# 内存不共享多机环境下Parallel sorting by regular sampling算法

单暖乔 21307130395

这里我用的MPI。在vscode中用的c++语言和<mpi.h>完成的

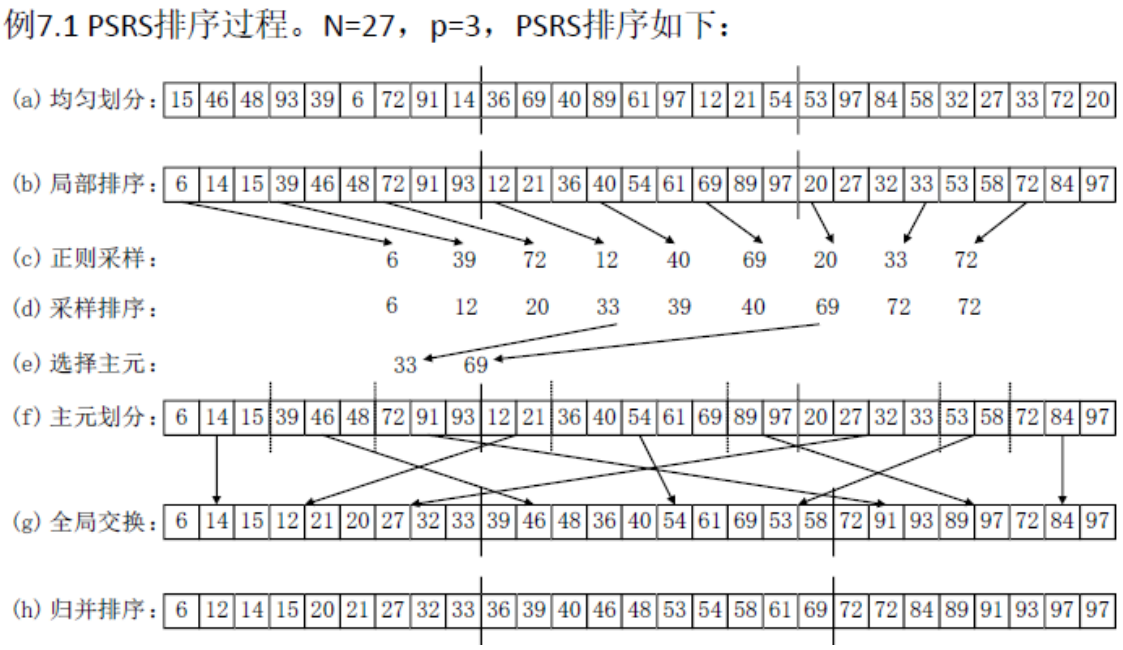
## 1.文件说明和环境配置

这个作业的文件夹中有源文件sort_quick.cpp和对应的可执行文件sort.exe。

我已经事先把c_cpp_properties.json和tasks.json都改好了。每一次进行编译的时候都要在vscode左上角的terminal中点击Run build task，之后在下方的终端输入 mpiexec -n {进程数} sort_quick就可以运行了。

## 2.代码思路

这里我参考了ppt上的这个示意图，基本按照这个过程实现的PSRS算法




就是先把数据分成p个（进程的个数）段。每一段给到一个进程进行局部排序。之后我尽量的规律的进行抽样，每个进程中都要选p个sample。之后把这些sample交给根进程（这里是编号为0的进程）进行排序。之后均匀的选出(p-1)主元播送给各个进程。每个进程根据这些主元把自己排好的数据拆段，之后把对应的段播送给对应的进程，并从别的进程得到自己需要的数据段。之后由于一个本地进程中的数据段都是分别有序的，因此可以直接进行merge排序，得到局部有序的大的数据段，又因为一个进程的所有数据段是被同一对相邻的主元进行切分的，且所有被这对相邻主元切分的数据段都在这个进程中，因此一个进程中的所有元素都是在一个区间之内的，且所有在这个区间中的元素都在这个进程的数据段中，因此可以把每个进程排好的数据段直接拼接就可以得到全有序的输出。

对于MPI中的函数，我主要用到了这几个

MPI_Comm_size 这个是为了得到进程的个数，因为进程个数本来是在命令行中指定的，所以如果要用到这个值的时候，需要用这个函数获取

MPI_Comm_rank 这个是得到pid，也就是进程号用的。

MPI_Wtime 这个是获取时间的，在调用函数之前计时，在调用完毕之后计时，可以算出整个算法的执行时间。

MPI_Gather 把所有的进程的数据集中到根进程中

MPI_Bcast 把数据从根进程中广播出去

MPI_Alltoall 全交换，指定发送位置的起始位置，和数量，接收的起始位置，接收的数量。对每个进程发送的数据的数量是一样的。比如这里具体实现全局交换的时候，要先把每个数据段的长度全交换，才能把数据全交换，这个把每个数据段长度全交换的时候，每个进程要给任意一个进程（包括自己）一个数，作为这个进程之后打算给对方的数据段的长度。

MPI_Alltoallv 全交换，这个可以让每个进程向外发送数据的时候不用每个数据段的长度都是一样的，并且要指定发送的每个数据段的起始位置。在全局交换的时候，每个进程要给别的进程传输数据段和接收数据段的时候用这个函数

MPI_Gatherv 每个进程发送出的数据长度可以不同 ，也是集中到根进程中，这个用在最后把局部归并排好的数据段发到根进程中直接拼接的步骤。

## 3.代码说明

我定义了六个函数

step和之前写快排的时候一样，都是用来做partition的

sort_num也和之前快排的一样，但是是串行版本的

local_sort_extract_sample是在把数据分块之后每一个进程进行本地的局部排序和样本抽取的。

sort_sample_extract_pivot是在pid=0的进程中运行的，是把所有的样本进行排序和选取主元用的

local_final_sort是在每个进程收到自己应该处理的局部排好序的数据段之后，把这些数据段进行merge，产生在本地有序的数据序列的函数。

sort函数是整个用mpi实现PSRS的逻辑，这里会直接或者间接调用上述的五个函数，其中的mpi通信都是在这里实现的，包括收集样本，播送主元，全交换数据段，收集merge之后的有序数据序列并进行拼接等等。

## 4.实验结果

这些是按30次取平均得到的，对2个线程、4个线程、8个线程的情况下不同数据量时的运算结果。由于每次运行的具体情况不同，和当时电脑的空闲程度也很有关系，在同一个数据量的时候串行的时间可能也会出现些许的出入。

**2个线程**

| 数据量 | 加速比 | 串行平均执行时间/s | 并行平均执行时间/s |
|---|---|---|---|
| 1K | 1.09486 | 7.57133e-005 | 6.91533e-005 |
| 5K | 1.55452 | 0.000409777 | 0.000263603 |
| 10K | 1.62348 | 0.000877153 | 0.000540293 |
| 100K | 1.76002 | 0.0102517 | 0.00582476 |

| 数据量 | 加速比 | 串行平均执行时间/s | 并行平均执行时间/s |
| --- | --- | --- | --- |
| 1M | 1.90583 | 0.107993 | 0.0566645 |
| 10M | 2.4302 | 1.83491 | 0.755045 |

**4个线程**

| 数据量 | 加速比 | 串行平均执行时间/s | 并行平均执行时间/s |
| --- | --- | --- | --- |
| 1K | 1.31366 | 7.81233e-005 | 5.947e-005 |
| 5K | 2.26054 | 0.00043217 | 0.00019118 |
| 10K | 2.72979 | 0.000965453 | 0.000353673 |
| 100K | 3.14408 | 0.0101461 | 0.00322705 |
| 1M | 3.52732 | 0.108817 | 0.0308496 |
| 10M | 5.37001 | 1.8316 | 0.34108 |

**8个线程**

| 数据量 | 加速比 | 串行平均执行时间/s | 并行平均执行时间/s |
| --- | --- | --- | --- |
| 1K | 0.813337 | 8.98033e-005 | 0.000110413 |
| 5K | 2.87377 | 0.000533947 | 0.0001858 |
| 10K | 3.38949 | 0.00105826 | 0.000312217 |
| 100K | 4.64714 | 0.0101192 | 0.0021775 |
| 1M | 5.87805 | 0.108707 | 0.0184938 |
| 10M | 10.2221 | 1.86201 | 0.182155 |

## 5.结果分析

可以看到当数据量很小的时候，固定数据量，加速比不一定随着线程数的增加而变大，因为这个线程数可能会相对于数据量而过多，因此其创建终止和调度的额外开销相比于数据的并行带来的节省是相对来说更大的，我在1K数据的时候之后也测试了16个进程的。

而在其他的情况下，也就是5K以及5K以上的情况下，在数据量不变的时候线程数越多加速比越大，这个就说明并行度越高，越快。

在线程数一定的情况下，数据量越大，加速比越大，这个也是因为数据量越大，并行的额外开销带来的影响就越不显著，加速比越大。

# 6.总结

感受和评价：

这个实验让我更好的理解了MPI，算法的思路很明晰，实现的代码每一步都很细致，也让我对这个算法里的消息传递的过程，也就是数据的流向，有了很清楚的了解。但是环境的配置感觉略难。

结论：

在线程数一定的时候，数据量越大加速比越大

在数据量一定的时候，在线程不过多的范围之内，增大线程有利于提高加速比