f_P(XY)=f_p(X)f_p(Y)$。下面给出了一个有效的并行算法来计算文本串 T 中所有子串的指纹。

对于每个 1 ≤ i ≤ n-m+1，令 $N_i=f_p(T(1,i))$，易得 $N_i=f_p(T(1))f_p(T(2))\cdots f_p(T(i))$。因为矩阵乘法是可结合的，所以此计算就是一种前缀

和的计算；同时每个矩阵的大小为2×2，因此这样的矩阵乘法可以在O（1）时间内完成。所以，所有的N_i都可以在O（log$n$）时间内，总共使用O（n）次操作计算。

定义 14.2： $g_p(0)=f_p(0)^{-1}=\begin{bmatrix}1 & 0\\ p-1 & 1\end{bmatrix}, g_p(1)=f_p(1)^{-1}=\begin{bmatrix}1 & p-1\\ 0 & 1\end{bmatrix}$。则乘积 $R_i=g_p(T(i))g_p(T(i-1))\cdots g_p(T(1))$ 也是一个前缀和计算，并且对于$1\le i\le n$，它可以在 O（log$n$）时间内运用 O（n）次操作计算。

容易看到，$f_p(T(i,i+m-1))=R_{i-1}N_{i+m-1}$，因此，一旦所有的$R_i$和$N_i$都计算出来了，则每个这样的矩阵均可在$O$（1）时间内计算之。所以，长度为n的正文串$T$中所有长度为m≤n的子串的指纹均可在$O$（log$n$）时间内使用$O$（n）次操作而计算之。

这样，函数 f_p 同时满足了前述三个性质。在此基础上我们给出了在分布式存储体系结构上随机串匹配的并行算法。

算法 14.9　并行随机串匹配算法

输入：数组 T(1，n)和 P(1，m)，整数 M。

输出：数组 MATCH，其分量指明 P在 T中出现的位置。

Begin

(1) for i=1 to n-m+1 par-do
　　MATCH(i)=0;
　end for

(2) Select a random prime number in [1，2，…，M]，then count $f_p(P)$。

(3) for i=1 to n-m+1 par-do
　　Li= $f_p(T(i,i+m-1))$;
　end for

(4) for i=1 to n-m+1 par-do
　　if Li= $f_p(P)$ then
　　　MATCH(i)=1
　　end if
　end for

End

该并行算法的计算复杂度为 O（log$n$）。处理期间的通信包括在对文本串到各个处理器的分派，其通信复杂度为 O（$n/p+m$）；以及匹配信息的收集，其通信复杂度为 O（n/p）。

MPI 源程序请参见所附光盘。

1.3 近似串匹配算法

1.3.1 近似串匹配及其串行算法

前两节所讨论的串匹配算法均属于精确串匹配技术，它要求模式串与文本串的子串完全

匹配，不允许有错误。然而在许多实际情况中，并不要求模式串与文本串的子串完全精确地匹配，因为模式串和文本串都有可能并不是完全准确的。例如，在检索文本时，文本中可能存在一些拼写错误，而待检索的关键字也可能存在输入或拼写错误。在这种情况下的串匹配问题就是近似串匹配问题。

近似串匹配问题主要是指按照一定的近似标准，在文本串中找出所有与模式串近似匹配的子串。近似串匹配问题的算法有很多，按照研究方法的不同大致分为动态规划算法，有限自动机算法，过滤算法等。但上述所有算法都是针对一般的近似串匹配问题，也就是只允许有插入、删除、替换这三种操作的情况。本节中还考虑了另外一种很常见的错误—换位，即，文本串或模式串中相邻两字符的位置发生了交换，这是在手写和用键盘进行输入时经常会发生的一类错误。为修正这类错误引入了换位操作，讨论了允许有插入、删除、替换和换位四种操作的近似串匹配问题。

1. 问题描述:

给定两个长度分别为m和n的字符串A[1，m]和B[1，n]，a_i，$b_j \in \Sigma$ ($1 \leqslant i \leqslant m$，$1 \leqslant j \leqslant n$，$\Sigma$是字母表)，串A与串B的**编辑距离**（Editor Distance）是指将串A变成串B所需要的最少编辑操作的个数。最常见的编辑操作有三种：①**插入**(Insert)，向串A中插入一个字符；②**删除**(Delete)，从串A中删除一个字符；③**替换**(Replace)，将串A中的一个字符替换成串B中的一个字符。其中，每个编辑操作修正的串A与串B之间的一个不同之处称为一个**误差**或者**错误**。

最多带k个误差的串匹配问题（简称为**k-differences**问题）可描述如下：给定一个长度为n的文本串$T=t_1 \cdots t_n$，一个长度为m的模式串$P=p_1 \cdots p_m$，以及整数k（$k \geqslant 1$），在文本串T中进行匹配查找，如果T的某个子串$t_i \cdots t_j$与模式串P的编辑距离小于等于k，则称在t_j处匹配成功，要求找出所有这样的匹配结束位置t_j。

另外一种很常见的编辑操作是**换位**(Exchange)：将串 A 中的两个相邻字符进行交换。该操作用于修正相邻两字符位置互换的错误。一个换位操作相当于一个插入操作加上一个删除操作。但是，当近似匹配允许的最大误差数 k 一定时，允许有换位操作的情况较之不允许有换位操作的情况，往往能够找出更多的匹配位置。

例如，假定文本串 T=abcdefghij，模式串 P=bxcegfhy，k=4，问在文本串的第 9 个字符处是否存在最多带 4 个误差的匹配？

首先考虑允许有换位操作的情况，文本串与模式串的对应关系如下：

	t_1	t_2		t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
T:	a	b	_	c	d	e	f	g	h	i	j
P:	b	x	c	_	e	g	f		h	y	
		①		②		③				④	

其中，①，②，③，④ 4 个位置分别对应于删除、插入、换位和替换操作，可见在t_9处确实有最多带 4 个误差的匹配。

不允许有换位操作的情况，对应关系如下：

	t_1	t_2		t_3	t_4	t_5	t_6	t_7		t_8	t_9	t_{10}
T:	a	b	_	c	d	e	f	g	_	h	i	j
P:		b	x	c	_	e	_	g	f	h	y	
			①		②		③		④		⑤	

可以看出，在t_9处不存在最多带 4 个误差的匹配，因为匹配成功所需要的最少编辑操作个数为 5。

由此可见，允许有换位操作的近似串匹配算法比以往未考虑换位操作的算法更加实用有效。尤其是在文本检索和手写体识别等实际应用中，新算法的检索成功率和识别率更高，准

确性更好，功能更强。

2. k-differences 问题的动态规划算法

一般的k-differences问题的一个著名算法采用了**动态规划**（Dynamic Programming）的技术，可以在O（mn）时间内解决该问题。其基本思想是：根据文本串T[1，n]和模式串P[1，m]计算矩阵$D_{(m+1)x(n+1)}$，其中D[i，j](0≤i≤m，0≤j≤n)是模式串P的前缀子串$p_1\cdots p_i$与文本串T的任意以t_j结尾的子串之间的最小误差个数（即最小编辑距离）。显然，如果D[m，j] ≤k，则表明在文本串T的t_j处存在最多带k个误差的匹配。D[i，j]由以下公式计算：

D[0，j]=0，0≤j≤n

D[i，0]=i，1≤i≤m

D[i，j]=min(D[i-1，j]+1，D[i，j-1]+1，D[i-1，j-1]+δ(p_i，t_j))

其中，δ(p_i，t_j)=0，如果$p_i=t_j$；否则，δ(p_i，t_j)=1。

可以看出，D[i，j]的计算是通过递推方式进行的。D[0，j]对应于模式串为空串，而文本串长度为j的情况，显然，空串不需要任何编辑操作就能在文本串的任何位置处匹配，所以D[0，j]=0。D[i，0]对应于模式串长度为i，而文本串为空串的情况，此时，至少需要对模式串进行i次删除操作才能与文本串（空串）匹配，所以D[i，0]=i。在计算D[i，j]时，D[i-1，j]，D[i，j-1]，D[i-1，j-1]都已计算出，D[i-1，j]+1，D[i，j-1]+1，D[i-1，j-1]+δ(p_i，t_j)分别对应于对模式串进行插入、删除、替换操作的情况，由于D[i，j]是最小编辑距离，所以取三者中的最小值。

上述动态规划算法只考虑了插入、删除、替换三种编辑操作，但很容易将其推广到允许有换位操作的情况。考虑D[i，j]的计算，若$p_{i-1}p_i=t_it_{i-1}$，则D[i，j]的一个可能取值是D[i-2，j-2]+1，即，将$p_{i-1}p_i$变成t_it_{i-1}只需要进行一次换位操作，从而最小编辑距离只增加 1。由此可对上述算法进行简单的修改，使之适用于允许有插入、删除、替换和换位四种编辑操作的情况。

算法 14.10 允许有换位操作的 k-differences 问题的动态规划算法。

输入：文本串 T[1，n]，模式串 P[1，m]，近似匹配允许的最大误差数 k。

输出：所有匹配位置。

```
K-Diff_DynamicProgramming(T，n，P，m，k)
Begin
(1)  for j=0 to n do D[0,j]=0 end for
(2)  for i=1 to m do D[i,0]=i end for
(3)  for i=1 to m do
     (3.1)for j=1 to n do
          (ⅰ) if (P[i]=T[j]) then
                   D[i,j]=D[i-1,j-1]
               else if (P[i]=T[j-1] and P[i-1]=T[j]) then
                        D[i,j]=min(D[i-2,j-2]+1,D[i-1,j]+1,D[i,j-1]+1)
                   else
                        D[i,j]=min(D[i-1,j]+1,D[i,j-1]+1,D[i-1,j-1]+1)
                   end if
               end if
          (ⅱ) if (i=m and D[i,j]<=k) then
                   找到 P 在 T 中的一个最多带 k 个误差的近似匹配，
                   输出匹配结束位置 j;
```

```
            end if
        end for
    end for
End
```

显然，该算法的时间复杂度为 $O(mn)$。

3. 基于过滤思想的扩展的近似串匹配算法:

经典的动态规划算法在实际应用中速度很慢，往往不能满足实际问题的需要。为此，S.Wu 和 U.Manber 以及 Gonzalo Navarro 和 Ricardo Baeza-Yates 等人先后提出了多个基于**过滤**(Filtration)思想的更加快速的近似串匹配算法。这些算法一般分为两步：①按照一定的过滤原则搜索文本串，过滤掉那些不可能发生匹配的文本段；②再进一步验证剩下的匹配候选位置处是否真的存在成功匹配。由于在第一步中已经滤去了大部分不可能发生匹配的文本段，因此大大加速了匹配查找过程。在实际应用中，这些过滤算法一般速度都很快。下面我们针对前面定义的扩展的近似串匹配问题，讨论了加入换位操作后的 k-differences 问题的过滤算法。

过滤算法基于这样一种直观的认识：若模式串P在文本串T的t_j处有一个最多带k个误差的近似匹配，则P中必然有一些子串是不带误差地出现在T中的，也就是说，P中必然有一些子串与T中的某些子串是精确匹配的。

引理 14.1：在允许有换位操作的最多带 k 个误差的串匹配问题中，如果将模式串 P 划分成 2k+1 段，那么对于 P 在 T 中的每一次近似出现（最多带 k 个误差），这 2k+1 段中至少有一段是不带误差地出现在 T 中的。

证明：这个引理很容易证明。试想，将 k 个误差，也就是对模式串 P 所做的 k 个编辑操作，分布于 2k+1 个子模式串中。由于一个插入，删除或替换操作只能改变一个子模式串，而一个换位操作有可能改变两个子模式串（例如，在子模式串 i 的最后一个字符与子模式串 i+1 的第一个字符之间进行一次换位操作），所以 k 个编辑操作最多只能改变 2k 个子模式串。这就是说，在这 2k+1 个子模式串中，至少有一个是未被改变的，它不带误差地出现在文本串 T 中，与 T 的某个（些）子串精确匹配。■

根据上面的引理，我们可设计过滤算法，其原理描述如下：

第 1 步:

(1) 将模式串P尽可能均匀地划分成 2k+1 个子模式串$P_1, P_2, \cdots, P_{2k+1}$，每个子模式串的长度为$\left\lfloor \frac{m}{2k+1} \right\rfloor$或$\left\lceil \frac{m}{2k+1} \right\rceil$。 具体地说，如果m=q(2k+1)+r，0≤r＜2k+1，那么可以将模式串P划分成r个长度为$\left\lceil \frac{m}{2k+1} \right\rceil$的子模式串和(2k+1)-r个长度为$\left\lfloor \frac{m}{2k+1} \right\rfloor$的子模式串。

(2) 采用一种快速的精确串匹配算法，在文本串T中查找$P_1, P_2, \cdots, P_{2k+1}$这 2k+1 个子模式串，找到的与某个子模式串$P_\ell$(1≤ℓ≤2k+1)精确匹配的位置也是可能与整个模式串P发生最多带k个误差的近似匹配的候选位置。

第 2 步:

采用动态规划算法（算法14.10）验证在各候选位置附近是否真的存在最多带k个误差

的近似匹配。假定在第一步中找到了以p_j(1≤j≤m)开头的子模式串P_ℓ(1≤ℓ≤2k+1)在文本串中的一个精确匹配，且该匹配的起始位置在文本串的t_i处，则需要验证从$t_{i-j+1-k}$到$t_{i-j+m+k}$之间的文本段T[i-j+1-k, i-j+m+k]。因为在本文所讨论的问题中，允许的编辑操作包括插入，删除，替换和换位，近似匹配允许的最大误差数为k，所以如果在候选位置t_i附近存在最多带k个误差的近似匹配，只可能发生在这个长度为m+2k的文本段范围内，超出这个范围，误差数就大于k了。因此在上述情况下，只需要验证T[i-j+1-k,i-j+m+k]就足够了。

算法 14.11 允许有换位操作的 k-differences 问题的过滤算法。

输入：文本串 T（长度 n），模式串 P（长度 m），近似匹配允许最大误差数 k。

输出：所有匹配位置。

K-Diff_Filtration(T，n，P，m，k)

Begin

(1) r=m mod (2k+1)

(2) j=1

(3) **for** ℓ=1 **to** 2k+1 **do**

(3.1) **if** (ℓ＜=r＝ **then** len＝$\left\lceil \frac{m}{2k+1} \right\rceil$

else len＝$\left\lfloor \frac{m}{2k+1} \right\rfloor$

end if

(3.2) **for each** exact matching position i **in** T reported by search(T,n,P[j,j+len-1],len) **do**

check the area T[i-j+1-k, i-j+m+k]

end for

(3.3) j＝j+len

end for

End

过滤算法的时间性能与模式串的长度 m，近似匹配允许的最大误差数 k，以及字母表中各字符出现的概率等因素都密切相关。误差率 α 定义为近似匹配允许的最大误差数 k 与模式串长度 m 的比值，即 α =k/m。在误差率 α 很小的情况下，算法 14.11 的平均时间复杂度为 O（kn）。

1.3.2 近似串匹配的并行算法

这一节首先介绍扩展近似串匹配过滤算法在 PRAM 模型上的并行化方法，然后给出了分布式存储模型的并行化过滤算法。

1. 扩展近似串匹配问题的过滤算法在 PRAM 模型上的并行化

首先假设有p个处理器。由于，最多带k个误差的串匹配问题要求在文本串中找出所有成功匹配的结束位置t_j，因此可以将整个问题划分成p个子问题，每个子问题的文本串长度为$\ell=\lceil n/p \rceil$（最后一段长度不够可以用特殊字符补齐），用p个处理器并行求解，每个处理器求解一个子问题。子问题i(1≤i≤p)对应于：求结束于$t_{(i-1)\ell+1}, t_{(i-1)\ell+2}, \cdots, t_{i\ell}$的最多带k个误差的近似匹配。

考虑第i个子问题。由于在定义的扩展近似串匹配问题中，允许的编辑操作包括插入、删除、替换和换位，而允许的最大误差数为k，因此结束于$t_{(i-1)\ell+1}$的最多带k个误差的近似匹配的最左起始位置应该是$t_{(i-1)\ell+1-(k+m-1)}$。这说明，在求解子问题i(2≤i≤p)时，必须结合前一文本段的最后k+m-1 个字符，才能找出所有的匹配位置。

算法 14.12 扩展近似串匹配问题的基于 PRAM 模型的并行化过滤算法

输入：文本串 T[1，n]，模式串 P[1，m]，近似匹配允许的最大误差数 k

输出：所有匹配位置

```
Begin
(1) ℓ=⌈n/p⌉
(2) K-Diff_Filtration(T[1,ℓ],ℓ,P,m,k)
(3) for i=2 to p par-do
        K-Diff_Filtration(T[(i-1)ℓ+1-(k+m-1),iℓ],ℓ+k+m-1,P,m,k)
    end for
End
```

算法 14.12 的平均时间复杂度的分析与上一节中串行过滤算法的分析完全类似。此时，用于查找 2k+1 个子模式串的时间开销为O((2k+1)ℓ+m)+O((2k+1)(ℓ+k+m-1)+m)=O(*kn/p+km*)，用于验证所有候选位置的时间开销约为$2m^3\alpha/\sigma^{1/2\alpha}$。通过类似的分析讨论可以得出如下结论：在误差率α很小的情况下，算法 14.12 的平均时间复杂度为O(*kn/p+km*)，其中，n、m、k和p分别是文本串长度、模式串长度、近似匹配允许的最大误差数和算法所使用的处理器个数。

2. 扩展近似串匹配问题的基于分布式存储模型的并行化过滤算法

扩展近似串匹配问题的基于分布式存储模型的并行化过滤算法与前述的基于 PRAM 模型的并行化过滤算法在设计原理和设计思路上是完全一样的。只不过由于是在分布式存储环境下，文本串和模式串分布存储于不同的处理器上，因此算法中涉及到处理器之间的通信。算法的设计思路是这样的：将长度为 n 的文本串 T 均匀地划分成长度相等且互不重叠的 p 段（最后一段长度不够可以用特殊字符补齐）。将这 p 个局部文本串按照先后顺序分布于处理器 0 到(p-1)上，也就是说，第 1 个局部文本串放在处理器 0 上，第 2 个局部文本串放在处理器 1 上，……，第 p 个局部文本串放在处理器 p-1 上。这样一来，相邻的局部文本串就被分布在相邻的处理器上，而且每个处理器上局部文本串的长度均为$\lceil n/p \rceil$。算法中假定长度为 m 的模式串 P 初始存储于处理器 0 上，所以必须将它播送到各个处理器上，以便所有处理器并行求解近似串匹配问题。但是根据上小节中的分析，结束位置在各局部文本串（第 1 个局部文本串除外)的前 m+k-1 个字符中的那些匹配位置必须跨越该局部文本串才能找到，具体地说，就是必须结合前一局部文本串的最后 m+k-1 个字符，才能找到结束于这些位置的近似匹配。因此，每个处理器（处理器 p-1 除外）应将它所存储的局部文本串的最后 m+k-1 个字符发送给下一处理器，下一处理器接收到上一处理器发送来的 m+k-1 个字符，再结合自身所存储的长度为$\lceil n/p \rceil$的局部文本串进行近似匹配查找，就可以找出所有的匹配位置。

在播送模式串 P 到各处理器上，以及发送局部文本串最后 m+k-1 个字符到下一处理器的通信操作结束之后，各处理器调用 K-Diff_Filtration 算法并行地进行匹配查找，处理器 0 求解文本串长度为$\lceil n/p \rceil$，模式串长度为 m 的子问题，其它各处理器求解文本串长度为

$\lceil n/p \rceil$+m+k-1，模式串长度为 m 的子问题。

算法 14.13 扩展近似串匹配问题的基于分布式存储模型的并行化过滤算法

输入：分布存储于处理器P_i上的文本串$T_i[1, \lceil n/p \rceil]$，

存储于处理器P_0上的模式串$P[1, m]$，
近似匹配允许的最大误差数 k

输出：分布于各个处理器上的匹配结果

```
Begin
(1) P0 broadcast m
(2) P0 broadcast P[1, m];
(3) for i=0 to p-2 par-do
        Pi send Ti[⌈n/p⌉-m-k+2,⌈n/p⌉] to Pi+1;
    end for
(4) for i=1 to p-1 par-do
        Pi receive Ti-1[⌈n/p⌉-m-k+2,⌈n/p⌉] from Pi-1;
    end for
(5) P0 call K-Diff_Filtration(T0[1,⌈n/p⌉],⌈n/p⌉,P,m,k);
(6) for i=1 to p-1 par-do
        Pi call K-Diff_Filtration
        (Ti-1[⌈n/p⌉-m-k+2,⌈n/p⌉]+Ti[1,⌈n/p⌉],⌈n/p⌉+m+k-1,P,m,k);
    end for
End
```

算法 14.13 的时间复杂度包括两部分，一部分是计算时间复杂度，另一部分是通信时间复杂度。算法中假定文本串初始已经分布于各个处理器上，最终的匹配结果也分布于各个处理器上。算法的计算时间由算法第(5)步中各处理器同时调用算法 14.12（K-Diff_Filtration 算法）的时间复杂度构成。根据对算法 14.11 的平均时间复杂度的分析，在误差率 α 很小的情况下，算法 14.13 的平均计算时间复杂度为 $O(kn/p+km)$，当模式串长度 m 远远小于局部文本串长度 n/p 时，平均计算时间复杂度为 $O(kn/p)$。算法的通信时间由算法第(1)步播送模式串长度 m 的时间，第(3)步播送模式串 P 的时间，以及第(4)步发送各局部文本串末尾 m+k-1 个字符到下一处理器的时间构成。通信时间复杂度与具体采用的播送算法有关。若以每次发送一个字符的时间为单位时间，则通信时间复杂度为 $O(m+k)$。

MPI 源程序请参见所附光盘。

1.4 小结

本章主要讨论了几个典型的并行串匹配算法，关于 KMP 算法的具体讨论可参考［1］，［2］，［3］，Karp 和 Rabin 在［4］中首先提出了随机串匹配的算法，关于该算法的正确性证明可参阅［5］；文献［6］和［7］分别讨论了近似串匹配，允许由换位操作的近似串匹配

算法见文献［8］。

参考文献

[1]. Knuth D E, Morris J H, Pratt V R. Fast Pattern Matching in Strings. SIAM Journal of Computing, 1977, 6(2):322-350

[2]. 朱洪等. 算法设计与分析. 上海科学技术出版社, 1989

[3]. 陈国良, 林洁, 顾乃杰. 分布式存储的并行串匹配算法的设计与分析. 软件学报, 2002.6, 11(6):771-778

[4]. Karp R M, Rabin M O. Efficient randomized pattern-matching algorithms. IBM J. Res. Develop., 1987,31(2):249-260

[5]. 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

[6]. Wu S, Manber U. Fast test searching allowing errors. Communications of the ACM, 1993, 35(10):83-91

[7]. Ricardo B Y, Gonzalo N. Faster Approximate String Matching. In Algorithmica, 1999. 23(2)

[8]. 姚珍. 带有换位操作的近似串匹配算法及其并行实现. 中国科学技术大学硕士论文,2000.5

附录 KMP 串匹配并行算法的 MPI 源程序

1. 源程序 gen_ped.c

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

/*根据输入参数生成模式串*/
int main(int argc,char *argv[])
{
    int strlen,pedlen,suffixlen,num,i,j;
    char *string;
    FILE *fp;

    strlen=atoi(argv[1]);
    pedlen=atoi(argv[2]);
    srand(atoi(argv[3]));

    string=(char*)malloc(strlen*sizeof(char));
    if(string==NULL)
    {
        printf("malloc error\n");
        exit(1);
    }

    for(i=0;i<pedlen;i++)
    {
        num=rand()%26;
        string[i]='a'+num;
    }

    for(j=1;j<(int)(strlen/pedlen);j++)
        strncpy(string+j*pedlen,string,pedlen);

    if((suffixlen=strlen%pedlen)!=0)
        strncpy(string+j*pedlen,string,suffixlen);

    if((fp=fopen(argv[4],"w"))!=NULL)
    {
        fprintf(fp,"%s",string);
        fclose(fp);
    }
    else
    {
```

```
            printf("file open error\n");
            exit(1);
        }
```

2. 源程序 kmp.c

```
#include <malloc.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#define    MAX(m,n)       (m>n?m:n)

typedef struct{
        int pedlen;
        int psuffixlen;
        int pednum;
}pntype;

/*对模式串进行周期分析，并计算相应的 new 和
        newval 值*/
void Next(char *W,int patlen,int *nextval,
        pntype *pped)
{
        int i,j,plen;
        int *next;

        if((next=(int *)malloc((patlen+1)*sizeof(int)))
                ==NULL)
        {
                printf("no enough memory\n");
                exit(1);
        }

        /*计算 next 和 nextval*/
       next[0]=nextval[0]=-1;
        j=1;
        while(j<=patlen)
        {
                i=next[j-1];
                while(i!=(-1) && W[i]!=W[j-1])
                        i=next[i];
                next[j]=i+1;

                if(j!=patlen)
                {
                        if( W[j]!=W[i+1])
                                nextval[j]=i+1;
                        else
                                nextval[j]=nextval[i+1];
                }
                j++;
        }

        pped->pedlen=patlen-next[patlen];
        pped->pednum=(int)(patlen/pped->pedlen);
        pped->psuffixlen=patlen%pped->pedlen;

        free(next);
}

/*改进的 KMP 算法*/
void kmp(char *T,char*W,int textlen,int patlen,
        int *nextval,pntype *pped,int prefix_flag,
        int matched_num,int *match,int *prefixlen)
{
        int i,j;

        i=matched_num;
        j=matched_num;

       while(i<textlen)
       {
                if((prefix_flag==1)&&((patlen-j)>
                        (textlen-i)))
                        break;
                while(j!=(-1) && W[j]!=T[i])
                        j=nextval[j];
                if(j==(patlen-1))
                {
                        match[i-(patlen-1)]=1;
                        if(pped->pednum+pped->psuffixlen
                                ==1)
        return;
}
```

```
                j = -1 ;
            else
                j=patlen-1-pped->pedlen;
        }
        j++;
        i++;
    }
    (*prefixlen)=j;
}

/*重构模式串以及 next 函数*/
void Rebuild_info(int patlen,pntype *pped,
    int *nextval,char *W)
{
    int i;
    if (pped->pednum == 1)
        memcpy(W+pped->pedlen,W,
            pped->psuffixlen);
    else
    {
        memcpy(W+pped->pedlen,W,
            pped->pedlen);
        for (i=3; i<=pped->pednum; i++)
        {
            memcpy(W+(i-1)*pped->pedlen,W,
                pped->pedlen);
            memcpy(nextval+(i-1)*
                pped->pedlen,
                nextval+pped->pedlen,
                pped->pedlen*sizeof(int));
        }

        if(pped->psuffixlen!=0)
        {
            memcpy(W+(i-1)*pped->pedlen,W,
                pped->psuffixlen);
            memcpy(nextval+(i-1)*
                pped->pedlen,nextval+
                pped->pedlen,
                pped->psuffixlen*sizeof(int));
        }
    }
}
```

```
/*生成文本串*/
void gen_string(int strlen,int pedlen,char *string,
    int seed)
{
    int suffixlen,num,i,j;

    srand(seed);
    for(i=0;i<pedlen;i++)
    {
        num=rand()%26;
        string[i]='a'+num;
    }
    for(j=1;j<(int)(strlen/pedlen);j++)
        strncpy(string+j*pedlen,string,pedlen);
    if((suffixlen=strlen%pedlen)!=0)
        strncpy(string+j*pedlen,string,suffixlen);
}

/*从文件读入模式串信息*/
void GetFile(char *filename,char *place, int *number)
{
    FILE *fp;
    struct stat statbuf;

    if ((fp=fopen(filename,"rb"))==NULL)
    {
        printf ("Error open file %s\n",filename);
        exit(0);
    }

    fstat(fileno(fp),&statbuf);
    if(((*place)=(char *)malloc(sizeof(char)*
        statbuf.st_size)) == NULL){
        printf ("Error alloc memory\n");
        exit(1);
    }

    if (fread((*place),1,statbuf.st_size,fp)
        !=statbuf.st_size){
        printf ("Error in reading num\n");
        exit(0);
    }
    fclose (fp);
    (*number)=statbuf.st_size;
```

```
}

/*打印运行参数信息*/
void PrintFile_info(char *filename,char *T,int id)
{
        FILE *fp;
        int i;

        if ((fp=fopen(filename,"a"))==NULL)
        {
                printf ("Error open file %s\n",filename);
                exit(0);
        }

        fprintf (fp,"The Text on node %d is %s .\n",
                id,T);
        fclose (fp);
}


/*打印匹配结果*/
void PrintFile_res(char *filename,int *t,int len,
        int init,int id)
{
        FILE *fp;
        int i;

        if ((fp=fopen(filename,"a"))==NULL)
        {
                printf ("Error open file %s\n",filename);
                exit(0);
        }

        fprintf (fp,"This is the match result on node %d
                \n",id);
        for (i=0; i<=len-1; i++)
                if(t[i]==1)
                        fprintf (fp,"(%d)    +\n",i+init);
                else
                        fprintf (fp,"(%d)    -\n",i+init);
        fclose (fp);
}

void main(int argc,char *argv[])
{
        char *T,*W;
        int     *nextval,*match;
        int     textlen,patlen,pedlen,nextlen_send;
        pntype pped;
        int     i,myid,numprocs,prefixlen,ready;
        MPI_Status    status;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,
                &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,
                &myid);

        nextlen_send=0;
        ready=1;
        /*读如文本串长度*/
        textlen=atoi(argv[1]);
        textlen=textlen/numprocs;
        pedlen=atoi(argv[2]);
        if((T=(char *)malloc(textlen*sizeof(char)))
                ==NULL)
        {
                printf("no enough memory\n");
                exit(1);
        }
        gen_string(textlen,pedlen,T,myid);

        if(myid==0)
        {
                PrintFile_info("match_result",T,myid);
                if(numprocs>1)
                        MPI_Send(&ready,1,MPI_INT,1,0,
                                MPI_COMM_WORLD);
        }
        else
        {
                MPI_Recv(&ready,1,MPI_INT,myid-1,
                        myid-1,MPI_COMM_WORLD,
                        &status);
                PrintFile_info("match_result",T,myid);
                if(myid!=numprocs-1)
                        MPI_Send(&ready,1,MPI_INT,
                                myid+1,myid,
```

```
                MPI_COMM_WORLD);
}

printf("\n");

if((match=(int *)malloc(textlen*sizeof(int)))
    ==NULL)
{
    printf("no enough memory\n");
    exit(1);
}

/*处理器 0 读入模式串，并记录运行参数*/
if(myid==0)
{
    printf("processor num = %d \n",
        numprocs);
    printf("textlen = %d\n",
        textlen*numprocs);
    GetFile("pattern.dat",&W,&patlen);
    printf("patlen= %d\n",patlen);

    if((nextval=(int *)malloc(patlen*
        sizeof(int)))==NULL)
    {
        printf("no enough memory\n");
        exit(1);
    }
    /*对模式串进行分析，对应于算法 14.6
        步骤（1）*/
    Next(W,patlen,nextval,&pped);
    if(numprocs>1)
    {
        if (pped.pednum==1)
            nextlen_send = patlen;
        else
            nextlen_send = pped.pedlen*2;
    }
}

/*向各个处理器播送模式串的信息，对应于算
    法 14.6 步骤（2）*/
if(numprocs>1)
{
    MPI_Bcast(&patlen, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
    if(myid!=0)
        if(((nextval=(int *)malloc(patlen*
            sizeof(int)))==NULL)
            ||((W=(char *)malloc(patlen*
            sizeof(char)))==NULL))
    {
        printf("no enough memory\n");
        exit(1);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(&pped,3,MPI_INT,0,
        MPI_COMM_WORLD);
    MPI_Bcast(&nextlen_send,1,MPI_INT,0,
        MPI_COMM_WORLD);
    MPI_Bcast(nextval,nextlen_send,
        MPI_INT,0,
        MPI_COMM_WORLD);
    MPI_Bcast(W,pped.pedlen,
        MPI_CHAR,0,
        MPI_COMM_WORLD);
}

MPI_Barrier(MPI_COMM_WORLD);

/*调用修改过的 KMP 算法进行局部串匹配，
    对应于算法 14.6 步骤（3）*/
if(numprocs==1)
{
    kmp(T,W,textlen,patlen,nextval,
        &pped,1,0,
        match+patlen-1,&prefixlen);
}
else
{
    if(myid!=0)
    /*各个处理器分别根据部分串数据以及
        周期信息重构模式串*/
    Rebuild_info(patlen,&pped,nextval,W);
    if(myid!=numprocs-1)
        kmp(T,W,textlen,patlen,nextval,
            &pped,0,0,match+patlen-1,
            &prefixlen);
```

```
        else
            kmp(T,W,textlen,patlen,nextval,
                &pped,1,0,match+patlen-1,
                &prefixlen);

        MPI_Barrier(MPI_COMM_WORLD);

        /*各个处理器进行段间匹配，对应于算
            法 14.6 步骤（4）*/
        if(myid<numprocs-1)
            MPI_Send(&prefixlen,1,MPI_INT,
                myid+1,99,
                MPI_COMM_WORLD);
        if(myid>0)
            MPI_Recv(&prefixlen,1,MPI_INT,
                myid-1,99,
                MPI_COMM_WORLD,
                &status);
        MPI_Barrier(MPI_COMM_WORLD);
        if((myid>0)&&(prefixlen!=0))
            kmp(T-prefixlen,W,
                prefixlen+patlen-1,
                patlen,nextval,&pped,1,
                prefixlen,
                match+patlen-1-prefixlen,
                &prefixlen);

        MPI_Barrier(MPI_COMM_WORLD);
    }

    /*输出匹配结果*/
    if(myid==0)
    {
        PrintFile_res("match_result",
            match+patlen-1,textlen-patlen+1,0,
            myid);
        if(numprocs>1)
            MPI_Send(&ready,1,MPI_INT,1,0,
                MPI_COMM_WORLD);
    }
    else
    {
        MPI_Recv(&ready,1,MPI_INT,
            myid-1,myid-1,
            MPI_COMM_WORLD,&status);
        PrintFile_res("match_result",match,
            textlen,myid*textlen-patlen+1,
            myid);
        if(myid!=numprocs-1)
            MPI_Send(&ready,1,MPI_INT,
                myid+1,myid,
                MPI_COMM_WORLD);
    }
    free(T);
    free(W);
    free(nextval);
    MPI_Finalize();
}
```

3. 运行实例

编译：gcc gen_ped.c –o gen_ped

mpicc kmp.c –o kmp

运行：首先运行 gen_ped 生成模式串，gen_ped Strlen Pedlen Seed Pattern_File。其中 Strlen 代表模式串的长度，Pedlen 代表模式串的最小周期长度，Seed 是随机函数使用的种子数，Pattern_File 是生成数据存储的文件，这里在 kmp.c 中固定指定的文件名为 pattern.dat。本例中使用了如下的参数。

gen_ped 3 2 1 pattern.dat

之后可以使用命令 mpirun –np SIZE kmp m n 来运行该串匹配程序，其中 SIZE 是所使用的处理器个数，m 表示文本串长度，n 为文本串的周期长度。本实例中使用了 SIZE=3 个处理器，m=18，n=3。

mpirun –np 3 kmp 18 2

运行结果：

存储于 pattern.dat 中的模式串为：**qmq**

存储于 match_result 中的匹配结果为：

The Text on node 0 is **asasas** .

The Text on node 1 is **qmqmqm** .

The Text on node 2 is **ypypyp .**

This is the match result on node 0

(0) -

(1) -

(2) -

(3) -

This is the match result on node 1

(4) -

(5) -

(6) +

(7) -

(8) +

(9) -

This is the match result on node 2

(10) -

(11) -

(12) -

(13) -

(14) -

(15) -

说明：该运行实例中，令文本串长度为 18，随机产生的文本串为 asasasqmqmqmypypyp，分布在 3 个节点上；模式串长度为 3，随机产生的模式串为 qmq。最后，节点 1 上得到两个匹配位置，由+表示出来。

3 图论

图论在计算机科学、信息科学、人工智能、网络理论、系统工程、控制论、运筹学和经济管理等领域有着广泛的应用。但很多图论问题虽易表达，却难以求解，其中有相当多的图论问题均属 NP 完全问题。本章主要介绍工程实用简单图论问题的并行算法及其 MPI 编程实现，包括传递闭包、连通分量、最短路径和最小生成树等。

1.1 传递闭包

设A是一个含N个顶点的有向图G的**布尔邻接矩阵**(Boolean Adjacent Matrix)，即元素 a_{ij}=1 当且仅当从顶点i到j有一条有向边。所谓A的**传递闭包**（Transitive Closure），记为 A^+，是一个N×N的布尔矩阵，其元素b_{ij}=1 当且仅当：①i=j；或②从i出发存在有向路径到j，又称为顶点i到j可达。从G的布尔邻接矩阵A求出G的传递闭包，就称为**传递闭包问题**。

传递闭包问题有很强的应用背景，在科学计算中广泛存在。传递闭包问题的经典解法之一就是利用布尔矩阵的乘法来求解。本节将把这一算法分别在串行和并行机上实现。

1.1.1 传递闭包串行算法

利用布尔矩阵相乘求解传递闭包问题的原理为：对于布尔矩阵$(A+I)^k$中的任一元素b_{ij}，b_{ij}=1 表示从i到j存在长度小于等于k的可达路径，否则，b_{ij}=0。显然对于k=1，$(A+I)^1$中b_{ij}=1 当且仅当从i到j路径长度为 0（i=j）或为 1（从i到j存在有向边）；$(A+I)^2$中，b_{ij}=1 当且仅当从i到j路径长度小于等于 2；$((A+I)^2)^2$中，b_{ij}=1 当且仅当从i到j路径长度小于等于 4，等等。因为任意两点间如果存在可达路径，长度最多为N-1，所以k≥N-1 时，$(A+I)^k$就是所求的传递闭包A^+。于是(A+I)矩阵的$\log$N次自乘之后所得的矩阵就是所求的传递闭包。

根据前面的叙述，很自然的有下面的传递闭包串行算法 15.1，其时间复杂度为$O(N^3\log N)$。

算法 15.1 传递闭包串行算法

输入：图 G 的布尔邻接矩阵 A

输出：A 的传递闭包 M

```
procedure closure
Begin
(1) 读入矩阵 A
/* 以下作 A = A+I 的运算 */
(2) for i=0 to N-1 do
        a(i, i) = 1
    endfor
/* 以下是 A 矩阵的log N 次自乘，结果放入 M 矩阵；每次乘后，结果写回 A 矩阵 */
(3) for k=1 to log N do
    (3.1)for i=0 to N-1 do
            for j=0 to N-1 do
                s=0
                while (s<N) and (a(i,s)=0 or a(s,j)=0) do
```

```
                    s = s+1
                endwhile
                if s<N then m(i,j)=1 else m(i,j)=0
            endfor
        endfor
        /* 计算结果从 M 写回 A */
    (3.2)for i=0 to N-1 do
            for j=0 to N-1 do
                a(i, j) = m(i, j)
            endfor
        endfor
    endfor
End
```

1.1.2 传递闭包并行算法

本小节将把上一小节里的算法并行化。在图论问题的并行化求解中，经常使用将 N 个顶点（或连通分量）平均分配给 p 个处理器并行处理的基本思想。其中每个处理器分配到 n 个顶点，即 n=N/p。无法整除时，一般的策略是在尽量均分的前提下，给最后一个处理器分配较少的顶点。为了使算法简明，在本章中，仅在算法的 MPI 实现中才考虑不能整除的情况。在以后的几节中，N、p、n 都具有上面约定的意义，不再多说。

为了并行处理，这里将矩阵(A+I)进行划分，分别交给p个处理器。在每次矩阵乘法的计算中，将N×N的结果矩阵$(A+I)^2$均匀划分成p×p个子块，每块大小为n×n。处理器i负责计算位于第i子块行上的p个子块。对整个子块行的计算由p次循环完成，每次计算一个子块。每个处理器为了计算一个n×n大小的子块，需要用到源矩阵(A+I)中对应的连续n行（局部数据a）和连续n列的数据（局部数据b）。计算完成后，各处理器循环交换局部数据b，就可以进行下一个子块的计算了。

于是，总体算法由 2 层循环构成，外层是矩阵 M=A+I 的$\log N$ 次自乘，每次首先完成矩阵 M 的一次自乘，然后将结果写回 M；内层是 p 个子块的计算，每次首先完成各自独立的计算，然后处理器间循环交换局部数据 b。算法运行期间，矩阵 M 的数据作为全局数据由处理器 0 维护。

根据以上思想，并行算法描述见下面的算法 15.2，使用了p个处理器，其时间复杂度为$O(N^3/p^{\log}N)$。其中myid是处理器编号，下同。

算法 15.2 传递闭包并行算法

输入：图 G 的布尔邻接矩阵 A

输出：A 的传递闭包 M

```
procedure closure-parallel
Begin
    /* 由处理器 0 读入矩阵 A 到 M 中，并进行 M=M+I 运算
       对应源程序中 readmatrix()函数 */
(1) if myid=0 then
    (1.1) 读入矩阵 A，放入 M 中
    (1.2) for i=0 to N-1 do
            m(i,i)=1
        endfor
```

endif

(2) 各处理器变量初始化

/* 以下是主循环，矩阵 M 的log N 次自乘 */

(3) **for** i=1 *to* log N **par_do**

/* 以下向各处理器发送对应子块行和列数据
对应源程序中 sendmatrix()函数 */

(3.1)**for** j=1 **to** p-1 **do**

(ⅰ) 处理器 0 发送第 j 个子块行的数据给处理器 j，成为 j 的局部数据 a

(ⅱ) 处理器 0 发送第 j 个子块列的数据给处理器 j，成为 j 的局部数据 b

endfor

/* 以下是各处理器接收接收和初始化局部数据 a 和 b
对应源程序中 getmatrix()函数 */

(3.2)处理器 0 更新自己的局部数据 a 和 b

(3.3)处理器 1 到 p-1 从处理器 0 接受数据，作为局部数据 a 和 b

/* 以下是乘法运算过程，对应源程序中 paramul()函数 */

(3.4)**for** j=0 **to** p-1 **do**

(ⅰ) 处理器 k 计算结果矩阵的子块(k, ((k+j) mod p))

(ⅱ) 各处理器将数据 b 发送给前一个处理器

(ⅲ) 各处理器从后一个处理器接收数据 b

endfor

/* 以下是各处理器将局部计算结果写回 M 数组
对应源程序中 writeback()函数 */

(3.5)**if** myid=0 **then**

(ⅰ) 计算结果直接写入矩阵 M

(ⅱ) 接受其它处理器发送来的计算结果并写入 M

else

发送计算结果的 myid 子块行数据给处理器 0

endif

endfor

End

MPI 源程序请参见所附光盘。

1.2 连通分量

图 G 的一个**连通分量**(Connected Component)是 G 的一个最大连通子图，该子图中每对顶点间均有一条路径。根据图 G，如何找出其所有连通分量的问题称为**连通分量问题**。解决该问题常用的方法有 3 种：①使用某种形式的图的搜索技术；②通过图的布尔邻接矩阵计算**传递闭包**；③顶点倒塌算法。本节将介绍如何在并行环境下实现顶点倒塌算法。

1.2.1 顶点倒塌法算法原理描述

顶点倒塌（Vertex Collapse）**算法**中，一开始图中的 N 个顶点看作 N 个孤立的**超顶点**(Super Vertex)，算法运行中，有边连通的超顶点相继合并，直到形成最后的整个连通分量。每个顶点属于且仅属于一个超顶点，超顶点中标号最小者称为该超顶点的**根**。

该算法的流程由一系列循环组成。每次循环分为三步：①发现每个顶点的最小标号邻接超顶点；②把每个超顶点的根连到最小标号邻接超顶点的根上；③所有在第 2 步连接在一起的超顶点倒塌合并成为一个较大的超顶点。

图 G 的顶点总数为 N，因为超顶点的个数每次循环后至少减少一半，所以把每个连通分量倒塌成单个超顶点至多$\log N$ 次循环即可。顶点 i 所属的超顶点的根记为 D(i)，则一开始时有 D(i)=i，算法结束后，所有处于同一连通分量中的顶点具有相同的 D(i)。

1.2.2 连通分量并行算法

算法中为顶点设置数组变量 D 和 C，其中 D(i)为顶点 i 所在的超顶点号，C(i)为和顶点 i 或超顶点 i 相连的最小超顶点号等，根据程序运行的阶段不同，意义也有变化。算法的主循环由 5 个步骤组成：①各处理器并行为每个顶点找出对应的 C(i)；②各处理器并行为每个超顶点找出最小邻接超顶点，编号放入 C(i)中；③修改所有 D(i)=C(i)；④修改所有 C(i)=C(C(i))，运行$\log N$ 次；⑤修改所有 D(i)为 C(i)和 D(C(i))中较小者。其中最后 3 步对应意义为超顶点的合并。

顶点倒塌算法是专为并行程序设计的，多个顶点的处理具有很强的独立性，很适合分配给多个处理器并行处理。这里让p个处理器分管N个顶点。则顶点倒塌算法的具体描述见算法 15.3，使用了p个处理器，其时间复杂度为$O(N^2/p\log N)$，其中步骤(2)为主循环，内含 5 个子步骤对应上面的描述。

算法 15.3 顶点倒塌算法

输入：图 G 的邻接矩阵 A

输出：向量 D(0 ： N-1)，其中 D(i)表示顶点 i 的标识

```
procedure collapse-vertices
Begin
    /* 初始化 */
(1) for 每个顶点 i par_do
        D(i) = i
    endfor
(2) for k=1 to log N do
        /* 以下并行为每个顶点找邻接超顶点中最小的
           对应源程序中 D_to_C()函数 */
  (2.1) for 每个 i, j : 0 ≤ i, j ≤ N-1 par_do
            C(i) = min{ D(j) | a(i,j)=1 and D(i)≠D(j) }
            if 没有满足要求的 D(j) then
                C(i) = D(i)
            endif
        endfor
        /* 以下并行求每个超顶点的最小邻接超顶点
           对应源程序中 C_to_C()函数 */
  (2.2) for 每个 i, j : 0 ≤ i, j ≤ N-1 par_do
            C(i) = min{ C(j) | D(j)=i and C(j)≠i }
            if 没有满足要求的 C(j) then
                C(i) = D(i)
            endif
        endfor
```

(2.3) **for** 每个 i : 0 ≤ i ≤ N-1 **par_do**
D(i) = C(i)
endfor
(2.4) **for** i=1 **to** log N **do**
/* 以下对应源程序中 CC_to_C()函数 */
for 每个 j : 0 ≤ j ≤ N-1 **par_do**
C(j) = C(C(j))
endfor
endfor
/* 以下对应源程序中 CD_to_D()函数 */
(2.5) **for** 每个 i : 0 ≤ i ≤ N-1 **par_do**
D(i) = min{ C(i), D(C(i)) }
endfor
endfor
End

MPI 源程序请参见章末附录。

1.3 单源最短路径

单源最短路径(Single Source Shortest Path)**问题**是指求从一个指定顶点 s 到其它所有顶点 i 之间的距离，因为是单一顶点到其它顶点的距离，所以称为单源。设图 G(V,E)是一个有向加权网络，其中 V 和 E 分别为顶点集合和边集合，其边权邻接矩阵为 W，边上权值 w(i,j) > 0，i,j∈V，V={0,1,…,N-1}。

设 dist(i)为最短的路径长度，即距离，其中 s∈V 且 i≠s。这里采用著名的 Dijkstra 算法，并将其并行化。

1.3.1 最短路径串行算法

Dijkstra 算法(Dijkstra Algorithm)是单源最短路径问题的经典解法之一，基本思想如下。

假定有一个待搜索的顶点表 VL，初始化时做： dist(s)←0，dist(i)=∞(i≠s)，VL=V。每次从 VL（非空集）中选取这样的一个顶点 u，它的 dist(u)最小。将选出的 u 点作为搜索顶点，对于其它还在 VL 内的顶点 v，若<u,v>∈E,而且 dist(u)+w(u,v)<dist(v)，则更新 dist(v)为 dist(u)+w(u,v)，同时从 VL 中将 u 删除，直到 VL 成为空集时算法终止。

算法 15.4 中给出了最短路径问题的Dijkstra串行算法，其时间复杂度为$O(N^2)$。

算法 15.4Dijkstra 串行算法

输入：边权邻接矩阵 W（约定顶点 i，j 之间无边连接时 w(i,j)=∞，且 w(i,i) = 0）、待计算顶点的标号 s

输出：dist(0 : N-1)，其中 dist(i)表示顶点 s 到顶点 i 的最短路径(1≤i≤N)

procedure Dijkstra
Begin
(1) dist(s) = 0
(2) **for** i=0 **to** N-1 **do**

```
        if  i≠s then dist(i) = ∞
    endfor
(3) VL = V
(4) for i=0 to N-1 do
    (4.1) 从 VL 中找一个顶点 u，使得 dist(u)最小
    (4.2) for (每个顶点 v∈VL) and (<u,v>∈E) do
              if dist(u)+w(u,v)<dist(v) then dist(v) = dist(u)+w(u,v)
              endif
          endfor
(5) VL = VL-{u}
    endfor
End
```

1.3.2 最短路径并行算法

在上一小节串行算法的基础上，这里将其并行化。思路如下：

⑴ 上述算法的(1)(2)两步中，每个处理器分派 n=N/p 个节点进行初始化。

⑵ 第(4.1)步可以并行化为：首先每个处理器分派 n 个节点分别求局部最小值，然后再 p 个处理器合作求全局最小值，最后再将这一最小值广播出去。p 个处理器合作方法如下：当 p 为偶数时，后 p/2 个处理器将自己的局部最小值分别发送到对应的前 p/2 个处理器中，由前 p/2 个处理器比较出 2 个局部最小值中相对较小者并保留。当 p 为奇数时，设 p=2h+1，则后 h 个处理器的值分别发送到前 h 个处理器中，比较并保留小值。一次这样的循环过后，p 个最小值减少了一半，两次后，变为 1/4，如此一层一层的比较，log P 次循环后，就可以得出唯一的全局最小值。

⑶ 第(4.2)步可以每个处理器分配 n 个顶点，然后独立进行更新 dist 的操作。

根据以上思路，最短路径的并行算法见算法 15.5，使用了p个处理器，其时间复杂度为 $O(N^2/p+N\log p)$。这里的实现算法和对应的源程序中，处理器 0 只进行输入输出工作，不参与任何其它计算；因此实际参加运算的处理器为p-1 个，所以有n=N/(p-1)；另外，布尔数组bdist用来标记各顶点是否已从VL中取出。

算法 15.5 Dijkstra 并行算法

输入：边权邻接矩阵 W（约定顶点 i，j 之间无边连接时 w(i,j)=∞，且 w(i,i) = 0）、待计算顶点的标号 s

输出：dist(0 : N-1)，其中 dist(i)表示顶点 s 到顶点 i 的最短路径(1≤i≤N)

```
procedure Dijkstra-parallel
Begin
    /* 以下对应源程序中 ReadMatrix()函数 */
(1) 处理器 0 读入边权邻接矩阵 W
    /* 以下初始化 dist 和 bdist 数组，对应源程序中 Init()函数 */
(2) for 每个顶点 i par_do
        if i=s then
            dist(i) = 0
            bdist(i) = TRUE
        else
            dist(i) = w(i,s)
            bdist(i) = FALSE
```

```
        endif
    endfor
    /* 以下是算法主循环，对应源程序中 FindMinWay()函数 */
(3) for i=1 to N-1 do
        /* 各处理器找出自己负责范围内未搜索节点中最小的 dist */
    (3.1) for 每个顶点 j par_do
            index = min{ j | bdist(j)=FALSE }
            num = dist(index)
        endfor
        /* 以下各处理器协作对 p-1 个 index 求最小 */
    (3.2) calnum = p-1
        for j=1 to log(p-1) par_do
            if calnum mod 2=0 then //本次循环参加比较的数据个数为偶数
            (ⅰ) calnum = calnum/2
            (ⅱ) 序号大于 calnum 的处理器将 index 和 num 发送给序号比自己小
                 calnum 的处理器
            (iii) 序号小于 calnum 的处理器（不包含 0 号）在自己原有的和收到
                 的两个 num 之间，比较并保留较小者，同时记录对应的 index
            else                    //本次循环参加比较的数据个数为奇数
            (ⅰ) calnum = (calnum+1)/2
            (ⅱ) 序号大于 calnum 的处理器将 index 和 num 发送给序号比自己小
                 calnum 的处理器
            (iii) 序号小于 calnum 的处理器（不包含 0 号）在自己原有的和收到
                 的两个 num 之间，比较并保留较小者，同时记录对应的 index
            endif
        endfor
    (3.3) 处理器 1 广播通知其它处理器自己的 num 和 index
        /* 以下并行更新 */
    (3.4) for 每个顶点 j par_do
            if bdist(j)=FALSE then dist(j) = min{ dist(j), num+w(j,index) }
            endif
        endfor
    (3.5) 顶点 index 对应处理器将对应的 bdist(index)更改为 TRUE
    endfor
End
```

MPI 源程序请参见所附光盘。

1.4 最小生成树

一个无向连通图 G 的生成树是指满足如下条件的 G 的子图 T：①T 和 G 顶点数相同；②T 有足够的边使得所有顶点连通，同时不出现回路。如果对 G 的每条边指定一个权值，那么，边权总和最小的生成树称为**最小代价生成树**，记为 MCST（Minimum Cost Spanning Tree），常简称为**最小生成树**（记为 MST）。**最小生成树问题**就是给定 G，找出 G 的一个最

小生成树 T 的问题。

本节将介绍用于求解 **MST 问题**的 Sollin 算法，并将其实现。

1.4.1 最小生成树串行算法

Sollin 算法(Sollin Algorithm)是众多用于求解 MST 问题的算法之一，其原理为：算法开始时，图中的 N 个顶点视为一片森林，每个顶点视为一棵树；算法主循环的流程中，森林里每棵树同时决定其连向其它树的最小邻边，并将这些边加入森林中，实现树的合并；循环到森林中只剩一棵树时终止。由于森林中树的数目每次至少减少一半，所以只要$\log N$ 次循环就可以找出 MST。

算法中以D(i)为顶点i所在的树的编号， edge(i) 和closet(i)为树i连向其它树的最小邻边和其长度，则Sollin算法的形式描述见算法 15.6，其时间复杂度为$O(N^2 \log N)$。

算法 15.6 Sollin MST 算法

输入：无向图 G 的边权矩阵 W

输出：G 的最小生成树的边集合 T

```
procedure Sollin-MST
Begin
    /* 以下所有顶点初始化为一棵孤立的树 */
(1) for i=0 to N-1 do
        D(i) = i
    endfor
(2) T 初始化为空集
(3) while |T| < N-1 do
        /* 以下为各树寻找连向其它树的最小边权 */
    (3.1) for 每棵树 i do
            closet(i) = ∞
        endfor
    (3.2) for 图 G 中每条边(v,u) do
            if D(v)≠D(u) and w(v,u)<closet(D(v)) then
            (ⅰ) closet(D(v)) = w(v,u)
            (ⅱ) edge(D(v)) = (v,u)
            endif
        endfor
        /* 以下是树的合并 */
    (3.3) for 每棵树 i do
        (ⅰ) (v,u) ← edge(i)
        (ⅱ) if D(v)≠D(u) then
                T = T + { (v,u) }
                树 D(v)和 D(u)合并
            endif
        endfor
    endwhile
End
```

1.4.2 最小生成树并行算法

在上一小节 Sollin 算法的基础上，本节将其并行化。基本思路是由多个处理器同时负责为多个树查找最短边。为了方便并行处理，各树寻找外连最短边的的任务分两步进行：①所有处理器独立并行为每个顶点找连向其它树的最短边，并将这些数据保存；②各处理器独立并行检索每棵树的各顶点，为整棵树找出外连最短边。注意在以上两步中，处理器间分别按照顶点和树进行分配，对象有了变化。

为了配合算法运行，设置了 4 个一维数组 D、C、J、W：① D(i)为顶点 i 所在的树的编号，也就是该树中最初的顶点号，初始化为 i；② C(i)为离顶点 i 最近的其它树的编号；③ J(i)为对应的顶点号；④ W(i)为对应的边权，即距离。以上变量约定对 MST 并行算法的源程序同样适用。运行期间每个处理器处理 n=N/p 个顶点或 n 个连通分量。

各树找出外连最短边后，接下来分 3 步进行树的合并：①让所有的 D(i)=C(i)；②对所有 i，进行 C(i)=C(C(i))运算，并循环$\log N$次；③对所有 i，更新 D(i)为 C(i)和 D(C(i))中较小者。以上合并过程类似顶点倒塌算法中超顶点的合并。

设MST为输出的最小生成树的邻接矩阵，运行期间由 0 号处理器维护，则并行算法的描述见算法 15.7，使用了p个处理器，其时间复杂度为$O(N^2/p\log N)$。

算法 15.7 并行 Sollin MST 算法

输入：原始图 G 的邻接矩阵 A

输出：所求最小生成树的邻接矩阵 MST

```
procedure Sollin-MST-parallel
Begin
    /* 以下初始化将图初始化为 N 棵树 */
(1) for 每个顶点 i par_do
    D(i) = i
  endfor
(2) while 图 G 未连通 do
        /* 以下各处理器为所负责的顶点找出距离最近的树
           对应源程序中 D_to_C()函数 */
    (2.1)for 每个顶点 i par_do
        (ⅰ) W(i) = MAX
        (ⅱ) for 每个顶点 j do
                if a(i, j)>0 and D(j)≠D(i) and a(i, j)<W(i) then
                    C(i) = D(j)
                    W(i) = a(i,j)
                    J(i) = j
                endif
            endfor
        (ⅲ) if W(i)=MAX then
                C(i) = D(i)
            endif
        endfor
        /* 以下各处理器为所负责的树找出外连的最短边
           对应源程序中 C_to_C()函数 */
    (2.2)for 每棵树 i par_do
        (ⅰ) tempj = N+1
```

```
        (ii) tempw = MAX
        (iii) for 每个顶点 j do
                if D(j)=i and C(j)≠i and W(j)<tempw then
                    C(i) = C(j)
                    tempw = W(j)
                    tempj = j
                endif
            endfor
        (iv) if tempj<N and J(tempj)<N then
                通知处理器 0 将边(tempj, J(tempj))加入 MST 数组
            endif
        endfor
    (2.3)for i=0 to N-1 par_do
            D(i) = C(i)
        endfor
    (2.4)for i=1 to log N do
            /* 以下对应源程序中 CC_to_C()函数 */
            for j=0 to N-1 par_do
                C(j) = C(C(j))
            endfor
        endfor
        /* 以下对应源程序中 CD_to_D()函数 */
    (2.5)for i=0 to N-1 par_do
            D(i) = min{ C(i), D(C(i)) }
        endfor
    endwhile
End
```

MPI 源程序请参见所附光盘。

1.5 小结

本章针对**传递闭包**、**连通分量**、**单源最短路径**、**最小生成树**等 4 个经典图论问题，分别介绍了它们的一种典型算法，以及在 MPI 并行环境下的算法实现。其中除连通分量外，其它 3 个问题的并行算法实现均由串行算法并行化而得到；通过这 3 个问题，较为详细的介绍了图论问题串行解法并行化的一般思路：均匀数据划分，独立计算。如何划分才能尽量做到计算的独立，是划分的关键。在最小生成树 Sollin 算法的并行化中，甚至在程序运行中按照不同的对象对数据进行了划分。

除了常见的串行算法并行化之外，本章还通过连通分量问题的解法，接触了根据并行机特点，直接为并行环境设计的图论问题解法。这种情况在非数值算法问题中并非特例。

本章的编写主要参考了文献[1]。其中，并行传递闭包和连通分量算法也可以参见文献[2]，单源最短路径Dijkstra算法也可以参见文献[3]，而最小生成树Sollin算法也可以参见文献[4]。

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]. Hirschberg D S. Parallel Algorithms for Transitive Closure and the Connected Component Problem. Proc. 8th Annu. ACM STOC, New York, 1976,55-57

[3]. Dijkstra E. A Note on Two Problems in Connetion with Graphs. Numerische Mathematic, 1959,1:269-271

[4]. Goodman S E and Hedetnimi S T ed. Introduction to the Design and Analysis of Algorithms(in An Algorithm Attributed to Sollin). McGraw-Hill, New York, 1977

附录 连通分量并行算法的 MPI 源程序

1. 源程序 connect.c

```
/*本算法中处理器数目须小于图的顶点数*/
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include <mpi.h>

/*使用动态分配的内存存储邻接矩阵，以下为宏定义*/
#define A(i,j) A[i*N+j]

/*以下为
    N:顶点数
    n:各处理器分配的顶点数
    p:处理器数*/
int N;
int n;
int p;
int *D,*C;
int *A;
int temp;
int myid;
MPI_Status status;

/*输出必要信息*/
void print(int *P)
{
    int i;
    if(myid==0)
    {
        for(i=0;i<N;i++)
            printf("%d ",P[i]);
        printf("\n");
    }
}

/*读入邻接矩阵*/
void readA()
{
    char *filename;
    int i,j;
    printf("\n");
    printf("Input the vertex num:\n");
    scanf("%d",&N);
    n=N/p;
    if(N%p!=0) n++;
    A=(int*)malloc(sizeof(int)*(n*p)*N);
    if(A==NULL){
    printf("Error when allocating memory\n");
    exit(0);
    }
    printf("Input the adjacent matrix:\n");
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            scanf("%d",&A(i,j));
    for(i=N;i<n*p;i++)
        for(j=0;j<N;j++)
            A(i,j)=0;
```

```
}

/*处理器 0 广播特定数据*/
void bcast(int *P)
{
    MPI_Bcast(P,N,MPI_INT,0,
        MPI_COMM_WORLD);
}

/*两者中取最小的数学函数*/
int min(int a,int b)
{
    return(a<b?a:b);
}

/*为各顶点找最小的邻接超顶点，对应算法步骤
    (2.1)*/
void D_to_C()
{
    int i,j;

    for(i=0;i<n;i++){
        C[n*myid+i]=N+1;
        for(j=0;j<N;j++)
        if((A(i,j)==1)
            &&(D[j]!=D[n*myid+i])
            &&(D[j]<C[n*myid+i])){
            C[n*myid+i]=D[j];
        }
        if(C[n*myid+i]==N+1)
            C[n*myid+i]=D[n*myid+i];
    }
}

/*为各超顶点找最小邻接超顶点，对应算法步骤
(2.2)*/
void C_to_C()
{
    int i,j;
    for(i=0;i<n;i++){
        temp=N+1;
        for(j=0;j<N;j++)
            if((D[j]==n*myid+i)
                &&(C[j]!=n*myid+i)
                &&(C[j]<temp)){
                temp=C[j];
            }
        if(temp==N+1) temp=D[n*myid+i];
        C[myid*n+i]=temp;
    }
}

/*调整超顶点标识*/
void CC_to_C()
{
    int i;
    for(i=0;i<n;i++)
        C[myid*n+i]=C[C[myid*n+i]];
}

/*产生新的超顶点，对应算法步骤(2.5)*/
void CD_to_D()
{
    int i;
    for(i=0;i<n;i++)
        D[myid*n+i]=
            min(C[myid*n+i],D[C[myid*n+i]]);
}

/*释放动态内存*/
void freeall()
{
    free(A);
    free(D);
    free(C);
}

/*主函数*/
void main(int argc,char *argv)
{
    int i,j,k;
    double l;
    int group_size;

    /*以下变量用来记录运行时间*/
    double starttime,endtime;

    MPI_Init(&argc,&argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD,
        &group_size);
MPI_Comm_rank(MPI_COMM_WORLD,
        &myid);

p=group_size;
MPI_Barrier(MPI_COMM_WORLD);
if(myid==0)
        starttime=MPI_Wtime();

/*处理器 0 读邻接矩阵*/
if(myid==0) readA();
MPI_Barrier(MPI_COMM_WORLD);

MPI_Bcast(&N,1,MPI_INT,0,
        MPI_COMM_WORLD);
if(myid!=0){
        n=N/p;
        if(N%p!=0) n++;
}

D=(int*)malloc(sizeof(int)*(n*p));
C=(int*)malloc(sizeof(int)*(n*p));
if(myid!=0)
        A=(int*)malloc(sizeof(int)*n*N);

/*初始化数组 D，步骤(1)*/
for(i=0;i<n;i++) D[myid*n+i]=myid*n+i;
MPI_Barrier(MPI_COMM_WORLD);
MPI_Gather(&D[myid*n],n,MPI_INT,D,n,
        MPI_INT,0,MPI_COMM_WORLD);
bcast(D);
MPI_Barrier(MPI_COMM_WORLD);

/*处理器 0 向其它处理器发送必要数据*/
if(myid==0)
        for(i=1;i<p;i++)
                MPI_Send(&A(i*n,0),n*N,
                        MPI_INT,i,i,
                        MPI_COMM_WORLD);
else
        MPI_Recv(A,n*N,MPI_INT,
                0,myid,MPI_COMM_WORLD,
                &status);
MPI_Barrier(MPI_COMM_WORLD);

l=log(N)/log(2);
/*主循环体：算法步骤(2)*/
for(i=0;i<l;i++){
        if(myid==0) printf("Stage %d:\n",i+1);

        /*算法步骤(2.1)*/
        D_to_C();
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Gather(&C[n*myid],n,MPI_INT,C,
                n,MPI_INT,0,
                MPI_COMM_WORLD);
        print(C);
        bcast(C);
        MPI_Barrier(MPI_COMM_WORLD);

        /*算法步骤(2.2)*/
        C_to_C();
        print(C);
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Gather(&C[n*myid],n,MPI_INT,C,
                n,MPI_INT,0,
                MPI_COMM_WORLD);
        MPI_Gather(&C[n*myid],n,MPI_INT,D,
                n,MPI_INT,0,
                MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);

        /*算法步骤(2.3)*/
        if(myid==0)
                for(j=0;j<n;j++)
                        D[j]=C[j];

        /*算法步骤(2.4)*/
        for(k=0;k<l;k++){
                bcast(C);
                CC_to_C();
                MPI_Gather(&C[n*myid],n,
                        MPI_INT,C,n,MPI_INT,0,
                        MPI_COMM_WORLD);
        }
        bcast(C);
        bcast(D);
```

```
            /*算法步骤(2.5)*/
            CD_to_D();
            MPI_Gather(&D[n*myid],n,
                  MPI_INT,D,n,MPI_INT,0,
                  MPI_COMM_WORLD);
            print(D);
            bcast(D);
      }

      if(myid==0) printf("Result: \n");
      print(D);
      if(myid==0){
            endtime=MPI_Wtime();
               printf("The running time is %d\n",
                        endtime-starttime);
      }

      freeall();
      MPI_Finalize();
}
```

2. 运行实例

编译：mpicc connect.c

运行：本实例中使用 3 个处理器。

mpirun –np 3 a.out

运行结果：

Input the vertex num:

8

Input the adjacent matrix:

0 0 0 0 0 0 0 1

0 0 0 0 0 1 1 0

0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 1

0 0 0 0 0 0 1 1

0 1 0 1 0 0 0 0

0 1 0 0 1 0 0 0

1 0 0 1 1 0 0 0

最终输出结果：

Result:

0 0 2 0 0 0 0 0

说明：使用了和上一小节传递闭包时相同的输入矩阵，得到的结果也相同。除顶点 2 外，其余顶点构成另一个连通分量

4 组合优化

组合优化问题在实践中有着广泛的应用，同时也是计算机科学中的重要研究课题。本章对于八皇后问题、SAT 问题、装箱问题、背包问题及 TSP 问题等五个经典的组合优化问题，给出其定义、串行算法描述、并行算法描述以及并行算法的 MPI 源程序。

1.1 八皇后问题

1.1.1 八皇后问题及其串行算法

所谓**八皇后问题**（Eight Queens Problem），是在 8*8 格的棋盘上，放置 8 个皇后。要求每行每列放一个皇后,而且每一条对角线和每一条反对角线上最多只能有一个皇后，即对同时放置在棋盘的任意两个皇后 (i_1, j_1) 和 (i_2, j_2)，不允许 $(i_1 - i_2) = (j_1 - j_2)$ 或者 $(i_1 + j_1) = (i_2 + j_2)$ 的情况出现。

八皇后问题的串行解法为如下的递归算法：

算法 16.1 八皇后问题的串行递归算法

```
/* 从 chessboard 的第 row 行开始放置皇后 */
procedure PlaceQueens(chessboard, row)
Begin
    if row > 8 then
        OutputResult(chessboard)          /* 结束递归并输出结果 */
    else
        for col = 1 to 8 do  /* 判断是否有列、对角线或反对角线冲突 */
            （1）no_collision = true
            （2）i = 1
            （3）while no_collision and i < row do
                    （3.1）if collides(i, chessboard[i], row, col) then
                                no_collision = false
                           end if
                    （3.2）i = i + 1
                  end while
            （4）if no_collision then
                    （4.1）chessboard[row] = col      /* 在当前位置放置一个皇后 */
                    （4.2）go(step + 1, place)        /* 递归地从下一行开始放置皇后 */
                  end if
        end for
    end if
End /* PlaceQueens */
```

1.1.2 八皇后问题的并行算法

该算法是将八皇后所有可能的解置于相应的棋盘上，主进程负责生成初始化的棋盘，并将该棋盘发送到某个空闲的从进程，由该从进程求出该棋盘上满足初始化条件的所有的解。这里，我们假定主进程只初始化棋盘的前两列，即在棋盘的前两列分别放上 2 个皇后，这样就可以产生 8 * 8 = 64 个棋盘。

算法 16.2 八皇后问题的并行算法

（1）主进程算法

```
procedure EightQueensMaster
Begin
    （1）active_slaves = n
    （2）while active_slaves > 0 do
            （2.1）从某个从进程 i 接收信号 signal
            （2.2）if signal = Accomplished then 从从进程 i 接收并记录解 end if
            （2.3）if has_more_boards then
                        (ⅰ)向从进程 i 发送 NewTask 信号
                        (ⅱ)向从进程 i 发送一个新棋盘
                   else
                        (ⅰ)向从进程 i 发送 Terminate 信号
                        (ⅱ)active_slaves = active_slaves - 1
                   end if
        end while
End /* EightQueensMaster */
```

（2）从进程算法

```
procedure EightQueenSlave
    Begin
  （1）向主进程发送 Ready 信号
  （2）finished = false
  （3）while not finished do
          （3.1）从主进程接收信号 signal
          （3.2）if signal = NewTask then
                    (ⅰ)从主进程接收新棋盘
                    (ⅱ)if 新棋盘合法 then
                            在新棋盘的基础上找出所有合法的解，并将解发送给主进程
                       end if
                 else /* signal = Terminate */
                       finished = true
                 end if
        end while
End /* EightQueenSlave */
```

MPI 源程序请参见章末附录。

1.2 SAT 问题

1.2.1 SAT 问题及其串行算法

1.SAT 问题描述

所谓**可满足性问题**（Satisfiability Problem）即 **SAT 问题**，其定义为：对于一个命题逻辑公式，是否存在对其变元的一个真值赋值公式使之成立。这个问题在许多领域都有着非常重要的意义，而且对其快速求解算法的研究成为计算机科学的核心课题之一。例如在机器定理证明领域，某命题是否是一个相容的公理集合的推论，这个问题归结为该命题的反命题与该公理集合一起是否是不可满足的。特别是 1971 年 Cook 首次证明了 SAT 是 NP-完全的，从而大量的计算问题都可以归结到 SAT。正是由于 SAT 重要的理论和应用地位，众多学者对它进行了广泛而深入的研究。

由命题逻辑知识可以知道，任何命题逻辑公式都逻辑等价与一个**合取范式**(Conjunctive Normal Form，简写为 CNF),因此本文只讨论 CNF 的 SAT 求解问题。

下面给出几种常见的SAT问题化简策略，以下均假定F是CNF范式：①**单元子句规则**：若C=｛L｝是F的子句，则F变为$F'=F[F/1]$；②**纯文字规则**：若文字L出现在F中，而$\neg L$不出现F中，则L称为F的纯文字。F变为$F'=F[F/1]$；③**重言子句规则**：若$C\in F$且C是重言子句，则从F中删去C得$F'=F-C$；④**两次出现规则**：若$L\in C_1$，$\neg L\in C_2$，且L和$\neg L$都不在其它子句中出现，则将C_1，C_2合并为$C'=(C_1-\{L\})\cup(C_2-\{\neg L\})$，$F$变为$F'=F-\{C_1, C_2\}\cup\{C'\}$;⑤ **包含规则**：若$C_1$，$C_2$是$F$的子句，且$C_1\in C_2$，则$F$中删去$C_2$，得$F'=F-\{C_2\}$。

2.SAT 问题串行算法

SAT问题的**DP算法**是由M.Davis和H.Putnam于 1960 年首次提出[2]，现在已经成为最著名的SAT算法，其描述如下：

算法 16.3 SAT 问题的 DP 算法
输入：合取范式 F。
输出：F 是否可满足。

```
procedure DP(F)
Begin
(1) if F 为空公式 then
        return Yes
    end if
(2) if F 包含一个空子句 then
        return No
    end if
(3) if F 包含一个子句{l} then          /* 单元子句规则 */
        return DP(F[l/1])
    end if
(4) if F 包含一个文字 l，但不包含 ¬l then    /* 纯文字规则 */
        return DP(F[l/1])
    end if
```

(5) 选择一个未赋值的文字 *l*
(6) /* 拆分 */
if DP(*F* [*l*/1]) = Yes **then**
return Yes
else
return DP(*F* [*l*/0])
end if
End /* DP ***/**

可以看出，DP 算法是对回溯算法的精化，其中应用了我们在前面提到的化简策略单元子句规则和纯文字规则。前面已经介绍过，这些策略并不能在数量级上降低算法的时间复杂度，但实验证明这些策略的应用可以极大的改善平均性能。其实，上面介绍的策略都可以应用于 SAT 的求解，而且已经有了这方面的工作。

1.2.2 SAT 问题的并行算法

1.并行化思路

通过我们在前面对串行DP算法的介绍可以看出，实际上DP算法仍然是利用深度优先方法对可能的解空间做深度优先搜索，这样我们仍然可以把这个解空间看成一棵二叉树。而它与回溯搜索算法的区别仅仅在于它利用了SAT的一些性质巧妙的找到了一些仅有一种赋值可能的文字，这样就有效地减少了搜索开销。同时在搜索一棵子树时，对某个文字的赋值可能导致新的单元子句的产生，这样，从平均意义上考虑，对这一性质的反复利用可以极大地加快搜索速度。容易知道，对于寻找单元子句和纯文字的工作是在多项式时间内完成的，因此我们可以由主进程预先把CNF的所有单元子句和纯文字找出来，对它们分别赋上可能使CNF得到满足的值，并按照某种策略选取n个文字对他们预先赋值，共得到 2^n组解。然后把这些信息分别发送给各个从进程进行计算，并收集运算结果。这样既避免了各个从进程重复寻找单元子句和纯文字，又有可能通过对选出的n个文字的赋值产生了新的单元子句，从而加快了整个程序的搜索速度。

2.并行 DP 算法

算法 16.4 SAT 问题的并行 DP 算法
输入：合取范式 *F*。
输出：*F* 是否可满足。

（1）主进程算法

procedure PDPMaster(*F*)
Begin
（1）找出 n 个文字，并初始化任务队列
（2）j = 0
（3）向每个从进程发送一个任务
（4）**while** true **do**
（4.1）某个从进程p_i接收结果
（4.2）**if** result = true **then**
（i）向所有从进程发送终止信号
（ii）**return** true

else

if (j > 2^n) **then**

（i）向所有从进程发送终止信号

（ii）**return** false

else

（i）向p_i发送下一个任务

end if

end if

end while

End /* PDPMaster */

（2）从进程算法

procedure PDPSlave

Begin

for each processor p_i，**where** $1 \le i \le k$ **do**

while true **do**

（1）从主进程接收信号

（2）**if** signal = task **then**

（i）用该任务更新 *F*

（ii）将 DP(*F*)的结果发送给主进程

else if signal = terminal **then**

return

end if

end if

end while

end for

End /* PDPSlave */

3.算法实现的说明

在这里我们实际上利用了集中控制的动态负载平衡技术，由主进程控制一个任务队列。首先给每个从进程发送一个任务，然后不断从这个队列中取出任务，并通过与相应的进程应答决定是发送给它新的任务，还是结束所有进程。而从进程不断从主进程中接收信号，决定是执行新的计算任务还是结束。

众所周知，DP 算法中的拆分文字是非常重要的，特别是早期的文字拆分更显得举足轻重，因为早期的错误会导致多搜索一个子树，工作量几乎增加一倍。例如，Gent 和 Walsh 就给出了一个这样的例子，早期的文字选择错误导致了先求解一个具有 350,000,000 个分枝的不可满足的子问题。如果在初始的文字拆分中，提高 DP 算法的拆分文字的命中率，无疑会达到事半功倍的效果。

为了提高拆分文字的命中率，编程实现中，主进程划分任务的时候，预先找出 CNF 范式中的纯文字和单元子句，这样就大大减少了需要搜索的子树数目。

MPI 源程序请参见所附光盘。

1.3 装箱问题

1.3.1 装箱问题及其串行算法

经典的一维**装箱问题**(Bin Packing Problem)是指，给定 n 件物品的序列 $L_n =< a_1, a_2, \cdots, a_n >$，物品 $a_i (1 \le i \le n)$ 的大小 $s(a_i) \in (0,1]$，要求将这些物品装入单位容量 1 的箱子 $B_1, B_2, \cdots, B_m$ 中，使得每个箱子中的物品大小之和不超过 1，并使所使用的箱子数目 m 最小。

下次适应算法（Next Fit Algorithm）是最早提出的、最简单的一个在线算法，[Joh73]首次仔细分析了下次适应算法的最坏情况性能。下次适应算法维持一个当前打开的箱子，它依序处理输入物品，当物品到达时检查该物品能否放入这个当前打开的箱子中，若能则放入；否则把物品放入一个新的箱子中，并把这个新箱子作为当前打开的箱子。算法描述如下：

算法 16.5 装箱问题的下次适应算法

输入：输入物品序列 $L =< a_1, a_2, \cdots, a_n >$。

输出：使用的箱子数目 m，各装入物品的箱子 $P =< B_1, B_2, \cdots, B_m >$。

```
procedure NextFit
Begin
    （1）s(B) = 1        /* 初始化当前打开的箱子 B，令 B 已满 */
    （2）m = 0           /* 使用箱子计数 */
    （3）for i = 1 to n do
            if s(a_i) + s(B) ≤ 1 then
                （i） s(B) = s(B) + s(a_i)      /* 把a_i放入B中 */
            else
                （i） m = m + 1                 /* 使用的箱子数加一 */
                （ii） B = B_m                  /* 新打开箱子B_m */
                （iii）s(B) = s(a_i)            /* 把a_i放入B中 */
            end if
        end for
End /* NextFit */
```

1.3.2 装箱问题的并行算法

算法 16.5 使用一遍扫描物品序列来实现，本身具有固有的顺序性，其时间复杂度为 O(n)。我们使用平衡树和倍增技术对其进行并行化。首先利用前缀和算法建立一个链表簇，令 *A[i]* 为输入物品序列中第 i 件物品的大小，如果链表指针由 *A[j]* 指向 *A[k]*，则表示 *A[j]*+*A[j*+1]+…+*A[k]*>1 且 *A[j]*+*A[j*+1]+…+*A[k*-1]≤1；其后利用倍增技术计算以 *A[*1]为头的链表的长度，而以 *A*(1)为头的链表的长度即是下次适应算法所使用的箱子数目。接下来利用在上一步骤中产生的中间数据反向建立一棵二叉树，使该二叉树的叶节点恰好是下次适应算法在各箱子中放入的第一个物品的序号；最后，根据各箱子中放入的第一个物品的序号，使

用二分法确定各物品所放入的箱子的序号。

算法 16.6 并行下次适应算法

输入：数组 $A[1,n]$，其中 $A[i]$为输入物品序列中第 i 个物品的大小。

输出：使用的箱子数目 m，每个物品应放入的箱子序号数组 $B[1,n]$。

procedure PNextFit

Begin

（1）调用求前缀和的并行算法计算 $F[j]= A[1]+A[2]+\cdots+A[j]$

（2）**for** j = 1 **to** n **pardo**

借助 $F[j]$，使用二分法计算 $C[0,j]= \max\{k|A[j]+A[j+1]+\cdots+A[k] \le 1\}$

end for

/* 以下（3）-（8），计算下次适应算法使用的箱子数目 m */

（3）$C[0, n+1] = n$

（4）$h = 0$

（5）**for** j=1 **to** n **pardo** $E[0, j]$=1 **end for**

（6）**while** $C[h,1] \neq n$ **do**

（6.1）$h = h + 1$

（6.2）**for** j = 1 **to** n **pardo**

if $C[h - 1, j] = n$ **then**

（i）$E[h, j] = E[h-1, j]$

（ii）$C[h,j] = C[h - 1,j]$

else

（i）$E[h, j] = E[h - 1, j] + E[h - 1,C[h - 1, j] + 1]$

（ii）$C[h, j] = C[h - 1,C[h - 1, j] + 1]$

end if

end for

end while

（7）$height = h$

（8）$m = E[height, 1]$

（9）/* 计算 $D[0, j]$=第 j 个箱子中第一件物品在输入序列中的编号 */

for $h = height$ **downto** 0 **do**

for j = 1 **to** $n / 2^h$ **pardo**

（i）**if** $j = even$ **then** $D[h,j] = C[h,D[h-1,j/2]]+1$ **endif**

（ii）**if** j = 1 **then** $D[h,1]$ = 1 **endif**

（iii）**if** $j = odd > 1$ **then** $D[h,j] = D[h-1,[j+1]/2]$ **endif**

end for

end for

（10）**for** j=1 **to** n **pardo** /* 计算 $B[j]$ = 第 j 个物品所放入的箱子序号 */

使用二分法计算 $B[j] = \max\{ k \mid D[0,k] \le j , D[0,k+1] > j$ 或者 $k = m \}$

end for

End /* PNextFit */

MPI 源程序请参见所附光盘。

1.4 背包问题

1.4.1 背包问题及其串行算法

0-1 背包问题（0-1 Knapsack Problem）的定义为：设集合 $A=\{a_1,a_2,\cdots,a_m\}$ 代表 m 件物品,正整数 p_i,w_r 分别表示第 i 件物品的价值与重量，那么 0-1 背包问题 KNAP(*A*,*c*)定义为，求 *A* 的子集，使得重量之和小于背包的容量 *c*，并使得价值和最大。也就是说求：

$$\begin{aligned}&\max\sum_{i=1}^{m}p_ix_i\\&subject\ \ to\ \ \sum_{i=1}^{m}w_ix_i\le c\end{aligned}\tag{16.1}$$

其中 $x_i\in\{0,1\}$。

解决 0-1 背包问题的最简单的方法是**动态规划**（Dynamic Programming）算法。我们先看看 KNAP(*A*, c)的一个子问题 KNAP(A_j,y)，其中 $A_j=\{a_1,a_2,\cdots,a_j\},y\le c$。显然，若 a_j 并不在最优解中，那么 KNAP (A_{j-1},y) 的解就是 KNAP (A_j,y) 的解。否则，KNAP $(A_{j-1},y-w_j)$ 就是 KNAP (A_j,y) 的解。设 $f_j(y)$ 是 KNAP (A_j,y) 的最优解，我们得到下面的最优方法：

$$\begin{aligned}&f_0(y)=\begin{cases}0 & if\ \ y\ge 0\\ -\infty & if\ \ y<0\end{cases}\\&f_j(y)=\max\{f_{j-1}(y),\ f_{j-1}(y-w_i)+p_j\}\\&\ \ for\ \ all\ \ 0\le y\le c\ \ and\ \ 1\le j\le m\end{aligned}\tag{16.2}$$

设 $\vec{F}_j(c)=(f_j(0),f_j(1),\cdots,f_j(c))\quad 0\le j\le m$，那么，动态规划算法就是依次地计算 $\vec{F}_1(c),\vec{F}_2(c),\cdots,\vec{F}_m(c)$ 来求解问题。其中 $\vec{F}_j(c)$ 是 KNAP(Aj，c)的最大价值向量。

动态规划算法是基于 Bellman 递归循环，它是求解决背包问题的最直接的方法。基于(16.2)式我们可以写出串行算法如下：

算法 16.7 0-1 背包问题的串行动态规划算法

输入：各件物品的价值$p_1,\cdots,p_m$，各件物品的重量$w_1,\cdots,w_m$，背包的容量c。

输出：能够放入背包的物品的最大总价值$f_m(c)$。

procedure Knapsack(p, w, c)

Begin

（1）**for** i = 0 **to** c **do** $f_0(i) = 0$ **end for**

（2）**for** i = 1 **to** m **do**

（2.1）$f_i(0) = 0$

（2.2）**for** j = 1 **to** c **do**

if $w_i \le j$ **then**

if $p_i + f_{i-1}(j - w_i) > f_{i-1}(j)$ **then**

$f_i(j) = p_i + f_{i-1}(j - w_i)$

else $f_i(j) = f_{i-1}(j)$

end if

else $f_i(j) = f_{i-1}(j)$

end if

end for

end for

End /* Knapsack */

可以看出，串行的动态规划算法需要占用很大的空间，其本质是 Bellman 递归，最坏和最好的情况下的时间差不多相同。虽然，它在问题规模比较小时，可以很好的解决问题，但是，随着问题规模的扩大，串行动态规划算法就显得力不从心了。

1.4.2 背包问题的并行算法

现在，我们要做的是把串行程序改成并行程序。首先，我们分析一下串行程序的特点。注意到第（2.2）步的for循环，$f_i(j)$只用到上一步$f_i(y)$的值，而与同一个i循环无关，这样，可以把第（2.2）步直接并行化。得到下面的并行算法：

算法 16.8 0-1 背包问题的并行算法

输入：各件物品的价值$p_1,\cdots,p_m$，各件物品的重量$w_1,\cdots,w_m$，背包的容量c。
输出：能够放入背包的物品的最大总价值$f_m(c)$。

procedure ParallelKnapsack(p, w, c)

Begin

（1）**for each** processor $0 \le j \le c$ **do** $f_0(j) = 0$ **end for**

（2）**for** i = 1 t**o** m **pardo**

（2.1）**for each** processor $0 \le j < w_i$ **do**

$$f_i(j) = f_{i-1}(j)$$

end for

（2.2）**for each** processor $w_i \le j \le c$ **do**

$$f_i(j) = \max\{f_{i-1}(j), f_{i-1}(j - w_i) + p_i\}$$

end for

end for

End /* ParallelKnapsack */

MPI 源程序请参见所附光盘。

1.5 TSP 问题

1.5.1 TSP 问题及其串行算法

TSP 问题（Traveling-Salesman Problem）可描述为：货郎到各村去卖货，再回到出发处，每村都要经过且仅经过一次，为其设计一种路线，使得所用旅行售货的时间最短。

TSP 问题至今还没有有效的算法，是当今图论上的一大难题。目前只能给出近似算法，其中之一是所谓“改良圈算法”，即已知$v_1v_2...v_vv_1$是 G 的 Hamilton 圈，用下面的算法把它的权减小：

算法 16.9 TSP 问题的改良圈算法

输入：任意的一个 Hamilton 圈。

输出：在给定的 Hamilton 圈上降低权而获得较优的 Hamilton 圈。

procedure ApproxTSP

Begin

（1）任意选定一个 Hamiltion 圈，将圈上的顶点顺次记为$v_1, v_2, \cdots, v_v$，

则该圈为$C = \{v_1v_2, v_2v_3, \cdots, v_{v-1}v_v, v_vv_1\}$

（2）**while** 存在 i≠j 使得($v_iv_{i+1}, v_jv_{j+1} \in C$，$v_iv_j, v_iv_{j+1} \in E(G)$) **do**

if $w(v_iv_j) + w(v_{i+1}v_{j+1}) < w(v_iv_{i+1}) + w(v_jv_{j+1})$ **then**

(2.1) $C = (C - \{v_iv_{i+1}, v_jv_{j+1}\}) \cup \{v_iv_j, v_{i+1}v_{j+1}\}$

(2.2) 将新的圈C上的顶点顺次记为$v_1, v_2, \cdots, v_v$

end if

end while

End /* ApproxTSP */

1.5.2 TSP 问题的并行算法

并行算法实际上是对串行算法的改进，主要是在算法 16.9 的步骤(1)上的改进。每个从进程检查原来的圈上不同的对边，即对进程$k(1 \le k \le n)$，检查下标索引为 i，j 的对边，$w(v_i, v_j) + w(v_{i+1}, v_{j+1}) < w(v_i, v_{i+1}) + w(v_j, v_{j+1})$是否成立；如果成立，则记下减小的权；在每一轮上，选择权减小最多的对边来改进原圈，得到新的改良圈；直到不能有任何改进为止。得到的改进算法仍然是近似算法。

算法 16.10 并行 TSP 算法

输入：任意不同两点之间的距离 w(i,j)。

输出：在这些给定点上的一个较优的回路。

（1）主进程算法

procedure ParallelApproxTSPMaster(w)

Begin

（1）任意选定一个 Hamilton 圈，将圈上的顶点顺次记为 $v_1, v_2, \cdots, v_v$，

则该圈为 $C = \{v_1v_2, v_2v_3, \cdots, v_{v-1}v_v, v_vv_1\}$

（2）将 C 发送给每个从进程 $k(1 \le k \le n)$

（3）**while** true **do**

（3.1）从每个从进程 $k(1 \le k \le n)$ 接收 Δw_k

（3.2）$\Delta w = \min\{\Delta w_k \mid 1 \le k \le n\}$，

并记录最小的 Δw_k 所对应的处理器编号，记为 m

（3.3）**if** $\Delta w = 0$ **then** /*没有任何改进*/

向所有从进程发送终止信号

return C

else

将 m 发送到各处理器

end if

end while

End /* PrallelApproxTSPMaster */

（2）从进程算法

procedure ParallelApproxTSPSlave

Begin /* 设当前从进程的编号为 $k(1 \le k \le n)$ */

（1）从主进程接收 Hamilton 圈，将圈上的顶点顺次记为 $v_1, v_2, \cdots, v_v$，

则该圈为 $C = \{v_1v_2, v_2v_3, \cdots, v_{v-1}v_v, v_vv_1\}$

（2）accomplished = false

（3）**while** not accomplished **do**

（3.1）**for** all $1 \le i, j \le v$ 且 $j - i > 1$ **do**

if (i + j) mod n = k **then**

$temp = (w(v_i, v_{i+1}) + w(v_j, v_{j+1})) - (w(v_i, v_j) + w(v_{i+1}, v_{j+1}))$

if temp > Δw_k **then**

$\Delta w_k = temp$

$p = i$
$q = j$
end if
end if
end for

（3.2）$C = (C - \{v_p v_{p+1}, v_q v_{q+1}\}) \cup \{v_p v_q, v_{p+1} v_{q+1}\}$，

将新的圈 C 上的顶点顺次记为 $v_1, v_2, \cdots, v_v$

（3.3）将 Δw_k 发送给主进程

（3.4）从主进程接收最优改良对应的处理器编号 m

（3.5）**if** k = 0 **then**
accomplished = true
else if k = m **then**
向其它所有从进程发送改良圈 C
else
从从进程 m 接收改良圈 C
end if
end if
end while
End /* ParallelTSPSlave */

MPI 源程序请参见所附光盘。

1.6 小结

本章主要讨论了八皇后、SAT、装箱、背包和 TSP 等经典组和优化问题的并行算法及其 MPI 编程实现。对这些问题读者如欲进一步学习可参考文献[1]、[2]、[3]、[4]和[5]。

参考文献

[1]. 朱洪, 陈增武, 段振华, 周克成 编著. 算法设计和分析. 上海科学技术文献出版社, 1989

[2]. Davis M, Putnam H. A Computing Procedure for Quantification Theory. J. of the ACM, 1960, 7: 201~215

[3]. Gu xiaodong, Chen Guoliang, Gu Jun, Huang Liusheng, Yunjae Jung. Performance Analysis and Improvement for Some Liner on-line Bin-Packing Algorithms. J. of Combinatorial Optimization, 2002,12, 6:455~471

[4]. 陈国良, 吴明, 顾钧. 搜索背包问题的并行分支界限算法. 计算机研究与发展, 2001.6, 38(6):741~745

[5]. 万颖瑜, 周智, 陈国良, 顾钧. Sizescale:求解旅行商问题(TSP)的新算法. 计算机研究与发展, 2002.10, 39(10):1294~1302

附录 八皇后问题并行算法的 MPI 源程序

1.源程序 Pqueens.c

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

#define QUEENS 8
#define MAX_SOLUTIONS 92

typedef int bool;
const int true = 1;
const int false = 0;

enum msg_content
{
        READY,
        ACCOMPLISHED,
        NEW_TASK,
        TERMINATE
};

enum msg_tag
{
        REQUEST_TAG,
        SEED_TAG,
        REPLY_TAG,
        NUM_SOLUTIONS_TAG,
        SOLUTIONS_TAG
};

int solutions[MAX_SOLUTIONS][QUEENS];
int solution_count = 0;

bool collides(int row1, int col1, int row2, int col2)
{
        return (col1 == col2)
                || (col1 - col2 == row1 - row2)
                || (col1 + row1 == col2 + row2);
} /* collides */

int generate_seed()
{
        static int seed = 0;
        do
        {
                seed++;
        } while (seed <= QUEENS * QUEENS – 1
                && collides(0, seed / QUEENS, 1,
                seed % QUEENS));

        if (seed > QUEENS * QUEENS - 1)
                return 0;
        else
                return seed;
} /* generate_seed */

void print_solutions(int count,
        int solutions[][QUEENS])
{
        int i, j, k;

        for (i = 0; i < count; i++)
        {
                printf("Solution %d :\n", i + 1);
                for (j = 0; j < QUEENS; j++)
                {
                        printf("%d ", solutions[i][j]);
                        for (k = 0; k < solutions[i][j]; k++)
                                printf("- ");
                        printf("* ");
                          for (k = QUEENS - 1;
                                k > solutions[i][j]; k--)
                                printf("- ");
                        printf("\n");
                }
                printf("\n");
        }
} /* print_solutions */

bool is_safe(int chessboard[], int row, int col)
{
        int i;
```

```
    for (i = 0; i < row; i++)
    {
        if (collides(i, chessboard[i], row, col))
            return false;
    } /* for */
    return true;
} /* is_safe */

void place_queens(int chessboard[], int row)
{
    int i, col;

    if (row >= QUEENS)
    {
        /* 记录当前解 */
        for (i = 0; i < QUEENS; i++)
        {
            solutions[solution_count][i] =
                chessboard[i];
        }
        solution_count++;
    }
    else
    {
        for (col = 0; col < QUEENS; col++)
        {
            if (is_safe(chessboard, row, col))
            {
                /* 在当前位置放置一个
                    皇后 */
                chessboard[row] = col;
                /* 递归放置下一个皇后 */
                place_queens(chessboard,
                    row + 1);
            } /* if */
        } /* for */
    } /* else */
} /* place_queens */

void sequential_eight_queens()
{
    int chessboard[QUEENS];

    solution_count = 0;
    place_queens(chessboard, 0);
    print_solutions(solution_count, solutions);
}

void eight_queens_master(int nodes)
{
    MPI_Status status;
    int active_slaves = nodes - 1;
    int new_task = NEW_TASK;
    int terminate = TERMINATE;
    int reply;
    int child;
    int num_solutions;
    int seed;

    while (active_slaves)
    {
        MPI_Recv(&reply, 1,
            MPI_INT,
            MPI_ANY_SOURCE,
            REPLY_TAG,
            MPI_COMM_WORLD,
            &status);

        child = status.MPI_SOURCE;

        if (reply == ACCOMPLISHED)
        {
            /* 从子进程接收并记录解 */
            MPI_Recv(&num_solutions, 1,
                MPI_INT,
                child,
                NUM_SOLUTIONS_TAG,
                MPI_COMM_WORLD,
                &status);

            if (num_solutions > 0)
            {
                MPI_Recv(solutions[solution_
                count],
                QUEENS * num_solutions,
                MPI_INT,
                child,
                SOLUTIONS_TAG,
```

```
                    MPI_COMM_WORLD,
                    &status);
                    solution_count +=
                    num_solutions;
                }
            }

            seed = generate_seed();
            if (seed) /* 还有新的初始棋盘 */
            {
                /* 向子进程发送一个合法的新棋
                盘 */
                MPI_Send(&new_task, 1,
                    MPI_INT,
                    child,
                    REQUEST_TAG,
                    MPI_COMM_WORLD);
                MPI_Send(&seed, 1,
                    MPI_INT,
                    child,
                    SEED_TAG,
                    MPI_COMM_WORLD);
            }
            else /* 已求出所有解 */
            {
                /* 向子进程发送终止信号 */
                MPI_Send(&terminate, 1,
                    MPI_INT,
                    child,
                    REQUEST_TAG,
                    MPI_COMM_WORLD);
                active_slaves--;
            }
        } /* while */

        print_solutions(solution_count, solutions);
} /* eight_queens_master */

void eight_queens_slave(int my_rank)
{
        MPI_Status status;
        int ready = READY;
        int accomplished = ACCOMPLISHED;
        bool finished = false;
        int request;
        int seed;
        int num_solutions = 0;
        int chessboard[QUEENS];

        MPI_Send(&ready, 1,
            MPI_INT,
            0,
            REPLY_TAG,
            MPI_COMM_WORLD);

        while (! finished)
        {
            /* 从主进程接收消息 */
            MPI_Recv(&request, 1,
                MPI_INT,
                0,
                REQUEST_TAG,
                MPI_COMM_WORLD,
                &status);

            if (request == NEW_TASK)
            {
                /* 从主进程接收初始棋盘 */
                MPI_Recv(&seed, 1,
                    MPI_INT,
                    0,
                    SEED_TAG,
                    MPI_COMM_WORLD,
                    &status);

                /* 在初始棋盘基础上求解 */
                chessboard[0] = seed / QUEENS;
                chessboard[1] = seed % QUEENS;

                solution_count = 0;
                place_queens(chessboard, 2);

                /* 将解发送给主进程 */
                MPI_Send(&accomplished, 1,
                    MPI_INT,
                    0,
                    REPLY_TAG,
                    MPI_COMM_WORLD);
```

```
                MPI_Send(&solution_count, 1,
                        MPI_INT,0,
                        NUM_SOLUTIONS_TAG,
                        MPI_COMM_WORLD);
                if (solution_count > 0)
                {
                        MPI_Send(*solutions,
                                QUEENS *
                                solution_count,
                                MPI_INT,
                                0,
                                SOLUTIONS_TAG,
                                MPI_COMM_WORLD)
                                ;
                }
        }
        else /* request == TERMINATE */
        {
                finished = true;
        }
    } /* whlie */
} /* eight_queens_slave */

/***** main *****/
int main(int argc, char* argv[])
{
    int nodes, my_rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
        &nodes);
    MPI_Comm_rank(MPI_COMM_WORLD,
        &my_rank);

    if (nodes == 1)
    {
        sequential_eight_queens();
    }
    if (! my_rank)
    {
        eight_queens_master(nodes);
    }
    else
    {
        eight_queens_slave(my_rank);
    }

    MPI_Finalize();

    return 0;
} /* main */
```

2.运行实例

编译：mpicc queen.c –o queen
运行：mpirun –np 4 queen
运行结果：

```
Solution 1 :
0 * - - - - - - -
4 - - - - * - - -
7 - - - - - - - *
5 - - - - - * - -
2 - - * - - - - -
6 - - - - - - * -
1 - * - - - - - -
3 - - - * - - - -

Solution 2 :
0 * - - - - - - -
6 - - - - - - * -
3 - - - * - - - -
5 - - - - - * - -
7 - - - - - - - *
1 - * - - - - - -
4 - - - - * - - -
2 - - * - - - - -

Solution 3 :
0 * - - - - - - -
6 - - - - - - * -
4 - - - - * - - -
7 - - - - - - - *
1 - * - - - - - -
3 - - - * - - - -
5 - - - - - * - -
2 - - * - - - - -
```

……(略)

```
Solution 91 :
1 - * - - - - - -
6 - - - - - - * -
2 - - * - - - - -
5 - - - - - * - -
7 - - - - - - - *
4 - - - - * - - -
```

```
0 * - - - - - - -
3 - - - * - - - -

Solution 92 :
1 - * - - - - - -
6 - - - - - - * -
4 - - - - * - - -
7 - - - - - - - *
0 * - - - - - - -
3 - - - * - - - -
5 - - - - - * - -
2 - - * - - - - -
```

说明：若指定进程数为 1，则使用串行算法 sequential_eight_queens()求解。

5 计算几何

计算几何是计算机科学中的一个分支，是专门研究有关几何对象问题的。它在图像分析、模式识别、计算机图形学等中应用甚广。本章主要介绍几个基本计算几何问题的简单并行算法和它们的 MPI 编程实现，包括包含问题、相交问题和凸壳问题等。

1.1 包 含 问 题

包含问题(Inclusion Problem)是计算几何中最基本的问题之一。简单的来说，包含问题就是判断点与多边形的位置关系，即点在多边形内，点在多边形外，和点在多边形上。我们这里讨论的仅是点与任意多边形之间的关系，不考虑与任意曲线封闭图形之间的关系。

一个能够画在平面图上而没有任何相交边的图形称为**平面图**（Planar Graph）。由一些相邻的多边形（称为**单图**）所组成的图形称为**平面细图**（Planar Subvision），其中各多边形中除了顶点外，无两边相交。给定一个平面细图和一个顶点 p，确定哪个多边形包含了 p，此即所谓的包含问题。最简单的情况是判断点 p 是否在一个多边形 Q 中。

1.1.1 包含问题及其串行算法

判断点在多边形中的算法的基本思想是先求点和边的交点个数，然后根据交点个数确定点和边的关系。基本步骤是：①过 p 点向 x 轴的负半轴方向做一条射线；②求此射线与多边型 Q 诸边的交点；③判断：交点的个数若为奇数，则 p 位于 Q 内；否则 p 位于 Q 外。这种测试，对于 n 边形可在 $O(n)$步内完成。很清楚，这是最优的串行算法，因为读入 n 条边也至少需要 $\Omega(n)$步。

但是需要注意几种特殊的边界情况：首先，过点 p 的射线与多边形顶点相交，则交点只计数一次；其次，若点 p 在多边形边上，则也认为点 p 在多边形中；最后，如果射线与水平边重叠且点 p 不在该边上，则交点不计数。

算法 17.1　单处理机上包含问题算法

输入：点p坐标(p_x,p_y)；多边形Q的n条边$E_1,E_2,,\cdots,E_n$的两个端点坐标集合S

输出：点和多边形的关系：true(p 位于 Q 内)；false(p 位于 Q 外)

```
Begin
(1)  count=0
(2)  while i<n do
         if p is on E_i then
             return true
         else if y=p_y(x≤p_x) intersects E_i left q and q is not E_i low-end then
                 count=count+1
             end if
         end if
     end while
(3)  if ((count mod 2)=1) then
         return true
     else
```

return false

end if

End

算法的总运行时间为 $t(n)=O(n)$。

1.1.2 包含问题并行算法

我们介绍两种简单方法来实现上面串行算法的并行化。一种方法是将串行算法中的步骤 2 并行化。假设系统有 p 个处理器，多边形有 n 条边，将 n 条边平均分配到这 p 个处理器中，每个处理器最多处理$\lceil n/p \rceil$条边。具体的处理方法为过 p 点向 x 的负半轴方向做一条射线，判断：如果点是在边上则返回为∞；线与边无交点时返回 0；如果有交点，那么就有 2 种情况，如果交点是边的下顶点，则返回为 0，否则，即交点在边上，则返回为 1；把一个处理器中的所有返回值加起来，然后将该值发送到主处理器(my_rank=0)，最后主处理器根据点的个数来判断点与多边形的关系。

另一种方法同样也是将串行算法中的步骤 2 并行化。假设有$p=2^{\alpha}-1$ 个处理器，α 为正整数。p个处理器的编号从根开始自上而下，自左而右逐级向下推进。每个处理器存储多边形Q的一条边，边由其两个端点的迪卡尔坐标表示，点p的坐标为(x_p, y_p)。开始时，根读进(x_p, y_p)，然后传送给树中的其余处理器。当P_j接收到p的坐标时，他就确定：穿过p的射线是否和Q的边e_j相交；对于特殊的情况需要利用图形学的知识处理之。如果条件满足，则P_j产生“1”输出；否则为“0”。将个处理器的输出相加，若和为奇数，则p位于Q内；如果为偶数，则p位于Q外；如果和为∞，则输出p在多边形上。

算法 17.2　多处理机上包含问题算法

输入：点p坐标(p_x,p_y)；多边形Q的n条边$E_1,E_2,,\cdots,E_n$的两个端点坐标集合S

输出：点和多边形的位置关系：true(p 位于 Q 内)；false(p 位于 Q 外)

Begin

(1) count=0

(2) **for** all P_j **where** $1 \leqslant j \leqslant p$ **par-do**

　　$result_j=0$

　end for

(3) **for** all P_j **where** $1 \leqslant j \leqslant p$ **par-do**

　　for i=1 **to** $\lceil n/p \rceil$ **do**

　　if p is on $E_{i\times p+j}$ **then**

　　return $result_j=\infty$

　　else

　　if $y=p_y(x \leqslant p_x)$ intersects $E_{i\times p+j}$ left q and q is not $E_{i\times p+j}$ low-end **then**

　　$result_j= result_j+1$

　　end if

　　end if

　　end for

end for

(4) **for** j=1 **to** p **par-do** P_j

sends $result_j$ to P_0

end for

(5) **for** j=1 **to** p **do**

count= count+ $result_j$

end for

(6) **if** ((count mod 2)=1) or (count=∞) **then**

return true

else return false

end if

End

在有 p 个处理器的情况下，上述算法的时间复杂度为 $O(\lceil n/p \rceil)$。

MPI 源程序请参见所附光盘。

1.2 相交问题

在很多实际应用中，要求确定一组几何物体(目标)是否相交。例如，在模式分类中，必须确定代表不同类别的空间中的不同区域是否具有共同的子区域；在集成电路设计中，重要的是要避免导线的交叉和元件的重叠；在计算机图形学中，要求消去三维景象的二维表示中的隐线和隐面等等。像如上的这些问题都可归结为物体的**相交问题**(Intersection Problem)。

设有平面上的两个多边形(允许有边相交)*R*和*Q*，如果多边形*R*的一条边和*Q*的一条边相交，则称*R*和*Q*是相交的。所以两个多边形的相交问题可以转化为线段与多边形的相交问题。三维空间的相交问题与二维平面上的相交问题并没有实质的区别，只是在判断边的相交时比二维问题上判断边的相交要麻烦，因为三维空间上的点坐标是与3个值有关的。

下面描述的算法都是针对二维平面上的多边形相交而言的。

1.2.1 两多边形相交问题及其串行算法

最基本的相交问题是判断两条线段是否相交。而边与多边形相交就是判断一条边和多条边中的某条边是否相交的算法。要是一个多边形的某条边与另一个多边形的一条边相交，则就称两个多边形相交。这样两个多边形相交的问题就转化为多条边与一个多边形相交的问题。

判断线段是否相交的关键是求两条直线的交点，即判断此交点是否在线段上。这和包含问题有些相似，需要判断点与边的关系。不同点是只须判断点是否在边上。

算法 17.3　单处理机上的两多边形相交问题算法

输入：多边形*R*的*n*条边$E_1,E_2,,\ldots,E_n$的两个端点坐标集合S_1，多边形*Q*的*m*条边$F_1,F_2,,\ldots,F_m$的两个端点坐标集合S_2

输出：两个多边形是否相交：true(两多边形相交)；false(两多边形不相交)

Begin

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

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

if (E_i intersects F_j) **then**

return true
end if
end for
end for
return false
End

显然上述算法所需时间为 $O(mn)$。

1.2.2 相交问题的并行算法

下面我们给出两多边形相交问题的朴素并行算法：对于多边形R的每一条边，要确定其是否与多边形Q相交；如果R的边中有一条边与Q相交，那么就可以断定多边形R与Q是相交的。假设R有n条边，Q有m条边，总共有p个处理器P_1 ,P_2 ,…,P_p。对于R中的每条边依次判断是否与Q相交。

算法 17.4　两多边形相交问题的并行算法

输入：多边形R的n条边E_1,E_2,,…,E_n的两个端点坐标集合S_1，多边形Q的m条边F_1,F_2,,…,F_m的两个端点坐标集合S_2

输出：两个多边形是否相交：true(两多边形相交)；false(两多边形不相交)

Begin

(1) **for** i=1 **to** m **do**
　　将F_i广播到所有处理器上
　end for

(2) **for** j=1 **to** m **do**
　　将E_j广播到所有处理器上
　end for

(3) **for** all P_k **where** $1 \leqslant k \leqslant p$ **par-do**
　　for i=1 **to** $\left\lceil n/p \right\rceil$ **do**
　　for j=1 **to** m **do**
　　if ($E_{i\times p+k}$ intersects F_j) **then**
　　$result_k$=true
　　end if
　　end for
　　end for
　end for

(4) 将各个处理器上$result_k$返回到主处理器，如果其中有一个为真，则两多边形相交，否则两多边形不相交

End

MPI 源程序请参见章末附录。

1.3 凸壳问题

凸壳（Convex Hull）问题是计算几何中一个重要问题，它的应用很广泛。例如，在图象处理中可以通过构造凸壳找到数字图象中的凹面；在模式识别中，可视模式的凸壳能够作

为描述模式外形的重要特征；在分类中，一组物体的凸壳就可勾画出这些物体的所属的类；在计算机图形学中，使用一组点的凸壳可以显示出点簇；在几何问题中，集合 S 中最远两点就是凸壳的顶点，等等。

1.3.1 凸壳问题及其串行算法

给定平面中的点的集合 $S=\{p_1,p_2,...,p_n\}$，所谓 *S* 之凸壳简记为 *CH(S)*，就是包含 S 所有点的最小的凸多边形。实际中，假定平面上的 n 个点，用 n 个钉在木板上的图钉表示，用一条橡皮带缠绕着这些图钉，然后放松橡皮带，在撑紧橡皮带所构成的平面图形就是凸多边形。

图 17.1 图示求凸壳的方法

参照图 17.1，串行算法的基本思路如下：①**识别极点**（Extreme Point）：S 中那些最大 X 坐标(XMAX)，最大 Y 坐标(YMAX)，最小 X 坐标(XMIN)，最小 Y 坐标(YMIN)的那些顶点称为极点；②**识别凸边**(Hull Edge)：线段 (p_i,p_j) 是 CH(S)的一条凸边，当且仅当 S 的其余 n－2 个顶点均位于穿过 p_i 和 p_j 的(无限长的)直线的同侧。由此定义可知，p_i 和 p_j 一定是 CH(S)的顶点；③**识别凸点**(Hull Point)：令 p_i 和 p_j 是 CH(S)上的两连续顶点。假定取 p_i 为坐标原点，那么在所有 S 中的点，p_j 和 p_i 相对于 X 轴所形成的正的或负的夹角是最小。这些点称为凸点。

很明显，极点都是 *CH(S)*的顶点，任何位于由极点所围成的多边形内的点都不是 *CH(S)* 的顶点，那些在多边形外的点可以归入由 XMAX、XMIN、YMAX、YMIN 所围成的四边形与由极点所围成的多边形所形成的四个三角区中。这样问题就归结成为求找这四个三角区中顶点的凸边，再把这些凸边连接起来就可求得 *CH(S)*。

算法 17.5　单处理机上求凸壳的串行算法

输入：n 点集合 $S=\{p_1,..,p_n\}$

输出：返回包含 S 的凸壳的顶点表列 CH(S)

Begin

(1)　识别极点，它们都是 *CH(S)*的顶点

(2)　将顶点归入有极点确定的四个三角区中，不在四个三角区中的顶点不需要处理

(3)　计算顶点的极角，并排序：

(3.1)依次对属于同一三角区域的点计算极角。然后将有最小极角点的序号作为自己

的 Nextindex 值

(3.2)然后处理器按照点的 Nextindex 索引输出点

End

1.3.2 凸壳问题并行算法

假定 n 个处理器排成网孔形状，处于第 *i* 行和第 *j* 列的处理器用 *P*(*i*,*j*)表示。点 *i* 的坐标用 (x_i, y_i) 表示。首先来介绍两个基本概念和术语：①**拓扑结构的转化**：给定的 n 个处理器，可以认为其是直线拓扑结构。现在将一维拓扑结构改变为 2 维的，则 P(i,j)的 i=1,2,3,4，而 j=1,2,…,n/4。②**行主处理器**：行主处理器是一个人为设定的处理器，纯粹是为了处理上的方便。在本算法中取 p(i,1)为行主处理器。

算法 17.6　凸壳问题并行算法

输入：n 点集合 $S = \{p_1, ..., p_n\}$

输出：返回凸壳顶点表列 CH(S)

Begin

(1) 计算极点：

(1.1)第 1 行的行主处理器向第 2, 3, 4 行的行主处理器发送顶点坐标

(1.2)第 1、2、3、4 行的行主处理器分别计算 XMAX、 YMIN、 YMAX、 XMIN；并把它们的坐标分别存储在 4 个行主处理器 P(1,1)、P(2,1)、P(3,1)、P(4,1)中

(1.3)确定极点后，将四条由极点组成的边存储到每一行的行主处理器上

(2) 确定 *S* 中的顶点是否在四个三角区中：

(2.1)四个行主处理器同时判断顶点是否处于自身所在的区域，其中属于自己区域内的点，存储到行主处理器中本行要处理的顶点表列

(2.2)将行主处理器上的表列传递到本行中其余的处理器上

(3) 计算顶点的极角，并排序输出：

(3.1)每一行上的第 i 个处理器计算表列中第 j 个点极角，其中 j 是 mod 行处理器数等于 i 的数。然后将有最小极角点的序号作为自己的 Nextindex 值

(3.2)处理器将计算的 Nextindex 值传递到每行中的主处理器

(3.3)然后每个行主处理器按照从极点开始按 Nextindex 索引依次输出所得到的极点上的点

End

MPI 源程序请参见所附光盘。

1.4 小结

本章主要介绍了计算几何中包含问题、相交问题和凸壳等三个基本问题的并行算法及其 MPI 编程及其实现。这些算法可参见文献[1]。读者如欲进一步学习可参考文献[2]、[3]和[4]。

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]. Akl S G. Parallel Computational Geometry. Prentic-Hall, 1992
[3]. Atallah M J. Goodrich M T. Efficient Parallel Solutions to Some Geometric Problems. J. of Parallel and Distributed Computing, 1986, 3: 492-507
[4]. 周培德 著. 计算几何-算法设计与分析. 清华大学出版社,广西科学技术出版社, 2000.5

附录 包含问题并行算法的 MPI 源程序

1. 源程序 including.c

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

int n;
double xtemp[20],ytemp[20];
double x,y;
int s,mys;
int group_size,my_rank;

/*判断射线与线段是否有交点*/
int cal_inter(int number,int i,double x,double y)
{
        double x1,y1,x2,y2,temp;
        int result;
        result=0;

        if(number+i*group_size>=n)
                return result;

        x1=xtemp[number+i*group_size];
        y1=ytemp[number+i*group_size];
        x2=xtemp[(number+1+i*group_size)%n];
        y2=ytemp[(number+1+i*group_size)%n];

        if(y1>y2)
        {
                temp=x1;
                x1=x2;
                x2=temp;
                temp=y1;
                y1=y2;
                y2=temp;
        }

      /*判断竖直边的情况*/
      if(x1==x2)
      {
            if((x>x1)&&(y<=y2)&&(y>y1))
                  result=1;
            else
                  result=0;
            /*点在竖直边上，应该对 result 赋一个比
                  较的大的值，这里是 100*/
            if((x==x1)&&((y-y1)*(y2-y)>=0))
                  result=100;
      }
      else
      {
          /*非竖直边，非水平边*/
          if (y1!=y2)
          {
                temp=x2+(y-y2)*(x2-x1)/(y2-y1);
                /*交点刚好在边上，且不为下顶
                      点*/
                if((temp<x)&&(y<=y2)&&(y>y1))
                      result=1;
                else
                      result=0;

                /*点在边上，应该对 result 赋一个
                      比较的大的值，这里是 100*/
                if((temp==x)&&((y-y2)*
                      (y1-y)>=0))
                      result=100;
           }
           else
           {
                /*点在水平边上，应该对 result 赋
                      一个比较的大的值，这里是
                      100*/
```

```
                if((y==y1)&&((x1-x)*(x-x2)>=0))
                    result=100;
            }
        }
        return result;
}

main(int argc,char* argv[])
{
        int i;

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

        /*各处理器计数器初始化，对应于
            算法 17.2 步骤(2)*/
        mys=0;

        /*主处理器读入多边形顶点和要判断的点
            的坐标*/
        if(my_rank==0)
        {
            printf("请输入点的个数:");
            scanf("%d",&n);
            printf("请输入各点的坐标\n");
            for(i=0;i<n;i++)
            {
                printf("%d:",i);
                scanf("%lf",&xtemp[i]);
                scanf("%lf",&ytemp[i]);
            }
            printf("请输入要判断点的坐标\n");
            scanf("%lf %lf",&x,&y);
        }

        MPI_Barrier(MPI_COMM_WORLD);

        /*把多边形的顶点数、顶点坐标与要判别的点
            的坐标播送给所有进程*/
        MPI_Bcast(&n,1,MPI_INT,0,
            MPI_COMM_WORLD);
        MPI_Bcast(&x,1,MPI_DOUBLE,0,
            MPI_COMM_WORLD);
        MPI_Bcast(&y,1,MPI_DOUBLE,0,
            MPI_COMM_WORLD);
        MPI_Bcast(xtemp,n,MPI_DOUBLE,0,
            MPI_COMM_WORLD);
        MPI_Bcast(ytemp,n,MPI_DOUBLE,0,
            MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORLD);

        /*每一个处理器处理 n/group_size 条边上的情
            况并求和，对应于算法 17.2 步骤(3)*/
        for(i=0;i<n/group_size+1;i++)
        {
            mys+=cal_inter(my_rank,i,x,y);
        }

        MPI_Barrier(MPI_COMM_WORLD);

        /*把 mys 的值规约到 s，对应于
            算法 17.2 步骤(4)和(5)*/
        MPI_Reduce(&mys,&s,1,MPI_INT,MPI_SUM,
            0,MPI_COMM_WORLD);

        /*根据 s 值确定输出结果，对应于
            算法 17.2 步骤(6)*/
        if(my_rank==0)
        {
            if(s>=100)
                printf("vertex p is in polygon\n");
            else
                if(s%2==1)
                    printf("vertex p is in
                        polygon\n");
                else
                    printf("vertex p is out of
                        polygon\n");
        }
        MPI_Finalize();
}
```

2. 运行实例

编译：mpicc including.c –o including

运行：可以使用命令 mpirun –np SIZE including 来运行该串匹配程序，其中 SIZE 是所使用的处理器个数。本实例中使用了 SIZE=4 个处理器。

mpirun –np 4 including

运行结果：

请输入点的个数:6

请输入各点的坐标

0：1 0

1：2 1

2：4 0

3：2 4

4：1 2

5：0 3

请输入要判断点的坐标

2 3

结果：vertex p is in polygon

说明：输入顶点的(x,y)坐标时中间用空格隔开。

6 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
```

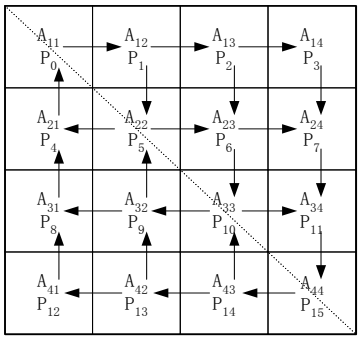


图 18.1　子块转置

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 18.1 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$n\times v$。

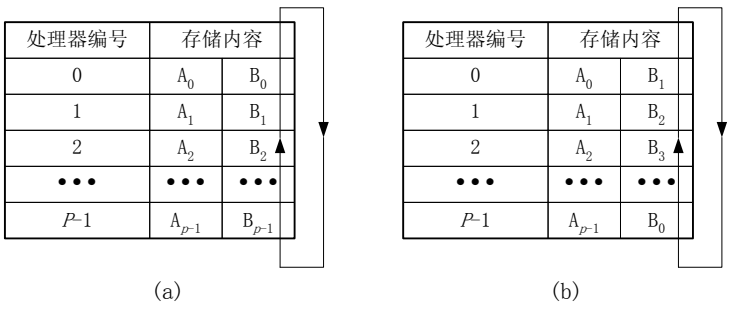


图 18.2 矩阵相乘并行算法中的数据交换

开始，各处理器的存储内容如图 18.2 (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 18.2 (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 18.3 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
```

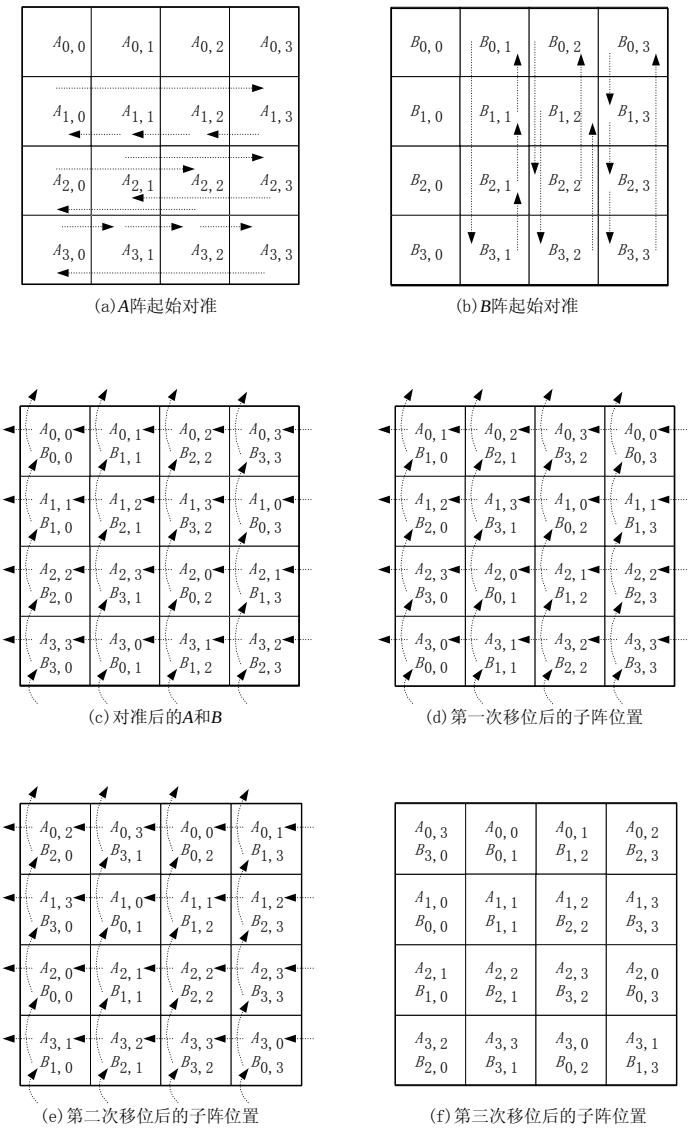


图 18.3　16 个处理器上的 Cannon 乘法过程

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：

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

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

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 18.4 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。

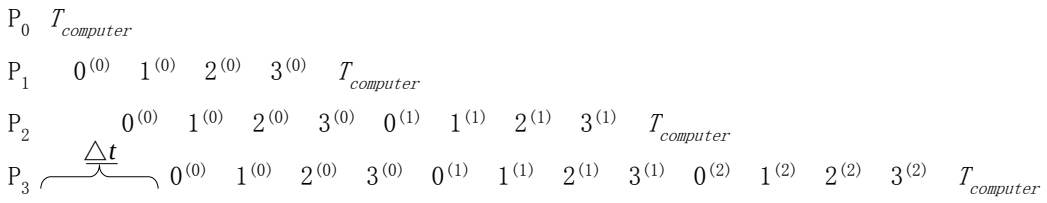


图 18.4　QR 分解并行计算的时间分布图

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：

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

即

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

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$\theta$=0；否则记 $l = \dfrac{b-d}{2e}, t = \dfrac{sign(l)}{|l| + \sqrt{1+l^2}}, \cos\theta = \dfrac{1}{\sqrt{1+t^2}}, \sin\theta = t\cos\theta$ ，由此可求得θ，满足$|\theta| \le \dfrac{\pi}{4}$。

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 $A=\begin{pmatrix} a_{11} & \vdots & \alpha \\ \cdots & \cdots & \cdots \\ \alpha^T & \vdots & A_1 \end{pmatrix}=\begin{pmatrix} g_{11} & \vdots & 0 \\ \cdots & \cdots & \cdots \\ \varphi^T & \vdots & G_1^T \end{pmatrix}\begin{pmatrix} g_{11} & \vdots & \varphi \\ \cdots & \cdots & \cdots \\ 0 & \vdots & G_1 \end{pmatrix}=G^TG$ 则有：

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

令$G^{'}=G, A'=A_1-\dfrac{\alpha^T\alpha}{a_{11}}$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：

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

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)$a_{ij}=a_{ij}^{'}(i,j=1,\cdots,n)$

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。

7 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。

1.1 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 $a_{k+1k},a_{k+2k},\cdots,a_{nk}$。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 $x_i=b_i-\sum_{j=i+1}^{n}a_{ij}x_j,\quad i=n-1,\cdots,2,1$

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。

1.2 约当消去法解线性方程组

1.2.1 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

1.2.2 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。

1.3 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。

8 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：

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

1.1 雅可比迭代

1.1.1 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

1.1.2 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)$x1[i]=(b[i]-sum)/a[i,my\_rank*m+i]$

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 $localmax$ ***/**

$localmax=|x1[0]-x[0]|$

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

if ($|x1[i]-x[i]|>localmax$) **then**

$localmax=|x1[i]-x[i]|$

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。

1.2 高斯-塞德尔迭代

1.2.1 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：

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

高斯-塞德尔迭代公式为：

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

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)$p=\varepsilon+1$

(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

1.2.2 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* ≠ *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。

1.3 松弛法

1.3.1 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：

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

然后计算第 k+1 次迭代值与第 k 次迭代值之差；即：

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

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：

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

其中 w 为一个常数。综合以上过程，可以得到如下迭代格式：

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

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，$\overline{x}_i^{(k+1)}$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

1.3.2 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。

1.4 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。

9 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。

1.1 求解矩阵最大特征值的乘幂法

1.1.1 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, …,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, …,做如下迭代：

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

直至$\left\|u_{k+1}\right\|_\infty-\left\|u_k\right\|_\infty<\varepsilon$为止，这时$v_{k+1}$就是$A$的绝对值最大的特征值$\lambda_1$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

1.1.2 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。

1.2 求对称矩阵特征值的雅可比法

1.2.1 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

$$D=\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

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：

$$tg2\theta = \frac{-a_{pq}}{(a_{qq} - a_{pp})/2}$$

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：

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

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$2a^2{}_{pq}$，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* ≠ *p*) and (*j* ≠ *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**

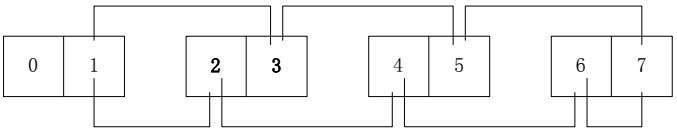
```
                if((i ≠ p) and (i ≠ q)) then
                    (i)b[i, p]= a[i,p]*co1+ a[i,q]*si1
                    (ii)b[i, q]= - a[i,p]*si1+ a[i,q]*co1
                end if
            end for
        (1.6)for i=1 to n do
                for j=1 to n do
                    a[i,j]=b[i,j]
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

1.2.2 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，…,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。



开　始 ： （0，1） （2，3） （4，5） （6，7）
第一步 ： （0，3） （1，5） （2，7） （4，6）
第二步 ： （0，5） （3，7） （1，6） （2，4）
第三步 ： （0，7） （5，6） （3，4） （1，2）
第四步 ： （0，6） （7，4） （5，2） （3，1）
第五步 ： （0，4） （6，2） （7，1） （5，3）
第六步 ： （0，2） （4，1） （6，3） （7，5）
第七步 ： （0，1） （2，3） （4，5） （6，7）

图 21.1 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 21.1 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 2*p*-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*] ≠ 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**sin*1)，

bpp= *a*[*r,i*]**cos*1**cos*1+*a*[*t,j*]**sin*1**sin*1+*a*[*r,j*]**sin*2，

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

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* ≠ *i*) and (*v* ≠ *j*)) **then**

br[*v*] = *a*[*r,v*]**cos*1 + *a*[*t,v*]**sin*1

a[*t,v*] = -*a*[*r,v*]* *sin*1 + *a*[*t,v*]* *cos*1

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**sin*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$n(n\text{-}1)/(2p)$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。

1.3 求对称矩阵特征值的单侧旋转法

1.3.1 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：$\delta_i = \lambda_i^2$ (i=1,2, …,n)。设Q的第i列为q_i, 则 $q_i^T q_i = \delta_i$，$\|q_i\|_2 = \sqrt{q_i^T q_i} = \sqrt{\delta_i} = |\lambda_i|$。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$n(n-1)/2*(12\ n+6n)= 9n(n-1)\ n=O(n^3)$。

1.3.2 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。

1.4 求一般矩阵全部特征值的 QR 方法

1.4.1 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：

$$\begin{pmatrix} \lambda_1 & * & * & * & \cdots & * \\ & \lambda_2 & * & * & \cdots & * \\ & & \lambda_3 & * & \cdots & * \\ & & & \cdots & \cdots & \cdots \\ & & & & & \lambda_n \end{pmatrix}$$

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。

1.4.2 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。

1.5 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,陈崚 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。

10 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]。

1.1 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]。

1.1.1 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 $W_n=e^{\frac{2\pi i}{n}}$，实际上，上式可写成矩阵 W 和向量 a 的乘积形式：

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} w^0 & w^0 & w^0 & \cdots & w^0 \\ w^0 & w^1 & w^2 & \cdots & w^{n-1} \\ w^0 & w^2 & w^4 & \cdots & w^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w^0 & w^{(n-1)} & w^{2(n-1)} & \cdots & w^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

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 22.1 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

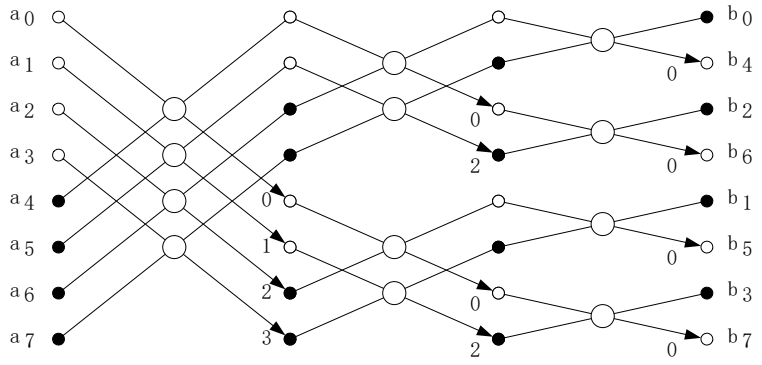


图 22.1 n=4 时的 FFT 蝶式变换图

显然, FFT 算法的计算复杂度为 O($n$log$n$)。

1.1.2 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)×m-1，其元素被分配至第 i 号处理器（i=0,1, …, p-1）。由图 22.1 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 logn-1 步至第 logm 步各处理器之间需要通信*/
(1.1) t=2^i, ,l=2^(i+logm)  ,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^(my_rank*m+k) mod 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=logm-1 downto 0 do  /*第二阶段，第 logm-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^(my_rank*m+k) mod  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。

1.2 离散小波变换 DWT

1.2.1 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 $\phi_{jk}(x) = 2^{-j/2}\phi(2^{-j}x - k)$，$\psi_{jk}(x) = 2^{-j/2}\psi(2^{-j}x - k)$，这里 $\phi(x)$ 与 $\psi(x)$ 分别称为定标函数与子波函数，$\{\phi_{jk}(x)\}$ 与

$\{\psi_{jk}(x)\}$ 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：

$$P_{j-1}f = P_j f + Q_j f = \sum_k c_k^j \phi_{jk} + \sum_k d_k^j \psi_{jk}$$

其中：$c_k^j = \sum_{n=0}^{p-1} h(n) c_{2k+n}^{j-1}$，$d_k^j = \sum_{n=0}^{p-1} g(n) c_{2k+n}^{j-1}$ $(j = 1,2,...,L, k = 0,1,...,N/2^j - 1)$，这里，$\{h(n)\}$ 与$\{g(n)\}$分别为低通与高通权系数，它们由基函数$\{\phi_{jk}(x)\}$与$\{\psi_{jk}(x)\}$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： $c^0=d^0(c_0^0, c_1^0,..., c_{N-1}^0)$

$h=(h_0, h_1,..., h_{L-1})$

$g=(g_0, g_1,..., g_{L-1})$

输出： c_i^j, d_i^j $(i=0, 1,..., N/2^{j-1}, j \geqslant 0)$

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：$c_n^0 = x_n, n = 1,2,\cdots,M$ (其余点的值都看成 0)，它的离散小波变换为:

$$c_k^{j+1} = \sum_{n \in Z} c_n^j h_{n-2k}$$

$$d_k^{j+1} = \sum_{n \in Z} c_n^j g_{n-2k}$$

$$j = 0,1,\cdots,J-1$$

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

$$c_n^{j-1} = \sum_{k\in Z} c_k^j h_{n-2k} + \sum_{k\in Z} c_k^j h_{n-2k}$$

$$j = J,\cdots,1$$

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 $d_k^j, j = 0,1,\cdots,J-1$ 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：$k = -\frac{N}{2}+1,\cdots,-1,0,\cdots,\left\lceil\frac{M}{2}\right\rceil-1$，即有 $k = -\frac{N}{2}-1+\left\lceil\frac{M}{2}\right\rceil = \left\lfloor\frac{M+N-1}{2}\right\rfloor$ 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。

1.2.2 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 $c_n^0 = x_n, (n = 0,1,\cdots,N-1)$ 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：

$$c_k^{j+1} = \sum_{n\in Z} c_n^j h_{n-2k} = \sum_{n=0}^{L-1} h_n c^{j}{}_{\langle n+2k \rangle}$$

$$d_k^{j+1} = \sum_{n=0}^{L-1} g_n d^{\,j}_{\langle n+2k \rangle}$$

$$j = 0,1,\cdots, J-1$$

其中<*n*+2*k*>表示 *n*+2*k* 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

$$c_n^{j-1} = \sum_{k\in Z} c_k^j h_{n-2k} + \sum_{k\in Z} c_k^j g_{n-2k}$$

$$j = J,\cdots,1$$

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*=2*k* 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 $c_n^j(n = 0,1,\cdots, N/2^j - 1)$ 按块分配给 *p* 台处理器，处理器 *i* 得到数据 $c_n^j(n = i\frac{N}{P2^j},\cdots,(i+1)\frac{N}{P2^j}-1)$，让处理器 *i* 负责 c_n^{j+1} 和 $d_n^{j+1}(n = i\frac{N}{P2^{j+1}},\cdots,(i+1)\frac{N}{P2^{j+1}}-1)$ 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 $c_n^j(n = 0,1,\cdots, N/2^j - 1)$ 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 $d_n^{j+1}(n = i\frac{N}{p2^{j+1}},\cdots,(i+1)\frac{N}{p2^{j+1}}-1)$ 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。

1.3 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。