Parallel Quick Sort using MPI

Qingkui Wang

Abstract

Sorting is one of the most fundamental problems in computer science. And sorting is an important part of high-performance multiprocessing. Parallel sorting has been studied in our csc503 course. We implemented the parallel merge sort. This project is parallelizing the quicksort algorithm using MPI. According to avoid the initial partitioning of data and merging step in all the processes, we could get a performance improvement for this implementation.

1. Introduction

Quicksort is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. It was developed in 1960 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University. When implemented well, it can be about two or three times faster than merge sort and heap sort.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-array: the low elements and the high elements. It can then recursively sort the sub-arrays.

1.1 Algorithm steps

1. Pick an element, called a pivot, from the array.
2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

1.2 Running time

Mathematical analysis of quicksort shows that, on average, the algorithm takes ***O (n log n)*** comparisons to sort n items. In the worst case, it makes **O (*n*^{2})** comparisons, though this behavior is rare.

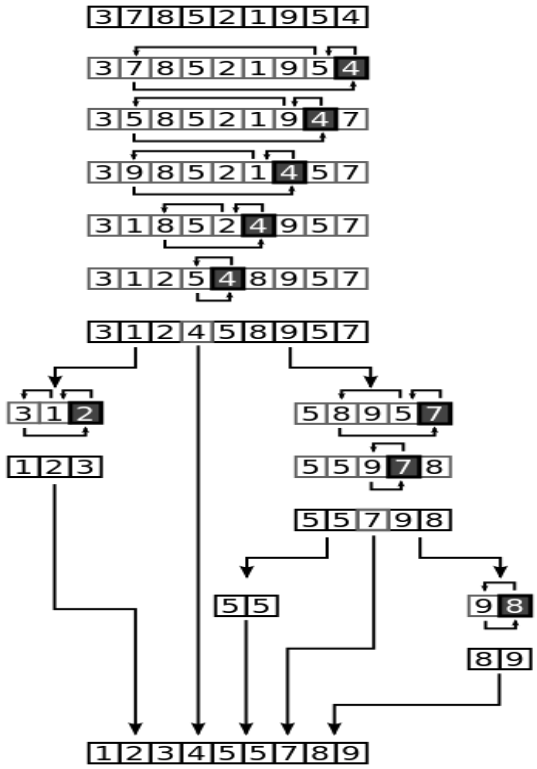
The base case of the recursion is arrays of size zero or one, which never need to be sorted.

1.3 Serial quicksort in C++

In C++ code, a quicksort that sorts elements ***left*** (l) through ***right***(r) (inclusive) of an array ***dataSet*** can be expressed compactly:

```
/*******************************************
Function: void quickSort(int *data, int start, int end)
Description: quicksort with recursion
Input: data: dataset to be sorted
       left: the start point of the dataset
       right: the end point of the dataset
Return: void
Others: recursion
*******************************************/
void quickSort(int *dataSet, int left, int right)
{
       int r;
       if (left < right)
       {
              r = partition(dataSet, left, right);
              quickSort(dataSet, left, r - 1);
              quickSort(dataSet, r + 1, right);
       }
}

/*******************************************
Function: int partition(int *data, int start, int end)
Description: divide array by the pivot
Input: dataSet: dataset to be sorted
       left: the start point of the dataset
       right: the end point of the dataset
Return: the partition position (int)
Others: pick the last element of the array as pivot
*******************************************/
int partition(int *dataSet, int left, int right)
{
       int pivot;
       int i, j;
       int tmp;
       pivot = dataSet[right];
       i = left - 1;
       for (j = left; j < right; j++)
       {
              if (dataSet[j] <= pivot)
              {
                     i++;
                     tmp = dataSet[i];
                     dataSet[i] = dataSet[j];
                     dataSet[j] = tmp;
              }
       }
       tmp = dataSet[i + 1];
       dataSet[i + 1] = dataSet[right];
       dataSet[right] = tmp;
       return i + 1;

}
```

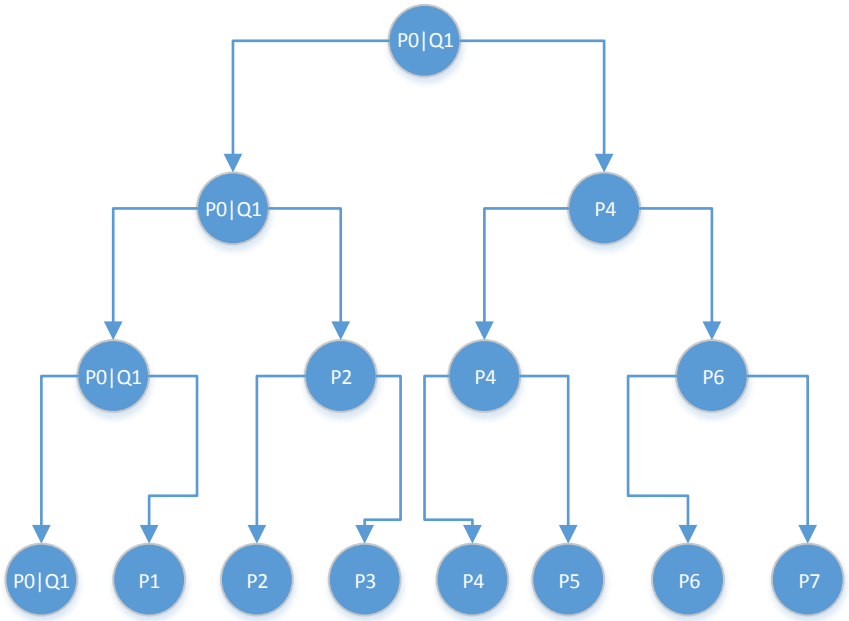
Here is an example of quicksort on a set of numbers. We can see the algorithm clearly through it.



2. Parallel Quicksort

Always, we can solve a problem with divide and conquer. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. In parallel algorithm, we can still use divide and conquer.

Let's explain it through the example and photograph below.



P is process, Q is question.

Assume we are solving a question Q1 using divide and conquer and parallel algorithm, we can divide and conquer it like this:

```
P0                                  // Started from process P0
divide(Q1, Q1, Q2);                 // Divide Q1 into Q1 and Q2
send(Q2, P4);                       // Send Q2 to process P4
divide(Q1, Q1, Q2);                 // Divide the Q1 into Q1 and Q2 again
send(Q2, P2);                       // Send Q2 to process P2
divide(Q1, Q1, Q2);                 // Divide the Q1 into Q1 and Q2 again
send(Q2, P1);                       // Send Q2 to process P1
sub_result = *S1;
recv(&sub_result1, P1);             // Gather the result from process P1
sub_result = sub_result + sub_result1;
recv(&sub_result2, P2);             // Gather the result from process P2
sub_result = sub_result + sub_result2;
recv(&sub_result4, P4);             // Gather the result from process P4
sub_result = sub_result + sub_result4;
```

Thus, we can get the final result, it equals to the ***sub_result*** above.

2.1 Implementation steps

2.1.1 Perform an initial data partition and give the two subsets to two available processes.

2.1.2 Each process performs the data partition if there is any available process. Then send a sub part of data to the available process.

2.1.3 Perform the ordinary quick sort on the sub data if no more process is available.

2.1.4 Send the sorted sub data to its sender.

2.1.5 Get the exact sorted data set.

2.2 Running time

On average, the serial quicksort algorithm takes ***O (n log n)*** comparisons to sort n items. Let's analysis the running time of parallel quicksort according this.

Assume that each partition creates two equal sub sets,

For 2 processes,

T(n) = (n/2)log(n/2).

For 3 processes,

T(n) = (n/2)log(n/2) + (n/4)log(n/4) = (n/2)log(n/2).

Actually, in my implementation, I consider log2(2) equals to log2(3).

So, for 3 processes,

T(n) = (n/2)log(n/2), indeed.

For 4 processes,

T(n) = (n/4)log(n/4)

Thus, for $\boldsymbol{p}$ processes,

T(n) = (n/k)log(n/k) where $\mathbf{2^{k-1} < p \leq 2^k}$

3. Description of MPI implementation

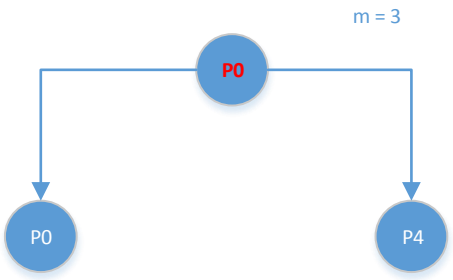
1. These code initialize the MPI environment. Get the rank of current process and the number of processes.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &CurrentID);
MPI_Comm_size(MPI_COMM_WORLD, &SumNum);
```

2. Take 8 processes as an example to show the process of the function ***praQuickSort()***

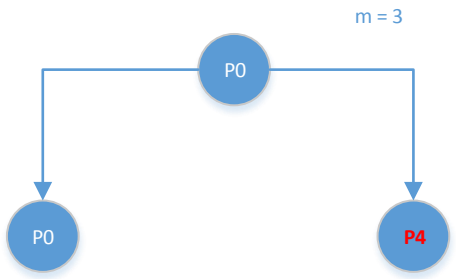
```
/*******************************************
Function: void paraQuickSort(int *dataSet, int left, int right, int m,
int id, int CurrentID)
Description: parallel quicksort using MPI
Input: dataSet: dataset to be sorted
           left: the start point of the dataset
           right: the end point of the dataset
           m: log2(number of processes)
           id: initial process id
           CurrentID: current process id
Return: void
*******************************************/
void paraQuickSort(int *dataSet, int left, int right, int m, int id, int
CurrentID)
```

3. The initial process id is ***0***. And the current process id is ***0*** too. So the process partitions the original dataset and send a subset to process ***4.***
   m = 3 and id = 0, so (id + pow(2, (m - 1)) = 4



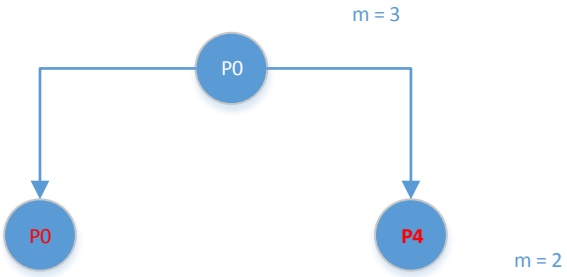
```
if (CurrentID == id)
{
                r = partition(dataSet, left, right);
                MyLength = right - r;
                MPI_Send(&MyLength, 1, MPI_INT, id + pow(2, (m - 1)),
CurrentID, MPI_COMM_WORLD);
                cout << "test here, "<< CurrentID << "MyLenght = " <<
MyLength << endl;
                if (MyLength != 0)
                {
                        MPI_Send(dataSet + r + 1, MyLength, MPI_INT, id
+ pow(2, (m - 1)), CurrentID, MPI_COMM_WORLD);
                }

}
```

4. For process ***4***, it received the dataset from process ***0***.



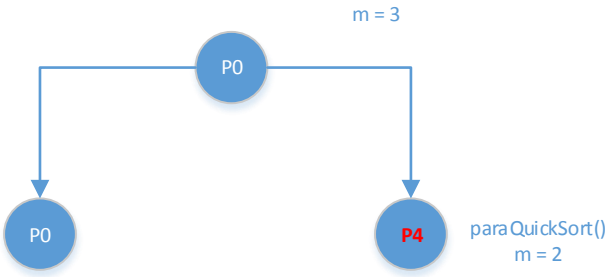

```
if (CurrentID == id + pow(2, (m - 1)))
{
                MPI_Recv(&MyLength, 1, MPI_INT, id, id, MPI_COMM_WORLD,
&status);
                if (MyLength != 0)
                {
                        tmp = (int *)malloc(MyLength*sizeof(int));
                        if (tmp == 0) perror("Malloc memory error!");
                        MPI_Recv(tmp, MyLength, MPI_INT, id, id,
MPI_COMM_WORLD, &status);
                }
}
```

5. Execute the recursion on process ***1*** and ***4***. And execute the ***Send()*** on process ***4***, ***Recv()*** on process ***0*** to finish the MPI communication.

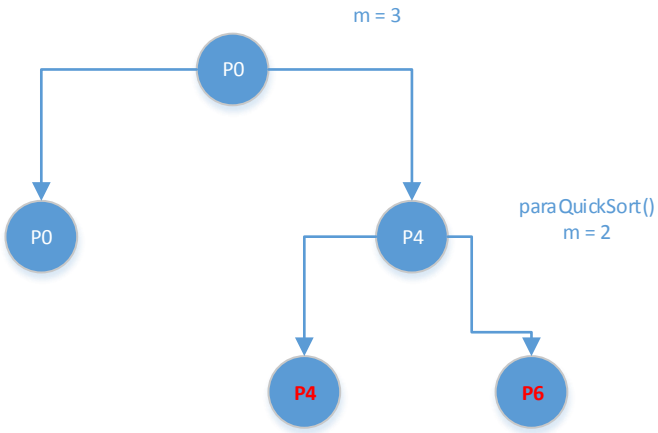



```
paraQuickSort(dataSet, left, r - 1, m - 1, id, CurrentID);
paraQuickSort(tmp, 0, MyLength - 1, m - 1, id + pow(2, (m - 1)), CurrentID);

if ((CurrentID == id + pow(2, (m - 1))) && (MyLength != 0))
        MPI_Send(tmp, MyLength, MPI_INT, id, id + pow(2,(m - 1)),
MPI_COMM_WORLD);
if ((CurrentID == id) && (MyLength != 0))
        MPI_Recv(dataSet + r + 1, MyLength, MPI_INT, id + pow(2, (m - 1)), id
        + pow(2, (m - 1)), MPI_COMM_WORLD, &status);
```

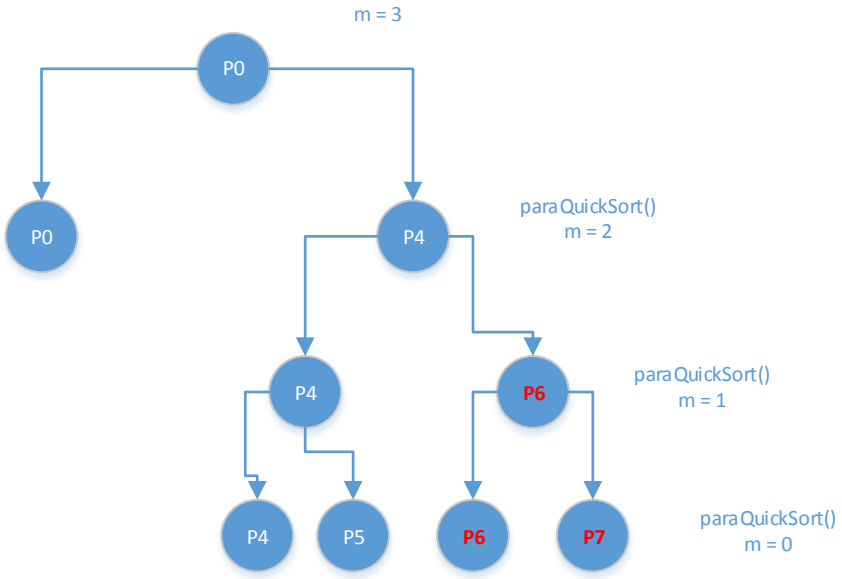
6. The recursion parallel quicksort on process ***4***.



7. The communication between process ***4*** and ***6***.



8. The communication between process ***6*** and ***7***.



Finally we finish the parallel quicksort with 8 processes using MPI.

4. Benchmark

I sorted 1MB and 2MB dataset in my experiment.
Since the size of integer is 4 bytes, I implement a java code for generating integer dataset. Take a 200000 integers dataset as 1M, 400000 as 2M.

Here is the execution time table.
1MB data set:

Number of processes	Parallel execution time(second)	Serial execution time(second)
2	0.115473	0.110023
4	0.121614	0.110023
8	0.110303	0.110023

2MB data set:

Number of processes	Parallel execution time(second)	Serial execution time(second)
2	0.338598	0.356089
4	0.273286	0.356089
8	0.252886	0.356089

From the result, we can conclude that parallel quicksort have advantage with big dataset. With the size of data set growing, it performance will be better and better. So is it with the growing of process number.

5. Conclusion

Through implementing the parallel quicksort using MPI, I got a chance to have a further understanding of MPI and parallel algorithm, especially sorting algorithm. I am thinking of implementing other parallel sorting algorithms. This is really interesting. To this quicksort, I am wondering if there is some better strategy of the pivot choice. Generally speaking, parallel algorithm is powerful in dealing with big data. Our world today is a big data world. We should bring more parallel algorithm into our real life, to improve the life quality.