

# 并行程序设计实验报告

姓 名: 孙 鑫

学 号: SA22010050

COMP6201P 并行程序设计

(秋季, 2022)

中国科学技术大学

计算机学院

2023 年 11 月 24 日

# 目 录

<b>1</b>	<b>OMP 实验内容</b>	<b>3</b>
1.1	OMP1 . . . . .	3
1.2	OMP2 . . . . .	4
1.3	OMP3 . . . . .	5
1.4	OMP4 . . . . .	6
1.5	OMP5 . . . . .	8
<b>2</b>	<b>MPI 实验内容</b>	<b>9</b>
2.1	分组与 MPI_Bcast . . . . .	9
2.2	蝶式全和 . . . . .	9
2.3	二叉树全和 . . . . .	10
2.4	fox 矩阵乘法 . . . . .	10
2.5	参数服务器 . . . . .	12
2.6	行块划分 . . . . .	13
2.7	棋盘划分 . . . . .	13
<b>3</b>	<b>个人实验内容</b>	<b>14</b>
3.1	串行版本 . . . . .	14
3.2	并行版本 . . . . .	14

# 1 OMP 实验内容

对于 omp 实验，我们测试 omp 程序与串行程序的性能，并保证 omp 与串行程序在循环结束后写入的数组完全相等。先使用串行跑一边循环，然后复制所有数据，使用 omp 并行化再进行循环，最后检测结果是否相同。下面展示部分重要代码，详细代码见源码文件。

## 1.1 OMP1

- i : 不存在方向向量为 (0, 1) 的依赖关系，所以可直接并行化内层循环，使用 `#pragma omp parallel for` 即可
- ii : 注意到循环中含有流依赖，故不能直接使用 `#pragma omp parallel for` 来并行化。同时注意到依赖向量为 (2)，所以相邻两个迭代之间并无依赖关系，因此可将迭代划分为奇数与偶数系数迭代，然后并行执行。

```
#pragma omp parallel num_threads(2) private(i)
{
    int tid = omp_get_thread_num();
    for (int i = 2 + tid; i <= 20; i+=2)
    {
        A[2 * i + 2] = A[2 * i - 2] + B[i];
    }
}
```

- iii: 当  $A[i-1] < 0 \&\& A[i] > 0$  时，存在方向向量为 (1) 的流依赖关系。可以将原循环的一层循环改为两层循环，即当  $A[i-1] < 0 \&\& A[i] > 0$  时，将循环断开，这样循环被分为 m 段，外层循环从 1 到 m，内层循环为第 i 段的所有元素，这样内层循环没有流依赖可并行，但是外层循环无法并行。重点代码：

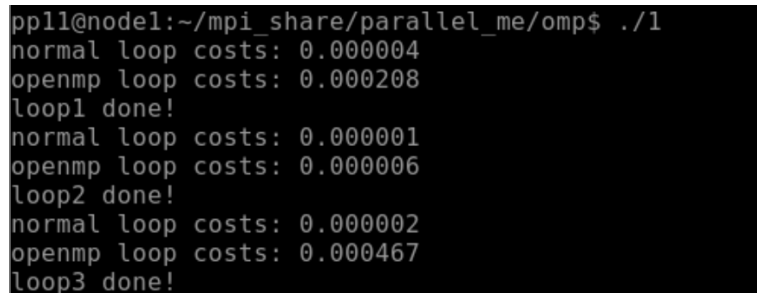
```
int m = 1;
for (int i = 2; i < 20; i++)
{
    if ((A[i - 1] < 0) && (A[i] > 0))
    {
        D[m] = i;
        m++;
    }
}
D[m] = 20;
for (int i = 0; i < m; i++)
{
    #pragma omp parallel for
    for (int k = D[i]; k < D[i + 1]; k++)
```

```

    {
        if (A[k] > 0)
        {
            B[k] = C[k - 1] + 1;
        }
        else
        {
            C[k] = B[k] - 1;
        }
    }
}

```

第一题实验结果如图所示, loop done 代表串行与并行结果相同。normal costs 是串行时间, openmp costs 是使用 openmp 程序的时间。



```

ppl1@node1:~/mpi_share/parallel_me/omp$ ./1
normal loop costs: 0.000004
openmp loop costs: 0.000208
loop1 done!
normal loop costs: 0.000001
openmp loop costs: 0.000006
loop2 done!
normal loop costs: 0.000002
openmp loop costs: 0.000467
loop3 done!

```

图 1: OMP1

由于循环的每次迭代计算代价太小, 因此 openmp 并不能带来时间的降低, 反而因为分配线程等额外开销时间更长。

## 1.2 OMP2

- i 依赖向量为 (1, 1), 最内层循环可以并行执行, 直接使用 omp parallel for 即可。
- ii : 分  $N \leq 100$  和  $N > 100$  两种情况, 若  $N \leq 100$  则可并行化为:

```

// #pragma omp parallel for
for (int i = 1; i <= 100; i++)
{
// #pragma omp parallel for
for (int j = 1; j <= 100; j++)
{
    B[j] = A[IN(j, N, 110)];
#pragma omp parallel for
for (int k = 1; k <= 100; k++)
{
    A[IN(j + 1, k, 110)] = B[j] + C[IN(j, k, 110)];
}
}
}

```

```

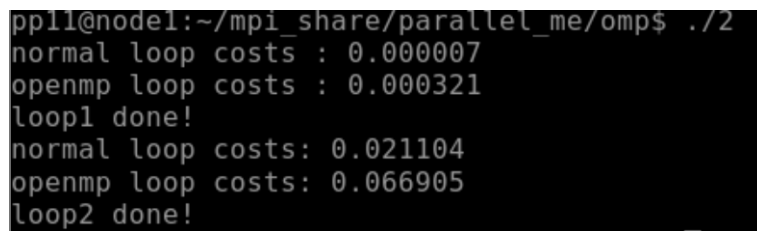
    }
  }
}

for (int i = 1; i <= 100; i++)
{
#pragma omp parallel for
  for (int j = 1; j <= 100; j++)
  {
    Y[i + j] = A[IN(j + 1, N, 110)];
  }
}

#pragma omp parallel for
  for (int i = 1; i <= 100; i++)
  {
    X[i] = Y[i] + 10;
  }
}

```

N>100, 则可去掉前两个 omp parallel 的注释。



```

pp11@node1:~/mpi_share/parallel_me/omp$ ./2
normal loop costs : 0.000007
openmp loop costs : 0.000321
loop1 done!
normal loop costs: 0.021104
openmp loop costs: 0.066905
loop2 done!

```

图 2: OMP2

### 1.3 OMP3

- i 无法并行化。由于存在方向为 (0, 1) 的依赖，内层也无法并行。
- ii: 语句 S4 的作用其实被 S2 所抵消 (除了 S4 写入的最后一个元素, 即 C[101]), 所以可直接删除语句 S4, 并在最后单独赋值 C[101], 语句 S2 和 S3 之间有流依赖, 因此可以将 S2 单独拿出, 先执行 (可并行)。然后再并行执行语句 S3 和 S5。重点代码:

```

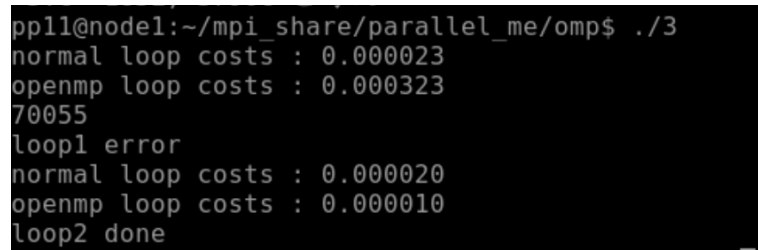
x = y * 2;
#pragma omp parallel for
  for (i = 1; i <= 100; i++)
  {
    C2[i] = B2[i] + x;
  }
#pragma omp parallel for private(i, j)
  for (i = 1; i <= 100; i++)

```

```

{
    A2[i] = C2[i - 1] + z;
    for (j = 1; j <= 50; j++)
    {
        D2[IN(i, j, 60)] = D2[IN(i, j - 1, 60)] + x;
    }
}
C2[101] = A2[100] + B2[100];
z = y + 4;

```



```

pp11@node1:~/mpi_share/parallel_me/omp$ ./3
normal loop costs : 0.000023
openmp loop costs : 0.000323
70055
loop1 error
normal loop costs : 0.000020
openmp loop costs : 0.000010
loop2 done

```

图 3: OMP3

loop1 error 是由于其无法并行化，error 是因为并行化内层循环与串行执行的结果不同。loop2 的 openmp 时间开销相较于串行更低，这是由于并行循环的内部还是一个循环，循环每次迭代的开销变大了，因此 openmp 的时间优越性得以显现。

#### 1.4 OMP4

- i 同时含有 (0, 1) 和 (1, 0) 两个依赖向量，因此内外循环均无法并行化，但是可以对角并行化，即将  $i, j$  坐标轴旋转  $45^\circ$ ，此时内层循环可以并行。

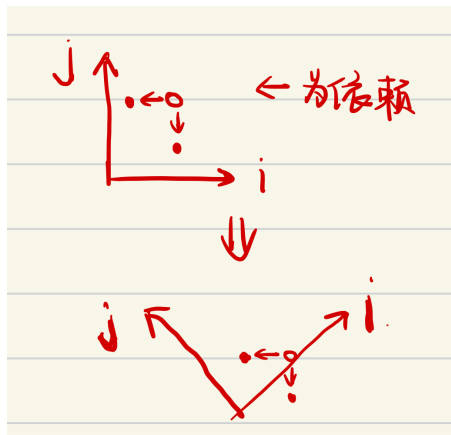


图 4: 4-1

重点代码为：

```

for (i = 4; i <= 20; i++)
{
#pragma omp parallel for
  for (j = max(2, i - 10); j <= min(i / 2, 10); j++)
  {
    B3[IN(j, i - j, 20)] = (B3[IN(j, i - j - 1, 20)] + B3[IN(j - 1, i - j, 20)]) * 0.5;
  }
}

```

- ii：最内层循环有流依赖关系因此无法直接并行化，但是也可采用之前分段的思想，将迭代区间每三个单位分为一段，这样内层循环没有依赖关系。重点代码：

```

for (int k = 1; k <= 16; k += 3)
{
#pragma omp parallel for
  for (int i = k; i <= min(16, k + 2); i++)
  {
    A2[i + 3] = A2[i] + B[i];
  }
}

```

也可采用另一种思路，由于只有  $i$  和  $i+3$  之间存在依赖，将原来的循环拆成三个循环，使用三个 `omp_threads`，每个循环串行执行，循环之间并行执行。

```

#pragma omp parallel num_threads(3) private(i)
{
  int tid = omp_get_thread_num();
  tid %= 3;

```

```

        for (i = tid + 1; i <= 16; i += 3) {
            a[i+3] = a[i] + b[i];
        }
    }
}

```

- iii: 分析依赖图得知，该循环与循环 4.ii 本质相同，因此 mpi 并行化程序与上面相同。

```

pp11@node1:~/mpi_share/parallel_me/omp$ ./4
normal loop costs : 0.000004
openmp diagonal parallel loop costs : 0.000306
loop1 done!
normal loop costs : 0.000002
openmp loop costs : 0.000025
loop2 done!
normal loop costs : 0.000002
openmp loop costs : 0.000025
loop3 done

```

图 5: OMP4

## 1.5 OMP5

- i : 将原循环拆分，S2 S3 之间互相流依赖，因此单独拆出串行执行，剩余部分可并行执行。
- ii : 将原循环按迭代变量 i 拆为两段 1-500, 501-999。然后分别并行执行。先执行 1-500 再执行 501-999。
- iii: 无依赖关系，可任意并行化。使用

```
#pragma omp parallel for collapse(2)
```

```

pp11@node1:~/mpi_share/parallel_me/omp$ ./5
normal loop costs : 0.000005
openmp loop costs : 0.000253
loop1 done!
normal loop costs : 0.000008
openmp loop costs : 0.000010
loop2 done!
normal loop costs : 0.000136
openmp loop costs : 0.000096
loop3 done!

```

图 6: OMP5



## 2 MPI 实验内容

### 2.1 分组与 MPI\_Bcast

先使用 MPI\_Comm\_split 将 world 通信域划分为两个子通信域，这里我们有八个进程，按照每个进程所在的节点（node1, node2）划分，子域中 rank0 的进程进行 MPI\_Bcast 广播。结果为：

```
id_procs: 0. process 0 of 4. comm: 0. host: node1
id_procs: 1. process 1 of 4. comm: 0. host: node1
id_procs: 2. process 2 of 4. comm: 0. host: node1
id_procs: 3. process 3 of 4. comm: 0. host: node1
id_procs: 4. process 0 of 4. comm: 1. host: node2
MPI Commmand 0, original id 0, size 4. The new msg is hello mpi!
MPI Commmand 1, original id 1, size 4. The new msg is hello mpi!
MPI Commmand 2, original id 2, size 4. The new msg is hello mpi!
MPI Commmand 3, original id 3, size 4. The new msg is hello mpi!
id_procs: 6. process 2 of 4. comm: 1. host: node2
id_procs: 7. process 3 of 4. comm: 1. host: node2
id_procs: 5. process 1 of 4. comm: 1. host: node2
MPI Commmand 3, original id 7, size 4. The new msg is hello mpi!
MPI Commmand 0, original id 4, size 4. The new msg is hello mpi!
MPI Commmand 2, original id 6, size 4. The new msg is hello mpi!
MPI Commmand 1, original id 5, size 4. The new msg is hello mpi!
```

图 7: 分组和广播

node1 上的进程全被分到 comm 0, node 2 上的进程全被分到 comm 1, 符合要求。

### 2.2 蝶式全和

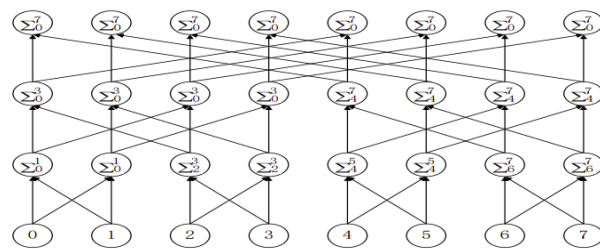


图 8: 蝶式求和

重点是第  $i$  轮，进程  $n$  与进程  $n(1 \ll (i-1))$  交换数据。每个进程既要发送又要接收，接收完数据后与原先的数据相加即可。这里每个进程的初始数据为其 rank，我们使用 8 个进程进行来求全和，得到的结果为：

```

pp11@node1:~/mpi_share/parallel_me/mpi$ mpirun -np 8 --hostfile config ./2_1
Proc:1 Sum is = 28
Proc:2 Sum is = 28
Proc:3 Sum is = 28
Proc:0 Sum is = 28
Proc:5 Sum is = 28
Proc:6 Sum is = 28
Proc:7 Sum is = 28
Proc:4 Sum is = 28

```

图 9: 蝶式求和结果

## 2.3 二叉树全和

重点为第  $i$  轮，相互通信的进程仅第  $i$  位不一致，使用 rank 较小的进程接收数据，rank 较大的进程发送数据。最后 rank 0 进程得到全和，然后将全和分发到其他进程，分发时迭代变量  $i$  从  $\text{num\_procs}$  开始循环，每次迭代除以 2，模  $i$  余 0 的进程发送数据，模  $i/2$  余 0 的进程接收数据，这样就完成了分发。实验结果为

```

pp11@node1:~/mpi_share/parallel_me/mpi$ mpirun -np 8 --hostfile config ./2_2
0 Sum is = 28
1 Sum is = 28
2 Sum is = 28
3 Sum is = 28
4 Sum is = 28
5 Sum is = 28
6 Sum is = 28
7 Sum is = 28

```

图 10: 二叉树求和结果

## 2.4 fox 矩阵乘法

fox 矩阵乘法原理图:

stage 0	$a_{00} \longrightarrow$	$c_{00} = a_{00}b_{00}$	$c_{01} = a_{00}b_{01}$	$c_{02} = a_{00}b_{02}$	$\begin{pmatrix} b_{00} \\ b_{10} \\ b_{20} \end{pmatrix}$	$\begin{pmatrix} b_{01} \\ b_{11} \\ b_{21} \end{pmatrix}$	$\begin{pmatrix} b_{02} \\ b_{12} \\ b_{22} \end{pmatrix}$
	$\longleftarrow a_{11}$	$c_{10} = a_{11}b_{10}$	$c_{11} = a_{11}b_{11}$	$c_{12} = a_{11}b_{12}$	$\begin{pmatrix} b_{10} \\ b_{11} \\ b_{21} \end{pmatrix}$	$\begin{pmatrix} b_{11} \\ b_{21} \\ b_{22} \end{pmatrix}$	$\begin{pmatrix} b_{12} \\ b_{22} \\ b_{22} \end{pmatrix}$
	$\longleftarrow a_{22}$	$c_{20} = a_{22}b_{20}$	$c_{21} = a_{22}b_{21}$	$c_{22} = a_{22}b_{22}$	$\begin{pmatrix} b_{20} \\ b_{21} \\ b_{22} \end{pmatrix}$	$\begin{pmatrix} b_{21} \\ b_{22} \\ b_{22} \end{pmatrix}$	$\begin{pmatrix} b_{22} \\ b_{22} \\ b_{22} \end{pmatrix}$
stage 1	$\longleftarrow a_{01}$	$c_{00} += a_{01}b_{10}$	$c_{01} += a_{01}b_{11}$	$c_{02} += a_{01}b_{12}$	$\begin{pmatrix} b_{10} \\ b_{20} \\ b_{00} \end{pmatrix}$	$\begin{pmatrix} b_{11} \\ b_{21} \\ b_{01} \end{pmatrix}$	$\begin{pmatrix} b_{12} \\ b_{22} \\ b_{02} \end{pmatrix}$
	$\longleftarrow a_{12}$	$c_{10} += a_{12}b_{20}$	$c_{11} += a_{12}b_{21}$	$c_{12} += a_{12}b_{22}$	$\begin{pmatrix} b_{20} \\ b_{21} \\ b_{00} \end{pmatrix}$	$\begin{pmatrix} b_{21} \\ b_{22} \\ b_{01} \end{pmatrix}$	$\begin{pmatrix} b_{22} \\ b_{22} \\ b_{02} \end{pmatrix}$
	$a_{20} \longrightarrow$	$c_{20} += a_{20}b_{00}$	$c_{21} += a_{20}b_{01}$	$c_{22} += a_{20}b_{02}$	$\begin{pmatrix} b_{00} \\ b_{10} \\ b_{20} \end{pmatrix}$	$\begin{pmatrix} b_{01} \\ b_{11} \\ b_{21} \end{pmatrix}$	$\begin{pmatrix} b_{02} \\ b_{12} \\ b_{22} \end{pmatrix}$
stage 2	$\longleftarrow a_{02}$	$c_{00} += a_{02}b_{20}$	$c_{01} += a_{02}b_{21}$	$c_{02} += a_{02}b_{22}$	$\begin{pmatrix} b_{20} \\ b_{21} \\ b_{00} \end{pmatrix}$	$\begin{pmatrix} b_{21} \\ b_{22} \\ b_{01} \end{pmatrix}$	$\begin{pmatrix} b_{22} \\ b_{22} \\ b_{02} \end{pmatrix}$
	$a_{10} \longrightarrow$	$c_{10} += a_{10}b_{00}$	$c_{11} += a_{10}b_{01}$	$c_{12} += a_{10}b_{02}$	$\begin{pmatrix} b_{00} \\ b_{10} \\ b_{20} \end{pmatrix}$	$\begin{pmatrix} b_{01} \\ b_{11} \\ b_{21} \end{pmatrix}$	$\begin{pmatrix} b_{02} \\ b_{12} \\ b_{22} \end{pmatrix}$
	$\longleftarrow a_{21}$	$c_{20} += a_{21}b_{10}$	$c_{21} += a_{21}b_{11}$	$c_{22} += a_{21}b_{12}$	$\begin{pmatrix} b_{10} \\ b_{11} \\ b_{21} \end{pmatrix}$	$\begin{pmatrix} b_{11} \\ b_{12} \\ b_{22} \end{pmatrix}$	$\begin{pmatrix} b_{12} \\ b_{22} \\ b_{22} \end{pmatrix}$

图 11: fox 矩阵乘法

由于我们需要分发子矩阵块,因此需要自定义一个子矩阵块的 MPI\_Datatype, 名字为 submat。相关代码为:

```
MPI_Type_vector(blksize, blksize, N, MPI_INT, &SubMat);
```

此 submat 只有 0 进程使用，剩余进程只需维护 A, B, C, A\_, B\_ 五个个矩阵即可，这五个矩阵也是自定义 mat 类型，

```
MPI_Type_vector(blksize, blksize, blksize, MPI_INT, &Mat);
```

与 submat 唯一不同的是 stride，因为其不需要从最初的大矩阵分发。然后根据计算过程继承计算即可。

为了通信方便，同时需要划分一个行通信域和列通信域便于行之间和列之间广播。

使用 4 进程，每个块矩阵的维度为  $4 * 4$ 。执行命令

```
mpirun -np 4 --hostfile config ./3 4
```

实验结果见图12，每个进程输出 8 行数据，上面四行为正确的子块（由串行程序得到），下面四行为计算得到的子块，两者相同。

```

|1 : 20783 |1 : 13018 |1 : 14351 |1 : 16809
|1 : 25451 |1 : 11306 |1 : 10882 |1 : 18674
|1 : 22472 |1 : 17327 |1 : 7286 |1 : 14156
|1 : 18209 |1 : 10478 |1 : 4857 |1 : 9353
|1 : 20783 |1 : 13018 |1 : 14351 |1 : 16809
|1 : 25451 |1 : 11306 |1 : 10882 |1 : 18674
|1 : 22472 |1 : 17327 |1 : 7286 |1 : 14156
|1 : 18209 |1 : 10478 |1 : 4857 |1 : 9353
Proc#1 Done.
|2 : 27116 |2 : 26342 |2 : 23214 |2 : 21131
|2 : 13943 |2 : 18835 |2 : 15679 |2 : 14653
|2 : 31919 |2 : 29791 |2 : 26970 |2 : 26005
|2 : 21910 |2 : 23879 |2 : 21987 |2 : 23239
|2 : 27116 |2 : 26342 |2 : 23214 |2 : 21131
|2 : 13943 |2 : 18835 |2 : 15679 |2 : 14653
|2 : 31919 |2 : 29791 |2 : 26970 |2 : 26005
|2 : 21910 |2 : 23879 |2 : 21987 |2 : 23239
Proc#2 Done.
|3 : 24370 |3 : 17921 |3 : 16568 |3 : 23294
|3 : 15661 |3 : 13747 |3 : 11218 |3 : 13082
|3 : 31912 |3 : 23598 |3 : 16459 |3 : 26007
|3 : 25672 |3 : 17491 |3 : 10174 |3 : 15839
|3 : 24370 |3 : 17921 |3 : 16568 |3 : 23294
|3 : 15661 |3 : 13747 |3 : 11218 |3 : 13082
|3 : 31912 |3 : 23598 |3 : 16459 |3 : 26007
|3 : 25672 |3 : 17491 |3 : 10174 |3 : 15839
Proc#3 Done.
|0 : 18720 |0 : 22203 |0 : 15168 |0 : 18135
|0 : 24887 |0 : 25858 |0 : 19990 |0 : 22459
|0 : 18542 |0 : 15062 |0 : 20582 |0 : 16732
|0 : 12990 |0 : 12859 |0 : 11472 |0 : 13841
|0 : 18720 |0 : 22203 |0 : 15168 |0 : 18135
|0 : 24887 |0 : 25858 |0 : 19990 |0 : 22459
|0 : 18542 |0 : 15062 |0 : 20582 |0 : 16732
|0 : 12990 |0 : 12859 |0 : 11472 |0 : 13841
Proc#0 Done.

```

图 12: fox 矩阵乘法结果

## 2.5 参数服务器

有  $PNum$  台服务器,  $numProcs - PNum$  个 worker, 首先为服务器划分一个服务通信域 `servecomm`, 里面包含所有的服务器。这样方便服务器之间通讯。对于随机数的收发, 直接使用 `world` 通信域,  $0-PNum-1$  都为服务器,  $0$  负责  $PNum, 2PNum, \dots$ ,  $1$  负责  $PNum + 1, 2Pnum + 1, \dots$  以此类推。所

有服务器都收到其管辖的所有 worker 的数据之后，使用 MPI\_Alltogether 来为每个服务器收集所有数据，然后每个服务器计算平均值返回给管辖的 worker。实验结果为（只截图了一轮）：

```
^C^Cpp11@node1:~/mpi_share/SA22010050/mpi$ mpirun -np 8 --hostfile
fig ./4
proc#2 send data = 60
proc#3 send data = 82
proc#4 send data = 23
proc#5 send data = 41
proc#6 send data = 37
proc#7 send data = 22
Proc#0 send average data = 44
Proc#1 send average data = 44
Proc#2 receive average data = 44
Proc#3 receive average data = 44
Proc#4 receive average data = 44
Proc#5 receive average data = 44
Proc#6 receive average data = 44
Proc#7 receive average data = 44
```

图 13: 参数服务器

## 2.6 行块划分

按行连续划分，即将矩阵以行为单位分割，并交给不同的进程进行处理。下图就是我们对矩阵 A 行连续划分的示意图。除主进程外，每个进程轮流地取出连续的三行（计算一行 B 只需要三行 A）。在 rank 最大的进程生成随机数据，然后以三行为单位（有重叠）分发给其他进程。主进程不参与计算，每个计算进程自行计算，然后各个进程将计算得到的结果再发送给主进程，完成计算。

我们先串行执行该程序，然后使用 mpi 并行化执行该程序，矩阵 A 的维数为 500 维，进程数为 8，如果 B 数组所有 entries 均相同输出 no error，结果如图14。

```
pp11@node1:~/mpi_share/parallel_me/mpi$ mpirun -np 8 --hostfile
config ./5_1
Done.No Error
```

图 14: 行块划分结果

## 2.7 棋盘划分

矩阵 B 的有效结果范围为  $[1 \ N-2, 1 \ N-2]$ 。将矩阵 B 划分为棋盘状，每个小格子由一个进程单独完成，rank 0 的进程生成随机数据并分发，每个格子可以用 fox 矩阵乘法中的 submat 表示：

```

MPI_Datatype SubMat;
MPI_Type_vector(a+2, b+2, N, MPI_DOUBLE, &SubMat);
MPI_Type_commit(&SubMat);

```

其中 a,b 为格子的长宽，如何向独立计算 B 中每个格子的值，A 的格子相较于 B 需要 pad 1，这也是代码中 +2 的原因。

然后计算进程计算 B 中每一个格子的值，在将得到的结果发给主进程，完成计算。

我们先串行执行该程序，然后使用 mpi 并行化执行该程序，如果 B 数组所有 entries 均相同输出 no error，结果如图15。

```

pp11@node1:~/mpi_share/parallel_me/mpi$ mpirun -np 8 --hostfile
config ./5_2
Done.No Error

```

图 15: 棋盘划分结果

### 3 个人实验内容

由于我在炼丹的时候经常使用聚类算法，因此实现了一个 MPI 版本的 K-means 算法。可聚类任意纬度向量。

#### 3.1 串行版本

串行版本即将所有操作在一个进程上进行，不在赘述。实验使用 80000 个样本（随机生成），向量维数为 64 维，聚类中心个数为 8。执行命令

```
time ./serial 10000 8 64
```

最终程序运行时间按如下图所示。

```

real    0m12.414s
user    0m12.393s
sys     0m0.020s

```

图 16: 串行运行时间

#### 3.2 并行版本

选取进程 0 为主节点，负责数据分发，同时也要进行计算。样本总数为 N，进程数为 numProcs，则每个进程被分发的数据个数为  $N/\text{numProcs}$ 。使用 MPI\_Scatter

```
MPI_Scatter(all_sites, d*sites_per_proc, MPI_FLOAT, sites, \
d*sites_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

进程 0 同时随机选择聚类中心点。并将聚类中心广播到每一个进程。

每个进程为自己所分配到的数据计算每个点到聚类中心的距离，取距离最小的那个类为该点所属的聚类，并计算每个聚类包含的数据量 counts，同时将每个数据的属性值叠加到对应聚类 i 的属性和 sums[i] 上。

最后将 counts 和 sums 两个数组使用 MPI\_Reduce 返回给主进程。规约操作为 MPI\_SUM。

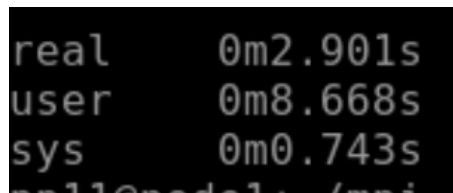
```
MPI_Reduce(sums, grand_sums, k * d, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(counts, grand_counts, k, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

最后进程 0 根据 sums 和 counts 计算新的聚类中心。重复上述步骤知道聚类中心不发生改变或者改变非常小。

并行程序执行的命令为

```
time mpirun -np 8 -hostfile ../mpi/config ../Kmeans_mpi 10000 8 64
```

结果如下



```
real    0m2.901s
user    0m8.668s
sys     0m0.743s
```

图 17: 并行运行时间

观察 real-time 可知，并行版本的 kmeans 算法在此实验设置下运行时间为串行版本的 1/4 不到，mpi 并行化大大提高了多核利用率，使程序更加时间高效。