高性能计算程序设计基础 秋季2021 lab2

1. 通过MPI实现通用矩阵乘法

> 通过MPI实现通用矩阵乘法（Lab1）的并行版本，MPI并行进程（rank size）从1增加至8，矩阵规模从512增加至2048.
>
> 通用矩阵乘法（GEMM）通常定义为：
>
> $$\mathrm{C} = \mathrm{AB}$$
>
> $$C_{\mathrm{m,n}} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$$
>
> 输入：M , N, K三个整数（512 ~2048）
>
> 问题描述：随机生成M*N和N*K的两个矩阵A,B,对这两个矩阵做乘法得到矩阵C.
>
> 输出：A,B,C三个矩阵以及矩阵计算的时间

点对点通信

代码

> 完整代码见lab2_1.c

1. 头文件：

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

其中`#include<mpi.h>`是我们本次实现MPI通信最主要的头文件。

2. 通过一个函数`Get_args()`来获得我们在终端输入运行指令时同时输入的参数m、k、n，他们是我们所计算的矩阵的大小。

```
int m,n,k;
void Get_args(int argc, char* argv[]) {
   //thread_count = strtol(argv[1], NULL, 10);
   m=strtol(argv[1], NULL, 10);
   k=strtol(argv[2], NULL, 10);
   n = strtoll(argv[3], NULL, 10);
}
```

3. 一个用来输出矩阵的函数`PRINT()`：

```
void PRINT(double *x,int m,int n){
    for(int i=0;i<m;++i){
        for(int j=0;j<n;++j) printf("%lf  ",x[i*n+j]);
        printf("\n");
    }
    printf("\n");
}
```

4. 在主函数中我们首先调用函数获得矩阵尺寸，接着定义一些所需的变量：

```
int main(int argc, char *argv[]){
    Get_args(argc,argv);
    int rank,numprocess,block_line;
    double *A,*B,*C,*send_A,*recv_C;
    ...
}
```

5. 首先调用函数`MPI_Init()`，完成一些初始化工作，接下来的两个函数是获得本次程序运行过程中的进程号和进程数量，并以次计算需要将矩阵A划分为几块，并且给存放矩阵元素的数组分配空间：

```
int main(int argc, char *argv[]){
    ...
    MPI_Init(0,0);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocess);
    block_line=m/numprocess;

    A=(double*)malloc(m*k*sizeof(double));
    B=(double*)malloc(k*n*sizeof(double));
    C=(double*)malloc(m*n*sizeof(double));
    send_A=(double*)malloc(block_line*k*sizeof(double));//将A划分后的部分
    recv_C=(double*)malloc(block_line*n*sizeof(double));//每个进程计算获得矩阵C的一部分
    ...
}
```

6. 判断进程号，假设是0号进程：

```
int main(int argc, char *argv[]){
    ...
    if(rank==0){
        int strtime,fintime;
        strtime=clock();
```

```
6
7           srand(time(NULL));
8           for(int i=0;i<m;++i)
9               for(int j=0;j<k;++j) A[i*k+j]=rand()%50;
10          for(int i=0;i<k;++i)
11              for(int j=0;j<n;++j) B[i*n+j]=rand()%50;
12
13          //send
14          for(int i=1; i<numprocess; ++i){
15              MPI_Send(A+i*block_line*k, block_line*k, MPI_DOUBLE, i, 0,
    MPI_COMM_WORLD);
16          }
17          for(int i=1; i<numprocess; ++i){
18              MPI_Send(B, k*n, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
19          }
20
21          //computer
22          for(int l=rank*block_line; l<(rank+1)*block_line; ++l){
23              for(int j=0; j<n; ++j){
24                  double temp=0;
25                  for(int i=0; i<k; ++i) temp+=A[(l-
    rank*block_line)*k+i]*B[i*n+j];
26                  C[l*n+j]=temp;
27              }
28          }
29
30          //接收计算结果
31          for(int i=1; i<numprocess; ++i){
32                  MPI_Recv(recv_C, block_line*n, MPI_DOUBLE, i, 2,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
33                  for(int x=i*block_line; x<(i+1)*block_line; ++x)
34                      for(int y=0; y<n; ++y)    C[x*n+y]=recv_C[(x-
    i*block_line)*n+y];
35          }
36
37          //if we are unable to divide the number of rows of matrix A
    evenly according to numprocess
38          if((m/numprocess)!=0){
39              for(int l=numprocess*block_line-1; l<m; ++l){
40                  for(int j=0; j<n; ++j){
41                      double temp=0;
42                      for(int i=0; i<k; ++i) temp+=A[l*k+i]*B[i*n+j];
43                      C[l*n+j]=temp;
44                  }
45              }
46          }
47
48          fintime=clock();
49          printf("completed:\n");
50          printf("the number of process is %d, the time cost = %lf s\n
    \n",numprocess,(double)(fintime - strtime)/CLOCKS_PER_SEC);
51          //print
52          printf("\n");
53          printf("A:\n");
54          PRINT(A,m,k);
55          printf("B:\n");
56          PRINT(B,k,n);
57          printf("C=A*B:\n");
```

```
58          PRINT(C,m,n);
59
60          free(A);
61          free(B);
62          free(C);
63          free(send_A);
64          free(recv_C);
65      }
66      ...
67  }
```

- 第4-5行，定义关于时间的变量并开始计时。
- 第7-11行，给矩阵A、B赋值50以内的随机数作为矩阵元素。
- 第14-16行，将矩阵A的划分后的分割部分传递给除0以外的其他进程，`A+i*block_line*k`是传送的元素的起始地址；`block_line*k`是传送的元素个数，只传每个进程需要用到的部分；`MPI_DOUBLE`表示传送的元素的数据类型是`double`；`i`表示见数据传给第i个进程；`0`是一个tag要将他和接收元素时的tag保持一致；`MPI_COMM_WORLD`用来表示一组进程间可以通信的进程组，此处是所有进程都可以进行通信的意思。
- 第17-19行，将矩阵B的元素传递给除进程0外的其他进程。`B`传的是矩阵B的地址，`k*n`是传送元素的个数，这里是把矩阵B全部传送过去；此处tag为`1`。后面的参数和上一个的意思一样，不再赘述。
- 第22-28行，进程0开始计算分配给他的计算任务。计算$C_{block_line,n}=\sum_{k=1}^{K}A_{block_line,k}B_{k,n}$，获得矩阵C的一部分，即`block_line`行`n`列。
- 第31-35行，接收其他进程计算得到的矩阵C的部分结果，并把他们放到矩阵C的对应位置。`recv_C`是接收元素的地址；`block_line*n`是接收的元素个数，也是`block_line`行`n`列；`MPI_DOUBLE`接收元素的数据类型；`i`表示从第i个进程获得的；`2`是一个tag，和当时传送这些元素时函数里的tag保持一致，`MPI_COMM_WORLD`时通信子；`MPI_STATUS`在此处不重要，设置为`MPI_STATUS_IGNORE`即可。这里和上面同样的原因判断是否要接收最后一个进程发来的元素，如果是那么就接受矩阵C剩余全部的元素，并放到矩阵C的对应位置上来。
- 第38-46行，表示如果我们无法将矩阵A的函数按照进程数将他平均分配给每个进程，那么就把最后剩余的没有被计算的A的最后几行交给进程0来完成计算并放在C的对应位置上。
- 第48-50行，表示整个矩阵乘法计算完毕，输出整个时长。
- 第52-58行，表示有需要时可以输出矩阵A、B、C的所有元素来验证计算结果的正确性。
- 第60-64行，表示释放之前分配的地址空间。

7. 除进程0外的其他进程，进行下面的操作：

```
1  int main(int argc, char *argv[]){
2      ...
3      else{
4          MPI_Recv(send_A, block_line*k, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
5          MPI_Recv(B, k*n, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
6
7          int x=0;
8          for(int l=rank*block_line; l<(rank+1)*block_line; ++l){
9              for(int j=0; j<n; ++j){
10                 double temp=0;
11                 for(int i=0; i<k; ++i) temp+=send_A[(l-
   rank*block_line)*k+i]*B[i*n+j];
12                 recv_C[x++]=temp;
13             }
14         }
```

```
15
16          MPI_Send(recv_C, block_line*n, MPI_DOUBLE, 0, 2,
    MPI_COMM_WORLD);
17
18          free(A);
19          free(B);
20          free(C);
21          free(send_A);
22          free(recv_C);
23      }
24      ...
25  }
```

- 第4行，接收进程0发过来的矩阵A的部分元素。在调用函数`MPI_Recv()`的一些参数。`send_A`是接受元素的地址；`block_line*k`是接收的元素个数；`MPI_DOUBLE`接收的元素类型；`0`表示此处接收的是从0号进程发过来的数据；`0`是一个tag与发送该元素时的tag对应；`MPI_COMM_WORLD`时通信子；此处`MPI_Recv`同样不重要，置为`MPI_STATUS_IGNORE`。
- 第5行，接收进程0发过来的矩阵B的元素。参数意义与上面的相同，只是对应的数据不同，此处不再赘述。
- 第7-14行，开始计算接收到的这部分矩阵A与矩阵B的乘积，将计算结果放入数组`recv_C`中。
- 第16行，将`recv_C`的元素传递给进程0。
- 第18-22行，释放之前分配的地址空间。

8. 调用函数`MPI_Finalize()`，终止MPI，程序结束。

```
1  int main(int argc, char *argv[]){
2      ...
3      MPI_Finalize();
4      return 0;
5  }
```

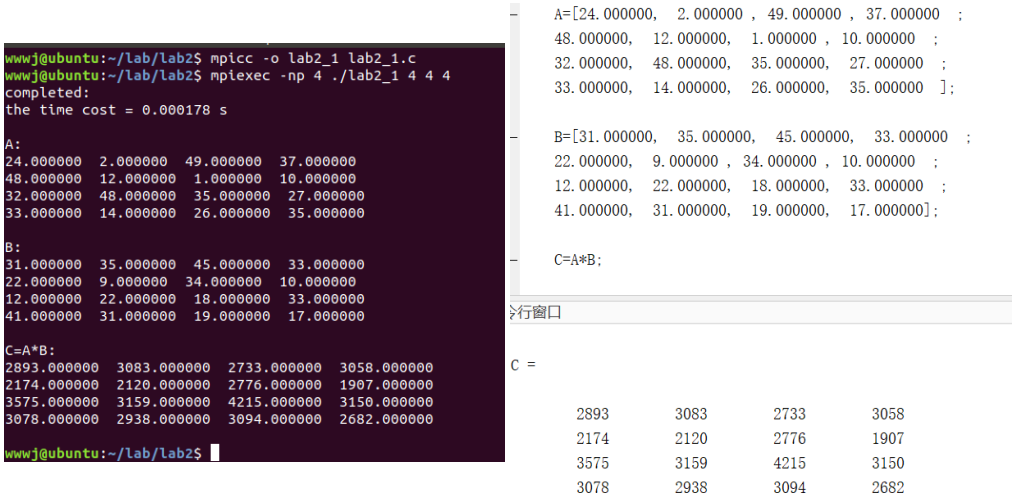
运行结果和时间

下面是编译和运行这段代码的指令：

```
1  /* Compile:  mpicc -o lab2_1 lab2_1.c
2   *
3   * Run:      mpiexec -np 4 ./lab2_1 <m> <k> <n>
4   *           m,k,n is size of matrix A,B,C
5   */
```

- 我们首先验证代码计算矩阵乘法的正确性：

  计算两个4×4矩阵的乘法，左边是代码的计算结果，右边是用matlab计算的结果。



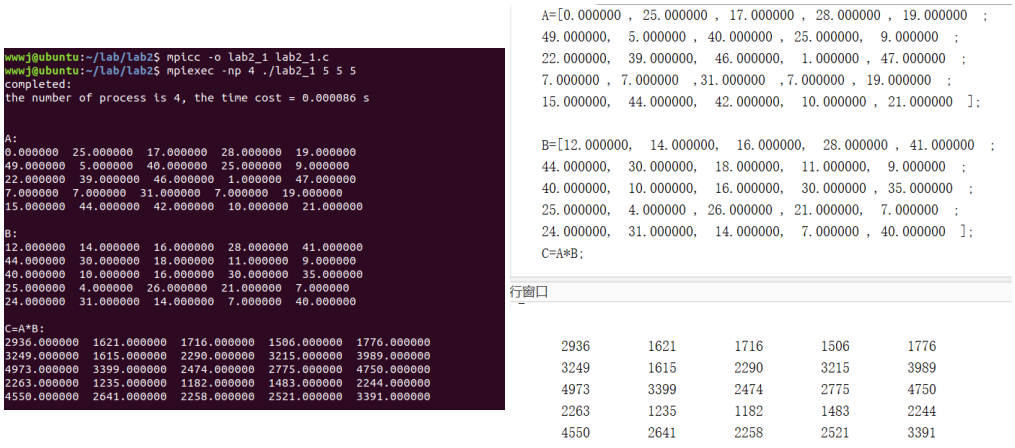
```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_1 lab2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_1 4 4 4
completed:
the time cost = 0.000178 s

A:
24.000000  2.000000  49.000000  37.000000
48.000000  12.000000  1.000000  10.000000
32.000000  48.000000  35.000000  27.000000
33.000000  14.000000  26.000000  35.000000

B:
31.000000  35.000000  45.000000  33.000000
22.000000  9.000000  34.000000  10.000000
12.000000  22.000000  18.000000  33.000000
41.000000  31.000000  19.000000  17.000000

C=A*B:
2893.000000  3083.000000  2733.000000  3058.000000
2174.000000  2120.000000  2776.000000  1907.000000
3575.000000  3159.000000  4215.000000  3150.000000
3078.000000  2938.000000  3094.000000  2682.000000

wwwj@ubuntu:~/lab/lab2$
```

```
A=[24.000000,  2.000000 , 49.000000 , 37.000000  ;
48.000000,  12.000000,  1.000000 , 10.000000  ;
32.000000,  48.000000,  35.000000,  27.000000  ;
33.000000,  14.000000,  26.000000,  35.000000  ];

B=[31.000000,  35.000000,  45.000000,  33.000000  ;
22.000000,  9.000000 , 34.000000 , 10.000000  ;
12.000000,  22.000000,  18.000000,  33.000000  ;
41.000000,  31.000000,  19.000000,  17.000000];

C=A*B;
```

行窗口

```
C =

    2893    3083    2733    3058
    2174    2120    2776    1907
    3575    3159    4215    3150
    3078    2938    3094    2682
```

如果是进程为4，A、B矩阵大小都为5×5（无法将A平均划分），查看计算结果：



```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_1 lab2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_1 5 5 5
completed:
the number of process is 4, the time cost = 0.000086 s


A:
0.000000  25.000000  17.000000  28.000000  19.000000
49.000000  5.000000  40.000000  25.000000  9.000000
22.000000  39.000000  46.000000  1.000000  47.000000
7.000000  7.000000  31.000000  7.000000  19.000000
15.000000  44.000000  42.000000  10.000000  21.000000

B:
12.000000  14.000000  16.000000  28.000000  41.000000
44.000000  30.000000  18.000000  11.000000  9.000000
40.000000  10.000000  16.000000  30.000000  35.000000
25.000000  4.000000  26.000000  21.000000  7.000000
24.000000  31.000000  14.000000  7.000000  40.000000

C=A*B:
2936.000000  1621.000000  1716.000000  1506.000000  1776.000000
3249.000000  1615.000000  2290.000000  3215.000000  3989.000000
4973.000000  3399.000000  2474.000000  2775.000000  4750.000000
2263.000000  1235.000000  1182.000000  1483.000000  2244.000000
4550.000000  2641.000000  2258.000000  2521.000000  3391.000000
```

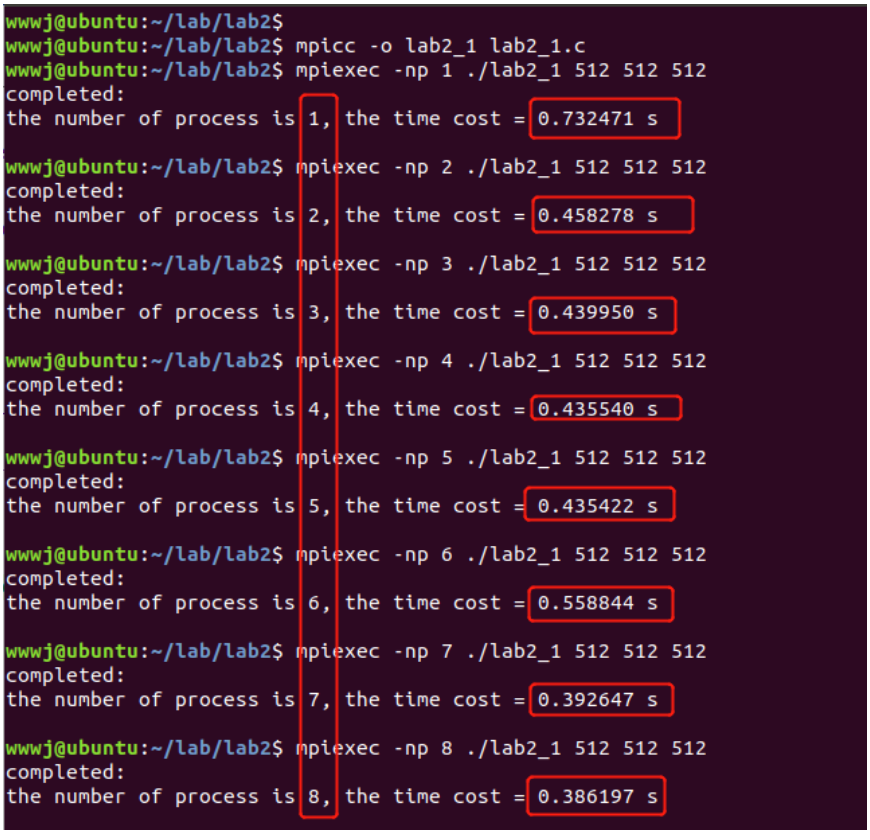
```
A=[0.000000 , 25.000000 , 17.000000 , 28.000000 , 19.000000  ;
49.000000,  5.000000 , 40.000000 , 25.000000,  9.000000  ;
22.000000,  39.000000,  46.000000,  1.000000 , 47.000000  ;
7.000000 , 7.000000  ,31.000000  ,7.000000 , 19.000000  ;
15.000000,  44.000000,  42.000000,  10.000000 , 21.000000  ];

B=[12.000000,  14.000000,  16.000000,  28.000000 , 41.000000  ;
44.000000,  30.000000,  18.000000,  11.000000,  9.000000  ;
40.000000,  10.000000,  16.000000,  30.000000 , 35.000000  ;
25.000000,  4.000000 , 26.000000 , 21.000000,  7.000000  ;
24.000000,  31.000000,  14.000000,  7.000000 , 40.000000  ];
C=A*B;
```

行窗口

```
    2936    1621    1716    1506    1776
    3249    1615    2290    3215    3989
    4973    3399    2474    2775    4750
    2263    1235    1182    1483    2244
    4550    2641    2258    2521    3391
```

计算结果正确。

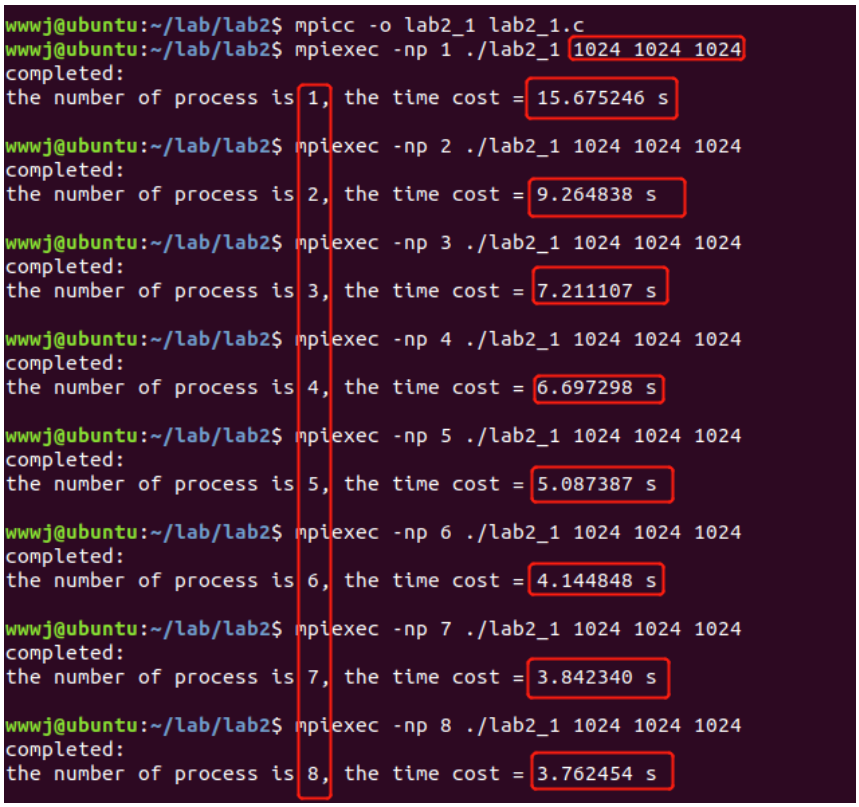
- 在矩阵大小都为512×512时，将MPI的进程数从1增加至8的耗时情况：



```
wwwj@ubuntu:~/lab/lab2$
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_1 lab2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_1 512 512 512
completed:
the number of process is 1, the time cost = 0.732471 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 2 ./lab2_1 512 512 512
completed:
the number of process is 2, the time cost = 0.458278 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 3 ./lab2_1 512 512 512
completed:
the number of process is 3, the time cost = 0.439950 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_1 512 512 512
completed:
the number of process is 4, the time cost = 0.435540 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 5 ./lab2_1 512 512 512
completed:
the number of process is 5, the time cost = 0.435422 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 6 ./lab2_1 512 512 512
completed:
the number of process is 6, the time cost = 0.558844 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 7 ./lab2_1 512 512 512
completed:
the number of process is 7, the time cost = 0.392647 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 8 ./lab2_1 512 512 512
completed:
the number of process is 8, the time cost = 0.386197 s
```

可以看到，进程数从1增加到2时耗时情况改变的最为明显，减少了0.3s左右的时间；之后进程从3到5，运行时间都没有什么明显的改变；进程变为6的时候，可能是由于通信的耗时等原因，运行时间反而变长了；但最后进程数变为7、8的时候运行时间又减少到了0.3s左右。总体来看，他们的运行时间随着进程数的增加而减少。

- 在矩阵大小都为1024×1024时，将MPI的进程数从1增加至8的耗时情况：



很明显可以看出，运行时间会随着进程数的增加而减少，在刚开始时随着进程数的增加运行时间会很明显减少，之后在进程从3-8时，每增加一个进程运行时间减少1s左右。

2. 基于MPI的通用矩阵乘法优化

> 分别采用MPI点对点通信和MPI集合通信实现矩阵乘法中的进程之间通信，并比较两种实现方式的性能。尝试用mpi_type_create_struct聚合MPI进程内变量后通信。

集合通信

代码

> 完整代码见lab2_2_1.c

代码前面的头文件、获取参数、输入输出函数等和点对点通信的完全相同（即前面介绍的代码1-5），此处不再赘述。

1. 判断，如果是进程0，就先来给矩阵A、B取随机数，并开始计时：

```
int main(int argc, char* argv[]){
    ...
    if(rank==0){
        srand(time(NULL));
        strtime=clock();

        for(int i=0;i<m;++i)
            for(int j=0;j<k;++j) A[i*k+j]=rand()%50;
        for(int i=0;i<k;++i)
            for(int j=0;j<n;++j) B[i*n+j]=rand()%50;
    }
    ...
}
```

2. 利用函数`MPI_Scatter()`和`MPI_Bcast()`，将矩阵A、B分别散播和广播到所有进程

```
int main(int argc, char* argv[]){
    ...
    //将A散播
    MPI_Scatter(A, block_line*k, MPI_DOUBLE, send_A, block_line*k,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    //将B广播
    MPI_Bcast(B,k*n,MPI_DOUBLE,0,MPI_COMM_WORLD);
    ...
}
```

- 函数`MPI_Scatter()`。其中`A`是要进行散播的矩阵的起始地址，`block_line*k`是要散播到每个进程的元素个数，`MPI_DOUBLE`是进行通信传递的数据的数据类型，`send_A`是接收数据的起始地址，`block_line*k`是每个进程要收到的元素个数，`MPI_DOUBLE`是进行通信传递的数据的数据类型，`0`表示发送消息的进程的标识,这里是进程0来进行发送消息，`MPI_COMM_WORLD`是通信域。

  注意这里的两个元素个数是相同的，即A按顺序每次发x个，每个进程接收x个。它的意思并不是说总共发送A地址后的x个元素，每个进程接收x个，即这两个参数要相同。

- 函数`MPI_Bcast()`。其中`B`是要进行广播的元素的起始地址，`k*n`是广播的元素个数，`MPI_DOUBLE`是数据类型，`0`表示发送消息的进程的标识,这里是进程0来进行发送消息，`MPI_COMM_WORLD`是通信域。

3. 然后每个进程进行自己负责的那部分的矩阵乘法运算：

```
int main(int argc, char* argv[]){
    ...
    //computer
    int x=0;
    for(int l=rank*block_line; l<(rank+1)*block_line; ++l){
        for(int j=0; j<n; ++j){
            double temp=0;
            for(int i=0; i<k; ++i) temp+=send_A[(l-
rank*block_line)*k+i]*B[i*n+j];
            recv_C[x++]=temp;
        }
    }
    ...
}
```

4. 将计算所得的运算结果进行“聚集”，来获得最终的矩阵C：

```
int main(int argc, char* argv[]){
    ...
    MPI_Gather(recv_C, block_line*n, MPI_DOUBLE, C, block_line*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ...
}
```

函数`MPI_Gather()`。其中`recv_C`是每个进程传递回去的数据的起始地址，`block_line*n`传递的元素个数，`MPI_DOUBLE`是进行通信传递的数据类型，`C`是接收数据的起始地址，`block_line*n`是从每个进程要收到的元素个数，`MPI_DOUBLE`是进行通信传递的数据的数据类型，`0`表示接收消息的进程的标识,这里是进程0来进行消息汇总，`MPI_COMM_WORLD`是通信域。

5. 最后在进程0处进行判断。意在如果我们无法将矩阵A的函数按照进程数将他平均分配给每个进程，那么就把最后剩余的没有被计算的A的最后几行交给进程0来完成计算并放在C的对应位置上。然后结束计时，并输出在x进程下的运行时间。如果需要的话也可以输出矩阵A、B、C来判断运算结果是否正确。

```
int main(int argc, char* argv[]){
    ...
    if(rank==0){
        //if we are unable to divide the number of rows of matrix A evenly according to numprocess
        if((m/numprocess)!=0){
            for(int l=numprocess*block_line-1; l<m; ++l){
                for(int j=0; j<n; ++j){
                    double temp=0;
                    for(int i=0; i<k; ++i) temp+=A[l*k+i]*B[i*n+j];
                    C[l*n+j]=temp;
                }
            }
        }

        fintime=clock();
        printf("completed:\n");
        printf("the number of process is %d, the time cost = %lf s\n\n",numprocess,(double)(fintime - strtime)/CLOCKS_PER_SEC);

        //print
        // printf("\n");
        // printf("A:\n");
        // PRINT(A,m,k);
        // printf("B:\n");
        // PRINT(B,k,n);
        // printf("C=A*B:\n");
        // PRINT(C,m,n);
    }
    ...
}
```

6. 释放地址空间，并终止MPI，程序运行结束：

```c
int main(int argc, char* argv[]){
    ...
    free(A);
    free(B);
    free(C);
    free(send_A);
    free(recv_C);

    MPI_Finalize();
    return 0;
}
```

运行结果和时间

下面是编译和运行这段代码的指令：

```c
/* Compile:  mpicc -o lab2_2_1 lab2_2_1.c
 *
 * Run:      mpiexec -np 4 ./lab2_2_1 <m> <k> <n>
 *           m,k,n is size of matrix A,B,C
 */
```

- 我们首先验证代码计算矩阵乘法的正确性：

  计算两个4×4矩阵的乘法，左边是代码的计算结果，右边是用matlab计算的结果。

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_2_1 lab2_2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_2_1 4 4 4
completed:
the time cost = 0.000077 s

A:
3.000000  39.000000  18.000000  43.000000
13.000000  10.000000  46.000000  12.000000
34.000000  41.000000  5.000000  24.000000
0.000000  16.000000  7.000000  25.000000

B:
14.000000  9.000000  0.000000  21.000000
23.000000  41.000000  18.000000  41.000000
4.000000  5.000000  44.000000  0.000000
5.000000  36.000000  23.000000  11.000000

C=A*B:
1226.000000  3264.000000  2483.000000  2135.000000
656.000000  1189.000000  2480.000000  815.000000
1559.000000  2876.000000  1510.000000  2659.000000
521.000000  1591.000000  1171.000000  931.000000
```

```
A=[3.000000 , 39.000000 , 18.000000 , 43.000000  ;
13.000000,  10.000000,  46.000000,  12.000000  ;
34.000000,  41.000000,  5.000000 , 24.000000  ;
0.000000 , 16.000000 , 7.000000  ,25.000000  ];

B=[14.000000,  9.000000 , 0.000000  ,21.000000  ;
23.000000,  41.000000,  18.000000,  41.000000  ;
4.000000 , 5.000000  ,44.000000  ,0.000000  ;
5.000000 , 36.000000 , 23.000000 , 11.000000  ];

C=A*B;
```

令行窗口

```
C =

        1226        3264        2483        2135
         656        1189        2480         815
        1559        2876        1510        2659
         521        1591        1171         931
```

如果是进程为4，A、B矩阵大小都为5×5（无法将A平均划分），查看计算结果：

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_2_1 lab2_2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_2_1 5 5 5
completed:
the time cost = 0.000043 s

A:
45.000000  15.000000  4.000000  12.000000  0.000000
17.000000  34.000000  49.000000  12.000000  21.000000
37.000000  9.000000  25.000000  21.000000  42.000000
4.000000  0.000000  6.000000  28.000000  1.000000
9.000000  45.000000  40.000000  24.000000  35.000000

B:
11.000000  21.000000  49.000000  33.000000  4.000000
44.000000  30.000000  21.000000  48.000000  42.000000
21.000000  17.000000  26.000000  20.000000  30.000000
47.000000  10.000000  41.000000  25.000000  33.000000
35.000000  29.000000  35.000000  41.000000  7.000000

C=A*B:
1803.000000  1583.000000  3116.000000  2585.000000  1326.000000
4011.000000  2939.000000  4048.000000  4334.000000  3509.000000
3785.000000  2900.000000  4983.000000  4400.000000  2263.000000
1521.000000  495.000000  1535.000000  993.000000  1127.000000
5272.000000  3474.000000  4635.000000  5292.000000  4163.000000
```

```
A=[45.000000,  15.000000,  4.000000 , 12.000000,  0.000000  ;
17.000000,  34.000000,  49.000000  ,12.000000 , 21.000000  ;
37.000000,  9.000000 , 25.000000  ,21.000000 , 42.000000  ;
4.000000 , 0.000000  ,6.000000 , 28.000000,  1.000000  ;
9.000000 , 45.000000 , 40.000000  ,24.000000 , 35.000000  ];
B=[11.000000,  21.000000,  49.000000 , 33.000000,  4.000000  ;
44.000000,  30.000000,  21.000000  ,48.000000 , 42.000000  ;
21.000000,  17.000000,  26.000000  ,20.000000 , 30.000000  ;
47.000000,  10.000000,  41.000000  ,25.000000 , 33.000000  ;
35.000000,  29.000000,  35.000000 , 41.000000,  7.000000  ];

C=A*B;
```

行窗口

```
        1803        1583        3116        2585        1326
        4011        2939        4048        4334        3509
        3785        2900        4983        4400        2263
        1521         495        1535         993        1127
        5272        3474        4635        5292        4163
```

计算结果正确。

- 在矩阵大小都为512×512时，将MPI的进程数从1增加至8的耗时情况：

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_2_1 lab2_2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_2_1 512 512 512
completed:
the number of process is 1, the time cost = 0.764076 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 2 ./lab2_2_1 512 512 512
completed:
the number of process is 2, the time cost = 0.553319 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 3 ./lab2_2_1 512 512 512
completed:
the number of process is 3, the time cost = 0.417501 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_2_1 512 512 512
completed:
the number of process is 4, the time cost = 0.442605 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 5 ./lab2_2_1 512 512 512
completed:
the number of process is 5, the time cost = 0.388621 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 6 ./lab2_2_1 512 512 512
completed:
the number of process is 6, the time cost = 0.379370 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 7 ./lab2_2_1 512 512 512
completed:
the number of process is 7, the time cost = 0.364101 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 8 ./lab2_2_1 512 512 512
completed:
the number of process is 8, the time cost = 0.450096 s
```

基本上运行时间随着进程数的增加而减少，只是进程数为8时，可能是由于通信的原因。

- 在矩阵大小都为1024×1024时，将MPI的进程数从1增加至8的耗时情况：

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_2_1 lab2_2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 1, the time cost = 12.942828 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 2 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 2, the time cost = 8.610211 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 3 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 3, the time cost = 6.707928 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 4, the time cost = 6.794914 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 5 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 5, the time cost = 5.200326 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 6 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 6, the time cost = 4.930436 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 7 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 7, the time cost = 3.985662 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 8 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 8, the time cost = 4.228503 s
```

加速效果较为明显。

聚合MPI进程内变量后通信

代码

> 完整代码见lab2_2_2.c

代码前面的头文件、获取参数、输入输出函数等和点对点通信的完全相同（即前面介绍的代码1-5），只是不再单独创造一下变量`double *A,*B`，因为我们要用一个结构体来传递这些变量，此处不再赘述。

1. 创建一个新数据类型`mytype`：

```
1  int main(int argc, char *argv[]){
2      ...
3      struct var{
4          double A_[block_line*k];
5          double B[k*n];
6      };
7
8      int var_count=2;
9      int var_everycount[2]={block_line*k, k*n};
10     MPI_Aint var_displace[2]={0, 8*block_line*k};
11     MPI_Datatype var_type[2]={MPI_DOUBLE,MPI_DOUBLE};
12     MPI_Datatype mytype;
13
14     MPI_Type_create_struct(var_count, var_everycount, var_displace,
   var_type, &mytype);
15     MPI_Type_commit(&mytype);
16     ...
17 }
```

   - 第3-6行是定义的一个结构体，里面用来存放即将传向各个进程的`A_`，以及`B`。他们的大小已经确定，由于MPI函数中传递的元素的起始地址之后的元素被要求是连续的，因此在这里一定要确定他们的大小。这里只在结构体中存放A、B，是因为我们如果将C也加进去，之后将C的计算结果传递回来的时候就要把整个结构体传递回来，但里面的A、B我们是不需要的，因此在这个结构体中只写入A、B。
   - 第8行，是新数据类型中元素个数，这里是2（double, double）。
   - 第9行，是每个数据项的元素个数（A_里有block_line*k个,B里有k*n个）。
   - 第10行，是每个数据项距离消息起始位置的偏移量（A_的偏移量是0，B的偏移量是8*block_line*k字节）
   - 第11行，是每个数据项的MPI类型。
   - 第12行，是一个自定义数据类型出口，命名为`mytype`。
   - 第14行，函数`MPI_Type_create_struct()`，它的每个参数前面几行都有介绍。
   - 第15行，函数`MPI_Type_commit()`表示允许MPI实现在通信函数内使用这一数据类型。

2. 在进程0中进行以下操作：

```
1  int main(int argc, char *argv[]){
2      ...
3      if(rank==0){
4          double A[m*k],B[k*n],C[m*n];
5          struct var tempvar;
6
7          int strtime,fintime;
8          strtime=clock();
9
10         srand(time(NULL));
```

```
        for(int i=0;i<m;++i)
            for(int j=0;j<k;++j) A[i*k+j]=rand()%50;
        for(int i=0;i<k;++i)
            for(int j=0;j<n;++j) B[i*n+j]=rand()%50;

        struct var ABC[numprocess];
        for(int i=0;i<numprocess;++i){
            for(int j=0;j<block_line*k;++j){
                ABC[i].A_[j]=A[i*block_line*k+j];
            }
            for(int j=0;j<n*k;++j){
                ABC[i].B[j] = B[j];
            }
        }

        for(int i=1; i<numprocess; ++i){
            MPI_Send(&ABC[i], 1, mytype, i, 0, MPI_COMM_WORLD);
        }


        for(int l=rank*block_line; l<(rank+1)*block_line; ++l){
            for(int j=0; j<n; ++j){
                double temp=0;
                for(int i=0; i<k; ++i) temp+=ABC[0].A_[(l-
block_line*rank)*k+i]*ABC[0].B[i*n+j];
                C[l*n+j]=temp;
            }
        }

        if((m/numprocess)!=0){
            for(int l=numprocess*block_line-1; l<m; ++l){
                for(int j=0; j<n; ++j){
                    double temp=0;
                    for(int i=0; i<k; ++i) temp+=A[l*k+i]*B[i*n+j];
                    C[l*n+j]=temp;
                }
            }
        }

        //接收计算结果
        for(int i=1; i<numprocess; ++i){
            MPI_Recv(recv_C, block_line*n, MPI_DOUBLE, i, 2,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for(int x=i*block_line; x<(i+1)*block_line; ++x)
                for(int y=0; y<n; ++y)    C[x*n+y]=recv_C[(x-
i*block_line)*n+y];
        }

        fintime=clock();
        printf("completed:\n");
        printf("the number of process is %d, the time cost = %lf s\n
\n",numprocess,(double)(fintime - strtime)/CLOCKS_PER_SEC);

        //print
        // printf("\n");
        // printf("A:\n");
        // PRINT(A,m,k);
        // printf("B:\n");
```

```
65          // PRINT(B,k,n);
66          // printf("C=A*B:\n");
67          // PRINT(C,m,n);
68      }
69      ...
70  }
```

- 第4-14行，定义一些需要的变量，给矩阵A、B取随机数。
- 第16-24行，定义一个结构体数组，表示分配给其他进程的结构体，并且给结构体里的变量按照需要的元素进行赋值。
- 第26-27行，用`MPI_Send()`函数将结构体传递给其他进程。`&ABC[i]`是给每个进程传递的元素的起始地址，`1`表示传递的结构体的数量是1，`mytype`表示传递的数据的数据结构是我们新创建的`mytype`。
- 第31-37行，计算分配给进程0的部分任务。
- 第39-47行，如果我们无法将矩阵A的函数按照进程数将他平均分配给每个进程，那么就把最后剩余的没有被计算的A的最后几行交给进程0来完成计算并放在C的对应位置上。
- 第50-68行，将C的部分传递回来、输出运行时间、输出矩阵等内容，和点对点通信中的一样，不再赘述。

3. 其他进程中的操作：

```
1  int main(int argc, char *argv[]){
2      ...
3      else{
4          struct var tempvar;
5          MPI_Recv(&tempvar, 1, mytype, 0, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
6
7          int x=0;
8          for(int l=rank*block_line; l<(rank+1)*block_line; ++l){
9              for(int j=0; j<n; ++j){
10                 double temp=0;
11                 for(int i=0; i<k; ++i) temp+=tempvar.A_[(l-
   rank*block_line)*k+i]*tempvar.B[i*n+j];
12                 recv_C[x++]=temp;
13             }
14         }
15
16         MPI_Send(recv_C, block_line*n, MPI_DOUBLE, 0, 2,
   MPI_COMM_WORLD);
17     }
18
19     MPI_Finalize();
20     return 0;
21 }
```

- 第4-5行，将接收到的结构体放在`tempvar`中，`MPI_Recv()`函数中的参数和之前不一样的地方就在与传递的数据类型是我们之前新创建的`mytype`。
- 设下的部分也都和之前的点对点通信一样，此处不再赘述。

还可以对集合通信用这种新数据类型。

> 完整代码见lab2_2_3.c

创建数据类型的过程和上面一样，这里只演示使用这个新数据类型的过程：

```
...
MPI_Scatter(&ABC, 1, mytype, &tempvar, 1, mytype, 0, MPI_COMM_WORLD);
...
```

运行：

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_2_3 lab2_2_3.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_2_3 256 256 256
completed:
the number of process is 4, the time cost = 0.037954 s
```

运行结果和时间

下面是编译和运行这段代码的指令：

```
/* Compile:  mpicc -o lab2_2_2 lab2_2_2.c
 *
 * Run:      mpiexec -np 4 ./lab2_2_2 <m> <k> <n>
 *           m,k,n is size of matrix A,B,C
 */
```

- 我们首先验证代码计算矩阵乘法的正确性：

  计算两个5×5矩阵的乘法，左边是代码的计算结果，右边是用matlab计算的结果。

```
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_2_2 5 5 5
completed:
the number of process is 4, the time cost = 0.000032 s


A:
6.000000  37.000000  41.000000  35.000000  40.000000
31.000000  30.000000  1.000000  15.000000  35.000000
2.000000  15.000000  8.000000  14.000000  17.000000
45.000000  40.000000  20.000000  17.000000  33.000000
33.000000  40.000000  5.000000  42.000000  3.000000

B:
5.000000  22.000000  2.000000  46.000000  7.000000
19.000000  5.000000  46.000000  12.000000  40.000000
37.000000  44.000000  23.000000  38.000000  11.000000
10.000000  40.000000  26.000000  19.000000  7.000000
45.000000  16.000000  47.000000  15.000000  35.000000

C=A*B:
4400.000000  4161.000000  5447.000000  3543.000000  3618.000000
2487.000000  2036.000000  3500.000000  2634.000000  2758.000000
1496.000000  1303.000000  2041.000000  1097.000000  1395.000000
3380.000000  3278.000000  4383.000000  4128.000000  3409.000000
1665.000000  2874.000000  3254.000000  3031.000000  2285.000000
```

```
A=[6.000000 , 37.000000 , 41.000000 , 35.000000 , 40.000000
31.000000,  30.000000,  1.000000 , 15.000000 , 35.000000  ;
2.000000 , 15.000000 , 8.000000  ,14.000000  ,17.000000  ;
45.000000,  40.000000,  20.000000,  17.000000,  33.000000  ;
33.000000,  40.000000,  5.000000 , 42.000000 , 3.000000  ];
B=[5.000000 , 22.000000 , 2.000000  ,46.000000  ,7.000000  ;
19.000000,  5.000000 , 46.000000 , 12.000000 , 40.000000  ;
37.000000,  44.000000,  23.000000,  38.000000,  11.000000  ;
10.000000,  40.000000,  26.000000,  19.000000,  7.000000  ;
45.000000,  16.000000,  47.000000,  15.000000,  35.000000 ];

C=A*B;
```

```
4400    4161    5447    3543    3618
2487    2036    3500    2634    2758
1496    1303    2041    1097    1395
3380    3278    4383    4128    3409
1665    2874    3254    3031    2285
```

计算结果正确。

- 在矩阵大小都为256×256时，将MPI的进程数从1增加至8的耗时情况：

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_2_2 lab2_2_2.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_2_2 256 256 256
completed:
the number of process is 1, the time cost = 0.095653 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 2 ./lab2_2_2 256 256 256
completed:
the number of process is 2, the time cost = 0.059996 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 3 ./lab2_2_2 256 256 256
completed:
the number of process is 3, the time cost = 0.046804 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 4 ./lab2_2_2 256 256 256
completed:
the number of process is 4, the time cost = 0.048192 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 5 ./lab2_2_2 256 256 256
completed:
the number of process is 5, the time cost = 0.088567 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 6 ./lab2_2_2 256 256 256
completed:
the number of process is 6, the time cost = 0.115310 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 7 ./lab2_2_2 256 256 256
completed:
the number of process is 7, the time cost = 0.100470 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 8 ./lab2_2_2 256 256 256
completed:
the number of process is 8, the time cost = 0.098566 s
```

可以看到在进程从1增加到4时，运算时间会随着进程数的增加而减少，但是从5增加到8时，运行时间反而变长，可能是他们之间进行通信的原因。

3. 改造Lab1成矩阵乘法库函数

> 将Lab1的矩阵乘法改造为一个标准的库函数 matrix_multiply（函数实现文件和函数头文件），输入参数为三个完整定义矩阵（A,B,C），定义方式没有具体要求，可以是二维矩阵，也可以是struct等。在Linux系统中将此函数编译为.so文件，由其他程序调用。

代码

1. 由于没有对输入参数进行定义，因此可以写多种参数的函数，实现重载，因此头文件：

> 代码见matrix_multiply.h

```
#ifndef _MATRIX_MULTIPLY_
#define _MATRIX_MULTIPLY_

#include<iostream>
#include<vector>
using namespace std;

vector<vector<double>> matrix_multiply(vector<vector<double>>
A,vector<vector<double>> B, int m,int n ,int k);

void matrix_multiply(vector<vector<double>> *A,vector<vector<double>>
*B,vector<vector<double>> *C, int m,int n ,int k);

void matrix_multiply(double *A, double *B, double *C,int m,int n ,int
k);

```

```
14 #endif
```

- 第1-2行，防止头文件重名，开始头文件。
- 第4-6行，一些需要的头文件。
- 第8行，输入的参数矩阵A、B的类型是二维向量，最后会将计算结果二维向量C返回。
- 第10行，输入的参数矩阵A、B的类型是二维向量的指针，同时矩阵C也是二维向量的指针传入。因此我们拥有矩阵C的地址，就可以直接获得计算结果矩阵C。
- 第12行，输入的参数是矩阵A、B、C的double类型的指针，最后也可以不用返回矩阵C而直接获得计算结果。
- 第14行，结束头文件的书写

2. 实现头文件中的三个函数

> 函数`vector<vector<double>> matrix_multiply(vector<vector<double>> A,vector<vector<double>> B, int m,int n ,int k)`在文件 lab2_3_vector.cpp 中实现
>
> 函数`void matrix_multiply(vector<vector<double>> *A,vector<vector<double>> *B,vector<vector<double>> *C, int m,int n ,int k)`在文件lab2_3_vectorpoint.cpp中实现
>
> 函数`void matrix_multiply(double *A, double *B, double *C,int m,int n ,int k)`在文件 lab2_3_doublepoint.cpp中实现

这里三个文件中对重载函数就是lab1中的`GEMM`的变体，实现实际上只是数据类型有些改变，只需要一些相应的修改，内在逻辑并没有改变，因此这里就对这三种函数不再进行赘述。

3. 写一个测试函数用来验证对我们实现的头文件的调用

> 完整代码见lab2_3_test.cpp

```
1  #include<iostream>
2  #include<vector>
3  #include<time.h>
4  #include"matrix_multiply.h"
5  using namespace std;
6  ...
7  int main(){
8      Get_args(argc, argv);
9
10     //vector
11     srand(time(NULL));
12     vector<vector<double>> A,B,C;
13     A=vector<vector<double>>(m,vector<double>(n,0));
14     B=vector<vector<double>>(n,vector<double>(k,0));
15     C=vector<vector<double>>(m,vector<double>(k,0));
16
17     for(int i=0;i<m;++i)
18         for(int j=0;j<n;++j) A[i][j]=rand()%50;
19     for(int i=0;i<n;++i)
20         for(int j=0;j<k;++j) B[i][j]=rand()%50;
21
22     C=matrix_multiply(A,B,m,n,k);
23
24     //vectorpoint
```

```
25      vector<vector<double>> *A2,*B2,*C2;
26      vector<vector<double>> A_=vector<vector<double>>(m,vector<double>
    (n,0));
27      vector<vector<double>> B_=vector<vector<double>>(n,vector<double>
    (k,0));
28      vector<vector<double>> C_=vector<vector<double>>(m,vector<double>
    (k,0));
29      A2=&A_,B2=&B_,C2=&C_;
30
31      srand(time(NULL));
32      for(int i=0;i<m;++i)
33          for(int j=0;j<n;++j)    A2->at(i)[j]=rand()%50;
34      for(int i=0;i<n;++i)
35          for(int j=0;j<k;++j)    B2->at(i)[j]=rand()%50;
36
37      matrix_multiply(A2,B2,C2,m,n,k);
38
39
40      //double point
41      double *A3,*B3,*C3;
42      A3=(double*)malloc(m*n*sizeof(double));
43      B3=(double*)malloc(n*k*sizeof(double));
44      C3=(double*)malloc(m*k*sizeof(double));
45
46      srand(time(NULL));
47      for(int i=0;i<m;++i)
48          for(int j=0;j<n;++j)    A3[i*m+j]=rand()%50;
49      for(int i=0;i<n;++i)
50          for(int j=0;j<k;++j)    B3[i*n+j]=rand()%50;
51
52      matrix_multiply(A3,B3,C3,m,n,k);
53
54      free(A3);
55      free(B3);
56      free(C3);
57
58      return 0;
59  }
60
```

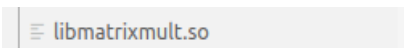
- 第4行，引入之前写的头文件，来调用我们头文件中的三个函数。
- 第10-22行，是对参数类型是二维向量的函数`matrix_multiply`进行调用。
- 第24-37行，是对参数类型是二维向量的指针的函数`matrix_multiply`进行调用。
- 第40-56行，是对参数类型是double类型的指针的函数`matrix_multiply`进行调用。

运行结果和时间

1. 首先我们先将三个实现重载的函数文件编译为动态链接库，生成libtest.so文件（它已经编译好了）：

```
1  g++ lab2_3_vector.cpp lab2_3_vectorpoint.cpp lab2_3_doublepoint.cpp --shared
   -fPIC -o libmatrixmult.so
```

生成的libtest.so文件：



2. 然后编译调用了`matrix_multiply.h`的测试文件`lab2_3_test.cpp`：

```
g++ lab2_3_test.cpp -L. -lmatrixmult -o test
```

((-L.)表示动态链接库在本目录， (-lmatrixmult)表示动态链接库的名字为lib(matrixmult).so

3. 然后通过`ldd ./test`查看有哪些动态链接库和可执行程序有关联。结果发现libmatrixmult.so没有关联:

```
wwwj@ubuntu:~/lab/lab2$ g++ lab2_3_doublepoint.cpp lab2_3_vector.cpp lab2_3_vect
orpoint.cpp -fPIC -shared -o libmatrixmult.so
wwwj@ubuntu:~/lab/lab2$ g++ lab2_3_test.cpp -L. -lmatrixmult -o test
wwwj@ubuntu:~/lab/lab2$ ldd test
        linux-vdso.so.1 (0x00007ffeefbe9000)
        libmatrixmult.so => not found
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fcb7a
426000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fcb7a20e000
)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcb79e1d000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fcb79a7f000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fcb7a9b9000)
wwwj@ubuntu:~/lab/lab2$
```

4. 然后将`libmatrixmult.so`文件挪入一般动态链接库所在的文件夹`/usr/lib/`中，然后再查看可执行文件能否找到我的动态链接文件：

```
sudo cp libmatrixmult.so /usr/lib/
```

```
wwwj@ubuntu:~/lab/lab2$ sudo cp libmatrixmult.so /usr/lib/
[sudo] password for wwwj:
wwwj@ubuntu:~/lab/lab2$ ldd test
        linux-vdso.so.1 (0x00007ffc945b5000)
        libmatrixmult.so => /usr/lib/libmatrixmult.so (0x00007fd3958c1000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fd395
538000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fd395320000
)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd394f2f000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fd394b91000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fd395cd6000)
wwwj@ubuntu:~/lab/lab2$
```

现在能够找到我的`.so`文件了。

5. 运行可执行文件：

```
wwwj@ubuntu:~/lab/lab2$ g++ lab2_3_test.cpp -L. -lmatrixmult -o test
wwwj@ubuntu:~/lab/lab2$ ./test 512 512 512
time cost = 1.42308 s
time cost = 3.22338 s
time cost = 0.644229 s
wwwj@ubuntu:~/lab/lab2$ ./test 1024 1024 1024
time cost = 11.7845 s
time cost = 26.3616 s
time cost = 7.74783 s
```

输入参数后可获得运算时间，三个有不同参数的函数运行时间不同。可以看到直接调用double类型的指针来计算（第三个）运行时间是最快的，而使用二维向量（第一个）和二维向量指针（第二个）慢了许多。

实验思考

MPI的理论和实践还是有很大的差别，实验过程中可以了解熟悉许多在课堂上没有注意到的问题，比如要求MPI函数中传入的参数是连续的，如果不注意这个问题，代码很容易就会写错。而且做实验的整个过程就是在发现问题、解决问题。发现问题的能力也是要不断学习加强的，有的bug坑是环境配置的问题，有些bug可能是代码的逻辑问题。之前在做点对点通信的时候他总是报错，但是我检查自己的代码并没有发现错误，最后才发现是之前安装过mpich和openmpi两个库，它们之间产生了冲突，在这一块我花费了许多时间。

文章参考

- MPI编程—集合通信MPI_Bcast、MPI_Gather、MPI_Scatter、MPI_Reduce
  https://blog.csdn.net/yiguagua/article/details/106908904
- MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[], MPI_Datatype *newtype) function (deino.net)
  http://mpi.deino.net/mpi_functions/MPI_Type_create_struct.html
- Linux下编写简单的动态链接库 https://blog.csdn.net/weixin_33696106/article/details/94457524)