

并行与分布式计算

第五次作业

姓名：张洪宾

班级：18 级计科超算

学号：18340208

1 问题描述

1.1

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.

1.2

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.

1.3

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。

2 实验环境

- 8 核心 16G 内存服务器

服务器操作系统 Ubuntu 20.04

gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2)

- Macbook Pro 2015

虚拟机: Parallels Desktop 下 Ubuntu 18.04

用两台虚拟机 node1 和 node2 搭建了一个简单的集群

3 解决方案

3.1 稀疏矩阵的向量乘法

3.1.1 稀疏矩阵

稀疏矩阵主要分为三元组，Compressed Row Storage(CRS) 格式和 Compressed Col Storage(CCS) 格式。三元组就是简单的将非 0 元素的坐标用三维的向量来表示，而后面两者则是抽取出行或列，将行或列的数组 Row 或 Col 改造成为表示第 i 行的首元素的位置。

对于稀疏矩阵，假设有 N 个非零元素，然后有 r 行和 c 列，三元组占用的空间是 $3 * N$ ，而 CRS 和 CCS 占用的空间是 $2 * N + r$ 和 $2 * N + c$ 。

当 $r \approx c \ll N \ll r * c$ 的时候，选择 CRS 或 CSS 较为合适。题目要求我们使用 CRS，经过分析也很好理解：我们可以非常直接地按照行划分问题，如果采用三元组的话，则我们无法准确地判断三元组之间会不会对结果的一个特定的位置造成写冲突。而使用 CRS 可以完美地避开这个问题。

3.1.2 结构体的定义

在这里我使用了 C 语言编写程序，结构体中包括了矩阵的行数、列数、非 0 元素的个数，然后还有三个指针，分别指向 CRS 的某一维向量。

结构体如下：

```
1 struct compressed_matrix
2 {
3     int row_size;
4     int col_size;
5     int element_size;
6     int* row;
7     int* col;
8     double* element;
9 };
```

然后我写了一个初始化函数，它可以根据流中的稀疏矩阵，初始化矩阵结构体，并将矩阵返回。这部分代码如下：

```
1 struct compressed_matrix init_matrix(FILE*stream){
2     int row_size,col_size,element_size;
3     fscanf(stream,"%d%d%d",&row_size,&col_size,&element_size);
4     struct compressed_matrix mat;
```

```

5     mat.col_size = col_size;
6     mat.row_size = row_size;
7     mat.element_size = element_size;
8     mat.row = (int*)malloc(row_size * sizeof(int));
9     mat.col = (int*)malloc(element_size * sizeof(int));
10    mat.element = (double*)malloc(element_size * sizeof(double));
11    for(int i = 0; i < row_size; i++){
12        fscanf(stream, "%d", &mat.row[i]);
13        mat.row[i]--;
14    }
15    for(int i = 0; i < element_size; i++){
16        fscanf(stream, "%d", &mat.col[i]);
17        mat.col[i]--;
18    }
19    for(int i = 0; i < element_size; i++){
20        fscanf(stream, "%lf", &mat.element[i]);
21    }
22    return mat;
23 }

```

流中的矩阵首先应该先包括三个整型数，矩阵的行数，列数，以及非零元素的个数。然后分别输入 CRS 稀疏矩阵格式对应的三个维度的向量。

3.1.3 写出串程序

在这里我们根据 CRS 的结构和矩阵的乘法很容易的推理出，对于有 r 行 c 列的矩阵 A ，当它与一个 c 维向量 v 相乘的时候，只需要将所有非 0 的元素 $A[i, j]$ ，结果向量记为 res 并初始化全部元素为 0。

则对于所有满足 $A[i, j] \neq 0$ 的元素， $res[i] += A[i, j] * c[j]$ 即可。

而对于 CRS 格式，我们可以将矩阵的同一行的元素做一个聚拢，也就是说在一个循环体中计算完矩阵一行的非零向量。

串程序的代码如下：

```

1 double* serial(struct compressed_matrix matrix, double* vector){
2     double*res = (double*)malloc(matrix.row_size * sizeof(double));
3     memset(res, 0, matrix.row_size * sizeof(double));
4     for(int i = 0; i < matrix.row_size; i++){
5         int begin = matrix.row[i];
6         int end;

```

```

7         if(i < matrix.row_size - 1)
8             end = matrix.row[i + 1] - 1;
9         else
10             end = matrix.element_size - 1;
11         for(int j = begin; j <= end; j++){
12             res[i] += vector[matrix.col[j]] * matrix.element[j];
13         }
14     }
15     return res;
16 }

```

3.1.4 对串行代码做并行化

在这里就体现了使用 CRS 格式的优势。如果采用三元组，则我们用 openMP 直接对串行程序修改做并行化，因为三元组我们会遍历所有非零元素，然后分配到每个循环体，这样子一来就可能会出现多个循环体修改结果向量中的同一个元素，就可能会出现数据竞争。

而在上面的串行代码中，第一层循环在不同的循环体之间不存在数据相关，不同的循环体对结果向量的不同的值进行修改，而第二层循环彼此之间存在数据相关，所以不能并行化。

所以我们只需要对串行的代码稍作修改就可以得到并行的代码：

```

1 double* parallel(struct compressed_matrix matrix, double* vector,
2     int thread_count){
3     double* res = (double*)malloc(matrix.row_size * sizeof(double));
4     memset(res, 0, matrix.row_size * sizeof(double));
5     #pragma omp parallel for num_threads(thread_count)
6     for(int i = 0; i < matrix.row_size; i++){
7         int begin = matrix.row[i];
8         int end;
9         if(i < matrix.row_size - 1)
10             end = matrix.row[i + 1] - 1;
11         else
12             end = matrix.element_size - 1;
13         for(int j = begin; j <= end; j++){
14             res[i] += vector[matrix.col[j]] * matrix.element[j];
15         }
16     }
17     return res;
18 }

```

3.2 生产者消费者问题

我们可以用一条队列来存储资源，生产者将资源放入资源队列中，然后消费者取出队列进行使用。这样子就可能会出现一个问题，就是如果这两者并行执行的话，队列就是一个临界资源，如果稍有不慎就会造成错误。所以我们需要用一个信号量来保护住这个队列。而如果队列为空的时候，如果消费者还试图从队列中取出不存在的资源，会出现错误，所以也需要一个信号量来做队列是否为空的标志。

3.2.1 单个生产者和单个消费者的模型实现

首先根据上面的分析，我确定了使用两个信号量，用 n 表示队列中资源的个数，如果 n 变成了 0，则会让消费者等待，直到生产者继续生产将 n 变回 1。而 s 则用于控制对 n 和队列的读写，防止出现写冲突。

于是最核心的生产者函数和消费者函数如下：

```
1 void producer(){
2     while(true){
3         resources x = produce();
4         sem_wait(&s);
5         all_resources.push(x);
6         printf("Resources %d is pushed.\n",x);
7         sem_post(&s);
8         sem_post(&n);
9     }
10 }
11 void consumer(){
12     while(true){
13         resources x;
14         sem_wait(&n);
15         sem_wait(&s);
16         x = all_resources.front();
17         all_resources.pop();
18         sem_post(&s);
19         consume(x);
20     }
21 }
```

划分问题的时候，因为目前只有一个生产者和一个消费者，所以我们可以采用 section 的方式，让两个 section 并行执行 producer 函数和 consumer 函数，代码如下：

```

1  #pragma omp parallel sections
2  {
3      #pragma omp section
4      {
5          producer();
6      }
7      #pragma omp section
8      {
9          consumer();
10     }
11 }

```

值得注意的是，信号量 n 应该初始化为 0，信号量 s 应该初始化为 1，这样子就可以保证最开始执行的是 produce 的部分，从未不会访问未定义的资源。

3.2.2 多个生产者和消费者模型

在单个生产者消费者模型中我们定义的 producer 函数和 consumer 函数还可以在这里继续使用，在多个生产者和消费者的模型中，生产者和生产者之间可以用信号量 s 来防止数据竞争，而消费者与消费者之间同样使用信号量 s 来防止数据竞争，生产者与消费者之间则使用信号量 n 来防止数据竞争。代码如下：

然后我们可以采用 openMP 的 parallel for 和 for 循环搭配，然后规定生产者和消费者的数量，然后开辟足够多的线程，给每个生产者和消费者都分配一个，然后根据线程 id 来确定它是生产者线程还是消费者线程。

```

1  #pragma omp parallel num_threads(p_count + c_count)
2  {
3      int id = omp_get_thread_num();
4      #pragma omp parallel sections
5      {
6          #pragma omp section
7          {
8              if(id < p_count)
9                  producer(id);
10             }
11             #pragma omp section
12             {
13                 if(id >= p_count)

```

```
14         consumer(id);
15     }
16 }
17 }
```

3.3 利用 MPI 通信来测试本地进程以及远程进程之间的通信时延和带宽

本来我想用尝试在服务器上使用 docker 来尝试搭建一个 MPI 的简单集群，但是由于实验室的空调坏了，实验室集群停机，服务器不能使用了，所以我只能在本地尝试用虚拟机搭建集群。

我在我的电脑上装了两个 Ubuntu 虚拟机，命名为 node1 和 node2。配置过程如下：

3.3.1 单个结点环境的搭建

在单个的结点是，我安装了 MPICH，ssh，nfs-kernel-server 等实验过程中需要用到的工具。

3.3.2 查询 IP 地址

在这里 Parallels Desktop 使用的是静态 IP，所以直接将 ifconfig 查到的 IP 地址拿来用就行了。

用 ifconfig 查询得到 node1 的 IP 是 10.211.55.25，node2 的 IP 是 10.211.55.26。

3.3.3 实现 ssh 免密码登录

在这里需要我们修改/etc/hosts 来使得 ssh 可以识别结点的名称。如图，分别在两个结点中用 sudo vim /etc/hosts，在其中加入 node1 和 node2 及它们的 IP 地址。

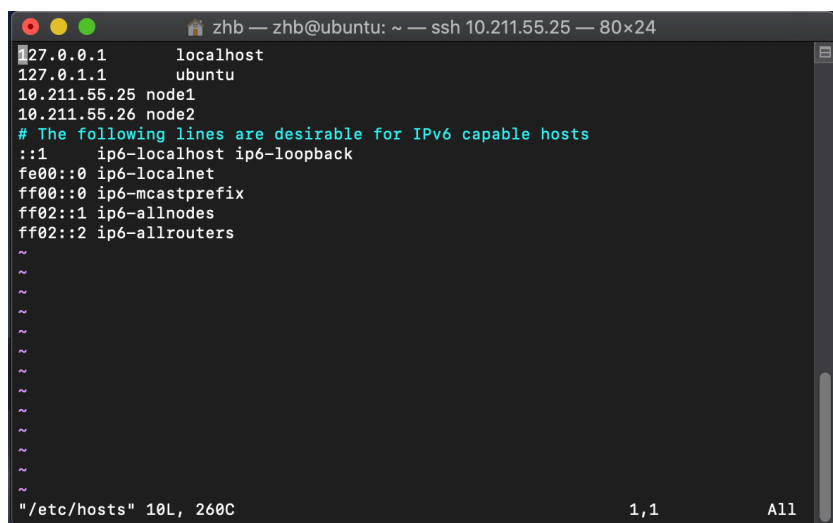
A terminal window titled 'zhh — zhh@ubuntu: ~ — ssh 10.211.55.25 — 80x24'. The terminal displays the content of the /etc/hosts file. The first four lines map IP addresses to hostnames: 127.0.0.1 to localhost, 127.0.1.1 to ubuntu, 10.211.55.25 to node1, and 10.211.55.26 to node2. A comment line follows: '# The following lines are desirable for IPv6 capable hosts'. Then, several IPv6 addresses are mapped to their respective names: ::1 to ip6-localhost and ip6-loopback, fe00::0 to ip6-localnet, ff00::0 to ip6-mcastprefix, ff02::1 to ip6-allnodes, and ff02::2 to ip6-allrouters. The rest of the file contains tilde (~) characters. At the bottom, the terminal shows the file path '/etc/hosts' with a size of 10L and a count of 260C. The status bar at the bottom right shows '1,1' and 'All'.

图 1: 修改/etc/hosts

然后在所有结点中都使用 `ssh-keygen -t rsa`, 将本机的钥匙保存在 `~/.ssh` 下, 然后用 `cat ~/.ssh/id_rsa.pub>>~/.ssh/authorized_keys` 认证。

再然后, 将 node1 作为主结点, 将 node2 和其他子结点 (如果存在的话) 的 `id_rsa.pub` 传到主结点 node1, 使用的命令如下:

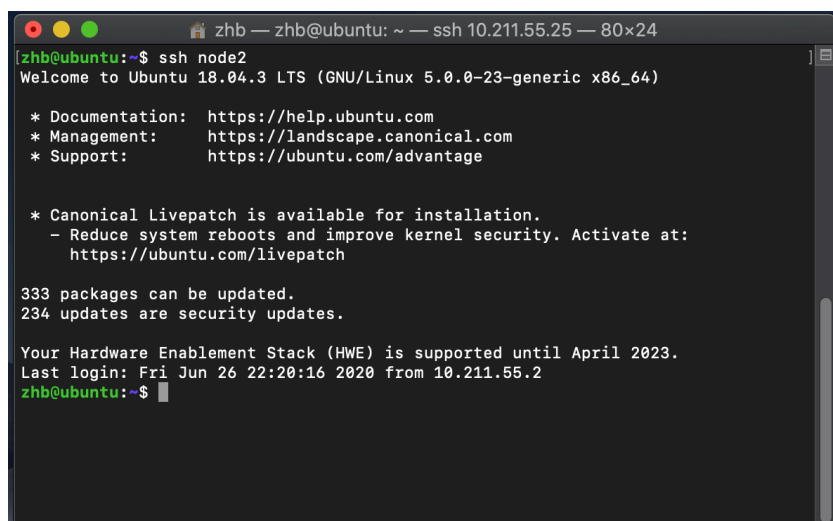
```
scp id_rsa.pub zhh@node1:~/.ssh/id_rsa.pub.node2
```

然后在主结点上, 再用以下两条命令认证并将认证文件传回每个子结点:

```
cat ~/.ssh/id_rsa.pub.node2>>~/.ssh/authorized_keys
```

```
scp authorized_keys zhh@node2:~/.ssh/authorized_keys
```

然后就可以实现免密登录了:



```
zhhb@ubuntu:~$ ssh node2
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 5.0.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

333 packages can be updated.
234 updates are security updates.

Your Hardware Enablement Stack (HWE) is supported until April 2023.
Last login: Fri Jun 26 22:20:16 2020 from 10.211.55.2
zhhb@ubuntu:~$
```

图 2: 在 node1 用 ssh node2 命令直接登录 node2

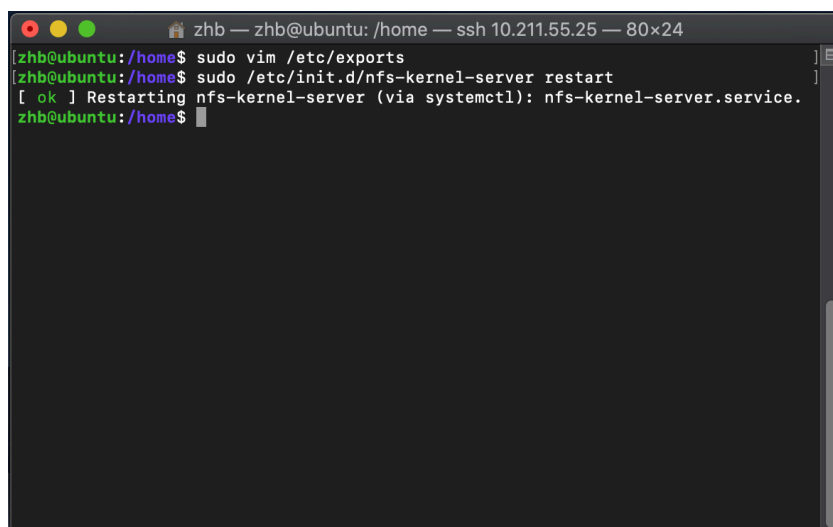
3.3.4 建立 NFS 共享目录并挂载

NFS(Network Files System) 即网络文件系统。它可以用于允许网络中的主机通过 TCP/IP 协议进行资源共享, 使得 NFS 客户端可以像使用本地资源一样读写远端 NFS 服务端的资料, 在这里我们要让 node1 的 MPI 程序可以调用 node2 的资源。

我在 node1 和 node2 的 /home 目录下新建来 nfs_dir 文件夹, 然后在 node1 使用 sudo vim /etc/exports, 来设置 NFS 服务器。

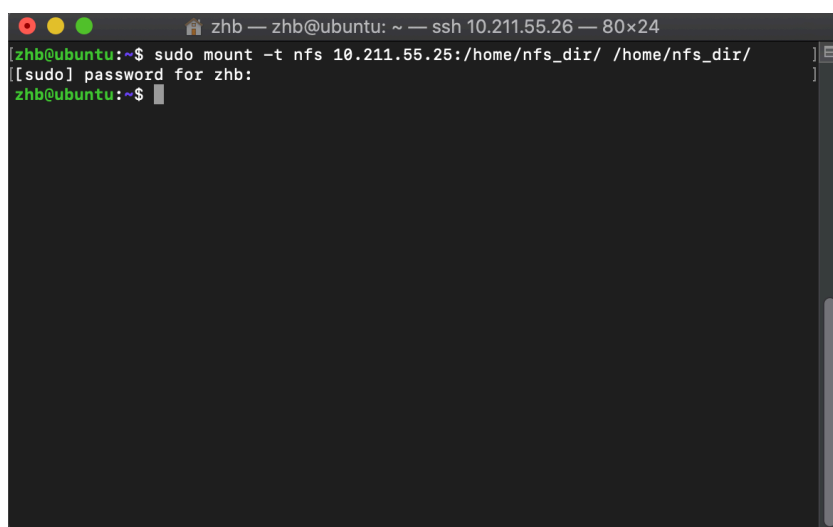
在其中加入一行 /home/nfs_dir *(rw,sync,no_root_squash,no_subtree_check), 然后用 sudo /etc/init.d/nfs-kernel-server restart 重启 NFS,

看到下面的提示, 说明 NFS 服务器配置完毕。



```
zhh — zhh@ubuntu: /home — ssh 10.211.55.25 — 80x24
[zhh@ubuntu:~/home$ sudo vim /etc/exports
[zhh@ubuntu:~/home$ sudo /etc/init.d/nfs-kernel-server restart
[ ok ] Restarting nfs-kernel-server (via systemctl): nfs-kernel-server.service.
zhh@ubuntu:~/home$
```

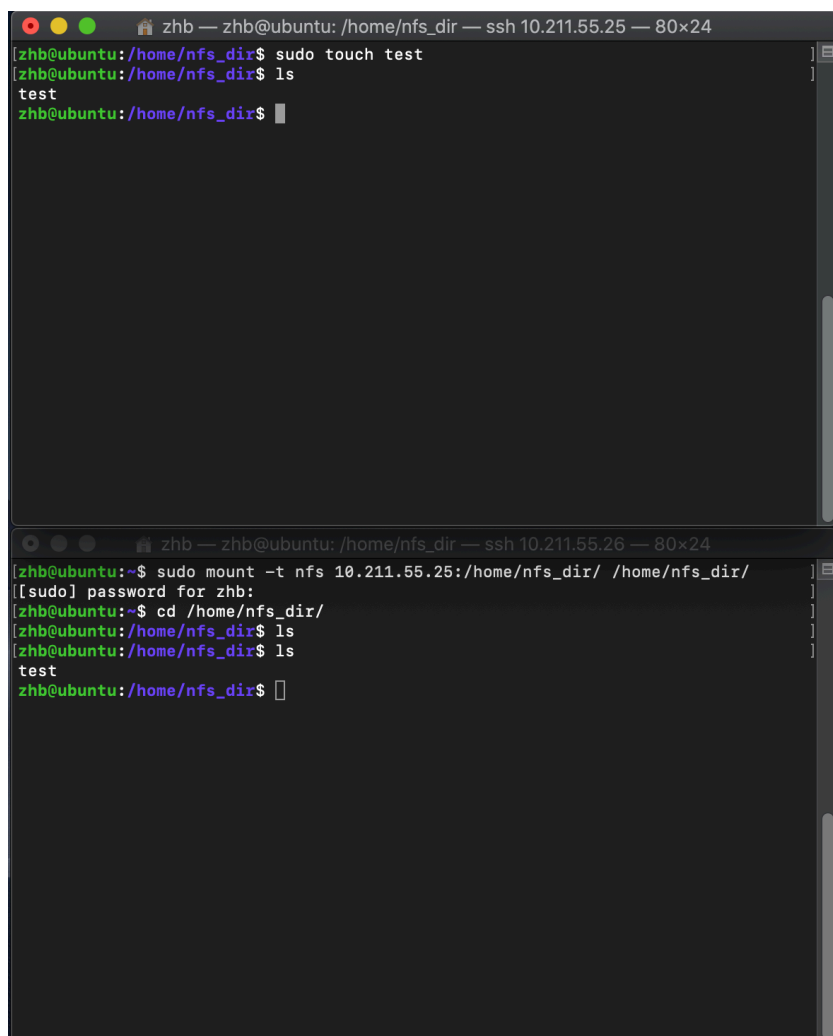
然后在所有的子结点中（在这里只有 node2）输入 `sudo mount -t nfs 10.211.55.25:/home-nfs_dir/ /home/nfs_dir/`



```
zhh — zhh@ubuntu: ~ — ssh 10.211.55.26 — 80x24
[zhh@ubuntu:~$ sudo mount -t nfs 10.211.55.25:/home/nfs_dir/ /home/nfs_dir/
[sudo] password for zhh:
zhh@ubuntu:~$
```

图 3: 在 node 2 挂载 node1 的 NFS 共享目录到本地

此时 `/home/nfs_dir/` 目录为空，在 node1 用 `sudo touch test` 创建一个空文件 `test`，再在 node2 的 `/home/nfs_dir/` 中查看，可以找到 node1 创建的文件，说明成功配置来 NFS。



The image contains two terminal window screenshots. The top window shows a user 'zhh' at 'zhh@ubuntu: /home/nfs_dir' connected via 'ssh 10.211.55.25'. The commands executed are 'sudo touch test', 'ls' (showing 'test'), and 'test'. The bottom window shows the same user at 'zhh@ubuntu: /home/nfs_dir' connected via 'ssh 10.211.55.26'. The commands executed are 'sudo mount -t nfs 10.211.55.25:/home/nfs_dir/ /home/nfs_dir/', '[sudo] password for zhh:', 'cd /home/nfs_dir/', 'ls' (showing 'test'), and 'test'.

```
zhh@ubuntu: /home/nfs_dir — ssh 10.211.55.25 — 80x24
zhh@ubuntu:/home/nfs_dir$ sudo touch test
zhh@ubuntu:/home/nfs_dir$ ls
test
zhh@ubuntu:/home/nfs_dir$

zhh@ubuntu: /home/nfs_dir — ssh 10.211.55.26 — 80x24
zhh@ubuntu:~$ sudo mount -t nfs 10.211.55.25:/home/nfs_dir/ /home/nfs_dir/
[sudo] password for zhh:
zhh@ubuntu:~$ cd /home/nfs_dir/
zhh@ubuntu:/home/nfs_dir$ ls
test
zhh@ubuntu:/home/nfs_dir$
```

图 4: 检验 NFS 是否成功配置

到这里，一个简单的集群就搭建完毕了。然后就可以开始写 MPI 程序来测试了。

3.3.5 编写 MPI 程序

在这里的 MPI 程序比较简单，我在上次作业的基础上稍微修改了一下，对 p 个进程（在这里 p 默认为 2），测量从主线程发送 `BUFLen` 长度的字节到各个子进程的时间。具体的代码如下：

```

1  #include <mpi.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define BUFLen 1000000
5
6  char buf[BUFLen];
7  int main(int argc , char **argv)
8  {
9      int my_id = 0;
10     int p;
11     MPI_Init(&argc , &argv);
12     MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
13     MPI_Comm_size(MPI_COMM_WORLD, &p);
14     MPI_Status status_p;
15     MPI_Barrier(MPI_COMM_WORLD);
16     double my_start,my_end,my_elapsed,elapsed;
17     my_start = MPI_Wtime();
18     if(my_id == 0){
19         for(int i = 1;i < p;i++){
20             MPI_Send(buf,BUFLen,MPI_CHAR,i,i,MPI_COMM_WORLD);
21         }
22     }
23     else{
24         MPI_Recv(buf,BUFLen,MPI_CHAR,0,my_id,MPI_COMM_WORLD,&status_p);
25     }
26     my_end = MPI_Wtime();
27
28     my_elapsed = my_end - my_start;
29
30     MPI_Reduce(&my_elapsed,&elapsed,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
31     if(my_id == 0){
32         printf("Time delay is %f s\n",elapsed);
33         printf("Bandwidth is %f Mbit/s\n", BUFLen * 1.0 / (1048576* elapsed));
34     }
35     MPI_Finalize();
36     return 0;
37 }

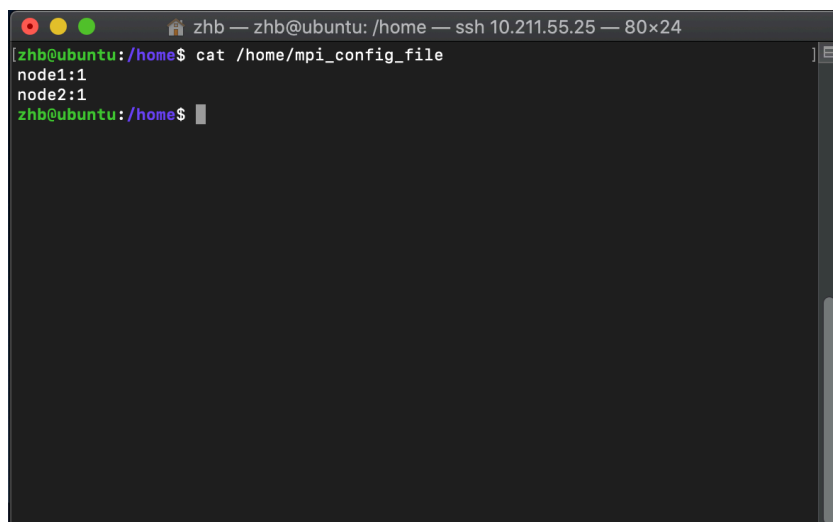
```

我们只要让运行时参数为 2，就可以测量我们需要的时延迟和带宽。

3.3.6 设置 MPI 的 Host 文件

MPI 使用 Host 文件来确定不同结点和结点的核心数量，用于将任务分配到结点上，可以用 `mpirun -f HostFile` 来使用它。

在 `/home` 文件夹下我创建了 `mpi_config_file`，然后在其中写入了分配给结点的核心。由于我的资源有限，我只给每个结点分配了一个核心，具体的文件如下：

A terminal window with a dark background and light text. The title bar shows 'zhh — zhh@ubuntu: /home — ssh 10.211.55.25 — 80x24'. The prompt is '[zhh@ubuntu: /home\$]'. The user has entered 'cat /home/mpi_config_file' and the output is 'node1:1' followed by 'node2:1' on the next line. The prompt is now 'zhh@ubuntu: /home\$' with a cursor.

```
[zhh@ubuntu: /home$ cat /home/mpi_config_file
node1:1
node2:1
zhh@ubuntu: /home$
```

图 5: 我的 Host 文件

至此，整个环境配置完毕。

4 实验结果

4.1 稀疏矩阵的向量乘法

为了比较不同的线程数和不同的矩阵规模对结果的影响，我在 MatrixMarket 上下载了 3 个不同规模的矩阵，将头部的冗余信息删除后，在不同线程数下执行我的出现，先测量了串行计算所需要的时间，然后再测量并行计算所需要的时间，通过比较来分析结果。

三个矩阵分别是 E05R0100、1138_bus.rsa 和 e40r5000.rua。三个矩阵的规模分布是：

矩阵	行数	列数	非零元素个数
E05R0100	236	236	5856
1138_bus.rsa	1138	1138	2596
e40r5000.rua	17281	17281	553956

4.1.1 编译并运行

在这里我设置了运行参数，第一个参数是线程的数量，第二个参数是矩阵的文件名。编译后执行，示例如下：

```
[ubuntu@ubuntu-20-04:~/parallel$ gcc 1.c -o 1 -fopenmp
[ubuntu@ubuntu-20-04:~/parallel$ ./1 2 E05R0100

Serial_time is:92µs

There are 2 threads.
Parallel_time is:238µs
[ubuntu@ubuntu-20-04:~/parallel$ ./1 4 E05R0100

Serial_time is:91µs

There are 4 threads.
Parallel_time is:357µs
[ubuntu@ubuntu-20-04:~/parallel$ ./1 8 E05R0100

Serial_time is:92µs

There are 8 threads.
Parallel_time is:752µs
ubuntu@ubuntu-20-04:~/parallel$ █
```

图 6: 编译并运行示例

4.1.2 运行结果

矩阵名称 线程数量	E05R0100	1138_bus.rsa	e40r5000.rua
2	92 μs	90 μs	2174 μs
4	91 μs	81 μs	2055 μs
8	92 μs	80 μs	2069 μs

表 1: 串行时的计算时间

矩阵名称 线程数量	E05R0100	1138_bus.rsa	e40r5000.rua
2	238 μs	242 μs	3748 μs
4	357 μs	438 μs	2098 μs
8	752 μs	726 μs	1298 μs

表 2: 并行时的计算时间

4.1.3 结果分析

由于在 Matrix Market 找不到规模特别大的矩阵，所以并行的效果并没有非常显著。但是从运行结果可以看出一些端倪。当数据比较小的时候，如矩阵 E05R0100 和矩阵 1138_bus.rsa，我们可以很容易看到，并行时间比串行的时间小，且随着线程数量的增加，并行所需的时间增加。而将行数和列数增加都增加约 10 倍后，在矩阵 e40r5000.rua 中，可以看到，串行时间增加了非常多，而并行的时间增加得较少，且在这时，并行计算的时间随着线程数量的增加而减少。这也在预料之中，并行程序的加速需要考虑线程调度的开销，所以如果问题规模不够大，增加线程数量对性能不增反降。

4.2 生产者消费者问题

4.2.1 单个生产者和单个消费者的情况

编译后运行，结果如下：


```
zhh — zhh@ubuntu: ~/Desktop/Q2 — ssh 10.211.55.10 — 80x24
Resources 8226 is pushed.
Resources 8227 is pushed.
Resources 4440 is popped.
Resources 8228 is pushed.
Resources 8229 is pushed.
Resources 8230 is pushed.
Resources 8231 is pushed.
Resources 8232 is pushed.
Resources 4441 is popped.
Resources 8233 is pushed.
Resources 4442 is popped.
Resources 8234 is pushed.
Resources 8235 is pushed.
Resources 4443 is popped.
Resources 8236 is pushed.
Resources 4444 is popped.
Resources 4445 is popped.
Resources 8237 is pushed.
Resources 4446 is popped.
Resources 8238 is pushed.
Resources 4447 is popped.
Resources 8239 is pushed.
Resources 8240 is pushed.
Resources 4448 is popped.
```

图 7: 执行单个生产者和单个消费者程序

可以看到生产者不断地向资源队列中放资源，然后消费者也会从中取出资源。

4.2.2 多个生产者和多个消费者的情况

在这里我令生产者个数为 8，消费者个数为 4，然后编译运行并执行程序，结果如下：

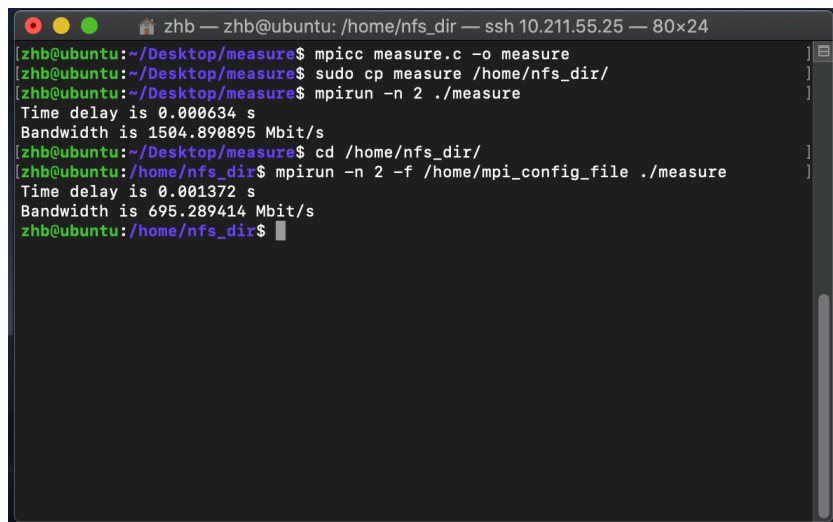
```
zhh — zhh@ubuntu: ~/Desktop/Q2 — ssh 10.211.55.10 — 80x24
Resources 3 is popped by thread 8.
Resources 5 is pushed by thread 4.
Resources 6 is pushed by thread 7.
Resources 4 is popped by thread 10.
Resources 5 is popped by thread 9.
Resources 7 is pushed by thread 6.
Resources 6 is popped by thread 11.
Resources 8 is pushed by thread 1.
Resources 7 is popped by thread 8.
Resources 9 is pushed by thread 2.
Resources 8 is popped by thread 10.
Resources 10 is pushed by thread 4.
Resources 9 is popped by thread 9.
Resources 11 is pushed by thread 3.
Resources 10 is popped by thread 11.
Resources 12 is pushed by thread 7.
Resources 13 is pushed by thread 2.
Resources 11 is popped by thread 10.
Resources 12 is popped by thread 10.
Resources 14 is pushed by thread 4.
Resources 13 is popped by thread 9.
Resources 15 is pushed by thread 3.
Resources 14 is popped by thread 11.
Resources 15 is popped by thread 10.
```

图 8: 执行多个生产者和多个消费者程序

其中 0 至 8 号线程为生产者线程，其余的线程是消费者线程，可以看到，不同的生产者不断地像资源队列放入资源，不同的消费者不断地从资源队列中取出资源。

4.3 利用 MPI 通信来测试本地进程以及远程进程之间的通信时延和带宽

搭建完环境后的工作显得格外的简单。我设定 BUFLLEN 为 1000000，先在 node1 编译程序，并运行一次程序，然后将可执行文件拷贝到 NFS 目录下，然后再 NFS 目录下选择 Host 文件，运行程序，结果如下：



```
zhh — zhh@ubuntu: /home/nfs_dir — ssh 10.211.55.25 — 80x24
[zhh@ubuntu:~/Desktop/measure$ mpicc measure.c -o measure
[zhh@ubuntu:~/Desktop/measure$ sudo cp measure /home/nfs_dir/
[zhh@ubuntu:~/Desktop/measure$ mpirun -n 2 ./measure
Time delay is 0.000634 s
Bandwidth is 1504.890895 Mbit/s
[zhh@ubuntu:~/Desktop/measure$ cd /home/nfs_dir/
[zhh@ubuntu:/home/nfs_dir$ mpirun -n 2 -f /home/mapi_config_file ./measure
Time delay is 0.001372 s
Bandwidth is 695.289414 Mbit/s
zhh@ubuntu:/home/nfs_dir$
```

图 9: 测量本地进程和远程进程之间的通信时延和带宽

如图，可以看出，在本地的进程之间的通信时延是 0.000634s，带宽为 1504Mbit/s。在远程进程之间的通信时延是 0.001372s，带宽为 695Mbit/s。

可以看出，远程通信比本地通信的费时更高，符合我们的预期。

5 遇到的问题及解决方法

这次实验的比较复杂，尤其是第三题，搭建环境的时候遇到了非常多的问题。最开始我配置 NFS 的时候，我是将 node1 和 node2 的 IP 作为网段的，但是重启 NFS 的时候出现了 fail，我通过反复查找资料，将这个 IP 地址改成 *，改成监视所有网段。

然后还有一些大大小小的问题，不过也通过查找资料解决了。