Parallel quicksort implementation

ZHIYING LIN 4155-9867-81
LINGHAO GU 6375-4201-53
Computer Science Department
University of Southern California

Contents

ABSTRACT

Sorting is vital important problem in that a lot of applications need the sorted lists and improving the sorting performance will have a great improvement on the application directly. After analyzing all the sorting algorithms features, we choose the quicksort because of its divide-and-conquer nature. A problem can be divided into several subproblems. And each subproblem can be solved parallelly. We implement three parallel approaches (1)multithreaded (2)message-passing, and (3)a hybrid of the two afore-mentioned approaches. In addition, we do the memory hierarchy optimization on the serial quicksort. The experiments are aimed on measuring these four implementations from both time cost and energy consumption. A comprehensive evaluation shows that (1) when the number of thread created is fixed in the mutithread implementation, bigger problem size has better performance on the time cost in that the overhead of thread creation can be ignored. (2)Creating more threads in the multithread implementation will not lead to higher speedup directly. (3) Multicores get better performance improvement than dual-cores processor in the multithread implementation. (4) The serial quicksort works better than the multithread implementation in a dual-core machine measured by both time cost and energy consumption. (5) The memory hierarchy optimization for the quicksort has less cost on the energy than the serial one, even it has no improvement on the time cost. (6)When the number of node is fixed, the speedup will increase with the increment of the problem size in MPI implementation. (7) The speedup will increase with the increment of the number of nodes first and then it will decrease when the host number reaches some point because of the huge overhead in MPI implementation. (8)When the problem size is fixed, MPI setup time will take most part of the total time and run time will take the least in MPI. (9) When the problem size is fixed, the energy consumption will increase with the increment of the nodes' number and running the code consumes most of the energy in MPI. (10) In a pseudo distributed system, optimal topology does not increase the transmission speed of MPI jobs, compared with a star topology in MPI. (11) In a pseudo distributed system, the speedup of hybrid implementation is lower than MPI implementation because of the large overhead.

Keywords: quicksort, parallel, POSIX Thread, MPI

1 Introduction

"A sorting algorithm is an algorithm that puts elements of a list in a certain order"[1]. Sorting is a vital important problem in that a lot of applications need the sorted list and improving the sorting performance will have a great improvement on the application directly. For example, government organizations, financial institutions and commercial company usually organize the information in a certain order. Another application is widely used in the website company. The website company often wants to get the ranking of the products popular among customers. Thus, how to using the parallel techniques to improve the performance of sorting has important meaning to the application. Actually, not all the sort algorithm has the features which can be paralleled. After analyzing all the sorting algorithms features, we choose the quicksort because of its divide-and-conquer nature. A problem can be divided into several subproblems. And each subproblem can be solved parallelly. Additionally, it sequential and localized memory references work well with a cache. Besides, it can be implemented with an in-place partitioning algorithm, which demands little for the memory size.

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently.[2] Based on different implementation level, the parallel computing can be divided into three category: instruction level, data parallelism and task parallelism. In this project, we implement the quicksort using the task parallelism. In the task parallelism, we choose the (1)message-passing,(2)multithreaded, and (3)a hybrid of the two afore-mentioned approaches. Also, we implement and analyze a model for the memory hierarchy optimization based on the serial version of the quicksort.

We measures different quicksort implementations' performance from two aspects. One is time cost: time speedup compared with the serial version; another is energy consumption. More recently another physical consideration, power consumption, has taken on importance. It is a factor in individual processors for cell phones and laptops computers, in sensor networks, and in supercomputers.[3] More energy consumption brings board negative effect on both environmental and economic aspects[4, 5, 6], especially in the distributed systems and clusters which use multiple computers to work on the same task.

The report is organized as follows. Section 2 describes the background of quicksort in more details. We present the three approaches for quicksort parallel implementations in section 3. Section 4 gives more details on implementations for different approaches. This is followed by a section reporting on experimental evaluation. Section 6 summarizes our findings and outlines work for the work.

2 Background

Quicksort is a sorting algorithm developed by Tony Hoare that, on average, make O(nlgon) comparisons to sort n items. In the worst case, it takes $O(n^2)$. [7]

Quicksort is a divide and conquer algorithm. It first divides a list into two parts: the low elements and high elements. And it can recursively sort the whole lists.

The steps are: [7]

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all the elements with the value which is less than the pivot and come before the pivot, while all the elements are greater than the pivot come after it. After the partitioning, the pivot is in the final and right position.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.
4. Go to step 1 until the size of list is zero or one, which never need to be sorted.

```
void quicksort(double * A, int p, int r)
{
    if(p<r)
    {
        int q= partition(A, p, r);
        quicksort(A, p, q-1);
        quicksort(A, q+1, r);

    }
}
```

The three-step divide-and-conquer process for sorting a subarray A[p…r]. [8]

Divide: Partition the array A[p…r] into two(possible empty) subarrays A[p….q-1] and A[q+1…r] such that each element of A[p…q-1] is less than or equal to A[p], which is, in turn, less than or equal to each element of A[q+1…r], shown in Figure 1. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p…q-1] and A[q+1…r] by recursive alls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array A[p…r] is now sorted.

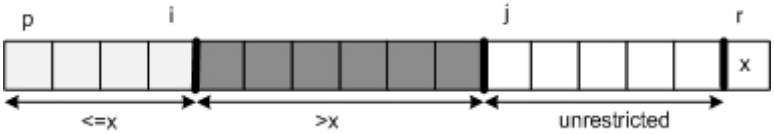

Figure 1. The four regions maintained by the procedure PARTITION on a subarray A[p…r].

3 Approaches Taken for Parallel Quicksort Implementation

3.1 Approach for POSIX multithreading(pThreads)

In pThreads, we divide the array size first. Based on the number of the child threads, divide the whole array averagely to subarrays. And assign each subarray to each thread. Then for each subarray, each thread uses the quicksort to sort the subarray. Each child thread does not need any auxiliary memory and communication in that they share the memory and it is easy to partition the problem to each thread for the parent thread. Besides, each subproblem is independent, which means that it has no race condition and no synchronization between the child threads. When each child threads finish sorting the subarray, the parent thread can start to uses the heap to merge the sorted subarray.

The steps for merging the sorted subarray are present as follows, shown in Figure 2:

1. Take the element with the maximal value in each subarray and build a max-heap.
2. And then extract the element, which comes from i subarray(1<=i<=number of the thread) with the maximal value from the heap.
3. Put this element as the next maximal element of the new array.
4. Then remove the next maximal element from the ith subarray and insert into the heap.
5. Repeat the steps 2-4 until we sort all the elements in each subarray.

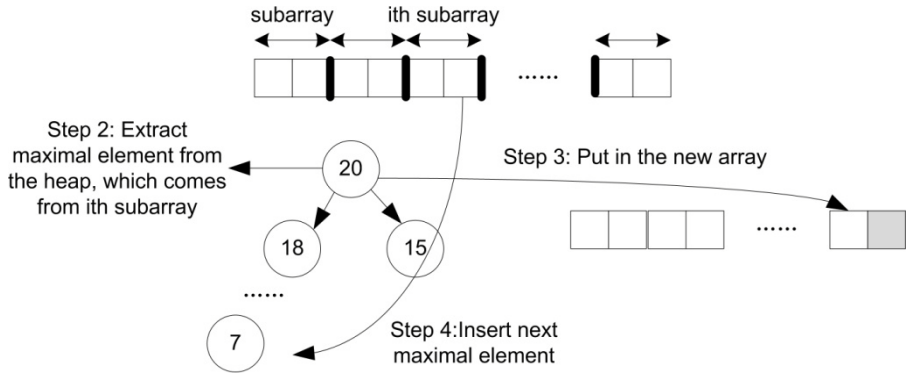

Figure 2. Steps taken for merging the sorted subarray.

In this approach, it will cause some overhead for the synchronization between the parent thread and child threads and creating child threads. The major advantage here is the load balancing is obtained since the threads are working on the subarray simultaneously. And they have no data dependency. They share the same memory and have no communication time. But compared with the serial version, the pThread approach needs merge operation for the whole array. Besides, more threads will lead to increase on the energy consumption of the processor.

Assume we create T child threads and the size of the array is N, on average, it will take $O(\frac{N}{T}\lg\left(\frac{N}{T}\right) + N\lg T)$, while the serial version will take $O(N\lg N)$.

3.2 Approach for memory hierarchy optimization

In memory hierarchy optimization, we design this approach based on the serial version of the quicksort. We try to make use of the features of the memory hierarchy to speed up the performance of the quicksort. Most programs have a high degree of data reuse in their accesses.(1) spatial reuse: accessing things nearby previous accesses. (2) temporal reuse: reusing an item that was previously accessed. In order to reuse the computation, we rearrange computation so that accesses to same data are closer together in time and the cache misses will be decreased.

We divide the whole array into several blocks. And each block have the fixed size. For each block, we do the sorting and then merge all the sorted blocks. The process of merging the sorted blocks is similar to the one of pThread's, which is shown in Figure 2. The rationale is that when the problem size is too huge, the processor cannot load all the data into the cache and it has a greater chance to have more read or write misses. It will increase the time cost of the quicksort and slow down the performance. Meanwhile, more cache misses also will lead to more power for reading the data from the memory.

However, this approach has overhead on the merging operation and this operation will increase the cache misses unavoidably when the problem size is huge. In this part, the processor cannot reuse the data and it should read the value from each blocks. And the positions of the elements from blocks are not close to each other. Also, merging operation takes a new element from blocks each time and there is no temporal reuse. This factor will slow down the performance of the memory hierarchy optimization compared with the serial version.

Assume we create T blocks and the size of the array is N, on average, it will take $O(N \lg\left(\frac{N}{T}\right) + N\lg T)=O(N\lg N)$, which is the same as the serial version. Though the equation seems that the block size is not related with the performance, a suitable block size will decrease the cache misses greatly and have an impact on the time cost.

3.3 Approach for MPI

In MPI, the master node separates its data and then sends data to different slave nodes. Slave nodes sort the given data set using quicksort and then send them back to the master node. The method used here is similar to the method used in pThreads. But here we use distributed memory rather than shared memory. That is, if we ignore the communication overhead, the speed of MPI is at least as fast as pThreads. As pThreads has limited resources, with the increasing number of nodes, MPI implementation will be much faster than pThreads implementation.

In MPI, we divide the array size first. Based on the number of the host, divide the whole array averagely to subarray. Then send each subarray to each host. In each host, use quicksort to sort the subarray. After that, slave node sends its data to the master node. After receiving all data, the master node uses heap to merge. The method to sort subarray is showing in section 3.1

One advantage of this technique is that communication time is kept low by sending only a subsequence to each process. We only send those data to a node which is relate to it which can efficiently reduce the overhead of communication.

However, this method has many disadvantages.(1)Load balancing may not be achieved because one node may finish its job faster than other nodes. So this node may wait for sending operation while other nodes are still doing their job. It results in process idle and is not efficient. (2) If the problem size is not large, the percentage of overhead cannot be ignored and it will have a negative effect on the performance.

3.4 Approach for the hybrid of MPI and pThreads

In the hybrid method, we divide the array size first. Based on the number of the host, divide the whole array averagely to subarrays. Then send each subarray to each host. In each host, distribute the array first and based on the number of the child threads, divide the array averagely to subarray. After assigning each subarray to each thread, each thread uses quicksort to sort the subarray. When each child thread finishes sorting the subarray, the parent thread can start to uses the heap to merge the sorted subarray. The method to sort subarray is shown in section 3.1

This approach makes use of the resources. By using MPI, it uses distributed system to speed up the running time. By using pThreads, it makes use of all the processors in a node. In general, almost computers have more than one processor.

Meanwhile, there are several disadvantages: (1)Since it uses many resources, the power it uses is more than any other approaches. (2)Like MPI, load balancing is also not achieved in this approach. (3)The overhead of this approach is much more than any other approaches because it has overhead for using both MPI and pThreads.

4 Implementation

All implementations and experiments are done by sorting double values generated by a random function. Using the double values increases the complexity of the computation in that the float pointing operations takes more time than the integer to the processor. The random input is saved in a file. Each implementation has the same input and results should also be the same. Size of double is considered as 8 bytes since a sizeof(double) operation reveals that double is 8 bytes on the cluster.

To sort N numbers, random numbers are created and saved in a file as follows:

```
srandom(clock());
for(i=0;i<n;i++)
    data[i] = random();
```

4.1 Implementation of pThread

The parent thread divides the whole problem into several subproblems and then creates several child threads. It assigns each subproblem to each child thread. Each child thread solves the subproblem individually and independently. Thus, there is no synchronization and race condition between child threads. This is implemented as follows:

```
void * quicksortSubRoutine(void * ptr){
    subproblem * sub=(subproblem *)ptr;
    quicksort(sub->array, sub->row_start, sub->row_end);
    return NULL;
}
```

Each child thread does not need any auxiliary memory and communication in that they share the memory and it is easy to partition the problem to each child thread for the parent thread. The parent thread gives the information to each child thread through shared memory. This is implemented as follows:

```
    int averageRow=arraySize/threadNumber;
    int remainder=arraySize%threadNumber;
    int offset=0;
    int size=0;

    int i;

    for(i=0;i<threadNumber;i++)
    {
        size=(i<remainder)?(averageRow+1):averageRow;
        sub[i].array=array;
        sub[i].row_start=offset;
```

```
        sub[i].row_end=offset+size-1;
        offset+=size;
        pthread_create( ( pthread_t * ) &(thread[i]), NULL,
quicksortSubRoutine, (void *) &sub[i] );
    }
```

The parent thread starts to do the merge operation until all the child threads finish the sorting. Then, the parent thread builds a maximum heap. The number of nodes in the heap is the number of the child threads. Extract the maximum node which belongs to the ith subarray and put it into the new array as the largest element. It continues to remove the element with the maximal value from the ith subarray. Then insert the element until all the elements are put in the new array. This is implemented as follows:

```
double * sortedArray=(double *)malloc(sizeof(double)*arraySize);
    build_max_heap(heap, heapSize);
    while(counter!=arraySize)
    {
        heapNode max=heap_extract_max(heap, &heapSize);
        counter++;
        sortedArray[arraySize-counter]=max.value;
        ptr[max.index]--;
        if(ptr[max.index]>=sub[max.index].row_start)
        {
            max.value=array[ptr[max.index]];
            max_heap_insert(heap, &heapSize, max);
        }
    }
```

4.2 Implementation of memory hierarchy optimization

We divide the whole array into several blocks. And each block have the fixed size. For each block, we do the sorting serially in order to observe the memory hierarchy's impact on the performance of this implementation, such as time cost and energy consumption, compared to the serial one. This is implemented as follows:

```
int number=(int)ceil((float)arraySize/(float)blockSize);

for(int i=0; i<number-1; i++)
{
        quicksort(array, i*blockSize, (i+1)*blockSize-1);
}
```

After getting the sorted blocks, we merge all the sorted blocks using a max-heap, which is same as the one used in the pThread implementation.

4.3 Implementation of MPI

Firstly, the master node divides the whole array into several parts and sends each part to different nodes. We need to send two kinds of data: size of the array and the array. By using the size of the array, we can dynamically allocate memory for the array in the slave node.

```
time=read_timer();
sub= (subproblem *)malloc(sizeof(subproblem)*numPro);
int i;
for(i=0;i<numPro;i++)
{
    size=(i<remainder)?(averageRow+1):averageRow;
    sub[i].array=array;
    sub[i].row_start=offset;
    sub[i].row_end=offset+size-1;
    offset+=size;
    int temp=size;
    if(i!=0){
        MPI_Send(&temp,1,MPI_INT,i,100,MPI_COMM_WORLD);
        MPI_Send(&array[sub[i].row_start],size,MPI_DOUBLE,i,101,MPI_COMM_WORLD);
    }
result.array=array;
result.row_start=sub[0].row_start;
result.row_end=sub[0].row_end;
}
(overhead)+=read_timer()-time;
```

Slave nodes have to receive data at the same time. After the slave node receives the size of the array, it should allocate memory for this array and then uses this piece of memory to store the array.

```
MPI_Recv(&size,1,MPI_INT,0,100,MPI_COMM_WORLD,&status);
subArray=(double*)malloc(size*sizeof(double));
MPI_Recv(&subArray[0],size,MPI_DOUBLE,0,101,MPI_COMM_WORLD,&status);
result.array=subArray;
result.row_start=0;
result.row_end=size-1;
```

After that, we use the quicksort algorithm mentioned in section 4.1 to sort the array in every node. Then, the slave nodes send back data to the master node. At the same time, the master node receives data and combines them into one whole array.

```
if(rank!=0){
    size=result.row_end-result.row_start+1;
    MPI_Send(&result.array[0],size,MPI_DOUBLE,0,200,MPI_COMM_WORLD);
}
if (rank==0) {
    time=read_timer();
    int i=0;
    int start=sub[i].row_start;
    int size=sub[i].row_end-sub[i].row_start+1;
    for (i=0; i<size; i++) {
        array[i]=result.array[i];
    }
    for (i=1; i<numPro; i++) {
        start=sub[i].row_start;
        size=sub[i].row_end-sub[i].row_start+1;
        MPI_Recv(&array[start],size,MPI_DOUBLE,i,200,MPI_COMM_WORLD,&status);
    }
```

After combining subarray into one whole array, we merge all the sorted blocks using a max-heap, which is same as the one used in the pThread implementation.

4.4 Implementation for hybrid of MPI and pThreads

As we have implemented MPI and pThreads, the hybrid method is quite simple to implement. We use MPI to send data to slave node which is same as the one used in section 4.3. In each node, we use the method mentioned in section 4.1 to sort the array.

5 Performance Evaluation

In this part, we did several experiments to evaluate the comprehensive performance of each parallel quicksort implementation from two aspects, time cost and energy consumption. We describe how they were done and draw our conclusions from the experiments' result.

5.1 pThread implementation

5.1.1 Performance with the problem size

To evaluate the relationship between the performance of the pThread implementation and the problem size of the array, we did the experiments on the aludra.usc.edu, which has 128 sparcv9 processor. And each processor operates at 1415MHz. We fixed the number of thread to two, three and four. And change the problem size to 128, 256, 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k and 512k to observe different problem size's impact on the performance of pThread implementation. In this experiment, we measured the speedup compared to the serial quicksort and the overhead needed to create threads.

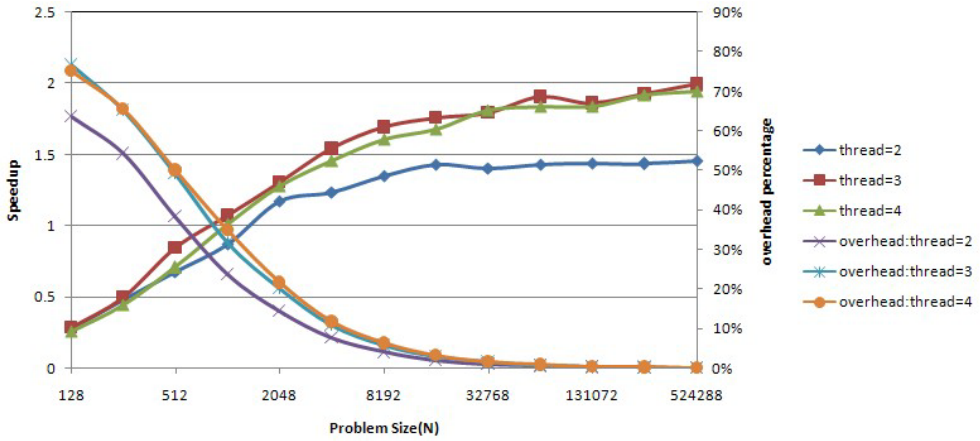

Figure 3. The relationship between problem size and speedup.

The results of pThread approaches with two, three and four threads are displayed in Figure 3. The diagram has two Y axes. The Y axis(left) is the value of speedup compared with the serial quicksort while the right one is the overhead(time cost to create threads) percentage. No matter there is two, three or four threads, the overhead percentage decreases with the increment on the problem size. From the figure, we can see that when the problem size is too small and it is less than 2048, the pThread approaches took more time than the serial one. The rationale is that the overhead of creating the threads is more than the decrement on the time cost of the pThread approaches. When we increase the problem size, the speedup is increasing correspondently. But the maximum speedup we can get is 2. There are several reasons to explain the result. First, the input array is created by random and decides the time complexity. Second, the aludra sever is public to all the USC students and is controlled by the ITS. Every student has limited resources to test his/her code. The limited resources include limited memory and CPU. Thus,

we cannot make use of all the 128 processors on the aludra. Thus, the results are what I have expected.

5.1.2 Performance with the thread number

Next, we do the experiments to explore the relationship between the performance and the number of the threads. We fixed the problem size to 256k, which means that the array contained 262144 double elements. We did the experiments on the aludra.usc.edu, which has 128 sparcv9 processor clocked at 1415 MHz. We make the problem size large enough to decrease the negative impact of the overhead of creating threads. We changed the number of threads from 2 to 100. And the experiments' result is shown in Figure 4. The diagram has two Y axes. The Y axis(left) is the value of speedup compared with the serial quicksort while the right one is the overhead(time cost to create threads) percentage. The blue line stands for the speedup while the red one stands for overhead percentage. From Figure 4, we can see that when we increase the number of threads, the overhead percentage is increasing correspondently. More threads do not lead to a better speedup. There are several reasons to explain it. First, the overhead of creating threads is too huge and it is more than the time we can save using the pThread approach. Another reason is that we use the aludra sever, which is public to all the USC students and is controlled by the ITS. Every student has limited resources to test his/her code. The limited resources include limited memory and CPU.

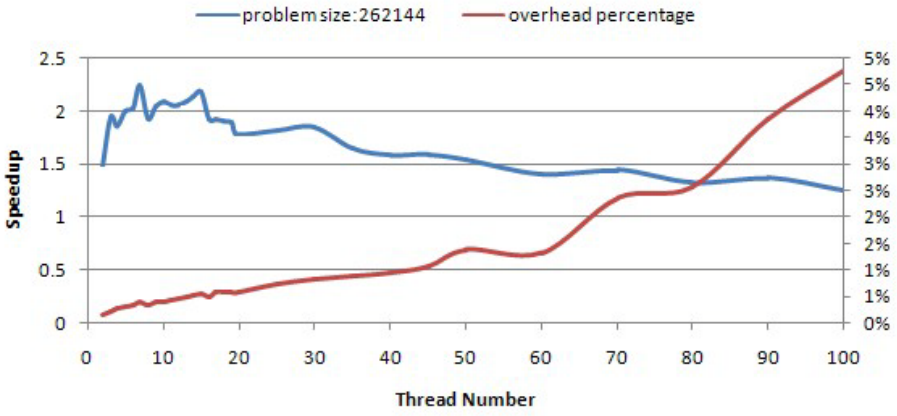

Figure 4. The relationship between the thread number and speedup.

5.1.3 Performance with the number of cores

In this part, we ran the code in different machines to explore the relationship between the number of cores and speedup. One is AMD Athlon(tm)X2 DualCore QL-66 processor clocked at 2200 MHz, another is Intel Core i7 4 cores clocked at 2600MHz and the last one is the aludra.usc.edu, which has 128 sparcv9 processor clocked at 1415 MHz. We fixed the thread number to two and changed the problem size from 128 to 256k. The Figure 5 describes the experiments' results. We can conclude that when the problem size is less than 1024, the performance of pThread approaches of dual or four cores is better than the one of 128 cores. More cores do not bring a better performance when the problem size is small. However, when

the problem size is huge enough, the 128 cores has much better speedup than the others because of more cores and more memory. From the figure, we find that the speedup of dual cores processor is less than one in most situations, which is worse than the serial version. If the number of threads is equal or more than the number of cores, the gaining on the performance will decrease. The speedup depends on the underlying hardware architecture greatly.

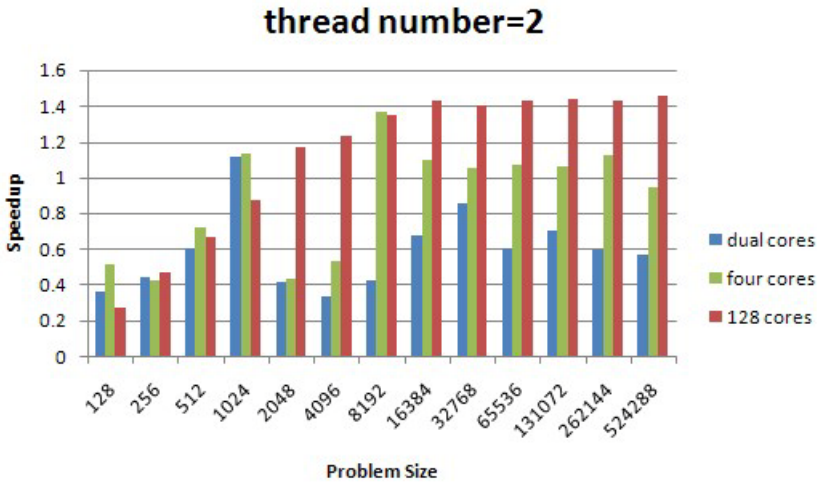

Figure 5. Performance in three different processors.

5.1.4 Performance compared with serial quicksort

In this experiment, we fixed the number of threads to two and ran the code on the AMD Athlon(tm)X2 DualCore QL-66 processor clocked at 2200 MHz. We measured the performance from two aspects, time cost and power consumption. We tested the code when the problem size is 64k, 128k, 256k and 512k. The Figure 6 summarizes the experiments' results. The diagram has two Y axes. The Y axis(left) is the value of speedup compared with the serial quicksort while the right one is the average power each approach cost. The figure shows that both time cost and average power of the parallel quicksort is more than the one of serial quicksort. If the number of threads is equal or more than the number of cores, the gaining on the performance will decrease. The speedup depends on the underlying hardware architecture greatly. Also, creating and scheduling threads will cost more energy than the serial quicksort. In a dual cores machine, a serial quicksort is a better choice than the parallel one.

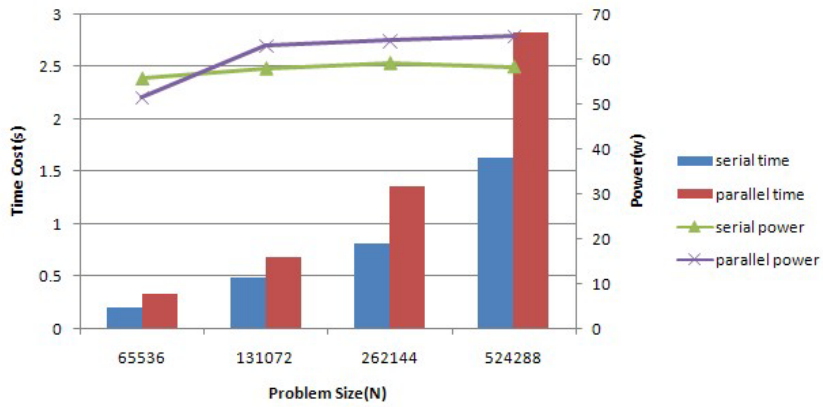

Figure 6. Performance on the dual cores machine

5.2 Memory hierarchy optimization implementation

5.2.1 Performance compared with serial quicksort

In this experiment, we did four kinds of problem size on the two implementations: serial quicksort and quicksort with memory hierarchy optimization. The problem sizes are 65536(64k), 131072(128k), 262144(256k) and 524288(512k). And each problem size was divided to small blocks with different size from 64 to 65536(64k). We ran the code on the AMD Athlon(tm)X2 DualCore QL-66 processor clocked at 2200 MHz. We measured the performance from two aspects. One is speedup compared with the serial quicksort shown in Figure 7. The other is the instantaneous power shown in Figure 8. From the Figure 7, we can find that in most cases, the serial version is better than the memory hierarchy. Dividing problem size into several blocks does not decrease the time cost in that the merge operation takes more time than the time saved from the memory misses. Given different problem size, when the block size is 32768(32k), the data in a block will occupies 256KB and the size is the same as L1 cache's size in this machine. The Figure 7 shows that when the block size reaches 32k, it has a higher speedup than others. It can fit in the fast memory and it will be still in fast memory when reused, thus increasing data locality. In Figure 8, we can conclude that the memory hierarchy optimization approach has less energy consumption compared the serial one, no matter from the average or range of instantaneous power.

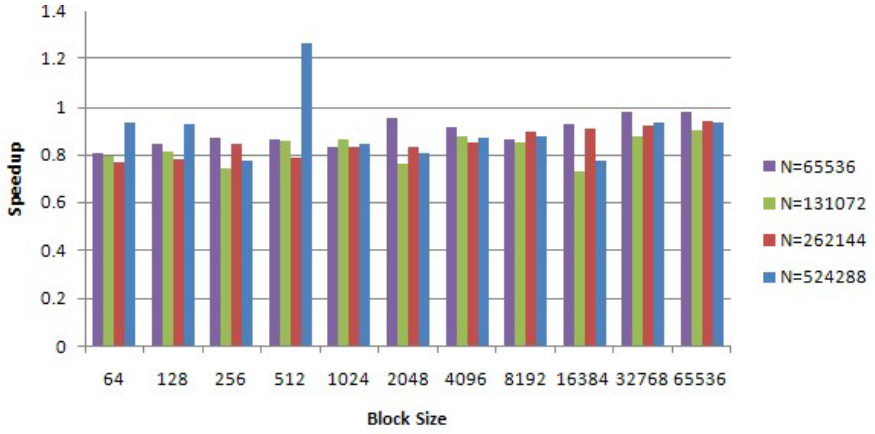

Figure 7. Speedup with different block sizes.

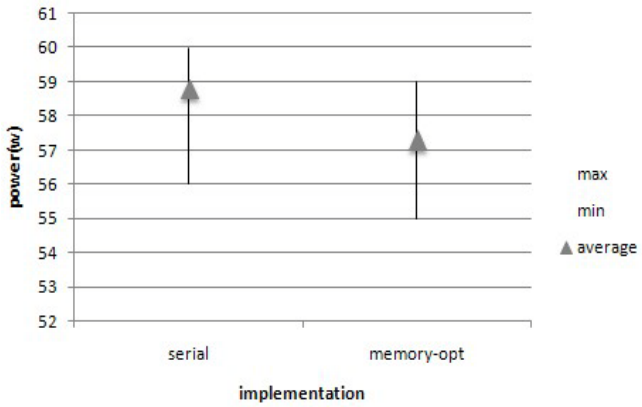

Figure 8. Instantaneous power with different implementations.

5.3 MPI implementation

We did the experiments on a machine which has four Inter Core i7 processor clocked at 2.6GHz. We build a virtual cluster on the machine. All the nodes use the Debian operating system. The master node is given 512MB RAM. Each slave node is given 256MB RAM.

5.3.1 Performance with the problem size

To evaluate the relationship between the performance of the MPI implementation and the problem size of the array, we fixed the number of nodes to two, four, six and eight. And change the problem size to 128, 256, 1k, 2k, 4k, 4k, 8k, 16k, 32k, 64k, 128k, 256k and 512k to observe different problem size's impact on the performance of MPI implementation. In this experiment, we measured the speedup compared to the serial quicksort and the overhead of communication.

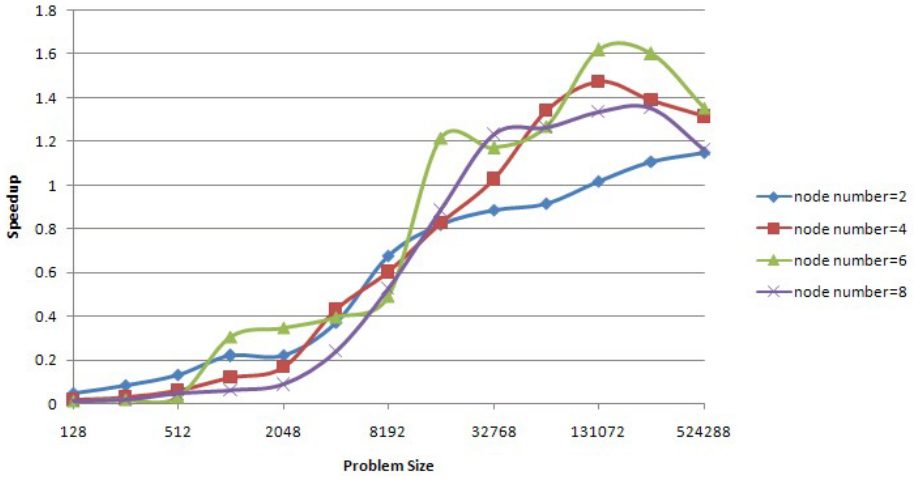

Figure 9 Speedup with problem size

The result of MPI approaches with two, four, six and eight nodes are displayed in Figure 9. The Y axe in the diagram is the value of speed up. From the figure, we can see that when the problem size is too small, the MPI approach takes more time than the serial one. The rationale is that the overhead of communication is more than the decrement on the time cost of the MPI approaches. When we increase the problem size, the speedup is increasing correspondently. But from Figure 9, we can see that the speedup will decrease after some point. For example, when node number is 8, the maximum value is 1.61. After that, the speedup will decrease. In section 5.3.3, we will discuss this problem. We find that one reason is that all experiments are done on one physical machine and all nodes shared the same CPU, memory and bus. So nodes are not independent. Besides, running so much operating system on the same machine produces a lot of overhead. We can figure out in the next section that the decrement has nothing to do with the problem size. So, with the increment of problem size, the performance will be better.

5.3.2 Performance with the node number

Next, we do the experiments to explore the relationship between the performance and the number of nodes. We fixed the problem size to 512K, which means that the array contained 524288 double elements. We changed the number of nodes from 2 to 8. And the experiments' result is shown in Figure 10. Y axis in the diagram is the value of speedup compared with the serial quciksort.

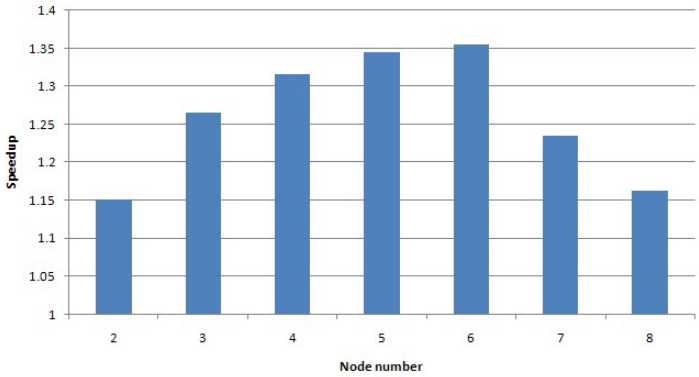

Figure 10. Speedup with different node number

From Figure 10, we can see that when we increase the number of slave nodes, the speedup will also increase. After we sent data set to different nodes, these nodes sorted subarray independently. So, the speedup will increase with the increment of the number of nodes. However, it will decrease when the host number reaches seven because of the huge overhead when MPI initialize its nodes.

5.3.3 Constitution of a MPI job

A MPI job has the following part: MPI setup, Communication, Running the program, Reading input file. In order to find out the relationship between them, we did a series of experiments to explore it. We fixed the problem size to 512K. And we change the host number to two, five and eight.

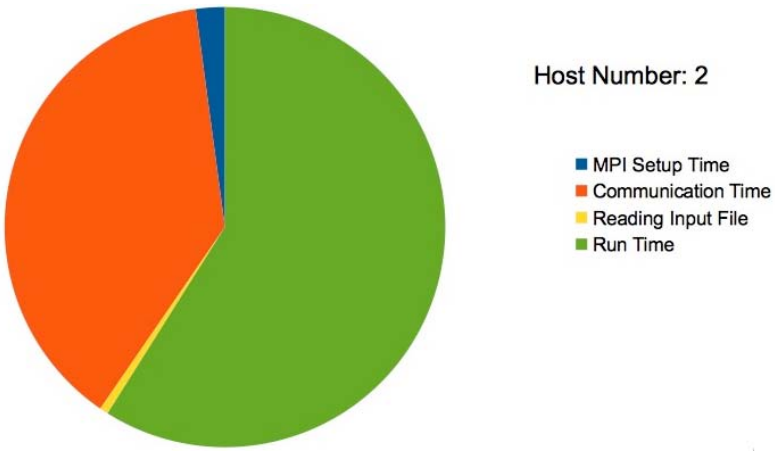

Figure 11. Constitution with 2 nodes

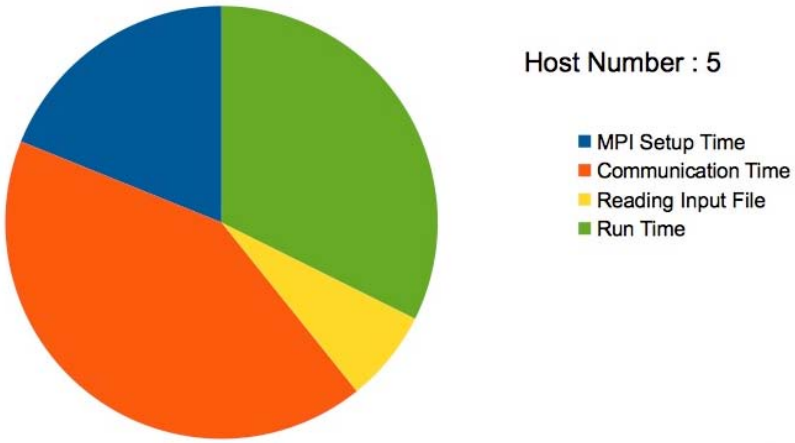

Figure 12. Constitution with 5 nodes

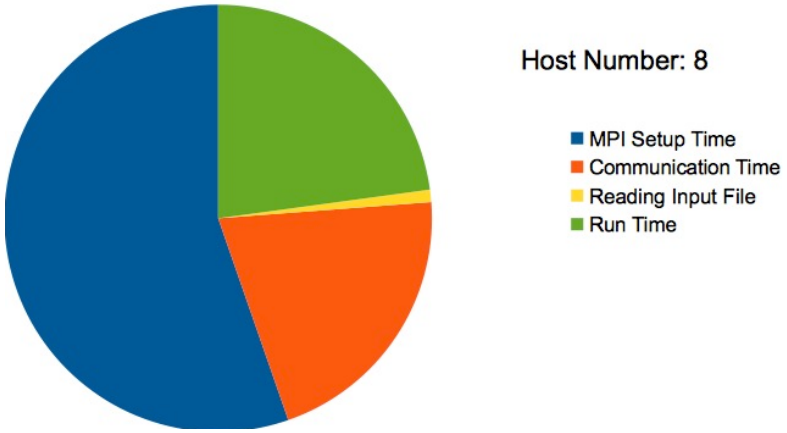

Figure 13. Constitution with 8 nodes

From Figure 11, 12 and 13, we can see that the part of running time became smaller when the number of host increases. The MPI setup time became larger when the number of host increases. We can make a conclusion that if the problem size is fixed. There exists a number that before this number, the speedup will increase while after this number, the speedup will decrease. Because after this point, the overhead's increment is higher than the running time's decrement. That can explain the problem which we encountered in section 5.3.1

5.3.4 Instantaneous power

In this part, we focus on the constitution of power consumption when we use MPI implementation. Here, we fixed our problem size to 512K, and changed our node to two, five, eight. Here, we turned on all eight nodes at the same time. In the real distributed system, nodes are always turning on and we don’t take into account the power consumption to keep these nodes working normally. We only consider the power consumption to run the code.

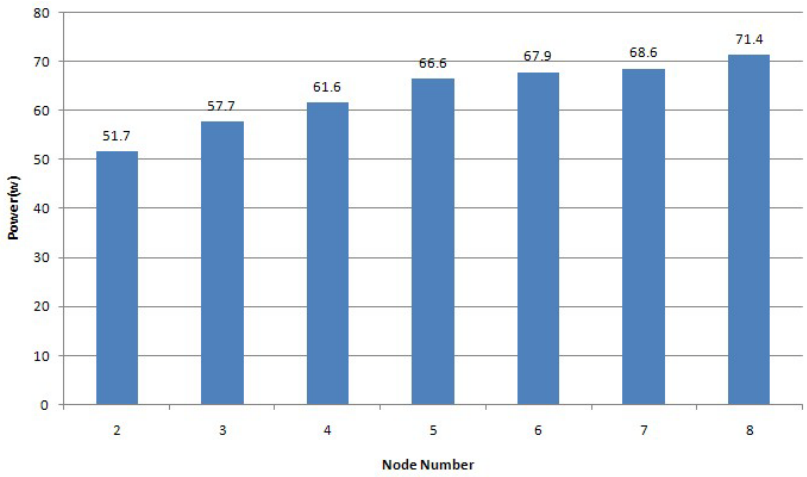

Figure 14. Instantaneous power with node number.

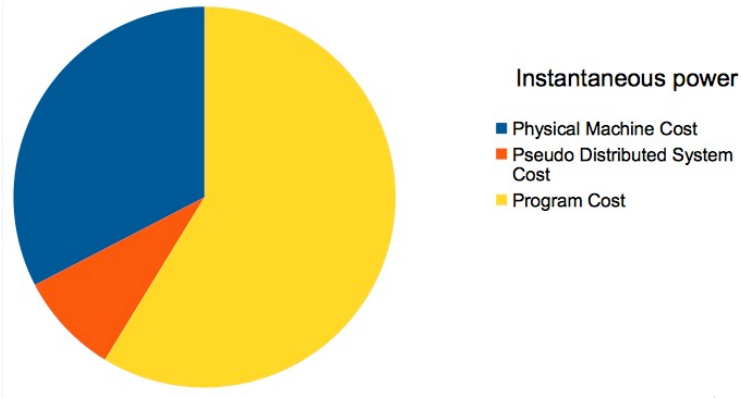

Figure 15. Constitution of the instantaneous power

In Figure 15, physical machine cost means the power that used to maintain the machine turning on. Pseudo distributed system cost means the power that used to maintain the virtual machine cluster. Program cost means the power to run the program on this cluster. From Figure 15, we can conclude that if we don't consider the power consumption of keeping nodes on, the number of nodes has impact on the power consumption if we only use a single machine. With the increasing number of nodes, the power consumption will also increase. Here we can't make a conclusion for a real distributed system. From Figure 15 we can concluded that the program part will use most of the power, keeping on the virtual machine only take a small part of the power consumption.

5.3.5 Performance with transmission topology

In this experiment, we explore the relationship between transmission topology of the MPI among the nodes and performance. We implemented anther approach of MPI using another topology among the nodes in the cluster, called optimal topology here. This method uses a different way to transfer data from the master node to the slave nodes which is faster than the original way.

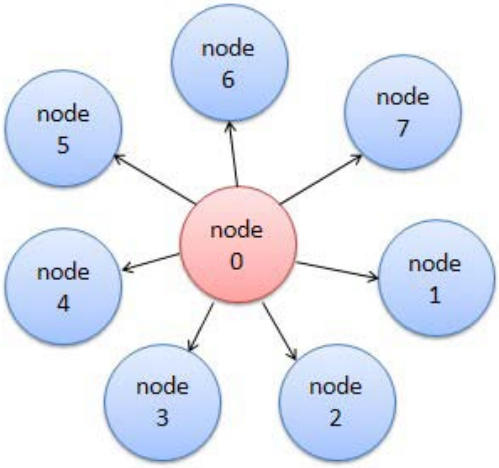

Figure 16. star topology

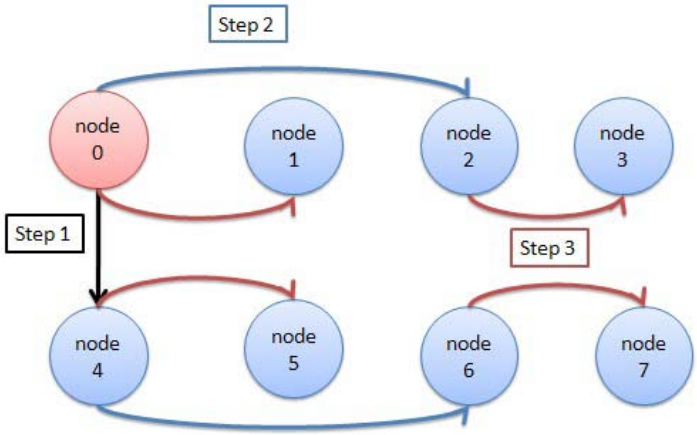

Figure 17. Optimal topology

Figure 16 shows the model we used to transfer data to slave nodes in the implementation of section 4.1. Here we will use Figure 4 as our model to transfer data. Assume that the master node will send the whole data set to the slave node and the time it used is a and the overhead of sending data using MPI is b. If nodes' number is 8, star topology will use $7*(a+b)$ time to transfer data. Optimal topology will only use $3*(a+b)$ time. But in this project, since every slave node will only get the data set belongs to its own. If we use star topology, time cost is $7b+a$. If we use optimal topology, time cost is $3b+a$. So, in this project, we will not save much time. But this method is excellent if we have a large data set and need to send the whole data set to every slave node.

In optimal topology, we assume that we have 8 nodes and node 0 is the master node shown in Figure 17. The first step described in black line is that node 0 sends data to node 4. The second step described in blue line is that node 0 sends data to node 2. At the same time, node 4 sends data to node 6. The third step described in red line is that node 0, node 2, node 4, node 6 send data to node 1, node 3, node 5, node 7 respectively.

In this part, we focus on how much optimal topology increases the speedup of MPI compared with the serial quicksort. Since it will impact the speed of transferring data, so here we only recorded time that the transferring time from master node to slave nodes. In this experiment, we fixed the host nodes to 8. And change the size of problem size to 128, 256, 1k, 2k, 4k, 4k, 8k, 16k, 32k, 64k, 128k, 256k and 512k.

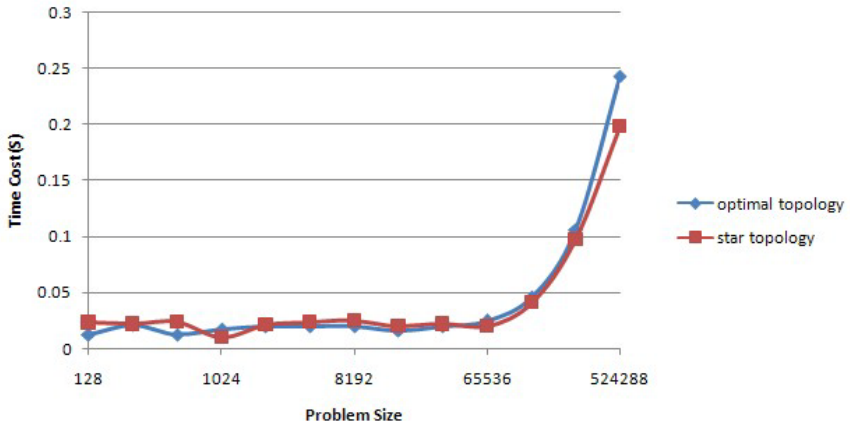

Figure 18. Constitution of the instantaneous power

From Figure 18 we can see that optimal topology always take longer time than star topology. That's not what we expected. But, since all the nodes are on only one physical machine, they shared the same I/O. So, instead of doing work parallelly, all works are done serially. But we believe in a real distributed system, using optimal topology for transmission in MPI has a better performance than the start topology.

5.4 Hybrid of MPI and pThread implementation

We did the experiments on a machine which has four Inter Core i7 processor clocked at 2.6GHz. We build a virtual cluster on the machine. All the nodes use the Debian operating system. The master node is given 512MB RAM. Each slave node is given 256MB RAM.

5.4.1 Performance compared with MPI implementation

In order to see the advantages and disadvantages of hybrid implementation, we compared it with MPI implementation. We fixed the number of hosts to 8. And change the problem size to 128, 256, 1k, 2k, 4k, 8k, 16k, 32k, 64k, 128k, 256k and 512k to observe different problem sizes' impact on the performance. In the hybrid implementation, we change the thread number to two, four, six.

In Figure.19, we can see that MPI implementations are always faster than hybrid implementation. That is not what we want to see. Because that we have talked about the relationship between the number of thread and the performance. With the increasing number of thread, the performance will also increase. But in this experiment, that is not the case. As we have mentioned in section 5.5.1, all experiments are done a single physical computer and all nodes use virtual machines sharing one physical machine. So, they shared the same memory and the same sets of cores. Besides, all of the nodes only use one core. Since hybrid implementation has more overhead than

MPI implementation, hybrid implementation is slower than MPI implementation. However, in fact, if we have enough physical machines for doing the experiments, hybrid implementation will be faster than MPI implementation because we have evaluated the relationship between nodes and performance and the relationship between thread number and performance.

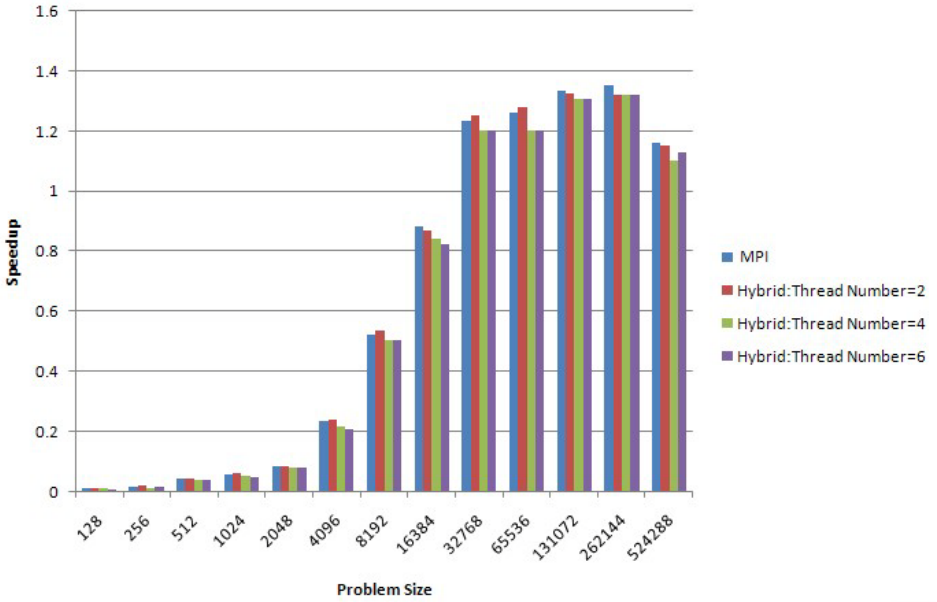

Figure 19.MPI and hybrid performance comparsion

6 Conclusion

We successfully implement the parallel quicksort using different methods (1)multithreaded (2)message-passing, and (3)a hybrid of the two afore-mentioned approaches. In our project, we implement quicksort by using these three methods to analyze their performance features and find out the best application situation for each method. In addition, we do the memory hierarchy optimization on the serial quicksort. We did several experiments to evaluate their performance from two aspects: time cost and energy consumption. Because of the limited experiment equipments, some analysis is not complete. In the future, we hope we can have and configure a real distribute systems to do all experiments in the MPI and hybrid approaches. Besides, we will use the CUDA architecture to implement the quicksort.

Reference

[1] Sorting algorithm https://en.wikipedia.org/wiki/Sorting_algorithm

[2]Parallel computing http://en.wikipedia.org/wiki/Parallel_computing#cite_note-3

[3]Quentin F.Stout. Algorithms Minimizing Peak Energy on Mesh-Connected Systems. Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)2006

[4] C. S. Ellis. The Case for Higher-Level Power Management. In Proceedings of the 7the workshop on Hot Topics in Operating Systems, March 1999.

[5] Jason Flinna and M. Satyanarayana. Energy-aware adaptation for mobile applications. In Symposium on Operating Systems Principles(SOSP), pages 48-63, December 1999.

[6] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahda. ECOSystem: Managing energy as a first class operating system resource. In Tenth for Programming Languages and Operating systems (ASPLOS X), October 2002.

[7]Quicksort. http://en.wikipedia.org/wiki/Quicksort

[8]Thomas H. Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, Second Edition, The MIT Press, page 145-146.