正则采样排序PSRS的MPI算法

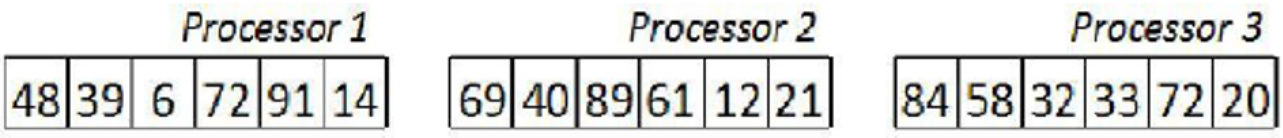
算法流程

假设有 p 个进程，有 N 条数据需要排序。

1. 均匀划分

将 N 条数据均匀划分为 p 段，每个进程处理一段数据。其中 i $(i = 0, 1, \ldots, p-1)$ 号进程处理 $\lfloor \frac{i \times N}{p} \rfloor$ 到 $\lfloor \frac{(i+1) \times N}{p} \rfloor - 1$ 行（行号从0开始计算）；

3个进程划分18个数据的结果如下，一个进程处理一段。



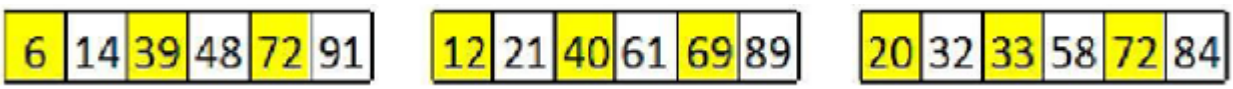
在实现上，由于数据较大（32G），为了充分并行节省时间，由各进程读取所需要的数据，同样达到均匀划分效果。实现代码如下：

```
#define BLOCK_LOW(my_rank, comm_sz, T) ((my_rank)*(T)/(comm_sz))
#define BLOCK_SIZE(my_rank, comm_sz, T) (BLOCK_HIGH(my_rank,comm_sz,T) -
BLOCK_LOW(my_rank,comm_sz,T) + 1)

// 打开文件
ifstream fin(fileName, ios::binary);
// 计算各进程读取文件的起始行号和大小
unsigned long myDataStart = BLOCK_LOW(my_rank, comm_sz, dataLength);
unsigned long myDataLength = BLOCK_SIZE(my_rank, comm_sz, dataLength);
// 将文件指针移动到起始行号
fin.seekg((myDataStart)*sizeof(unsigned long), ios::beg);
unsigned long *myData = new unsigned long[myDataLength];
// 读取数据
for(unsigned long i=0; i<myDataLength; i++)
    fin.read((char*)&myData[i], sizeof(unsigned long));
fin.close();
```

2. 局部排序

各进程对各自的数据进行排序。

例子如下，各段数据各自有序：



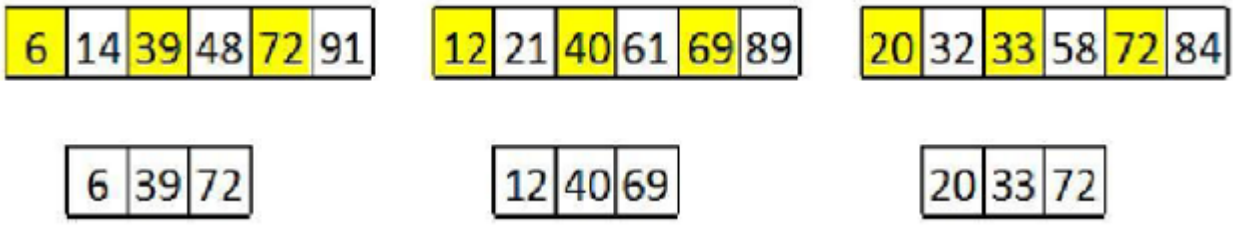
代码如下，可以调用算法库：

```
sort(myData, myData+myDataLength);
```

3. 选取样本

p 个进程中，每个进程需要选取出 p 个样本（regular samples），选取规则为 $\lfloor \frac{i \times dataLength}{p} \rfloor$, $i = 0, 1, \ldots p-1$。`dataLength` 是进程各自的数据长度。注意此时选取的 p 个样本也是有序的。

如下图三个进程中，每个进程选取出三个样本：



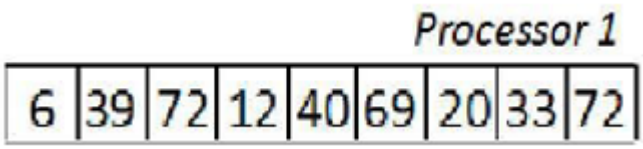
代码如下：

```
unsigned long *regularSamples = new unsigned long[comm_sz];
for(int index=0; index<comm_sz; index++)
    regularSamples[index] = myData[(index*myDataLength)/comm_sz];
```

4. 样本排序

用一个进程对 p 个进程的共 $p \times p$ 个样本进行排序，此时样本都是局部有序的，使用**归并**能减少时间复杂度。

如下图，processor 1将9个样本排序：



在实现中，由0号进程负责样本排序，首先需要 `Gather` 操作，将样本收集起来，然后要进行归并。

`Gather` 操作：

```
unsigned long *gatherRegSam;
if(my_rank == 0)
    gatherRegSam = new unsigned long[comm_sz*comm_sz];
// sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm
MPI_Gather(regularSamples, comm_sz, MPI_UNSIGNED_LONG, gatherRegSam, comm_sz,
MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
```

归并函数：

```
struct mmdata {
    // 待归并数组序号
```

```
    int stindex;
    // 在数组中的序号
    int index;
    unsigned long stvalue;
    mmdata(int st=0, int id=0, unsigned long stv =
0):stindex(st),index(id),stvalue(stv){}
};

// mmdata比较运算符重载，在优先队列的表现是mmdata中stvalue小的排在队列前面
bool operator<( const mmdata & One, const mmdata & Two) {
    return One.stvalue > Two.stvalue;
}

// 各进程regularSamples二维数组，各进程regularSamples长度，待归并数组数量，结果数组，待归并总
数据量
void multiple_merge(unsigned long* starts[], const int lengths[], const int Number,
unsigned long newArray[], const int newArrayLength) {
    priority_queue< mmdata> priorities;

    // 将每个待归并数组的第一个数加入优先队列，同时保存它所在待归并数组序号和数字在数组中的序号
    for(int i=0; i<Number;i++) {
        if (lengths[i]>0) {
            priorities.push(mmdata(i,0,starts[i][0]));
        }
    }

    int newArrayindex = 0;
    while (!priorities.empty() && (newArrayindex<newArrayLength)) {
        // 拿下最小的数据
        mmdata xxx = priorities.top();
        priorities.pop();

        // 将拿下的数据加入到结果数组中
        newArray[newArrayindex++] = starts[xxx.stindex][xxx.index];

        // 如果被拿下数据不是所在的待归并数组的最后一个元素，将下一个元素push进优先队列
        if ( lengths[xxx.stindex]>(xxx.index+1)) {
            priorities.push(mmdata(xxx.stindex, xxx.index+1, starts[xxx.stindex]
[xxx.index+1]));
        }
    }
}
```

归并样本：

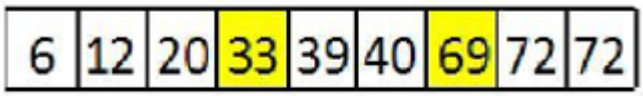
```
if(my_rank == 0) {
    // start用于存储gatherRegSam中各进程RegularSamples开始下标，相当于二维数组
    unsigned long **starts = new unsigned long* [comm_sz];
    // gatherRegSam中各进程RegularSamples长度，都一样是comm_sz
    int *lengths = new int[comm_sz];
    for(int i=0; i<comm_sz; i++) {
        starts[i] = &gatherRegSam[i*comm_sz];
        lengths[i] = comm_sz;
```

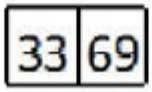
```
    }

    // 因为各进程的的ragularSamples就是有序的，因此只需要将gatherRegSam中的各进程数据归并即可
    unsigned long *sortedGatRegSam = new unsigned long[comm_sz*comm_sz];
    multiple_merge(starts, lengths, comm_sz, sortedGatRegSam, comm_sz*comm_sz);
}
```

5. 选取主元

一个进程从排好序的样本中选取 $p-1$ 个主元。选取方法是 $i \times p,\ i = 1, 2, \ldots, p-1$。

例子如下，从9个样本中选取2个主元：





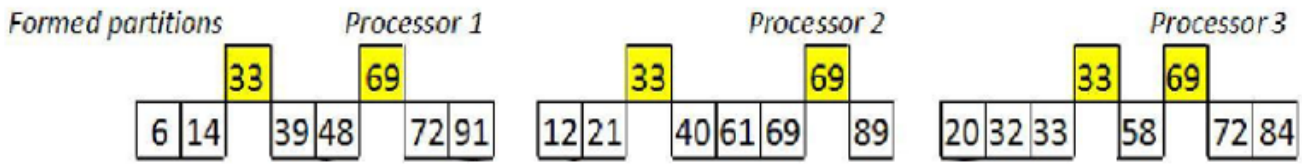
选取主元代码如下：

```
for(int i=0; i<comm_sz-1; i++)
    privots[i] = sortedGatRegSam[(i+1)*comm_sz];
```

6. 主元划分

p 个进程的数据按照 $p-1$ 个主元划分为 p 段。

例子如下，3个进程的数据被都被2个主元划分为3段：



在具体实现中，0号进程要将 $p-1$ 个主元广播到其它所有进程。然后所有进程将按照主元划分。

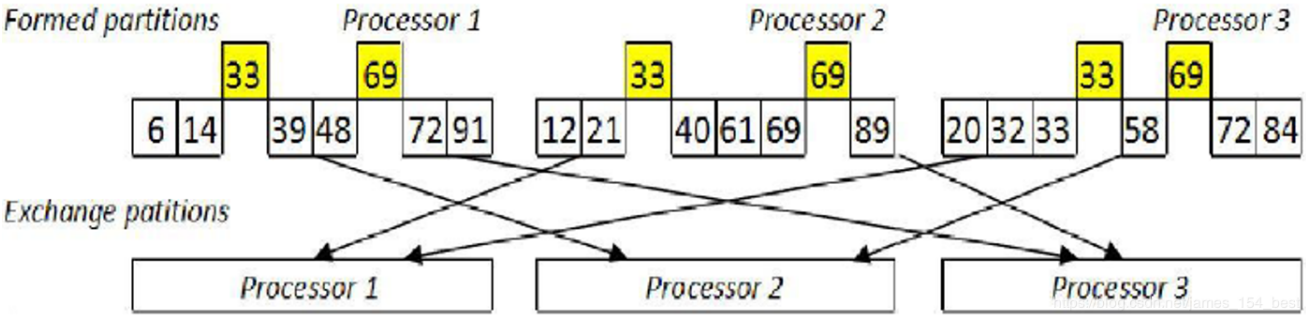
```
// 将主元广播到其它进程
MPI_Bcast(privots, comm_sz-1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
// partStartIndex保存每段开始下标
int *partStartIndex = new int[comm_sz];
// partLength保存每段的长度
int *partLength = new int[comm_sz];
unsigned long dataIndex = 0;
for(int partIndex = 0; partIndex<comm_sz-1; partIndex++) {
    partStartIndex[partIndex] = dataIndex;
    partLength[partIndex] = 0;

    while((dataIndex<myDataLength) && (myData[dataIndex]<=privots[partIndex])) {
        dataIndex++;
        partLength[partIndex]++;
```

```
    }
}
// 最后一段数据补齐（防止主元溢出）
partStartIndex[comm_sz-1] = dataIndex;
partLength[comm_sz-1] = myDataLength - dataIndex;
```

7. 全局交换

进程 i $(i=0,1,\ldots p-1)$ 将第 j $(j=0,1,\ldots,p-1)$ 段发送给进程 j。也就是每个进程都要给其它所有进程发送数据段，并且还要从其它所有进程中接收数据段，所以称为全局交换。

例子如下：



在具体实现中，可以使用 `MPI_Alltoall` 和 `MPI_Alltoallv` 简化操作，进程 i 先使用 `MPI_Alltoall` 将第 j 段数据的长度发送给进程 j（或者说进程 j 使用 `MPI_Alltoall` 接收进程 i 的第 j 段数据的长度），进程 j 将这个长度保存在 `recvRankPartLen[i]` 中，进程 j 知道长度后便可以使用 `MPI_Alltoallv` 接收进程 i 的第 j 段数据了。

```
// 接收各进程数据段长度到recvRankPartLen中
int *recvRankPartLen = new int[comm_sz];
MPI_Alltoall(partLength, 1, MPI_INT, recvRankPartLen, 1, MPI_INT, MPI_COMM_WORLD);

// 计算接收到的数据段在缓存中的摆放位置（距离起始地址的偏移量）
int rankPartLenSum = 0;
int *rankPartStart = new int[comm_sz];
for(int i=0; i<comm_sz; i++) {
    rankPartStart[i] = rankPartLenSum;
    rankPartLenSum += recvRankPartLen[i];
}
unsigned long *recvPartData = new unsigned long[rankPartLenSum];
// 发送数据段和接收数据段
MPI_Alltoallv(myData, partLength, partStartIndex, MPI_UNSIGNED_LONG,recvPartData,
recvRankPartLen, rankPartStart, MPI_UNSIGNED_LONG, MPI_COMM_WORLD);
```

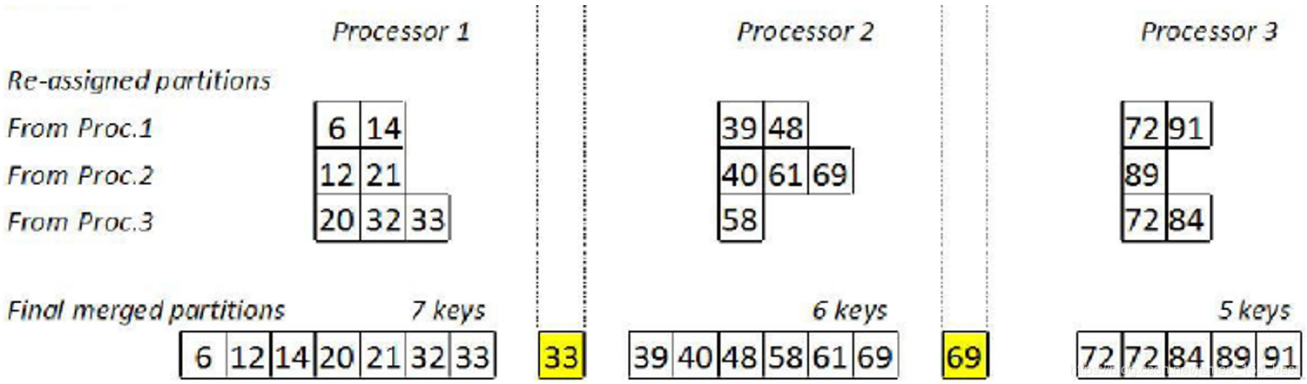
> 需要注意的是
>
> ```
> int MPI_Alltoallv(const void *sendbuf, const int *sendcounts, const int
> *sdispls, MPI_Datatype sendtype, void *recvbuf, const int *recvcounts, const int
> *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
> ```

> 其中 `sendcounts, sdispls, recvcounts, rdispls` 都是 `const int *` 类型，不能是 `unsigned long *` 类型，导致发送的数据段长度不能超过 $2^{31}-1$，而总数据量却有 2^{32}。因此在进程数 $p=2$ 和总数据量为 2^{32} 的情况下运行一定会出现错误（因为分成的两段数据一定有一段要超过 $2^{31}-1$）。实际上，当数据量为 2^{32} 时，尽管总进程数大于2，程序也往往会意外中断（怀疑是此处的问题）

8. 归并排序

各处理器对接收到的 p 个数据段进行归并，因为 p 个数据段已经是局部有序的。

例子如下：



代码如下：

```
unsigned long **mulPartStart = new unsigned long*[comm_sz];
for(int i=0; i<comm_sz; i++)
    mulPartStart[i] = &recvPartData[rankPartStart[i]];

unsigned long *sortedRes = new unsigned long[rankPartLenSum];
multiple_merge(mulPartStart, recvRankPartLen, comm_sz, sortedRes, rankPartLenSum);
```

完成以上操作后，将进程 0 到 $p-1$ 的数据按顺序拼接起来，就是全部数据排序结果。

结果展示

T_p 表（单位：秒）

> X 代表不能运行

n\p	1	2	4	8	16	32	64	112
14	0.007	0.005	0.003	0.004	0.018	0.193	0.564	0.208
18	0.124	0.079	0.046	0.029	0.016	0.312	0.611	0.222
22	2.231	1.348	0.757	0.418	0.24	0.49	0.821	0.311
26	39.197	22.935	12.992	7.132	3.897	3.366	3.762	3.309
30	1561.28	387.574	221.611	121.713	68.411	44.623	46.147	68.628
31	X	808.719	461.703	248.49	201.645	147.301	185.333	84.508
32	X	X	X	X	X	X	375.605	222.34