并行算法试题答题卷

林宇健 2018202296

2019 年 2 月 10 日

1 并行矩阵向量乘法

1.1 问题描述和算法分析

编程计算Ax，其中A是$m \times n$的稠密矩阵，x是n维列向量，分别采用$1, 4, 8, 16$台处理机计算。给出并行算法，及并行效率分析。

记$y = Ax$。为简便起见，取A是$n \times n$的方阵，$n = 2048$可以整除$1, 4, 8, 16$，从而保证每个进程储存的向量块维度相同。对于不能整除$4, 8, 16$的n，可通过循环存储等方式为各个进程分配矩阵和向量的数据。在计算时编程随机生成了$matrix_{2048\times2048}$和$vector_{2048\times1}$作为待计算的矩阵和向量（计算程序略去）。

我们采用一维行划分的方式并行计算矩阵向量乘法。假设矩阵A按逐行一维块划分为p个块（p表示进程数），即$A = [A_1, A_2, \cdots, A_p]^T, A_k = [A_{k,0}, A_{k,1}, \cdots, A_{k,p}]$。其中

$$A_{k,j} = \begin{bmatrix} a_{k\times n/p+1,j\times n/p+1} & a_{k\times n/p+1,j\times n/p+2} & \cdots & a_{k\times n/p+1,j\times n/p+n/p} \\ a_{k\times n/p+2,j\times n/p+1} & a_{k\times n/p+2,j\times n/p+2} & \cdots & a_{k\times n/p+2,j\times n/p+n/p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k\times n/p+n/p,j\times n/p+1} & a_{k\times n/p+n/p,j\times n/p+2} & \cdots & a_{k\times n/p+n/p,j\times n/p+n/p} \end{bmatrix}$$

与之相对应，向量x和向量y也分为p个块，其第$k(0 \le k \le p-1)$个块分别为

$$x_k = \begin{bmatrix} x_{k\times n/p+1} & x_{k\times n/p+2} & \cdots & x_{k\times n/p+n/p} \end{bmatrix}^T$$

$$y_k = \begin{bmatrix} y_{k\times n/p+1} & y_{k\times n/p+2} & \cdots & y_{k\times n/p+n/p} \end{bmatrix}^T$$

假设A_k、x_k和y_k储存在进程k上，下图表示了对应的数据分布情况。

A				X	
$A_{1,1}$	$A_{1,2}$	$\cdots$	$A_{1,p}$	x_1	P_1
$A_{2,1}$	$A_{2,2}$	$\cdots$	$A_{2,p}$	x_2	P_2
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$A_{p,1}$	$A_{p,2}$	$\cdots$	$A_{p,p}$	x_p	P_p

在特定的进程k上，需要计算的是对应的y_k，而y_k的计算公式如下

$$y_k = A_k x = \sum_{j=0}^{p-1} A_{k.j} x_j = \sum_{j=0}^{p-1} A_{k,(k+j)\%p} x_{(k+j)\%p}$$

其中%表示取余运算。可以看出在计算y_k时给定的子矩阵A的行标号始终为k。在进行第j步的计算时，需要用到的x的子向量$x_{(k+j)\%p}$,所以可以通过每计算一次将x的块在同列进程中循环上移一个位置的方法实现，在计算到第j步的时候x的块正好循环上移了j次，即计算所用的x的子向量在当前进程上，从而实现并行的稠密矩阵向量乘法。具体的算法由如下的伪代码给出。

在MPI2.0及其以上的版本中，提供了各个进程并行访问读写文件的I/O函数，称为MPI并行I/O函数。通过在并行矩阵向量乘法中引入并行I/O，可以进一步提高并行度。其中主要用到的MPI并行I/O函数Fortran原型如下：

/* MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
INTEGER FH, WHENCE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

该函数的作用是从链接到FH的文件中指定起始位置WHENCE处读取指定偏移量OFFSET的数据。在编程调用的过程中，每个进程读取矩阵中的$n \times n/p$个数据（生成的矩阵中数据类型为REAL(4)）,则第j个进程的起始读取位置$\text{WHENCE}_j = (j-1) \times n \times n/p + 1$（Fortran语言默认的数组下标从1开始），偏移量$\text{OFFSET}_j = n \times n/p \times \text{SIZEOF(REAL(4))}$。同理可以并行读取分配给各进程的向量块。并行读取的方式不仅提高了并行度，也节约了各进程需要分配的内存空间，并且并行I/O时不需要主进程与从进程的设计，各个进程都是等价的，在计算并行程序运行时间时可以用各进程运行时间的平均较准确的表示并行程序的实际运行时间。

依据以上的算法，可以编写基于一维行划分的稠密矩阵向量并行乘法的程序。

1.2 算法的编程实现

首先随机生成待相乘的矩阵和向量，并储存在二进制文件*matrix*和*vector*中（程序省略）。串行程序直接调用Fortran内置的矩阵乘法函数**matmul**即可。

```
!*****************************************************************************
!
!  matrix_mul_vector_serial.f90
!  串行矩阵向量乘法
!
!*****************************************************************************

program serial_Mat_mul_Vec

    implicit none

    integer, parameter :: N = 2048
    integer :: i, j
    real(4) :: startwtime, endwtime
    real(4) :: matrix(N, N), vector(N, 1), answer(N, 1) = 0

    call cpu_time(startwtime)

    ! 读取矩阵
    open(10, file = 'matrix', access = 'direct', form = 'unformatted', recl = 4*N*N)
    read(10, rec = 1)  ((matrix(i,j), j = 1, N), i = 1, N)
    close(10)
    matrix = transpose(matrix)  ! Fortran的矩阵储存方式为列储存，需要
                                ! 进行一次转置

    ! 读取向量
    open(20, file = 'vector', access = 'direct', form = 'unformatted', recl = 4*N)
    read(20, rec = 1) (vector(i, 1), i = 1, N)
```

```
    close(20)

    answer = matmul(matrix, vector)

    ! 输出结果到向量文件
    open(30, file = 'answer', access = 'direct', form = 'unformatted', recl = 4)
    do i = 1, N
        write(30, rec = i) answer(i, 1)
    end do

    call cpu_time(endwtime)
    open(40, file = 'walltime', access = 'direct', form = 'unformatted', recl = 4)
    write(40, rec = 1) (endwtime - startwtime) * 1000
    close(40)

end program serial_Mat_mul_Vec
```

在实际计算时，出现了串行程序耗时比1进程并行程序多出三倍的情况，经分析产生该情况的原因是串行程序和并行程序用了不同的读取文件方式。为了得到实际的加速比，将串行程序中文件读取的部分修改为和并行程序相同的方式，即调用MPI的I/O函数。更改后的程序如下。

```
!*************************************************************************
!
!  matrix_mul_vector_serial_with_mpi.f90
!  串行矩阵向量乘法
!
!*************************************************************************

program serial_Mat_mul_Vec_mpiio

    use mpi
    implicit none

    integer, parameter :: N = 2048
    integer :: i, j
    integer :: IERR, NPROC, NSTATUS(MPI_STATUS_SIZE)
    integer :: myrank, myleft, myright, myfile, buf_size, cnt
    real(4) :: startwtime, endwtime, wtime
    real(4) :: matrix(N, N), vector(N, 1), answer(N, 1) = 0

    call cpu_time(startwtime)

    call mpi_init(IERR)
    call mpi_comm_rank(MPI_COMM_WORLD, myrank, IERR)
    call mpi_comm_size(MPI_COMM_WORLD, NPROC, IERR)

    ! 读取矩阵
    call mpi_file_open(MPI_COMM_WORLD, "matrix", MPI_MODE_RDONLY, MPI_INFO_NULL, &
    &  myfile, IERR)
    call mpi_file_seek(myfile, myrank*N*N*sizeof(MPI_REAL), MPI_SEEK_SET, &
    &  IERR)
    call mpi_file_read(myfile, matrix, N*N, MPI_REAL, NSTATUS, IERR)
    call mpi_file_close(myfile, IERR)
    matrix = transpose(matrix)

    ! 读取向量
    call mpi_file_open(MPI_COMM_WORLD, "vector", MPI_MODE_RDONLY, MPI_INFO_NULL, &
    &  myfile, IERR)
    call mpi_file_seek(myfile, myrank*N*sizeof(MPI_REAL), MPI_SEEK_SET, IERR)
    call mpi_file_read(myfile, vector, N, MPI_REAL, NSTATUS, IERR)
    call mpi_file_close(myfile, IERR)


    answer = matmul(matrix, vector)

    ! 结果向量输出到文件
    call mpi_file_open(MPI_COMM_WORLD, "answer", MPI_MODE_CREATE+MPI_MODE_WRONLY, &
    &  MPI_INFO_NULL, myfile, IERR)
    call mpi_file_seek(myfile, myrank*N*sizeof(MPI_REAL), MPI_SEEK_SET, IERR)
    call mpi_file_write(myfile, answer, N, MPI_REAL, &
    &  MPI_STATUS_IGNORE, IERR)
    call mpi_file_close(myfile, IERR)

    ! 运行时间记录到文件
    call cpu_time(endwtime)
    wtime = (endwtime - startwtime) * 1000
```

```
56      call mpi_file_open(MPI_COMM_WORLD, "walltime_mpiio", MPI_MODE_CREATE &
57     &  +MPI_MODE_WRONLY, MPI_INFO_NULL, myfile, IERR)
58      call mpi_file_seek(myfile, myrank*sizeof(MPI_REAL), MPI_SEEK_SET, IERR)
59      call mpi_file_write(myfile, wtime, 1, MPI_REAL, MPI_STATUS_IGNORE, IERR)
60      call mpi_file_close(myfile, IERR)
61
62      call mpi_finalize(IERR)
63
64  end program serial_Mat_mul_Vec_mpiio
```

该程序和原来串行程序的差别仅在于I/O不同，虽然声明了mpi但是并没有并行计算的部分，实际上仍旧是串行程序。

对于并行的矩阵向量程序，其核心在于向量块的向上传递（程序60-71行部分）。在编程时每次计算玩后对所有的向量块做一次上传，执行NPORC次（NPROC为进程数）。循环完成后，每个进程中都计算了全部的向量块，再通过全规约操作将计算结果发送到每个进程，最终并行输出。并行程序如下。

```
1   !*****************************************************************************
2   !
3   !  matrix_mul_vector_parallel.f90
4   !  并行矩阵向量乘法，基于行划分方法进行并行化
5   !
6   !*****************************************************************************
7
8   program parallel_Mat_mul_Vec
9
10      use mpi
11      implicit none
12
13      integer, parameter :: N = 2048
14      integer :: my_left, my_right
15      integer :: IERR, NPROC, NSTATUS(MPI_STATUS_SIZE)
16      integer :: myrank, myleft, myright, myfile, buf_size, cnt
17      real(4) :: startwtime, endwtime, wtime
18      real(4), allocatable :: matrix(:, :), vector(:, :), answer(:, :)
19      real(4), allocatable :: matrix_buf(:, :), vector_buf(:, :), answer_buf(:, :)
20      character(len = 2) :: sTemp
21
22      call cpu_time(startwtime)
23
24      call mpi_init(IERR)
25      call mpi_comm_rank(MPI_COMM_WORLD, myrank, IERR)
26      call mpi_comm_size(MPI_COMM_WORLD, NPROC, IERR)
27
28      buf_size = N / NPROC
29      myleft = my_left(myrank, NPROC)
30      myright = my_right(myrank, NPROC)
31
32      allocate(matrix_buf(N, buf_size)) !  Fortran的矩阵储存方式为列储存，需要
33                                        !  进行一次转置，因此设置读取缓存空间
34      allocate(vector_buf(buf_size,1))  !  接收其他进程储存的向量所需要的缓存空间
35      allocate(matrix(buf_size, N))
36      allocate(vector(buf_size, 1))
37      allocate(answer(N, 1))
38      allocate(answer_buf(N, 1))
39
40      !  读取矩阵
41      call mpi_file_open(MPI_COMM_WORLD, "matrix", MPI_MODE_RDONLY, MPI_INFO_NULL, &
42      &  myfile, IERR)
43      call mpi_file_seek(myfile, myrank*N*buf_size*sizeof(MPI_REAL), MPI_SEEK_SET, &
44      &  IERR)
45      call mpi_file_read(myfile, matrix_buf, N*buf_size, MPI_REAL, NSTATUS, IERR)
46      call mpi_file_close(myfile, IERR)
47      matrix = transpose(matrix_buf)
48
49      !  读取向量
50      call mpi_file_open(MPI_COMM_WORLD, "vector", MPI_MODE_RDONLY, MPI_INFO_NULL, &
51      &  myfile, IERR)
52      call mpi_file_seek(myfile, myrank*buf_size*sizeof(MPI_REAL), MPI_SEEK_SET, IERR)
53      call mpi_file_read(myfile, vector, buf_size, MPI_REAL, NSTATUS, IERR)
54      call mpi_file_close(myfile, IERR)
55
56      answer = 0
```

```
    deallocate(matrix_buf) ! 释放矩阵缓存空间用于储存每一次计算时的矩阵块
    allocate(matrix_buf(buf_size, buf_size))
    ! 循环进程中储存的所有矩阵块
    do cnt = 0, NPROC
        ! 计算对应矩阵块与向量的乘积
        matrix_buf = matrix(:, mod(myrank+cnt, NPROC)*buf_size+1:(mod(myrank+cnt, NPROC) &
        &  +1)*buf_size)
        answer(myrank*buf_size+1:(myrank+1)*buf_size, :) = matmul(matrix_buf,vector) &
        &  + answer(myrank*buf_size+1:(myrank+1)*buf_size, :)
        ! 进行一次向量块的传递(向上)
        call mpi_send(vector, buf_size, MPI_REAL, myleft, myrank, MPI_COMM_WORLD, IERR)
        call mpi_recv(vector_buf, buf_size, MPI_REAL, myright, myright, &
        &  MPI_COMM_WORLD, NSTATUS, IERR)
        vector = vector_buf
    end do

    ! 全规约结果向量, 并行输出到文件
    call mpi_allreduce(answer, answer_buf, N, MPI_REAL, MPI_SUM, MPI_COMM_WORLD, IERR)
    call mpi_file_open(MPI_COMM_WORLD, "answer", MPI_MODE_CREATE+MPI_MODE_WRONLY, &
    &  MPI_INFO_NULL, myfile, IERR)
    call mpi_file_seek(myfile, myrank*buf_size*sizeof(MPI_REAL), MPI_SEEK_SET, IERR)
    call mpi_file_write(myfile, answer_buf(myrank*buf_size+1, 1), buf_size, MPI_REAL, &
    &  MPI_STATUS_IGNORE, IERR)
    call mpi_file_close(myfile, IERR)

    ! 将各进程的运行时间记录到文件中
    call cpu_time(endwtime)
    wtime = (endwtime - startwtime) * 1000
    write(sTemp, '(i2)') NPROC
    call mpi_file_open(MPI_COMM_WORLD, "walltime"//trim(adjustl(sTemp)), MPI_MODE_CREATE &
    &  +MPI_MODE_WRONLY, MPI_INFO_NULL, myfile, IERR)
    call mpi_file_seek(myfile, myrank*sizeof(MPI_REAL), MPI_SEEK_SET, IERR)
    call mpi_file_write(myfile, wtime, 1, MPI_REAL, MPI_STATUS_IGNORE, IERR)
    call mpi_file_close(myfile, IERR)

    deallocate(matrix)
    deallocate(vector)
    deallocate(answer)
    deallocate(matrix_buf)
    deallocate(vector_buf)
    deallocate(answer_buf)
    call mpi_finalize(IERR)

end program parallel_Mat_mul_Vec


!-------------------子程序与函数部分-----------------------------------------
integer function my_left(myrank, nproc) result(ans)

    implicit none
    integer, intent(in) :: myrank, nproc

    ans = myrank - 1
    if (0 == myrank) ans = nproc - 1

end function my_left


integer function my_right(myrank, nproc) result(ans)

    implicit none
    integer, intent(in) :: myrank, nproc

    ans = myrank + 1
    if (nproc-1 == myrank) ans = 0

end function my_right
```

并行程序与串行程序运行完成后，程序walltime.for读取不同进程数并行程序的执行时间，并输出到终端。

```
C**********************************************************************
C
C     walltime.for
C     计算串行程序和并行程序的运行时间并显示
C
C**********************************************************************
```

```
7
8       PROGRAM WALLTIME_FOR
9
10      REAL*4 T, TM, T1, T4(4), T8(8), T16(16)
11
12      OPEN(8, FILE = 'walltime', ACCESS = 'DIRECT', FORM = 'UNFORMATTED
13     & ', RECL = 4)
14      READ(8, REC = 1) T
15      CLOSE(8)
16      PRINT *, "serial program walltime: ", T, "ms"
17
18      OPEN(8, FILE = 'walltime_mpiio', ACCESS = 'DIRECT',
19     & FORM = 'UNFORMATTED', RECL = 4)
20      READ(8, REC = 1) TM
21      CLOSE(8)
22      PRINT *, "serial program(MPI I/O) walltime: ", TM, "ms"
23
24      OPEN(8, FILE = 'walltime1', ACCESS = 'DIRECT', FORM = 'UNFORMATTED
25     & ', RECL = 4)
26      READ(8, REC = 1) T1
27      CLOSE(8)
28      PRINT *, "parallel program walltime (1 process): ", T1, "ms"
29
30      OPEN(8, FILE = 'walltime4', ACCESS = 'DIRECT', FORM = 'UNFORMATTED
31     & ', RECL = 4*4)
32      READ(8, REC = 1) T4
33      CLOSE(8)
34      SUM4 = .0
35      DO 40 I = 1, 4
36         SUM4 = SUM4 + T4(I)
37   40 CONTINUE
38      PRINT *, "parallel program walltime (4 process): ", SUM4/4, "ms"
39
40      OPEN(8, FILE = 'walltime8', ACCESS = 'DIRECT', FORM = 'UNFORMATTED
41     & ', RECL = 4*8)
42      READ(8, REC = 1) T8
43      CLOSE(8)
44      SUM8 = .0
45      DO 80 I = 1, 8
46         SUM8 = SUM8 + T8(I)
47   80 CONTINUE
48      PRINT *, "parallel program walltime (8 process): ", SUM8/8, "ms"
49
50      OPEN(8, FILE = 'walltime16', ACCESS = 'DIRECT', FORM =
51     & 'UNFORMATTED', RECL = 4*16)
52      READ(8, REC = 1) T16
53      CLOSE(8)
54      SUM16 = .0
55      DO 160 I = 1, 16
56         SUM16 = SUM16 + T16(I)
57  160 CONTINUE
58      PRINT *, "parallel program walltime (16 process): ", SUM16/16,
59     & "ms"
60
61      END PROGRAM WALLTIME_FOR
```

将以上全部程序整理为一个脚本文件如下，运行脚本文件即可得到结果。

```
1  #!/bin/bash
2  # nohup sh autoexec.sh  > /dev/null 2>&1 &
3
4  gfortran random_matrix.f90 -o matrix.out
5  gfortran random_vector.f90 -o vector.out
6  ./matrix.out
7  ./vector.out
8  mpif90 matrix_mul_vector_parallel.f90
9  mpirun -np 1 a.out
10 mpirun -np 4 a.out
11 mpirun -np 8 a.out
12 mpirun -np 16 a.out
13 gfortran matrix_mul_vector_serial.f90 -o b.out
14 ./b.out
15 mpif90 matrix_mul_vector_serial_with_mpi.f90 -o c.out
16 mpirun -np 1 c.out
17 gfortran walltime.for -o d.out
18 ./d.out
```

1.3 运行结果与分析

运行autoexec.sh，输出的结果如下：

```
[zww05@cow Q01]$ sh reset.sh
[zww05@cow Q01]$ sh autoexec.sh
 serial program walltime:    360.94400     ms
 serial program(MPI I/O) walltime:    86.986000     ms
 parallel program walltime (1 process):    91.985001     ms
 parallel program walltime (4 process):    27.995750     ms
 parallel program walltime (8 process):    21.122000     ms
 parallel program walltime (16 process):    26.683750     ms
```

在程序执行的过程中，初始的串行程序和并行程序采用了不同的I/O方式，从而导致串行程序比1进程并行程序用时多出3倍的问题。通过将串行程序的I/O从Fortran自带的I/O更改为MPI的I/O之后，程序的计算结果符合估计。

从程序执行结果可以看出，当进程数为1时，并行程序慢于串行程序；随着进程数的增加，程序运行时间并没有越来越短，而是在减少到一定程度后反而随着进程数的增加而变长。产生该现象的原因是单个进程并行时计算量与串行相差不多，并行程序比串行程序多出了进程通信的开销；进程数过多时由于计算的规模并不是很大，通信开销的增长快于计算量的下降。可以估计，当矩阵A的维度增大时，采用16进程计算的并行程序用时会短于8进程的并行程序。

根据程序输出结果，计算并行度和效率并列成如下表格：

进程数 / 运行结果	–(串行程序)	–(串行程序+MPI I/O)	1	4	8	16
运行时间(ms)	360.94	86.99	91.99	28.00	21.12	26.68
加速比	–	–	0.95	3.11	4.12	3.26
效率(%)	–	–	94.56	77.67	51.24	20.69

从表格中的结果也可以看出，随着进程数的增加，并行效率越来越低，主要是由于通信开销增大导致。由于机器的限制，并未能成功测试10240×10240的矩阵向量乘法，可以推测计算较大的矩阵向量相乘时效率和加速比都会提高。

2 并行五点差分法

2.1 问题描述与算法编程实现

已知Possion方程

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = xy, \quad (x,y) \in (0,1) \times (0,1) \\ u(0,y) = y, u(1,y) = 1-y, u(x,0) = 0, u(x,1) = 0 \end{cases} \tag{1}$$

采用五点差分法将方程离散为线性方程组，其中步长为$\frac{1}{50}$,写成矩阵运算的形式；采用红黑迭代法并行求解，给出并行算法的流程图，在并行机上编程实现，且针对处理机台数为$1,2,4,6,8$的情况下，给出并行率，讨论算法的可扩发性；提出改进的方法。

五点差分法在求解区域的网格节点上用其相邻四个节点上解的差商近似代替偏导数的离散方法，对于给定的Possion方程，五点差分格式为

$$\frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{h^2}+\frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{h^2}=h^2ij$$

其中$h=1/N=1/50$。将其写为矩阵运算$Au=F$，u表示各个节点的近似解。计算时需要将待求解的二维节点区域用一个向量表示，参考C语言中二维数组的表示方法，用向量u的$i+j\times(N+1)$个元素表示节点(i,j)处的近似解。根据五点差分的格式，(i,j)对应的A中的系数为-4，$(i+1,j),(i-1,j),(i,j+1),(i,j-1)$对应的系数为1，$A$是一个$(N+1)^2\times(N+1)^2$的矩阵。据此可以编写如下生成系数矩阵$A$和右端项$F$的程序。

```
subroutine generate_matrix(matrix_A, vector_F)

    use constant_
    implicit none

    real(4), intent(inout) :: matrix_A(0: matrix_size-1, 0: matrix_size-1)
    real(4), intent(inout) :: vector_F(0: matrix_size-1, 1)

    ! 构成五点差分的系数矩阵(不含边界)
    do i = 1, N
        do j = 1, N
            matrix_A(i+j*(N+1), i+j*(N+1)) = -4
            matrix_A(i+(j-1)*(N+1), i+(j-1)*(N+1)) = 1
            matrix_A(i+(j+1)*(N+1), i+(j+1)*(N+1)) = 1
            matrix_A(i-1+j*(N+1), i-1+j*(N+1)) = 1
            matrix_A(i+1+j*(N+1), i+1+j*(N+1)) = 1
            vector_F(i+j*(N+1), 1) = i*j*h**4
        end do
    end do

end subroutine generate_matrix
```

其中MODULE constant_中申明了程序中的常量。

```
!------------------------常数声明部分----------------------------------

module constant_

    implicit none
    integer, parameter :: N = 50
    integer, parameter :: matrix_size = (N+1)*(N+1)
    real(4), parameter :: h = 1.0/N
    integer, parameter :: num_not_bd = matrix_size-2*(N+1)-2*(N-1)
    integer :: i, j, row, col

end module constant_
```

生成系数矩阵后，将给出的边界条件整合到系数矩阵中。经过实际计算发现，如果不对矩阵的边界条件进行处理，得到的矩阵无法进行红黑排序，因此需要通过行变换将边界非对角元消去，同时对右端项进行相同的处理。这样处理之后的矩阵在去掉边界部分后可以通过红黑排序变换为一个2×2分块矩阵，其两个对角块均是对角矩阵。处理边界条件的程序如下。

```
subroutine boundray_condition(matrix_A, vector_F)

    ! 处理完后仅保留A和F的一部分，其余部分置零

    use constant_
    implicit none

    real(4), intent(inout) :: matrix_A(0: matrix_size-1, 0: matrix_size-1)
    real(4), intent(inout) :: vector_F(0: matrix_size-1, 1)
    real(4), allocatable :: buf_A_1(:, :), buf_A_2(:, :), buf_F(:, :)
```

```
    integer :: index_zero(2*(N+1)-2*(N-1)), index_notzero(num_not_bd)

    ! 初始化边界条件
    ! 上边界: U(x,1) = 0
    j = N
    do i = 0, N+1
        matrix_A(i+j*(N+1), i+j*(N+1)) = 1
        vector_F(i+j*(N+1), 1) = 0
    end do
    ! 下边界: U(x,0) = 0
    j = 0
    do i = 0, N+1
        matrix_A(i+j*(N+1), i+j*(N+1)) = 1
        vector_F(i+j*(N+1), 1) = 0
    end do
    ! 左边界: U(0,y) = y
    i = 0
    do j = 0, N+1
        matrix_A(i+j*(N+1), i+j*(N+1)) = 1
        vector_F(i+j*(N+1), 1) = h*i
    end do
    ! 右边界: U(1,y) = 1-y
    i = N
    do j = 0, N+1
        matrix_A(i+j*(N+1), i+j*(N+1)) = 1
        vector_F(i+j*(N+1), 1) = 1-h*i
    end do

    ! 整合边界条件
    ! 各个边界减去对应列非零元所在行的元素
    ! 结果是将矩阵化为除边界外可以红黑排序的矩阵
    ! 处理上边界
    j = N
    do i = 0, N+1
        do col = 0, matrix_size
            if ( 0 /= matrix_A(i+j*(N+1), col))  then
                do row = 0, matrix_size
                    matrix_A(row, col) = matrix_A(row, col) - matrix_A(row, i+j*(N+1))
                end do
                vector_F(col, 1) = vector_F(col, 1) - vector_F(i+j*(N+1), 1)
            end if
        end do
        matrix_A(i+j*(N+1), i+j*(N+1)) = 0
        vector_F(i+j*(N+1), 1) = 0
    end do
    ! 处理下边界
    j = 0
    do i = 0, N+1
        do col = 0, matrix_size
            if ( 0 /= matrix_A(i+j*(N+1), col))  then
                do row = 0, matrix_size
                    matrix_A(row, col) = matrix_A(row, col) - matrix_A(row, i+j*(N+1))
                end do
                vector_F(col, 1) = vector_F(col, 1) - vector_F(i+j*(N+1), 1)
            end if
        end do
        matrix_A(i+j*(N+1), i+j*(N+1)) = 0
        vector_F(i+j*(N+1), 1) = 0
    end do
    ! 处理左边界
    i = 0
    do j = 0, N+1
        do col = 0, matrix_size
            if ( 0 /= matrix_A(i+j*(N+1), col))  then
                do row = 0, matrix_size
                    matrix_A(row, col) = matrix_A(row, col) - matrix_A(row, i+j*(N+1))
                end do
                vector_F(col, 1) = vector_F(col, 1) - vector_F(i+j*(N+1), 1)
            end if
        end do
        matrix_A(i+j*(N+1), i+j*(N+1)) = 0
        vector_F(i+j*(N+1), 1) = 0
    end do
    ! 处理右边界
    i = N
    do j = 0, N+1
        do col = 0, matrix_size
            if ( 0 /= matrix_A(i+j*(N+1), col))  then
                do row = 0, matrix_size
                    matrix_A(row, col) = matrix_A(row, col) - matrix_A(row, i+j*(N+1))
                end do
```

```
92                  vector_F(col, 1) = vector_F(col, 1) - vector_F(i+j*(N+1), 1)
93              end if
94          end do
95          matrix_A(i+j*(N+1), i+j*(N+1)) = 0
96          vector_F(i+j*(N+1), 1) = 0
97      end do
98      ! 去掉多余部分，得到可以红黑排序的矩阵
99      ! 计算并记录全零行的位置
100     col = 0
101     ! 下边界
102     j = 0
103     do i = 0, N
104         if (col >= 2*(N-1) + 2*(N+1)) exit
105         index_zero(col) = i+j*(N+1)
106         col = col + 1
107     end do
108     ! 上边界
109     j = N
110     do i = 0, N
111         if (col >= 2*(N-1) + 2*(N+1)) exit
112         index_zero(col) = i+j*(N+1)
113         col = col + 1
114     end do
115     ! 左边界
116     i = 0
117     do j = 0, N
118         if (col >= 2*(N-1) + 2*(N+1)) exit
119         index_zero(col) = i+j*(N+1)
120         col = col + 1
121     end do
122     ! 右边界
123     i = N
124     do j = 0, N
125         if (col >= 2*(N-1) + 2*(N+1)) exit
126         index_zero(col) = i+j*(N+1)
127         col = col + 1
128     end do
129     ! 计算非全零行位置
130     i = 0
131     do j = 0, matrix_size
132         do col = 0, 2*(N+1)+2*(N-1)
133             if (j == index_zero(col)) exit
134         end do
135         if (col >= 2*(N+1)+2*(N-1)) then
136             index_notzero(i) = j
137             i = i + 1
138         end if
139     end do
140     ! 去掉全零行
141     allocate(buf_A_1(0:matrix_size-1, 0:num_not_bd-1))
142     do i = 0, num_not_bd
143         do j = 0 ,matrix_size
144             buf_A_1(j, i) = matrix_A(j, index_notzero(i))
145         end do
146     end do
147     ! 去掉全零列
148     allocate(buf_A_2(0:num_not_bd-1, 0:num_not_bd-1))
149     do i = 0, num_not_bd
150         do j = 0 ,num_not_bd
151             buf_A_1(j, i) = matrix_A(index_notzero(j), i)
152         end do
153     end do
154     ! 对应处理右端项
155     allocate(buf_F(0: num_not_bd-1, 1))
156     do i = 0, num_not_bd
157         buf_F(i, 1) = vector_F(index_notzero(i), 1)
158     end do
159
160     matrix_A = 0
161     vector_F = 0
162     matrix_A(0: num_not_bd-1, 0: num_not_bd-1) = buf_A_2(0: num_not_bd-1, 0: num_not_bd-1)
163     vector_F(0: num_not_bd, 1) = buf_F(0: num_not_bd, 1)
164
165     deallocate(buf_F)
166     deallocate(buf_A_2)
167     deallocate(buf_A_1)
168
169 end subroutine boundray_condition
```

在编写程序时，处理完边界条件后的矩阵依旧赋值给了原矩阵，但是处理完边界条件后的矩阵和原矩阵维度并不相同，在后续调用的时候需要注意调用该函数后系数矩阵和右端项矩阵只有一部分可以作为后续红黑排序的传入参数。更好的写法是返回一个制定大小的数组指针，避免产生该问题。

处理完边界条件后，为了并行化SOR迭代，需要对矩阵进行红黑排序。红黑排序以及多色排序的基本思想是利用排序技术，使排序后的矩阵可以分块为对角块是对角矩阵的形式，对网格点进行分类，使得任何一类网格点上值的更新只依赖于其他类网格点的值，同一类网格点之间的最新迭代值之间互不依赖。分组过程可以具体描述如下：用l记录所采用颜色的种数，I记录所有节点组成的集合，c记录各个节点所分配的颜色号，T记录已标记过颜色的节点集合，S记录尚未标颜色的节点集合，adj_k表示节点k的邻节点所组成的集合，辅助数组t用$t_i = 0$或1表示颜色i在邻节点中没有/已经被使用过。多色排序的算法由如下的伪代码给出。

算法 1 多色排序

$l \leftarrow 0, c_1 \leftarrow l, I \leftarrow \{1, 2, \cdots, n\}, T \leftarrow \{1\}$
for $i = 2 \rightarrow n$ **do** $c_i \leftarrow 0$
end for
for $i = 1 \rightarrow n$ **do** $t_i \leftarrow 0$
end for
while $T \neq \{1, 2, \cdots, n\}$ **do**
    $S \leftarrow I\ T, \mathrm{Select} k \in S$
    for $j \in \mathrm{adj}_k \cap T$ **do** $t_{c_j} \leftarrow 1$
    end for
    for $j = 1 \rightarrow l$ **do**
        if $t_j = 0$ **then** Break
        end if
    end for
    $c_j \leftarrow j$
    if $j = l + 1$ **then** $l \leftarrow l + 1$
    end if
    for $j \in \mathrm{adj}_k \cap T$ **do** $t_{c_j} \leftarrow 0$
    end for
    $T \leftarrow T \cup \{k\}$
end while

在对矩阵多色排序之后，矩阵变为如下形式：

$$\begin{bmatrix} D_R & C_1 \\ C_2 & D_B \end{bmatrix}$$

其中D_R, D_B是对角矩阵。对右端项做相同的分类，则方程$Au = F$可以写成

$$\begin{bmatrix} D_R & C_1 \\ C_2 & D_B \end{bmatrix} \begin{bmatrix} U_R \\ U_B \end{bmatrix} = \begin{bmatrix} F_R \\ F_B \end{bmatrix}$$

对应的SOR迭代格式为

$$\begin{bmatrix} D_R & \\ \omega C_2 & D_B \end{bmatrix} \begin{bmatrix} U_R^{k+1} \\ U_B^{k+1} \end{bmatrix} = \begin{bmatrix} (1-\omega)D_R & -\omega C_1 \\ & (1-\omega)D_B \end{bmatrix} \begin{bmatrix} U_R^{k} \\ U_B^{k} \end{bmatrix} + \omega \begin{bmatrix} F_R \\ F_B \end{bmatrix}$$

写成方程的形式如下。

$$\begin{cases} U_R^{k+1} = D_R^{-1}\omega F_R + (1-\omega)D_R U_R^k - \omega C_1 U_B^k \end{cases}$$

$\mathrm{U}_B^{k+1} = D_B^{-1}\omega F_B + (1-\omega)D_B U_B^k - \omega C_2 U_R^{k+1}$
与SOR迭代$(D-\omega L)U^{k+1} = (\omega U + (1-\omega)D)U^k + \omega F$

2.2 运行结果与分析