

# 《并行与分布式计算》实验报告

（2024年6月版）

**郑重声明：**

**1、本报告中的所有内容均由本人合作独立编码、调试和测试。本报告中给出的实验数据和结果均由本人所完成的程序得出。**

**2、本人了解：不按照前述要求所完成的实验报告已经构成了抄袭或造假行为，本人将承担相应的不良后果。**

**卡通人物

描述已自动生成**

**姓名 （签名） 学号 202321046098**

**成绩**

MPI是高性能计算的基本编程方法。请使用MPI在多机系统上实现奇偶排序、PSRS(Parallel Sorting by Regular Sampling)算法，并进行性能分析和比较。

### 1、MPI计算环境的搭建（5分）

你的集群包括了 2 个节点，每个节点的配置是 如备注1 。

你使用的MPI平台是 openmpi-4.1.6 。

MPI搭建的过程包括： 如备注2 。

下文中设MPI的进程数为*P*，参与排序的总元素数量为*N*。

**硬件设置要求：至少包含两个节点。**

**备注1：**

1. 节点1（ubuntu节点）配置

|  |  |  |
| --- | --- | --- |
| **节点参数** | | **具体配置** |
| 操作系统 | | Ubuntu 20.04.6 LTS |
| CPU | CPU型号 | Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz |
| 核心数 | 40 |
| 每核线程数 | 2 |
| L1d cache | 640KB |
| L1i cache | 640KB |
| L2 cache | 5MB |
| L3 cache | 50MB |
| 最小频率 | 1.20GHz |
| 最大频率 | 3.10GHz |
| 内存 | 物理内存大小 | 251GB |
| 最大频率 | 3100MHz |
| 虚拟内存大小 | 8GB |
| 网络最大带宽(两根网线) | | 250MB/s |

1. 节点2（pc节点）配置

|  |  |  |
| --- | --- | --- |
| **节点参数** | | **具体配置** |
| 操作系统 | | Ubuntu 20.04.6 LTS |
| CPU | CPU型号 | Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz |
| 核心数 | 80 |
| 每核线程数 | 2 |
| L1d cache | 1.3MB |
| L1i cache | 1.3MB |
| L2 cache | 40MB |
| L3 cache | 55MB |
| 最小频率 | 0.8GHz |
| 最大频率 | 4.0GHz |
| 内存 | 物理内存大小 | 251GB |
| 最大频率 | 4500MHz |
| 虚拟内存大小 | 8GB |
| 网络最大带宽(一根网线) | | 125MB/s |

**备注2：**

**（注意：两个节点上都需要运行mpi平台安装步骤与多节点配置步骤）**

1. **openmpi平台安装**
2. 首先卸载系统原有的mpi平台，具体参照各平台卸载方法。
3. 安装具体版本MPI，实验中选用的是openmpi-4.1.6，运行bash脚本即可，脚本内容如下，最后需要输入sudo权限用户密码。
4. #!/bin/bash
5. wget https://download.open-mpi.org/release/open-mpi/v4.1/openmpi-4.1.6.tar.gz
6. tar -zvxf openmpi-4.1.6.tar.gz
7. cd openmpi-4.1.6
8. ./configure --prefix=/usr/local/openmpi
9. make -j64
10. sudo make install
11. 配置环境变量，在用户目录（例如：/home/tanwen）下的.bashrc文件中添加以下内容：
12. export PATH="$PATH:/usr/local/openmpi/bin"
13. export LD\_LIBRARY\_PATH="$LD\_LIBRARY\_PATH:/usr/local/openmpi/lib/"

添加后可以运行source .bashrc或者重开终端mpi即可使用。

1. **openmpi多节点配置**

**（注意：这里多节点默认是相同用户名，且用户拥有sudo权限）**

1. 往/etc/hosts文件中添加主机名与对应IP地址（需要sudo权限），添加内容例如：

1. 202.38.247.203 ubuntu
2. 202.38.247.204 pc

2. 需要为每台服务器配置ssh密钥登录，这里不详细介绍ssh密钥生成方法，具体实现就是生成一对密钥，包含公钥和私钥，将公钥放置在用户目录的.ssh文件夹中，命名为authorized\_keys，将私钥放在同一个位置，命名为id\_rsa。为了操作简洁，这里两个节点使用同一套公钥和私钥，操作可以如下：

1. mkdir ~/.ssh
2. cd ~/.ssh/
3. ssh-keygen -t rsa
4. cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys
5. chmod 600 ~/.ssh/authorized\_keys
6. chmod 600 ~/.ssh/id\_rsa

在任意一个节点执行上述操作，然后将生成的authorized\_keys文件与id\_rsa文件复制到另一个节点相同目录下，赋予同样的600权限。

3. 接着运行在各节点上运行ssh 主机名（例如 ssh pc）指令配置节点信息。

4. 所有节点都需要关闭防火墙，运行下列指令：

1. sudo systemctl stop firewalld

5. 还有一点需要注意，多节点运行时，可执行文件需要在不同节点上的相同位置，这里可以通过配置NFS共享目录来完成，但在本次实验中，我采取的是手动复制代码到节点相同位置上，再采用同样编译指令，将可执行文件生成在不同节点的相同位置。

**备注3：**

所有代码都已上传到github仓库，网址：<https://github.com/xiaotantanya/MPI_EXP.git>。

### 2、数据的产生和串行快速排序的性能（10分）

请使用程序-1产生*N*个随机浮点数，并写入数据文件，其中*seed*为外部输入的随机数种子，需要设置为(0,1.0)区间中的浮点数。

float rand\_float(float s){

return 4\*s\*(1-s);

}

void matrix\_gen(float \*a,uint64\_t N,float seed){

float s=seed;

for(uint64\_t i=0;i<N;i++){

s=rand\_float(s);

a[i]=s;

}

}

程序-1 随机数产生

**注意：在实验中，初始随机种子我设置为0.344。**

2.1 使用C语言中的qsort函数按照递增方式对*N*个浮点数从小到大排序。测量排序所需要的时间（不包括文件读入时间，仅仅包含排序时间，下同）。（5分）

表-1 节点1（ubuntu节点）上串行排序的时间

|  |  |
| --- | --- |
| *N* | 排序时间（s） |
| 256M | 48.576996 |
| 1G | 193.835233 |
| 4G | 817.400633 |

排序命令行截图：

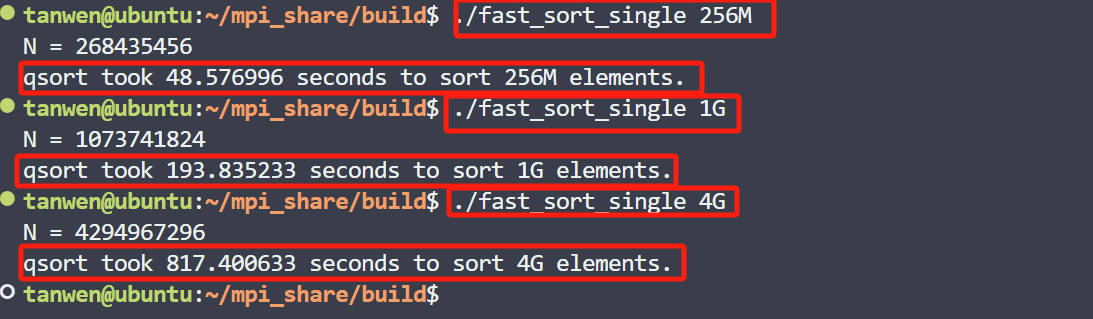


表-2 节点2（pc节点）上串行排序的时间

|  |  |
| --- | --- |
| *N* | 排序时间（s） |
| 256M | 39.391448 |
| 1G | 164.013461 |
| 4G | 684.248653 |

排序命令行截图：

图片包含 图形用户界面

描述已自动生成

2.2 请分析当数据量增加时，排序时间的变化规律。（5分）

首先我们知道快速排序的时间复杂度是，而从256M到1G，从1G到4G，数据量都是4倍关系。我们可以大概计算一下它们运行时间之间的关系：

上述实验结果也验证了这一点，运行256M和1G浮点数排序耗时时间倍数大约为（节点1为3.990倍，节点2为4.164倍），而运行1G和4G浮点数排序耗时时间倍数大约为（节点1为4.217倍，节点2为4.172倍）。

### 3、奇偶排序（30分）

3.1 请简述奇偶排序算法。（5分）

奇偶排序（Odd-Even Sort）是一种简单的并行排序算法，适用于并行计算环境。它的基本思想是通过交替进行奇数位和偶数位的比较和交换来逐步将数组排序。下面是该算法的工作原理：

1. **初始化**：给定一个待排序，长度为的数组。
2. **奇数阶段**：比较并交换索引为奇数的相邻元素。即比较和交换。
3. **偶数阶段**：比较并交换索引为偶数的相邻元素。即比较和交换 。
4. **重复**：重复奇数阶段和偶数阶段一起共次。

串行复杂度分析：1）**时间复杂度**：最坏情况下，奇偶排序的时间复杂度为 ，其中是数组的长度。这是因为每次迭代最多进行 次比较和交换，而在最坏情况下可能需要进行 次迭代。2）**空间复杂度**：奇偶排序是一种原地排序算法，空间复杂度为 。

**奇偶排序算法的特点是易于并行化**，这意味着在并行计算环境中可以高效地实现和执行。在奇偶排序中，有两个主要的阶段：奇数阶段和偶数阶段。在每个阶段中，例如在奇数阶段，所有奇数索引和其相邻元素的比较和交换都是独立的，可以并行执行。还有一种并行方式就是可以通过将数据分成多个部分，每个进程负责一部分数据，并在本地先对其拥有的部分数据进行排序（这里我们使用快速排序方法），然后进程间进行奇偶排序，当两个进程进行比较时，使用归并排序计算出每个进程自己需要的部分（从而减少进程间数据通信）。这种方法通常用于分布式计算环境，如多核处理器或集群计算。

在本次实验中，我们采取后一种，适用于分布式计算环境的并行方法。

3.2 对于*P*个MPI进程上*N*个元素的奇偶排序，其计算复杂度是多少？进程之间通信的数据量有多少？（请使用*N*和*P*表示）。（10分）

对于个元素的排序，假设能够整除，每个MPI进程会分到个元素，**由于我每个进程内局部排序采用的是快速排序法**，所以每个进程单独排序时间复杂度为，每个进程空间复杂度为。

接着每个进程间使用奇偶排序法，在最坏情况下需要经过次迭代，每次迭代最多需要进行次比较和交换，因此时间复杂度为，由于两个进程需要先合并它们的数据并放到一个能存放个浮点数的临时区域内，因此空间复杂度为。

所以，基于多个进程的奇偶排序时间复杂度为，也就是。空间复杂度为，也就是。

对于进程间的奇偶排序，首先需要将0号进程生成的随机数据分块并发送给不同进程，0号进程需要发送个浮点数，其他进程需要接收个浮点数。

然后进程间奇偶排序需要进行P次迭代，每个进程得到另一个进程的数据后使用归并排序计算出自己需要的部分，不需要计算另一个进程的部分（从而减少进程间数据通信），每次迭代每个进程需要发送和接收个浮点数。

但进程间奇偶排序有个特殊情况，对于边缘两个进程，也就是0号进程和P-1号进程，当进程总数为奇数时，0号进程只会发送和接收个浮点数，而P-1号进程则会发送和接收个浮点数。当进程总数为偶数时，0号进程和P-1号进程则会都发送和接收个浮点数。

最后除0进程外的所有进程排序好的数据需要收集到0号进程，除0进程外每个进程需要发送个浮点数，0号进程需要接收个浮点数.

总结如下：

1）当进程总数为奇数时：

|  |  |  |
| --- | --- | --- |
| 进程ID | 发送浮点数数目 | 接收浮点数数目 |
| 0 |  |  |
| 1 |  |  |
| … |  |  |
| P-1 |  |  |

2）当进程总数为偶数时：

|  |  |  |
| --- | --- | --- |
| 进程ID | 发送浮点数数目 | 接收浮点数数目 |
| 0 |  |  |
| 1 |  |  |
| … |  |  |
| P-1 |  |  |

3.3 请使用MPI完成奇偶排序算法，实际测量各个步骤的计算时间和进程之间通信的数据量，并与前述理论分析进行对比。（5分）

在实验中，我们将该排序过程分为1）原始数据分发；2）局部排序；3）进程间奇偶排序；4）排序数据收集。并在256M数据量，4进程基础上记录各步骤的计算时间和进程之间的通信量。

|  |  |  |
| --- | --- | --- |
| **排序步骤** | **进程ID** | **计算时间（s）** |
| 原始数据分发 | 0 | 4.883609 |
| 局部排序 | 0 | 11.463303 |
| 1 | 11.462927 |
| 2 | 11.460780 |
| 3 | 11.462673 |
| 进程间奇偶排序 | 0，1，2，3 | 10.934026 |
| 排序数据收集 | 0 | 4.853328 |
| 总排序时间 | 0，1，2，3 | 32.146442 |

排序过程中，每个进程数据的发送量和接收量：

|  |  |  |
| --- | --- | --- |
| **进程ID** | **发送量（Byte）** | **接收量（Byte）** |
| 0 | 1342177280 = 1.25 GB | 1342177280 = 1.25 GB |
| 1 | 1342177280 = 1.25 GB | 1342177280 = 1.25 GB |
| 2 | 1342177280 = 1.25 GB | 1342177280 = 1.2s5 GB |
| 3 | 805306368 = 768 MB | 805306368 = 768 MB |

结论分析：

1）耗时分析：从数据中可以看出，原始数据分发与排序数据收集耗时基本相同，这也非常正常。而耗时的主力则是局部排序和进程间奇偶排序，两者开销非常大，最后由于排序函数调用的开销也记录到总排序时间，导致排序步骤时间之和不等于总排序时间。

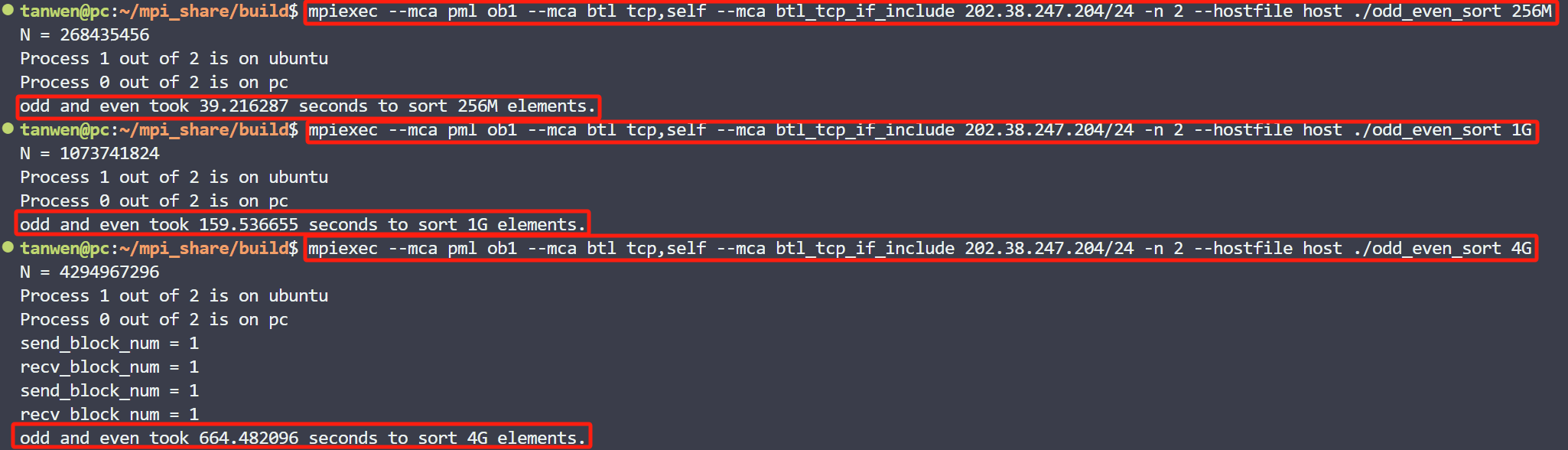
2）进程通信数据分析：奇偶排序并行的进程间通信数据量理论上是固定的，公式如上所介绍。由于一个浮点数占用4 Byte空间。根据公式计算可以发现，实验结果完全符合理论计算。

3.4 测试不同进程数和不同数据规模下的排序时间，结果如表-2所示。（5分）

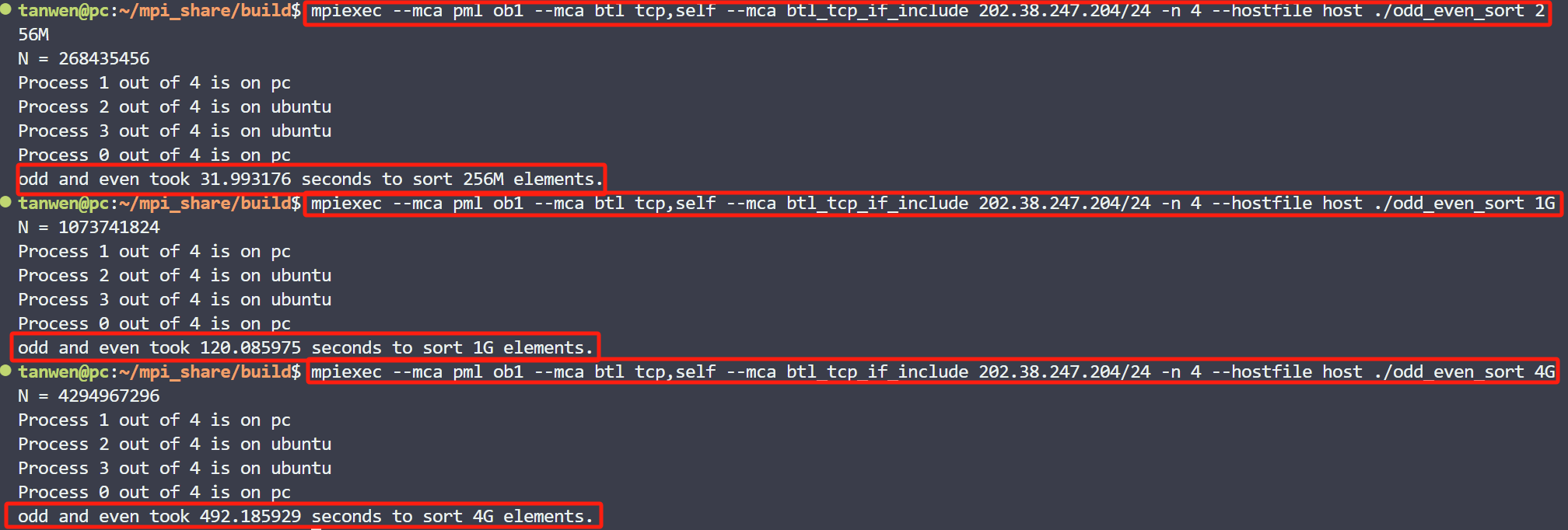
表-2 奇偶排序并行排序的时间

|  |  |  |
| --- | --- | --- |
| *N* | *P* | 奇偶并行排序时间(s) |
| 256M | 2 | 39.216287 |
| 4 | 31.993176 |
| 8 | 24.270029 |
| 1G | 2 | 159.536655 |
| 4 | 120.085975 |
| 8 | 97.700582 |
| 4G | 2 | 664.482096 |
| 4 | 492.185929 |
| 8 | 396.730125 |

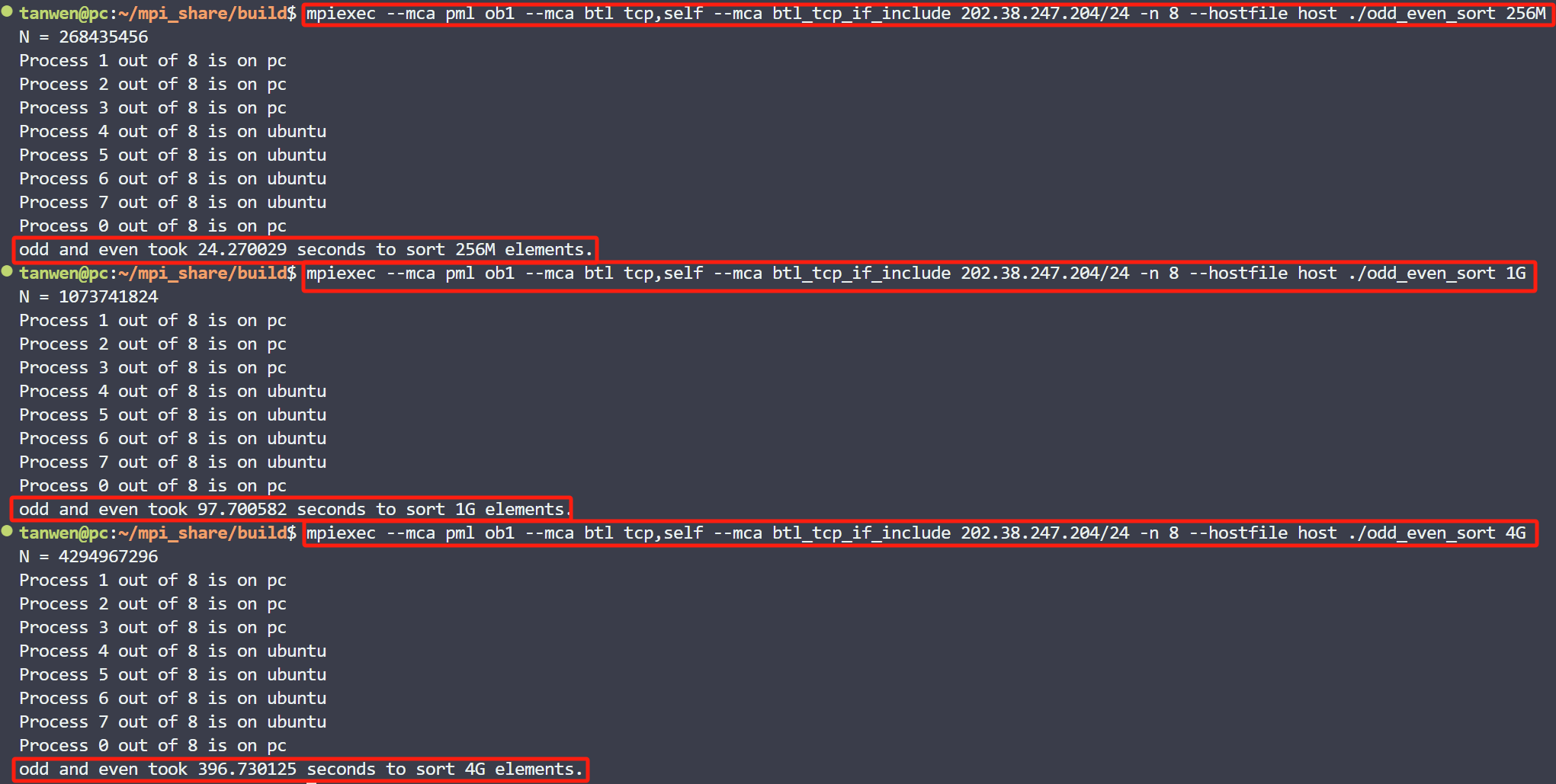
2进程排序运行命令行截图：



4进程排序运行命令行截图：



8进程排序运行命令行截图：



3.5 请分析排序时间和数据量*N*，进程数*P*之间的关系。（5分）

进程数相同的情况下，数据量变为原来的4倍且远远大于的情况下，如下公式计算，排序时间应为原来的倍数。从实验结果可以看出，当进程数为4和8的时候都是都是基本遵循规律的。但是当进程数为2时，并没有遵守这条规律。我认为是当进程数太少，排序过程中每个进程之间的数据交换时间占据了主导，从而导致规律不成立。

数据量相同的情况下，可以看到通过增加进程数，排序时间并没有显著减少，这里排序过程中数据交换对排序时间影响很大，进程数变为原来的2倍，排序时间约为原来的0.6~0.7倍；进程数变为原来的2倍，排序时间约为原来的0.5~0.6倍，单纯添加进程数无法显著减少排序时间。

### 4、PSRS排序（30分）

4.1 请简述PSRS排序算法。（5分）

假设待排序的元素有个，处理器有个。PSRS排序首先要将带排序的元素均匀的分成个部分，每部分包含个元素。每个处理器处理其中的一部分，并对其进行局部排序。为确定局部有序序列在整个序列中的位置，每个处理器从各自的局部有序序列中选取几个代表元素，将这些代表元素进行排序后选出个主元将自己的局部有序序列分成最多段。然后通过全局交换的方式，将段有序序列分发给对应的处理器，使第个处理器都拥有各个处理器的第段，共段有序序列。每个处理器对着段有序序列进行排序。最后，将各个处理器的有序段依次汇合起来，就是全局有序序列了。

算法的基本描述如下：

**输入**：个待排序的序列

**输出**：分布在各个处理器上，得到全局有序的数据序列

**1）无序序列的划分及局部排序**

根据数据块的划分方法，将无序序列划分为个部分，每个处理器对其中的一部分进行串行快速排序，这样每个处理器就会拥有一个局部有序序列。

**2）选取代表元素**

每个处理器从局部有序序列中选取第共个代表元素，其中。

**3）确定主元**

每个处理器都将自己选取好的代表元素发送给处理器。对这段有序序列做多路归并排序，再从这排序后的序列中选取第共个元素作为主元。

**4）分发主元**

将个主元分发给各个处理器。

**5）局部有序序列划分**

每个处理器在接收到主元后，根据主元将自己的局部有序序列划分成段。

**6）****段有序序列的分发**

每个处理器将自己的第段发送给第个处理器，是处理器都拥有所有处理器的第段。

**7）多路排序**

每个处理器将上一步得到的段有序序列做多路归并。

经过这7步后，一次将每个处理器的数据取出，这些数据就是有序的。

4.2 对于*P*个MPI进程上*N*个元素的排序，其计算复杂度是多少？进程之间通信的数据量有多少？（请使用*N*和*P*表示）。（10分）

算法的时间复杂度可达。

通信量（算法如上所述）：

在第步，进程需要给其他每个进程发送个元素也就是共个元素，然后其他每个进程需要接收个元素。

在第步，其他每个进程需要向进程发送个元素，进程需要接收个元素。

在第步，进程需要给其他每个进程发送个元素也就是个，然后其他每个进程需要接收个元素。

在第6步，每个进程需要发送段，接收段。但是这里每段的大小不确定，假设每段大小一样的话，也就是每一段包含个元素。那就是每个进程需要发送个元素，需要接收个元素。

4.3 请使用MPI完成PSRS排序算法，实际测量各个步骤的计算时间和进程之间通信的数据量，并与前述理论分析进行对比。（5分）

在实验中，我们将该排序过程分为1）原始数据分发；2）局部排序；3）选取代表元素并发送给0进程；4）确定并分发主元；5） 局部有序序列划分；6）P段有序序列的分发；7）多路排序。并在256M数据量，4进程基础上记录各步骤的计算时间和进程之间的通信量。

|  |  |  |
| --- | --- | --- |
| **排序步骤** | **进程ID** | **计算时间（s）** |
| 原始数据分发 | 0 | 4.867936 |
| 局部排序 | 0 | 11.333184 |
| 1 | 15.876101 |
| 2 | 13.587880 |
| 3 | 11.314442 |
| 选取代表元素并发送给0进程 | 0 | 0.000309 |
| 1 | 0.000079 |
| 2 | 0.000097 |
| 3 | 0.000068 |
| 确定并分发主元 | 0 | 0.000029 |
| 1 | 0.000797 |
| 2 | 0.000146 |
| 3 | 0.000819 |
| 局部有序序列划分 | 0 | 0.133496 |
| 1 | 0.132709 |
| 2 | 0.142217 |
| 3 | 0.142835 |
| P段有序序列的分发 | 0，1，2，3 | 4.028674 |
| 多路排序 | 0 | 1.062747 |
| 1 | 1.014439 |
| 2 | 1.423781 |
| 3 | 1.395212 |
| 总排序时间 | 0，1，2，3 | 21.462055 |

排序过程中，每个进程数据的发送量和接收量：

|  |  |  |
| --- | --- | --- |
| **进程ID** | **发送量（Byte）** | **接收量（Byte）** |
| 0 | 1006571156 ≈ 960 MB | 201512012 ≈ 192 MB |
| 1 | 201326676 ≈ 192 MB | 469762144 ≈ 448 MB |
| 2 | 201326660 ≈ 192 MB | 469762020 ≈ 448 MB |
| 3 | 201388416 ≈ 192 MB | 469576732 ≈ 448 MB |

结论分析：

1）耗时分析：可以看出，整个过程中最为耗时的步骤是局部排序，其次是原始数据分发过程，P段有序序列的分发过程。其他步骤耗时非常短。但当进程数增大时，每个进程负责的数据量减少，局部排序时间会减少，而P段有序序列分发耗时则会增加，无脑增加进程数量不一定能够加快排序。

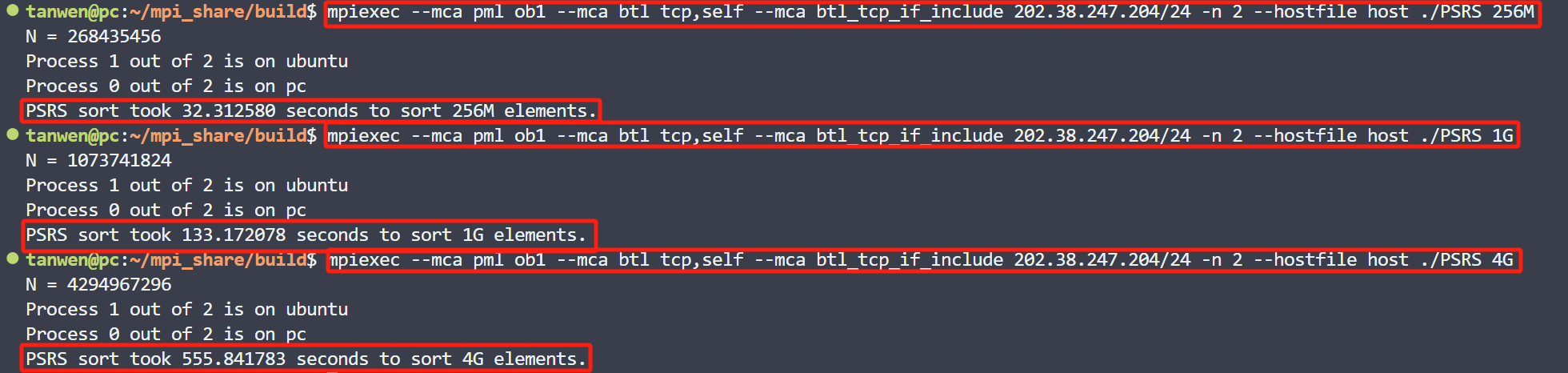
2）进程间通信数据量分析：PSRS排序过程中由于主元的选择对于每个进程有序序列的分发影响很大，加上每个进程与0号进程之间传递代表元素和主元，以及代码实现过程中有序序列分发中需要发送接收每个序列的长度，开始索引等信息，还有主元对分发序列长度的影响，导致进程间通信数据量基本没有一个理论的值。但是我们可以看到，由于我们的代码实现需要0号进程需要分发原始数据，其他进程需要接收原始数据之外，在排序过程每个进程的发送量和接收量都非常均匀，进程通信对于每个进程的负载都差不多。

4.4 测试不同进程数和不同数据规模下的排序时间，结果如表-3所示。（5分）

表-3 PSRS并行排序的时间

|  |  |  |
| --- | --- | --- |
| *N* | *P* | PSRS排序时间(s) |
| 256M | 2 | 32.312580 |
| 4 | 21.632976 |
| 8 | 15.417021 |
| 1G | 2 | 133.172078 |
| 4 | 87.471762 |
| 8 | 61.349537 |
| 4G | 2 | 555.841783 |
| 4 | 358.507321 |
| 8 | 255.437697 |

2进程排序运行命令行截图：



4进程排序运行命令行截图：

文本

描述已自动生成

8进程排序运行命令行截图：

文本, 日程表

描述已自动生成

4.5 请分析排序时间和数据量*N*，进程数*P*之间的关系。（5分）

从数据量来看，当，计算复杂度为，可以看到当数据量变为原来的4倍，相同进程数的情况下，按如下公式计算，理论上来说时间应为原来的倍。从实验结果也能看出来，数据量翻了4倍情况下，排序时间也基本约为4倍关系。

而当数据量不变，进程数变为原来的2倍的情况下，最好的情况下排序时间应为之前的一半，但是由于各进程之间需要互换信息，从而达不到理想情况。从实验结果可以看出，进程数变为原来的2倍，时间约为之前的0.65倍；进程数变为原来的4倍，时间约为之前的0.5倍。可以看出，进程之间的信息交换对排序时间影响很大。

### 5、并行排序的性能分析（25分）

5.1 对于上述输入，请比较数据快速排序、奇偶排序、PSRS排序方法的性能，并从理论上加以解释。（10分）

快速排序的时间复杂度为。奇偶排序由于局部排序我们使用的是快排，时间复杂度为，PSRS的时间复杂度为。由于快排我们只使用单进程，可以看出，快排的排序时间都最长，这也说明多进程对比单进程的优势。而对于奇偶排序和PSRS排序来说，由于奇偶排序的局部排序也采用了快排，所以时间复杂度差距不大，但是奇偶排序在进程之间进行奇偶排序时，每个进程要与相邻进程进行次信息交换，每次信息交换需要发送和接收个浮点数；而PSRS在排序过程中，只用在确定主元时进行少量数据交换，以及1次有序序列的分发，进程间数据通信量大大减少。从实验可以看出，在相同和的情况下，PSRS的性能优于奇偶排序。

5.2 待排序的数据具有以下特征时，上述三种算法的性能是否会发生变化，请从理论和实际测试角度说明。（15分）

A.完全逆序，即输入数据是从大到小序列

B.完全顺序，即输入数据是从小到大序列

C.近似顺序，即对于一个从小到大的数据序列，从中随机抽取1%的数据与相邻且不相等的数据交换位置。

在256M数据量，4进程（串行快速排序为节点2（pc节点）上单进程）基础上测试上述情况对于排序算法的影响，记录耗时情况如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **测试情况** | **串行快速排序耗时（s）** | **并行奇偶排序耗时（s）** | **PSRS排序耗时（s）** |
| 随机（默认） | 39.339267 | 30.392449 | 21.600049 |
| A | 13.992077 | 24.337871 | 15.799564 |
| B | 11.464677 | 23.805543 | 10.671319 |
| C | 11.501978 | 24.527121 | 10.630406 |

**总体情况**：1）从排序时间可以看出，当原始数据随机的情况下，各排序算法耗费的时间是最多的。当数据是A，B，C情况下，排序时间在串行快速排序上耗时大大减少，而并行奇偶排序耗时减少的不多，PSRS排序耗时也都大大减少。

**串行快速排序耗时分析**：在快速排序的早期版本中，最左面或者是最右面的那个元素被选为枢轴，那最坏的情况就会在下面的情况下发生：

1）数组已经是正序（same order）排过序的。  
2）数组已经是倒序排过序的。  
3）所有的元素都相同（1、2的特殊情况）

所以这个问题可以通过要么选择一个随机的枢轴，或者选择一个分区中间的下标作为枢轴，或者（特别是对于相比更长的分区）选择分区的第一个、中间、最后一个元素的中值作为枢轴。有了这些修改，那快排的最差的情况就不那么容易出现了。在本次实验中，我采取的是<stdlib.h>中提供的qsort()函数，应该也不是最原始的快速排序了，考虑了最坏情况并做出了对应处理。

**并行奇偶排序耗时分析**：A，B，C的情况都比随机好一些，因为局部快排的时候耗时更短，但是进程间奇偶排序时，由于数据交换以及较高的计算复杂度，耗时比较多，这部分并没有被原始数据的特征所影响。

**PSRS排序耗时分析：**对于该算法，A，B，C的情况都会让每个进程局部排序时耗时更短，但更关键的是，在原始数据完全顺序情况B或近似顺序情况C时，每个进程基本拥有最后归类到该进程数据的全部或者大部分，从而有效减少了进程间通信。从实验结果可以看到，在情况A下，耗时比情况B、情况C多了大概5秒，这就是由于进程间数据交换带来的影响。