

# 并行与分布式作业 五

年级	班级	学号	姓名
18级	计科（超算方向）	18340052	何泽

## 并行与分布式作业 五

### I. Sparse Matrix

1. 运行环境
2. 程序实现
3. 运行结果
4. 结果可视化
5. 结果分析

### II. Producer-Consumer

1. 程序实现
2. 运行结果
3. 结果可视化
4. 结果分析

### III. MPI测量通信时延和带宽

1. 程序实现
2. 运行

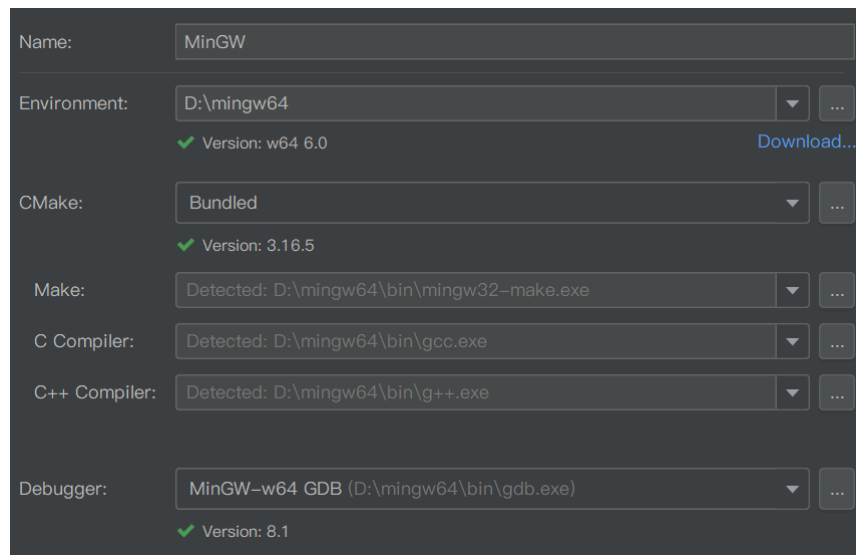
### IV. 实验总结

## I. Sparse Matrix

Consider a sparse matrix stored in the compressed row format (you may find a description of this format on the web or any suitable text on sparse linear algebra). Write an OpenMP program for computing the product of this matrix with a vector. Download sample matrices from the Matrix Market (<http://math.nist.gov/MatrixMarket/>) and test the performance of your implementation as a function of matrix size and number of threads.

### 1. 运行环境

- 我的运行系统是 Win10，编译器为 MinGW-w64，版本如下：



- 我使用的 IDE 为 JetBrains 的 Clion，该 IDE 需要写 CMakeLists，内容如下：

```
1 cmake_minimum_required(VERSION 3.16)
2 project(Sparse_Matrix)
3
4 set(CMAKE_CXX_STANDARD 20)
5 set(CMAKE_CXX_FLAGS "-fopenmp")
6
7 add_executable(Sparse_Matrix main.cpp)
8 add_executable(Sparse_Matrix2 main2.cpp)
```

其中 c++ 标准为 c++20，并指明需要 openmp，此外有两个 cpp 文件，main.cpp 为同一矩阵用不同线程数运行，main2.cpp 为不同大小的矩阵用相同线程数运行。

- CPU 为 i7-9750H，有 6 核 12 线程

基准速度:	2.59 GHz
插槽:	1
内核:	6
逻辑处理器:	12
虚拟化:	已启用
L1 缓存:	384 KB
L2 缓存:	1.5 MB
L3 缓存:	12.0 MB

## 2. 程序实现

- 首先矩阵从[https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/psadmit/1138\\_bus.html](https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/psadmit/1138_bus.html)下载，第一行为信息，第二行为行数、列数以及非零元素个数，从第三行开始每行为一个元素，格式为行数+列数+元素值；
- 那么首先将文件内容读到矩阵中来，并输出行数、列数和非零元素个数

```

1 ifstream matrix;
2 matrix.open("D:/1138_bus.txt");
3 cout << "Reading from the file" << endl;
4 char a[100];
5 matrix.getline(a,100); //读第一行信息
6 int row,col,num;
7 matrix>>row>>col>>num; //读行数、列数和非零元素个数
8 cout<<"There are "<<row<<" rols and "<<col<<" cows. "<<endl;
9 cout<<"There are "<<num<<" entries."<<endl;
10 static double m[size][size]; //将文件中的矩阵读到m中
11 for(int i=1;i<=num;i++){
12     int p,q;
13     double b;
14     matrix>>p>>q>>b;
15     m[p][q]=b;
16 }

```

- 然后随机生成元素值为 0-10 的向量 n

```

1 static double n[size];
2 for(int i=1;i<=row;i++) {
3     n[i] = rand()%10;
4 }

```

- 然后调用 `omp_get_num_procs()` 输出 CPU 线程数（非物理核心数）

```

1 cout<<"There are "<<omp_get_num_procs()<<" cores in this computer."<<endl;

```

- 为了让效果更明显，我将矩阵向量乘法进行 10 次，并计时,计时的函数 `GET_TIME` 如下（我写成了 `define` 的形式）：

```

1 #define GET_TIME(now) { \
2     struct timeval t; \
3     gettimeofday(&t, NULL); \
4     now = t.tv_sec + t.tv_usec/1000000.0; \
5 }

```

- 首先是串行计算，并输出 10 次的运行时间

```

1 int test_times=10;
2 GET_TIME(start);
3 for(int w=0;w<test_times;w++) {
4     for (int i = 1; i <= row; i++) {
5         for (int u = 1; u <= col; u++) {
6             double buf = 0;
7             for (int y = 1; y <= row; y++) {
8                 buf += n[y] * m[y][u];
9             }
10            res[i][u] = buf;
11        }
12    }
13 }
14 GET_TIME(end);
15 double t1=end-start;

```

```
16 | cout<<"Serial run time: "<<t1<<"s."<<endl;
```





- 然后使用 **OpenMP** 计算，此外最外层循环表示线程数从 **2-18** 递增，每一个线程数的值计算一次并输出该线程数时的运行时间以及相对于串行计算的加速比

```
1 |     for (int thread_num=2;thread_num<=18;thread_num++) { //线程数递增循环
2 |         GET_TIME(start);
3 |         for(int w=0;w<test_times;w++) { //计算10次
4 |             for (int i = 1; i <= row; i++) {
5 | #pragma omp parallel for num_threads(thread_num)schedule(guided)
6 |                 for (int u = 1; u <= col; u++) {
7 |                     double buf = 0;
8 |                     for (int y = 1; y <= row; y++) {
9 |                         buf += n[y] * m[y][u];
10 |                     }
11 |                     res[i][u] = buf;
12 |                 }
13 |             }
14 |         }
15 |         GET_TIME(end);
16 |         double t2 = end - start;
17 |         cout << "OpenMP run time: " << t2 << "s with " << thread_num << "
cores running. SpeedUP is "<<t1/t2<<". "<< endl;
18 |     }
```

以上便是同一矩阵用不同线程数运行的设计，对于不同大小的矩阵用相同线程数运行，我下载了<https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/psadmit/psadmit.html>下的四种稀疏矩阵，如下图：

Four symmetric matrices used in the modeling of power system networks.

Matrices in this set:

-  [662 BUS](#) (real symmetric indefinite symmetric positive definite, 662 by 662, 1568 entries), Power system networks
-  [494 BUS](#) (real symmetric indefinite symmetric positive definite, 494 by 494, 1080 entries), Power system networks
-  [685 BUS](#) (real symmetric indefinite symmetric positive definite, 685 by 685, 1967 entries), Power system networks
-  [1138 BUS](#) (real symmetric indefinite symmetric positive definite, 1138 by 1138, 2596 entries), Power system networks

算法设计与上面相同，只不过是读不同的文件。

### 3. 运行结果

- 同一矩阵用不同线程数运行，输出不同线程数的运行时间以及相对于串行计算的加速比

```
Run: Sparse_Matrix x
D:\CLionProjects\Sparse-Matrix\cmake-build-debug\Sparse_Matrix.exe
Reading from the file
There are 1138 rols and 1138 cows.
There are 2596 entries.
There are 12 cores in this computer.
Serial run time: 71.4002s.
OpenMP run time: 34.6003s with 2 cores running. SpeedUP is 2.06357.
OpenMP run time: 20.7806s with 3 cores running. SpeedUP is 3.4359.
OpenMP run time: 15.3325s with 4 cores running. SpeedUP is 4.65679.
OpenMP run time: 12.0547s with 5 cores running. SpeedUP is 5.92303.
OpenMP run time: 10.3829s with 6 cores running. SpeedUP is 6.87669.
OpenMP run time: 11.0066s with 7 cores running. SpeedUP is 6.48702.
OpenMP run time: 11.9277s with 8 cores running. SpeedUP is 5.9861.
OpenMP run time: 11.7853s with 9 cores running. SpeedUP is 6.05843.
OpenMP run time: 11.5312s with 10 cores running. SpeedUP is 6.19194.
OpenMP run time: 11.408s with 11 cores running. SpeedUP is 6.25878.
OpenMP run time: 11.3458s with 12 cores running. SpeedUP is 6.29309.
OpenMP run time: 11.5096s with 13 cores running. SpeedUP is 6.20354.
OpenMP run time: 11.6642s with 14 cores running. SpeedUP is 6.12131.
OpenMP run time: 11.3249s with 15 cores running. SpeedUP is 6.30473.
OpenMP run time: 11.4035s with 16 cores running. SpeedUP is 6.26124.
OpenMP run time: 11.5054s with 17 cores running. SpeedUP is 6.20579.
OpenMP run time: 11.463s with 18 cores running. SpeedUP is 6.22877.

Process finished with exit code 0
```

- 不同大小的矩阵用相同线程数运行，输出不同矩阵大小所对应的运行时间

```
Run: Sparse_Matrix2 x
D:\CLionProjects\Sparse-Matrix\cmake-build-debug\Sparse_Matrix2.exe
*****
Reading from the file 1138_bus.txt.
There are 1138 rols and 1138 cows.
There are 2596 entries.
OpenMP run time: 11.3523s with 6 cores running.
*****
*****
Reading from the file 685_bus.txt.
There are 685 rols and 685 cows.
There are 1967 entries.
OpenMP run time: 1.86467s with 6 cores running.
*****
*****
Reading from the file 662_bus.txt.
There are 662 rols and 662 cows.
There are 1568 entries.
OpenMP run time: 1.71292s with 6 cores running.
*****
*****
Reading from the file 494_bus.txt.
There are 494 rols and 494 cows.
There are 1080 entries.
OpenMP run time: 0.761967s with 6 cores running.
*****
*****
Process finished with exit code 0
```

#### 4. 结果可视化

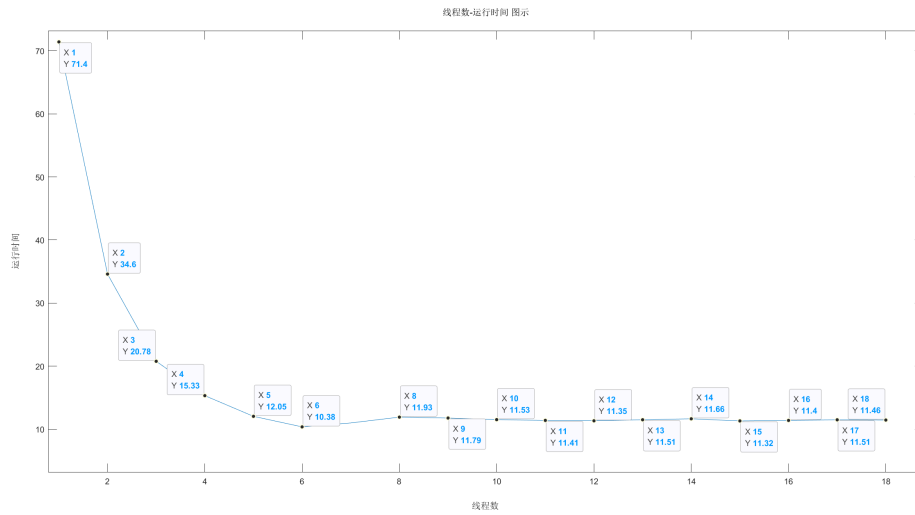
- 我将上面的运行结果在 **Matlab** 中输出可视化结果图像，命令如下：

```

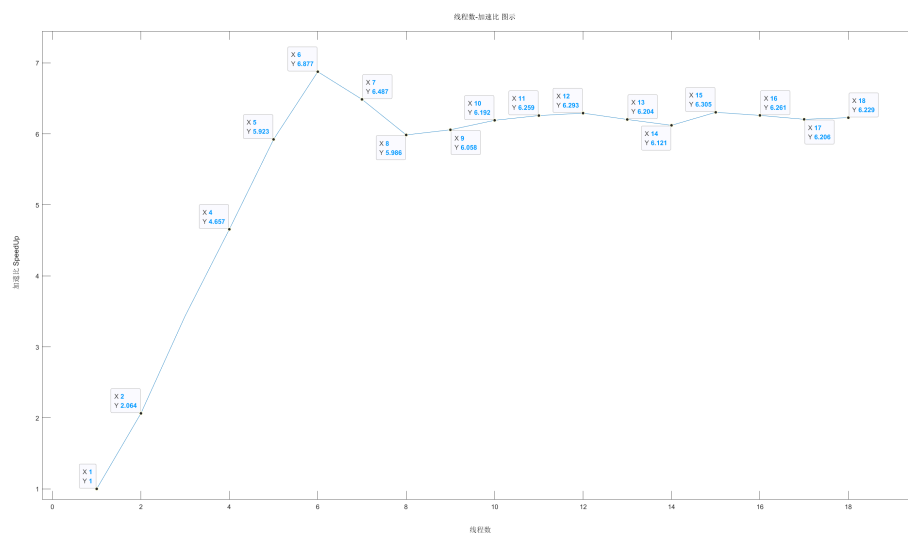
命令窗口
>> core=[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18];
>> time1=[71.4002 34.6003 20.7806 15.3325 12.0547 10.3829 11.0066 11.9277 11.7853 11.5312 11.408 11.3458 11.5096 11.6642 11.3249 11.4035 11.5054 11.463];
>> plot(core,time1)
>> sd=[1 2.06357 3.4359 4.65679 5.92303 6.87669 6.48702 5.9861 6.05843 6.19194 6.25878 6.29309 6.20354 6.12131 6.30473 6.26124 6.20579 6.22877];
>> plot(core,sd)
>> en=[1138 685 662 494];
>> t2=[11.3523 1.86467 1.71292 0.761967];
>> plot(en,t2)
>>

```

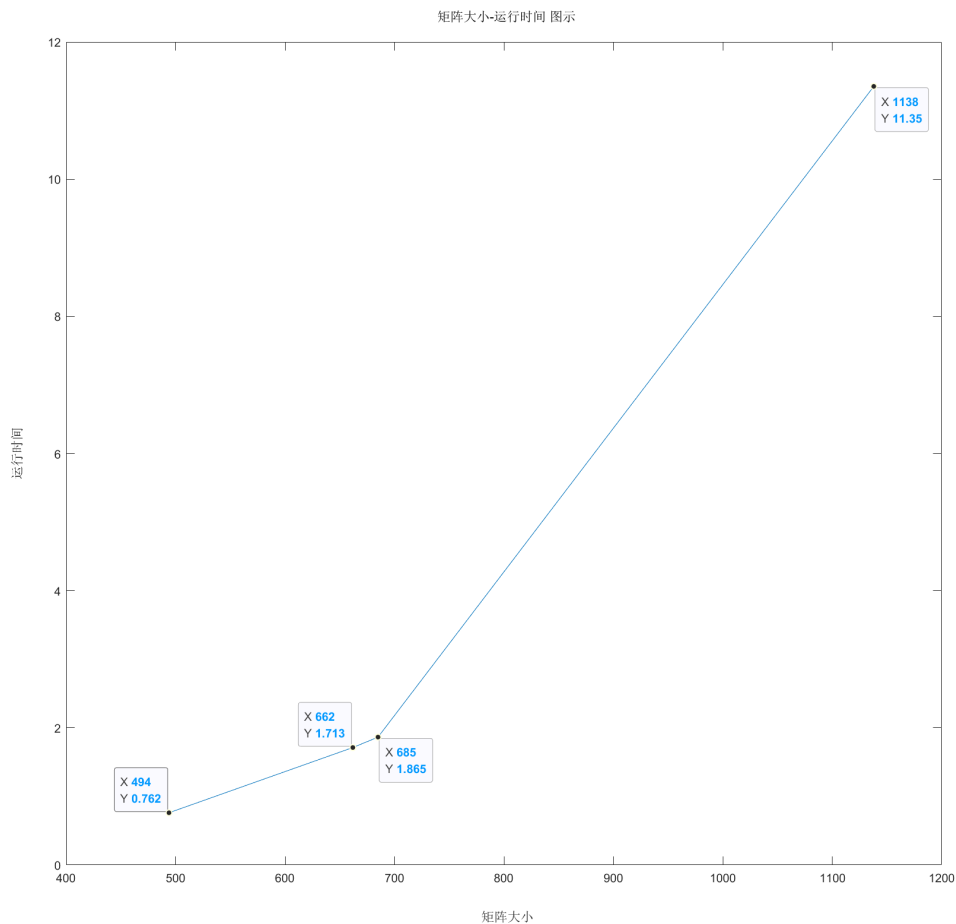
- 线程数—运行时间



- 线程数—加速比



- 矩阵大小-运行时间



## 5. 结果分析

- 对于相同大小的矩阵，由于我的计算机有 6 个物理核心，所以以 6 个线程运行的时候速度最快，加速比最高，而超过 6 个线程之后由于线程间通信的时间开销导致加速比比 6 个的时候略小，但 6-18 个速度接近，加速比趋于平稳
- 而对于不同大小的矩阵，矩阵越大运行时间越多

## II. Producer-Consumer

Implement a producer-consumer framework in OpenMP using sections to create a single producer task and a single consumer task. Ensure appropriate synchronization using locks. Test your program for a varying number of producers and consumers.

### 1. 程序实现

- 首先设计一个类代表缓冲区，然后根据缓冲区的状态制定生产者消费者规则

- 首先，生产者、消费者不能对缓冲区同时访问，实现方法是用 OpenMP 的 `critical` 对缓冲区加锁，同一时间只能有一个线程访问
- 之后定义一个变量 `p` 表示缓冲区元素最大个数，定义变量 `b` 表示现有元素个数，用来记录缓冲区用来判断缓冲区是否满或者空，如果缓冲区空，`b==0`，消费者不能访问，忙等 `sleep`；缓冲区满，`b==p-1`，生产者不能访问，忙等 `sleep`。
- 上面就是缓冲区的实现描述，代码如下：

```

1  const int p = 1000;
2
3  class buf {
4  public:
5      buf():b(0){}
6      void push(int n){
7          while(b==(p - 1))
8              sleep(1);
9      #pragma omp critical
10         {
11             a[b++]=n;
12         }
13     }
14     void pop(){
15         while(b==0)
16             sleep(1);
17     #pragma omp critical
18         {
19             b--;
20         }
21     }
22 private:
23     int a[p];
24     int b;
25 }t;

```

- 生产者、消费者就是调用上面的push和pop

```

1  void producer(int n) {
2      for (int i = 0; i < n; ++i)
3          t.push(i);
4  }
5
6  void consumer(int n) {
7      for (int i = 0; i < n; ++i)
8          t.pop();
9  }

```

- 主函数就是调用上面两个函数，为了测试生产者消费者不同数量，我用变量`n`代表，从25开始每次循环乘2，直到800，并输出运行时间

```

1  for(int n=25;n<=800;n*=2) {
2      cout<<"The number of Producer-Consumer is "<<n<<". ";

```

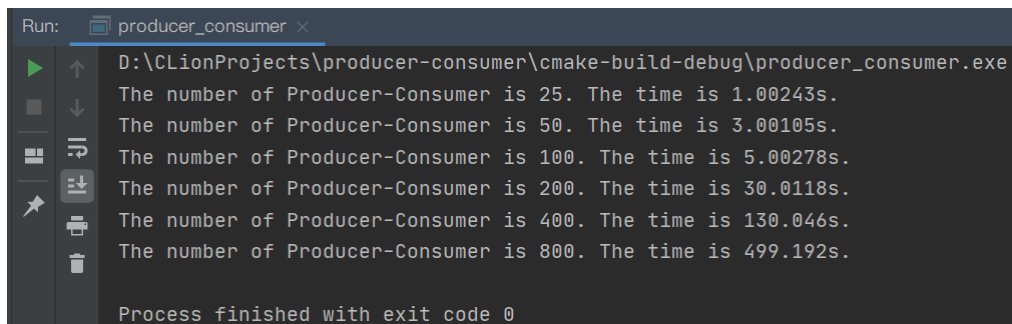


```

3     double start, end;
4     GET_TIME(start);
5     #pragma omp parallel sections
6     {
7     #pragma omp section
8     {
9     #pragma omp parallel for
10        for (int i = 0; i < n; ++i) {
11            producer(i);
12        }
13    }
14    #pragma omp section
15    {
16    #pragma omp parallel for
17        for (int i = 0; i < n; ++i) {
18            consumer(i);
19        }
20    }
21    }
22    GET_TIME(end);
23    cout<<"The time is "<<end-start<<"s."<<endl;
24 }

```

## 2. 运行结果



```

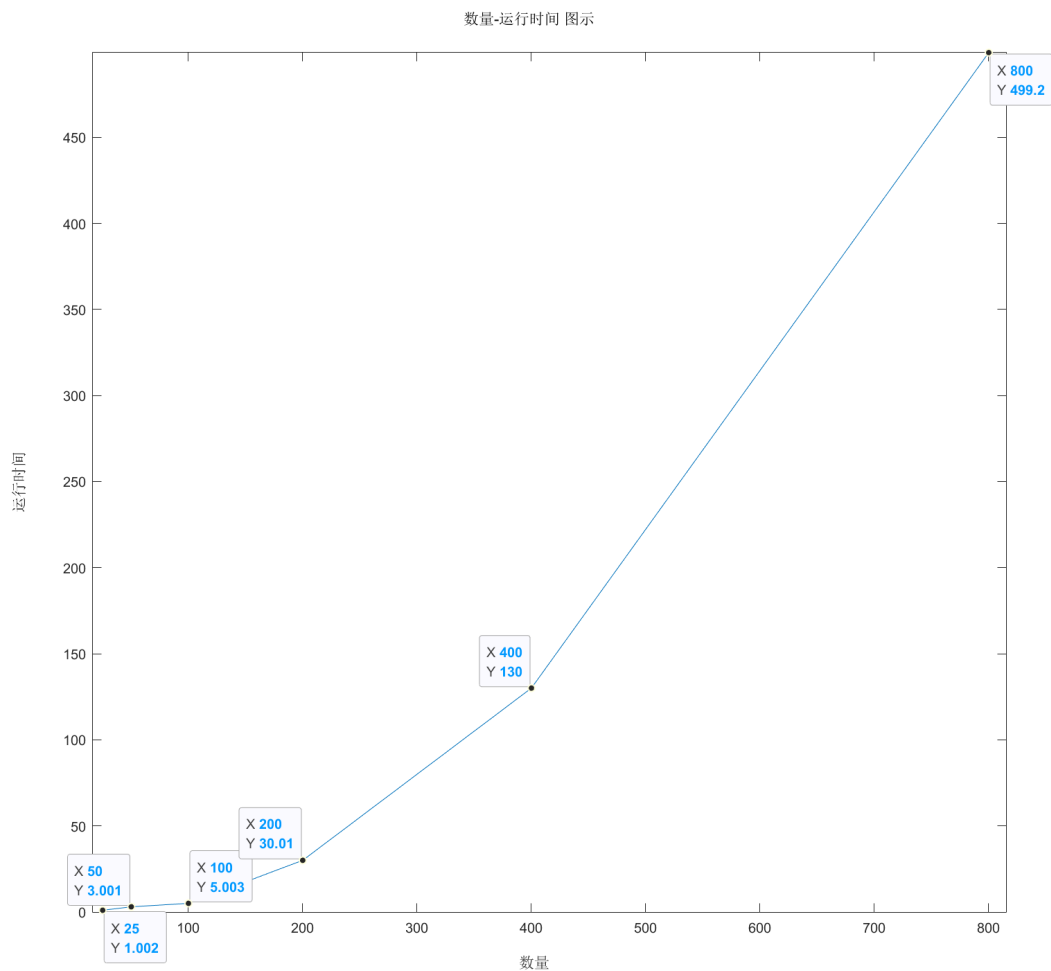
Run: producer_consumer x
D:\CLionProjects\producer-consumer\cmake-build-debug\producer_consumer.exe
The number of Producer-Consumer is 25. The time is 1.00243s.
The number of Producer-Consumer is 50. The time is 3.00105s.
The number of Producer-Consumer is 100. The time is 5.00278s.
The number of Producer-Consumer is 200. The time is 30.0118s.
The number of Producer-Consumer is 400. The time is 130.046s.
The number of Producer-Consumer is 800. The time is 499.192s.

Process finished with exit code 0

```

## 3. 结果可视化

和上一个一样，使用Matlab输出图像结果：



#### 4. 结果分析

可以看出，随着数量的增加，运行时间增长得也越来越快。

### III. MPI测量通信时延和带宽

利用MPI通信程序测试本地进程以及远程进程之间的通信时延和带宽。

#### 1. 程序实现

- 总体思路就是两个线程之间传送数据，一个发一个收，统计发送的大小和接收需要的时间，从而计算出带宽和延时。
- 所以要求进程数必须是2的倍数，并提前分配好组合，最后统计平均值得出结果
- 主要代码如下：

- 首先，主进程打印一些相关信息

```
1 if (my_rank == 0) {
2     printf("There are %d threads totally in the test.\n", size);
3     printf("The message size is %d bytes.\n", BUF_SIZE);
4     printf("-----\n");
5 }
```

- 对于前一半进程，发送并接收数据后计算时间并发送到主进程

```
1 double best = .0, worst = .99E+99, total = .0;
2 double total_time = .0;
3 for (int i = 0; i < TEST_TIMES; ++i) {
4     double nbytes = sizeof(char) * BUF_SIZE;
5     double start_time = MPI_Wtime();
6     MPI_Send(&buffer, BUF_SIZE, MPI_CHAR, partner_rank, tag,
7 MPI_COMM_WORLD);
8     MPI_Recv(&buffer, BUF_SIZE, MPI_CHAR, partner_rank, tag,
9 MPI_COMM_WORLD, &status);
10    double end_time = MPI_Wtime();
11    double run_time = end_time - start_time;
12    double bw = (2 * nbytes) / run_time;
13    total += bw;
14    best = bw > best ? bw : best;
15    worst = bw < worst ? bw : worst;
16    total_time += run_time;
17 }
18 best /= 1000000.0;
19 worst /= 1000000.0;
20 double avg_bw = (total / 1000000.0) / TEST_TIMES;
21 total_time /= TEST_TIMES;
```

接下来，判断是不是主进程即rank=0，如果不是，则给主进程发送时间信息

```
1 double tmp_timings[4];
2 tmp_timings[0] = best;
3 tmp_timings[1] = avg_bw;
4 tmp_timings[2] = worst;
5 tmp_timings[3] = total_time;
6 MPI_Send(tmp_timings, 4, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
```

如果是主进程，则接收各个进程发送过来的信息，并打印每一组的信息，最后，计算各组的平均值得出结果

```
1 timings[0][0] = best;
2 timings[0][1] = avg_bw;
3 timings[0][2] = worst;
4 timings[0][3] = total_time;
5
6 double best_all = .0, worst_all = .0, avg_all = .0;
7 double time_all = .0;
8 for (int j = 1; j < size / 2; ++j) {
9     MPI_Recv(&timings[j], 4, MPI_DOUBLE, j, tag, MPI_COMM_WORLD,
10 &status);
11 }
```

```

11 for (int j = 0; j < size / 2; ++j) {
12     printf("Test between %d and %d, best bandwidth is %lfMBps, worst
    bandwidth is %lfMBps, average bandwidth is %lfMBps, time is %lfs.\n", j,
    task_pair[j], timings[j][0], timings[j][2], timings[j][1], timings[j]
    [3]);
13     best_all += timings[j][0];
14     avg_all += timings[j][1];
15     worst_all += timings[j][2];
16     time_all += timings[j][3];
17 }
18     printf("-----
    -----\n");
19     printf("Averagely, best bandwidth is %lfMBps, worst bandwidth is
    %lfMBps, average bandwidth is %lfMBps, time is %lfs.\n", best_all / (size
    / 2), worst_all / (size / 2), avg_all / (size / 2), time_all / (size /
    2));
20     printf("-----
    -----\n");
21 }

```

- 对于后一半进程，只需要先接收数据，再发出去即可

```

1 for (int i = 0; i < TEST_TIMES; ++i) {
2     MPI_Recv(&buffer, BUF_SIZE, MPI_CHAR, partner_rank, tag,
    MPI_COMM_WORLD, &status);
3     MPI_Send(&buffer, BUF_SIZE, MPI_CHAR, partner_rank, tag,
    MPI_COMM_WORLD);
4 }

```

## 2. 运行

因为我没有多个节点，所以运行测试是在单机本地测试的，远程测试也只需要写host文件即可，代码都是一样的，所以本地测试原理一致，就用本地测试代替了。

- 使用 `mpicc` 编译再用 `mpirun` 运行，结果如下：

```

heze@ubuntu:~/mpi$ mpicc mpitest.cpp -o mpitest
heze@ubuntu:~/mpi$ mpirun -n 12 /home/heze/mpi/mpitest
There are 12 threads totally in the test.
The message size is 100000 bytes.
-----
Test between 0 and 6, best bandwidth is 11491.243836MBps, worst bandwidth is 704.333165MBps, average bandwidth is 10735.273121MBps, time is 0.000026s.
Test between 1 and 7, best bandwidth is 11491.243836MBps, worst bandwidth is 7.865328MBps, average bandwidth is 10272.282055MBps, time is 0.000819s.
Test between 2 and 8, best bandwidth is 10754.625641MBps, worst bandwidth is 10.561273MBps, average bandwidth is 9665.675823MBps, time is 0.000612s.
Test between 3 and 9, best bandwidth is 11491.243836MBps, worst bandwidth is 1076.843132MBps, average bandwidth is 10383.549489MBps, time is 0.000025s.
Test between 4 and 10, best bandwidth is 10894.296104MBps, worst bandwidth is 1200.086981MBps, average bandwidth is 10234.716494MBps, time is 0.000024s.
Test between 5 and 11, best bandwidth is 10754.625641MBps, worst bandwidth is 17.101808MBps, average bandwidth is 9639.080212MBps, time is 0.000385s.
-----
Averagely, best bandwidth is 11146.213149MBps, worst bandwidth is 502.798615MBps, average bandwidth is 10155.096199MBps, time is 0.000315s.
-----

```

我用了12个线程跑，每一组的结果都已输出，最后得到平均带宽为 `10155MBps`，延为 `0.000315s`，测试结束。

## IV. 实验总结

---

这次实验使用了 `OpenMP` 和 `MPI` 完成了很多任务，让我对并行编程更加了解，使用起来也更加熟练。