并行分布式计算实验报告

20307130112 马成

实验一 多线程并行快速排序算法

代码分析

1. 在这次实验中我选用了OpenMP和C++语言来完成实验。
2. 首先，为了实现并行的计算先将数组划分成不相交的若干部分交给不同的进程分别排序

```
for (int i=1;i<=numThr;i++){
    bj[i]=bj[i-1]+MAXN/numThr+((MAXN%numThr)>(i-1));
    id[i]=bj[i];
    if (bj[i]-bj[i-1]) mxn[mt++]=i-1;
}
#pragma omp parallel num_threads(numThr)
{
    sort(a+bj[omp_get_thread_num()],a+bj[omp_get_thread_num()+1]);
}
```

3. 完成分别的排序之后在每一个进程中都已经是有序的了，这时只要使用归并排序即可完成任务

```
sort(mxn,mxn+mt,cmp);
for (int k=0;k<MAXN;k++){
    b[k]=a[id[mxn[0]]];
    id[mxn[0]]++;
    int now=mxn[0];
    if (id[now]<bj[now+1]){
        int j;
        for (j=1;j<mt;j++){
            if (a[id[mxn[j]]]<a[id[now]]) mxn[j-1]=mxn[j];
            else break;
        }
        mxn[j-1]=now;
    }else{
        for (int j=1;j<mt;j++) mxn[j-1]=mxn[j];
        mt--;
    }
}
```

4. 最后只需要加入一下计时代码和比对是否排序正确即可，此处省略。

实验结果

1. 我在数据量为5K、10K、100K、1M、10M、100M的时候分别统计了串行、并行线程数为2、4、8、16的运行时间，实验结果如下。其中由于1K中串行程序运行时间过快无法捕捉，这里就不测了，由于希望更好的体现并行计算的优势增加了1M、10M、100M的测试组别

	串行	2线程	4线程	8线程	16线程
5K	0.0010005s	0.0029968s	0.0019586s	0.001997s	0.0040095s
10K	0.0020308s	0.0039833s	0.004005s	0.0040769s	0.0059953s
100K	0.0240567s	0.0189897s	0.0199692s	0.0220026s	0.0268808s
1M	0.251089s	0.18024s	0.168378s	0.184134s	0.183079s
10M	2.61047s	1.71436s	1.2194s	1.27739s	1.43819s
100M	30.6709s	18.9519s	13.2613s	13.7519s	16.6175s

2. 加速比

	串行	2线程	4线程	8线程	16线程
5K	/	0.333856	0.510824	0.501002	0.249532
10K	/	0.509829	0.507066	0.498124	0.338732
100K	/	1.266829	1.20469	1.093357	0.89494
1M	/	1.393081	1.491222	1.363621	1.371479
10M	/	1.522708	2.140783	2.043597	1.815108
100M	/	1.618355	2.312813	2.230303	1.845699

实验分析

1. 在数据较小的时候由于串行算法本身就可以快速计算，而并行算法需要经过启动线程等一系列准备操作，在数据范围较小的时候运行速度比串行算法慢。
2. 在数据较大的时候并行算法展现出了较好的性能。如果记排序的元素个数是n，线程个数是m，那么串行计算的复杂度是$O(nlgn)$而并行计算的复杂度是$O((n/m)lg(n/m)+nm)$其中前半部分是并行快速排序的复杂度，后半部分是在主线程进行多路归并的复杂度。空间复杂度是$O(n)$的
3. 一般拥有较为理想的计算速度的是4线程并行计算，因为我计算机CPU本身逻辑处理数量是4。线程数为4应该可以高效的利用CPU的计算资源，其次如果有过高的线程数将在多路归并阶段花费较多的时间。
4. 其实可以在多路归并的时候使用堆的数据结构将多路归并的复杂度降到$O(nlg(m))$但是一般进行排序的时候不会启用过多的线程，堆也会有一些常数的提升因此实验在多路归并时还是使用了朴素的插入排序。

实验二 内存不共享多机环境下的PSRS算法

代码分析

1. 在这次实验中我选用了MPI和C++语言来完成实验。
2. 需要对数据做一些预处理，因为再使用`MPI_scatter`函数的时候如果每一个线程需要从主线程接受的元素数量不同会出现一些问题，因此我先将元素数量补充成了可以正好被线程数整除的数量，最后完成排序之后再一次处理节课

```
// 排序前的预处理
int i;
for (i=N;i%numThr;i++) array[i]=INT_MIN;
addnum=i-N;
N=i;

// 排序后还原
for(int k=addnum;k<N;k++) array[k-addnum]=array[k];
```

3. 每一个处理器获得自己需要局部排序的数据并进行正则采样

```
MPI_Scatter(array,localSize,MPI_INT,localArray,localSize,MPI_INT,0,MPI_COMM_
WORLD);
sort(localArray,localArray+localSize);
for (int i=0;i<numThr;i++){
    pivots[i]=localArray[(i*(N/(numThr*numThr)))];
}
```

4. 主线程接受每一个处理器的采样并挑选主元

```
MPI_Gather(pivots,numThr,MPI_INT,collectedPivots,numThr,MPI_INT,0,MPI_COMM_W
ORLD);
if (myId == 0){
    sort(collectedPivots,collectedPivots+numThr*numThr);
    for (int i=0;i<(numThr-1);i++){
        phase2Pivots[i]=collectedPivots[(((i+1)*numThr)+(numThr/2))-1];
    }
}
MPI_Bcast(phase2Pivots,numThr-1,MPI_INT,0,MPI_COMM_WORLD);
```

5. 每一个处理器根据选取的主元计算每一个划分的元素个数

```
for (int i=0;i<localSize;i++){
    while (index<numThr-1&&localArray[i]>phase2Pivots[index]) index+=1;
    if (index==numThr-1){
        partitionSizes[numThr-1]=localSize-i;
        break;
    }
    partitionSizes[index]++;
}
```

6. 每一个处理器接受需要自己多路归并的部分

```
MPI_Alltoall(partitionSizes,1,MPI_INT,newPartitionSizes,1,MPI_INT,MPI_COMM_W
ORLD);
for (int i=0;i<numThr;i++) totalSize+=newPartitionSizes[i];
*newlocalArray=(int*)malloc(totalSize*sizeof(int));
// 计算发送位置和接受位置的数组用于传递真实的元素值
sendDisp[0]=0;
recvDisp[0]=0;
for (int i=1;i<numThr;i++){
    sendDisp[i]=partitionSizes[i-1]+sendDisp[i-1];
    recvDisp[i]=newPartitionSizes[i-1]+recvDisp[i-1];
}
MPI_Alltoallv(localArray,partitionSizes,sendDisp,MPI_INT,*newlocalArray,
              newPartitionSizes,recvDisp,MPI_INT,MPI_COMM_WORLD);
```

7. 计算出每一个线程发送过来的数据的下标范围

```
indexes[0]=0;
totalListSize=partitionSizes[0];
for (int i=1;i<numThr;i++){
    totalListSize+=partitionSizes[i];
    indexes[i]=indexes[i-1]+partitionSizes[i-1];
    partitionEnds[i-1]=indexes[i];
}
partitionEnds[numThr-1]=totalListSize;
```

8. 每一个处理器各自多路归并

```
for (int i=0;i<totalListSize;i++){
    int lowest=INT_MAX,ind=-1;
    for (int j=0;j<numThr;j++){
        if ((indexes[j]<partitionEnds[j]) && (newlocalArray[indexes[j]]
<lowest)){
            lowest=newlocalArray[indexes[j]];
            ind=j;
        }
    }
    sortedSubList[i]=lowest;
    indexes[ind]+=1;
}
```

9. 主线程接受所有处理器的排序最终结果

```
MPI_Gather(&totalListSize,1,MPI_INT,subListSizes,1,MPI_INT,0,MPI_COMM_WORLD)
;
if (myId == 0){
    recvDisp[0]=0;
    for (int i=1;i<numThr;i++){
        recvDisp[i]=subListSizes[i-1]+recvDisp[i-1];
    }
}
MPI_Gatherv(sortedSubList,totalListSize,MPI_INT,array,subListSizes,
            recvDisp,MPI_INT,0,MPI_COMM_WORLD);
```

实验结果

1. 我在数据量为5K、10K、100K、1M、10M、100M的时候分别统计了串行、并行线程数为2、4、8、16的运行时间，实验结果如下。（说明：由于串行程序跑的较慢，我这里仍然使用了Lab1中测试出来的时间）

	串行	2线程	4线程	8线程	16线程
5K	0.0008088s	0.000962s	0.0025529s	0.0027232s	0.0046455s
10K	0.0020308s	0.0017298s	0.0026672s	0.0042272s	0.0147132s
100K	0.0240567s	0.0180336s	0.0167115s	0.0192499s	0.0255933s
1M	0.251089s	0.200412s	0.131606s	0.125617s	0.152689s
10M	2.61047s	1.90652s	1.41617s	1.34185s	1.68886s
100M	30.6709s	21.411s	13.6822s	14.9693s	15.1609s

2. 加速比

	串行	2线程	4线程	8线程	16线程
5K	/	0.840749	0.316816	0.297004	0.174104
10K	/	1.174008	0.761398	0.480413	0.138026
100K	/	1.333993	1.43953	1.249705	0.939961
1M	/	1.252864	1.907884	1.998845	1.644447
10M	/	1.369233	1.843332	1.945427	1.5457
100M	/	1.432484	2.241665	2.04892	2.023026

实验分析

1. 在数据较小的时候由于串行算法本身就可以快速计算，而并行算法需要经过启动线程和通信等一系列准备操作，在数据范围较小的时候运行速度比串行算法慢。
2. 在数据较大的时候并行算法展现出了较好的性能。如果记排序的元素个数是n，线程个数是m，那么串行计算的复杂度是$O(nlgn)$而并行计算的复杂度是$O((n/m)lg(n/m)+n)$其中前半部分是并行快速排序的复杂度，后半部分是在每一个线程并行的进行多路归并的复杂度，期望上我们认为每一个线程处理了n/m个元素，因此这里的复杂度是$O(n/m \times m)$。