**实验3**

学号 姓名

**实验题目:**

**利用MPI实现PSRS算法**

**实验环境(操作系统,编译器,硬件配置等):**

**本次实验运行在NVIDIA DGX（本次实验只使用了CPU：Intel Xeon E5-2698 v4 @ 2.2GHz, 20 cores/40 threads）硬件平台的ubuntu 16.04 RTS系统上，采用了支持C++11标准的gcc 7.3编译器来进行编译工作和cmake 3.9来辅助编译（只是为了程序写起来比较简单）。**

**算法设计与分析(写出解题思路和实现步骤)**

**PSRS算法有6个步骤，总的来说就是从初步分部排序的结果中选择若干合适的取样点，根据这些取样点划分数据，把划分出来的同样部分的数据分发到同一处，而后进行再排序，使得在回收数据时，能保证从某处拿到的数据一定是有序的，并且比其他若干部分的所有数大并且比剩余部分的所有数少。这样就不需要复杂的全数组多路合并实现，而只需要直接合并各部分结果即可。具体的步骤在实验配发的tutorial documentation上可以找到。**

**实现上来说，即是按部分逐步实现即可。在第三步的采样点p路合并时，使用了多路合并算法。第二步和第五步的部分排序中都采用了快速排序。**

**核心代码(写出算法实现的关键部分,如核心的循环等)**

**具体的代码都可以在https://github.com/thoh-testarossa/pc-exp上找到**

**程序在部分排序时使用的快速排序**

**void** quicksort(**int** \*array, **int** spos, **int** epos)  
{  
 **if**(spos >= epos);  
 **else** {  
 **int** c = spos, d = epos, tmp;  
 **while**(c < d)  
 {  
 **while** (c < d && array[c] <= array[d]) c++;  
 tmp = array[c];  
 array[c] = array[d];  
 array[d] = tmp;  
 **while** (c < d && array[c] <= array[d]) d--;  
 tmp = array[c];  
 array[c] = array[d];  
 array[d] = tmp;  
 }  
 quicksort(array, spos, c - 1);  
 quicksort(array, d + 1, epos);  
 }  
}

**程序在p路采样点归并时使用的最小堆**

**typedef struct** minHeapNode  
{  
 **int** value;  
 **int** source;  
}minHeapNode;  
  
/\*  
 \* Notice: The minHeap begin at element 1  
 \*/  
**void** adjustMinHeap(minHeapNode \*minHeap, **int** pos, **int** heapSize)  
{  
 **int** i, minpos = pos;  
 minHeapNode tmpNode;  
 **do**{  
 i = minpos;  
 **if**(2 \* i <= heapSize && minHeap[minpos].value > minHeap[2 \* i].value) minpos = 2 \* i;  
 **if**(2 \* i + 1 <= heapSize && minHeap[minpos].value > minHeap[2 \* i + 1].value) minpos = 2 \* i + 1;  
 **if**(i != minpos)  
 {  
 tmpNode = minHeap[i];  
 minHeap[i] = minHeap[minpos];  
 minHeap[minpos] = tmpNode;  
 }  
 }**while**(i != minpos);  
}  
  
/\*  
 \* The function which build a minHeap from raw data  
 \*/  
**void** buildMinHeap(minHeapNode \*minHeap, **int** heapSize)  
{  
 **for**(**int** i = heapSize / 2; i >= 1; i--) adjustMinHeap(minHeap, i, heapSize);  
}  
  
/\*  
 \* The function which delete the minimum element from the minHeap (this operation changes the size of the heap)  
 \*/  
**void** deleteHeapElement(minHeapNode \*minHeap, **int** &heapSize)  
{  
 **if**(heapSize <= 0);  
 **else** {  
 minHeap[1] = minHeap[heapSize--];  
 adjustMinHeap(minHeap, 1, heapSize);  
 }  
}

**p路归并部分**

//Multimerge part using minHeap  
 **int** \*sortedTotalSamplingPoint = **nullptr**;  
 **int** \*samplingPointArrayPointer = **nullptr**;  
 **if**(MASTER\_PROC == myid)  
 {  
 samplingPointArrayPointer = **new int** [numprocs];  
 **for**(**int** i = 0; i < numprocs; i++) samplingPointArrayPointer[i] = i \* numprocs;  
  
 minHeapNode \*minHeap = **new** minHeapNode [numprocs + 1];  
 **int** heapSize = numprocs;  
 **for**(**int** i = 1; i <= numprocs; i++)  
 {  
 minHeap[i].source = i - 1;  
 minHeap[i].value = totalSamplingPoint[samplingPointArrayPointer[i - 1]];  
 }  
  
 buildMinHeap(minHeap, heapSize);  
  
#endif  
  
 sortedTotalSamplingPoint = **new int** [numprocs \* numprocs];  
 **int** pt\_sTSP = 0;  
 **while**(heapSize > 0)  
 {  
 //Fetch the top element of the heap  
 sortedTotalSamplingPoint[pt\_sTSP++] = minHeap[1].value;  
 //Replace the element from the same sorted table if possible  
 samplingPointArrayPointer[minHeap[1].source]++;  
 //If the corresponding sorted table has some elements left  
 **if**(samplingPointArrayPointer[minHeap[1].source] < (minHeap[1].source + 1) \* numprocs)  
 {  
#ifdef TEST\_PART  
 cout << "replace" << endl;  
#endif  
 minHeap[1].value = totalSamplingPoint[samplingPointArrayPointer[minHeap[1].source]];  
 adjustMinHeap(minHeap, 1, heapSize);  
 }  
 //If the corresponding sorted table doesn't have any element left  
 **else** {  
 deleteHeapElement(minHeap, heapSize);  
 }  
 }  
 }

**程序的第五步的通信部分。由于划分出来的数组显然不是定长的，因此先进行的是各线程各划分部分的size的相互通信，而后根据这些size数据在每个线程内生成对应大小的对应部分的划分数据的临时数组，并且直接将通信过来的数组数据拷贝到该临时数组对应的位置。（第六步的通信道理类似，所以就不额外放了）**

//Step 5: All \*ith\* classes are gathered and merged  
//The num of ith class in this process sent to other processes  
**int** \*numOfCorrespondingPartFromEachProcess = **new int** [numprocs];  
**for**(**int** p\_send = 0; p\_send < numprocs; p\_send++)  
{  
 **for**(**int** p\_recv = 0; p\_recv < numprocs; p\_recv++)  
 {  
 **if**(p\_send == myid)  
 {  
 **if**(p\_send != p\_recv)  
 MPI\_Send(&numOfEachPart[p\_recv], 1, MPI\_INT, p\_recv, p\_send, MPI\_COMM\_WORLD);  
 **else** numOfCorrespondingPartFromEachProcess[p\_recv] = numOfEachPart[p\_send];  
 }  
 **if**(p\_recv == myid)  
 {  
 **if**(p\_recv != p\_send)  
 MPI\_Recv(&numOfCorrespondingPartFromEachProcess[p\_send], 1, MPI\_INT, p\_send, p\_send, MPI\_COMM\_WORLD, &s);  
 **else**;  
 }  
 }  
}  
**int** \*startPosOfCorrespondingPartFromEachProcess = **new int** [numprocs];  
startPosOfCorrespondingPartFromEachProcess[0] = 0;  
**for**(**int** i = 1; i < numprocs; i++)  
 startPosOfCorrespondingPartFromEachProcess[i] = startPosOfCorrespondingPartFromEachProcess[i - 1] + numOfCorrespondingPartFromEachProcess[i - 1];  
**int** correspondingPartSize = startPosOfCorrespondingPartFromEachProcess[numprocs - 1] + numOfCorrespondingPartFromEachProcess[numprocs - 1];  
  
//The corresponding part of data will be delivered to the corresponding process  
**int** \*correspondingPartDataSet = **new int** [correspondingPartSize];  
**for**(**int** p\_send = 0; p\_send < numprocs; p\_send++)  
{  
 **for**(**int** p\_recv = 0; p\_recv < numprocs; p\_recv++)  
 {  
 **if**(p\_send == myid)  
 {  
 **if** (p\_send != p\_recv)  
 MPI\_Send(&myDataset[posOfEachPartStartFrom[p\_recv]], numOfEachPart[p\_recv], MPI\_INT, p\_recv, p\_send, MPI\_COMM\_WORLD);  
 **else** {  
 **for** (**int** i = posOfEachPartStartFrom[p\_recv]; i < posOfEachPartStartFrom[p\_recv] + numOfEachPart[p\_recv]; i++)  
 {  
 **int** pos = i - posOfEachPartStartFrom[p\_recv];  
 correspondingPartDataSet[startPosOfCorrespondingPartFromEachProcess[p\_send] + pos] = myDataset[i];  
 }  
 }  
 }  
 **if**(p\_recv == myid)  
 {  
 **if**(p\_recv != p\_send)  
 MPI\_Recv(&correspondingPartDataSet[startPosOfCorrespondingPartFromEachProcess[p\_send]], numOfCorrespondingPartFromEachProcess[p\_send], MPI\_INT, p\_send, p\_send, MPI\_COMM\_WORLD, &s);  
 **else**;  
 }  
 }  
}

**实验结果:**

**运行时间**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 规模, 步数\线程 | 1 | 2 | 4 | 8 | 16 | 24 | 32 |
| 33554432 | **29.537** | **11.419** | **4.509** | **2.109** | **1.361** | **3.674** | **4.89** |

**加速比**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 规模, 步数\线程 | 1 | 2 | 4 | 8 | 16 | 24 | 32 |
| 33554432 | **1** | **2.587** | **6.551** | **14.005** | **21.702** | **8.039** | **6.04** |

**以上的时间与加速比都是排序部分的时间统计。IO时间通过对比实验排除了。**

**分析与总结**

**该实验给出了一种可行的充分利用并行性能的排序方法。利用该方法以及多路归并算法可以轻松的实现很大的数据集的排序。**

**由于排序时使用的是qsort，因此当数据划分后，根据算法的平均时间复杂度O(nlgn)，(n/k)lg(n/k)的计算时间将比单纯的nlgn / k还要低，因此能出现超过线性加速比的现象出现。**

**另外，当核数多到一定程度的时候，通信的成本将过于庞大，使得算法的运行性能反倒下降了。**

**实际实现时，通信控制（也就是数据共享）其实才是最复杂的部分，这也是并行程序从古至今的一个老大难问题。如何巧妙的设计这一部分是并行程序能否高效执行的关键。**