Parallel Computing, Comp633-Fall 2015 PA2

Dong Nie
Code are submitted by dongnie

1. Summary

In this assignment, we are required to implement a parallel quicksort algorithm. We first implement a sequential quicksort algorithm, and make it parallel using openmp. We tried several solutions, also, we implemented a parallel simulated parallel quicksort algorithm using binary search tree algorithm.

2. Description of implementations

We will describe our implementation in the following subsections.

2.1 Sequential quicksort

The sequential quick sort we implemented contains: partition is maintaining two index: one from left (to search the one whose value is larger than pivot) and other (to search the one whose value is smaller than pivot) from right. Then use quicksort function recursively call themselves. The codes are listed below.

```
int partition( double A[], int left, int right )
{
    double pivot = A[left];
    int i = left - 1;
    int j = right + 1;
    while(1)
    {
        do
        {
            j = j - 1;
        } while ( A[j] > pivot );
        do
        {
            i = i + 1;
        } while ( A[i] < pivot );
        if (i < j)//swap A[i] with A[j]
        {
            double temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
        else
        {
            return j;
        }
    }
}
```

```
void qsort_seq( double A[], int left, int right )
{
    if ( left < right )
    {
        int key = partition(A, left, right);
        qsort_seq(A, left, key);
        qsort_seq(A, key + 1, right);
    }
}
```

2.2 Parallel quicksort 1

This part, we use “sections” directives to form the parallel quicksort algorithm.

```
void qsort_v1( double A[], int left, int right )
{
    if ( left < right )
    {
        int p = partition(A, left, right);
        int size = right - left + 1;
        #pragma omp parallel
        {
            #pragma omp sections
            {
                #pragma omp section
                qsort_v1(A, left, p);
                #pragma omp section
                qsort_v1(A, p + 1, right);
            }
        }
    }
}
```

2.3 Parallel quicksort 2

In this part, we use “task” directives to form the parallel quicksort algorithm.

```
void qsort_v2( double A[], int left, int right )
{
    if ( left < right )
    {
        int p = partition(A, left, right);
        int size = right - left + 1;
        #pragma omp
        {
        #pragma omp task
        qsort_v2(A, left, p);
        #pragma omp task
        qsort_v2(A, p + 1, right);
        #pragma omp taskwait
        }
    }
}
```

2.4 Parallel quicksort 3

In this part, we not only parallel the conquer part, also parallel the partition part.

```
void qsort_v3( double A[], int left, int right )
{
    if ( left < right )
    {
        int p = partition_parallel(A, left, right);
        int size = right - left + 1;
        #pragma omp
        {
            #pragma omp task
            qsort_seq(A, left, p);
            #pragma omp task
            qsort_seq(A, p + 1, right);
        }
    }
}
```

For the partition part, we scan the data first and get the index of data which are smaller than pivot, larger than pivot. Then update the partition for the array.

```
#pragma omp parallel for private(i)
for (i = left; i <= right; ++i)
{
    if (smallArray[i] == 1)
    {
        A[smallLocations[i]-1] = sArray[i];
    }
    else if (equalArray[i] == 1)
    {
        A[equalLocations[i]-1+sCnt] = sArray[i];
    }
    else if (largArray[i] == 1)
    {
        A[largeLocations[i]-1+sCnt+ncount] = sArray[i];
    }
    else {}
}
```

2.5 Simulated parallel quicksort

In this part, we tried to simulate parallel quicksort with binary search tree. In essence, the binary search tree is the same with quicksort. In sequential sorting style, the binary search tree algorithm will suffer greatly from cache locality problem, so that's why people use quicksort much more frequently. However, for parallel style, binary search tree is very easy to implement and suffer less from cache locality.

```
    beginTime=omp_get_wtime();
    omp_set_num_threads(th);//set the number of threads to be used
#pragma omp parallel private(i) shared(root)
    {
#pragma omp for
        for(i=0;i<n;++i)
        {
            root=i;// compete to be the root of the balance tree
            LC[i]=RC[i]=n+1;
        }
#pragma omp for
        for(i=0;i<n;++i)
            f[i]=root;
    }
```

```
#pragma omp parallel private(i,flag) shared(A,f,LC,RC)
    {
#pragma omp for
        for(i=0;i<n;++i)
        if(i!=root)
        {
            flag=false;
            while(!flag)
            {
                if(A[i]<A[f[i]] || A[i]==A[f[i]] && i<f[i]) //go to left
                {
                    if(LC[f[i]]>n)// if no left child
                    {
                        LC[f[i]]=i;
                        flag=true;
                    }
                    else f[i]=LC[f[i]];
                }
                else //go to right
                {
                    if(RC[f[i]]>n)// if no right child
                    {
                        RC[f[i]]=i;
                        flag=true;
                    }
                    else f[i]=RC[f[i]];
                }
            }
        }
    }
    endTime=omp_get_wtime();
```

3. Result Analysis

We compile run our program on two different platforms, one is bass, the other is classroom (classroom.cs.unc.edu). Surprisingly, the result is largely different.

3.1 Result Correctness

At first, the sequential qsort function is executed correctly. We generated a simple array randomly, and the first row below is the generated data, we perform our sort algorithms, and list sorted result below (each line represents a quicksort implementation).

```
0.767926 0.091512 0.236710 0.015262 0.975065 0.410828 0.714024 0.798339 0.959383 0.420027
0.015262 0.091512 0.236710 0.410828 0.420027 0.714024 0.767926 0.798339 0.959383 0.975065
0.015262 0.091512 0.236710 0.410828 0.420027 0.714024 0.767926 0.798339 0.959383 0.975065
0.015262 0.091512 0.236710 0.410828 0.420027 0.714024 0.767926 0.798339 0.959383 0.975065
0.015262 0.091512 0.236710 0.410828 0.420027 0.714024 0.767926 0.798339 0.959383 0.975065
0.015262 0.091512 0.236710 0.410828 0.420027 0.714024 0.767926 0.798339 0.959383 0.975065
```

We can observe that all implementations are correct. In fact, we also test for large dataset, they are also right.

3.2 Performance Analysis **(Note, this one is compiling and running on bass)**

In this part, we will present our analysis for factors which may affect the performance. The measurement we used is as recommended by Prof. Jan Prins.

$$P(n) = (c_1 n \lg n)/T(n)$$

We consider the number of processors, problem size, also, we compare the result with sequential quicksort algorithm. Note, we set the constant c1 to be 1/100000.

3.2.1 Number of Processors

We test the parallel quicksort program with different number of processers, and recorded their performance. Then we presented the results in Fig.1 and Fig.2 (Note, here we take simulated parallel quick sort as example). Obviously, with the number of processors increasing, the graph performance improved. However, cases are not always like this. If the problem size is too small (or too large), the performance curves is different.

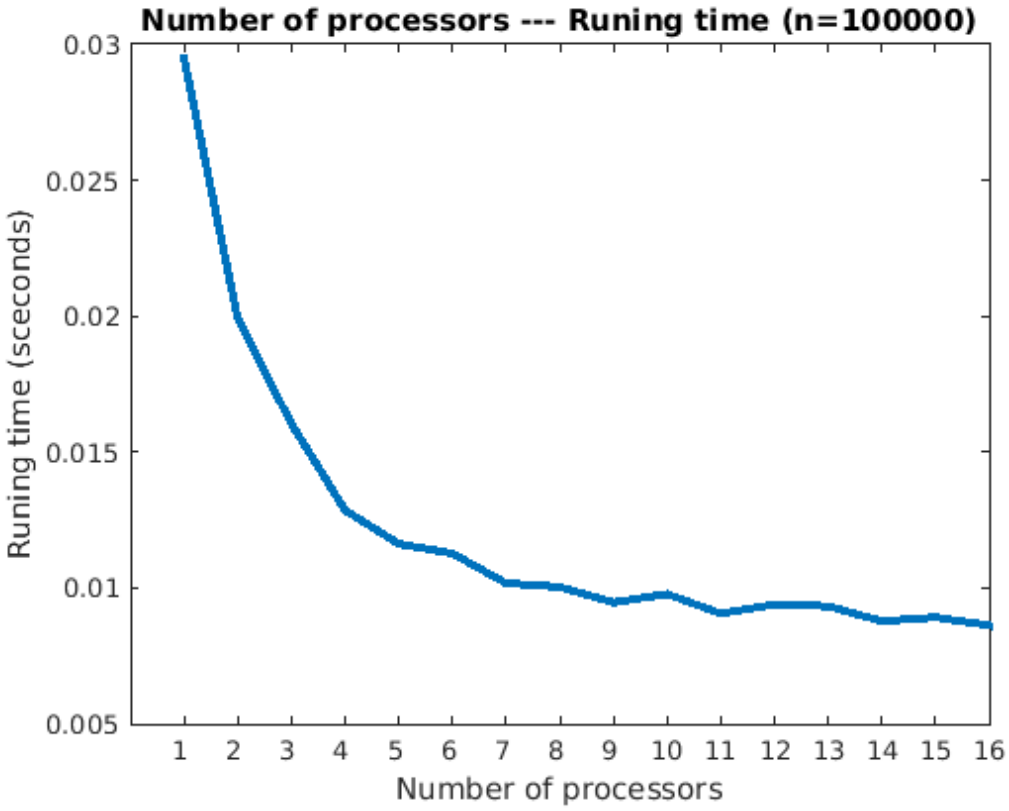

Fig.1 the running time over different number of processors for parallel quicksort

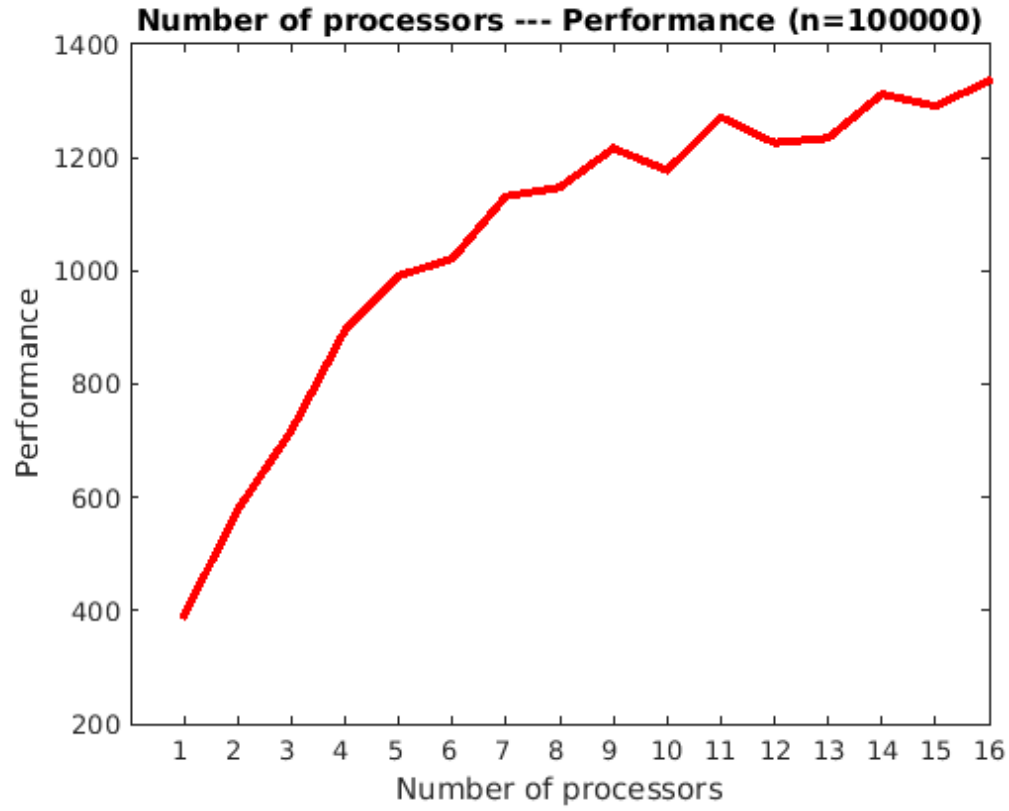

Fig.2 the graph performance over different number of processors for parallel quicksort

3.2.2 Problem Size

We test the parallel quicksort program with different problem size, and recorded their performance. Then we presented the results in Fig.3 and Fig.4. Obviously, with problem size increasing, the cost time goes up, and the graph performance first increased, then decreased.

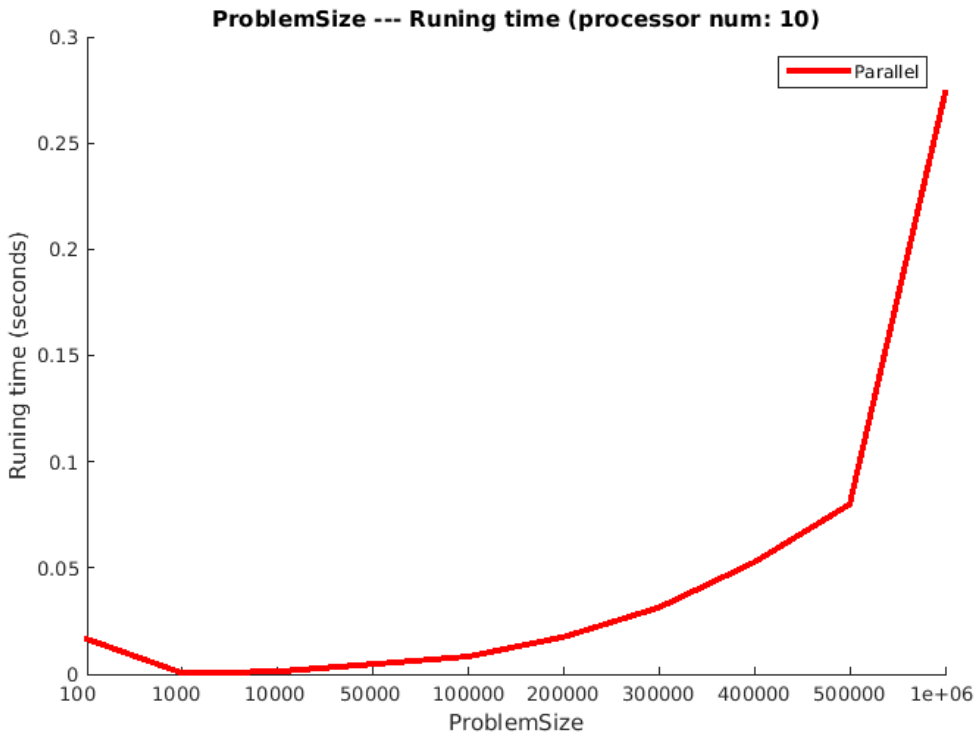

Fig.3 the running time over different problem size for parallel quicksort

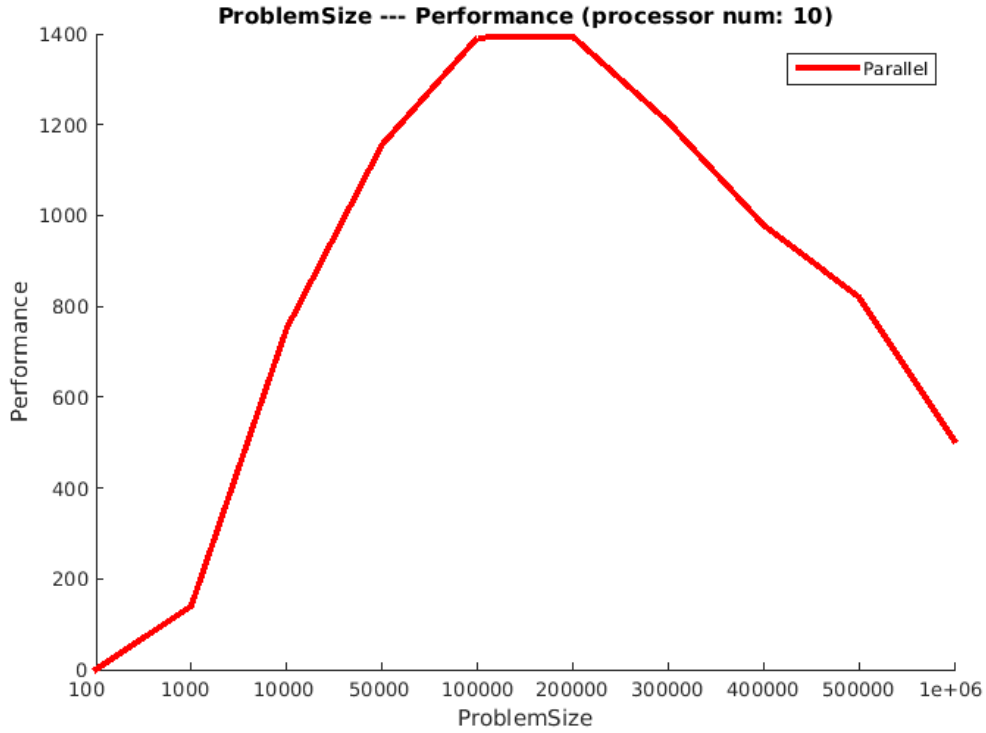

Fig.4 the graph performance over different problem size for parallel quicksort

3.2.3 Comparison with Sequential QuickSort

We compared the results of parallel quicksort with sequential quicksort. The result is showed in Fig.5.

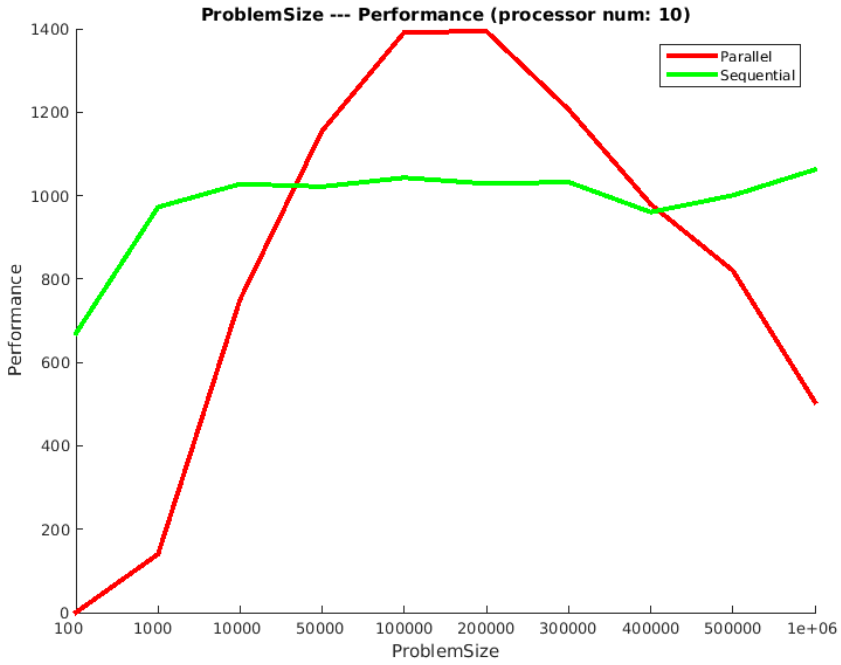

Fig.5 the graph performance: parallel quicksort vs. sequential quicksort

3.3 Performance Analysis **(This one is compiling and running on classroom platform)**

With the same measurement, but different compiling and running platform.

3.3.1 Number of Processors

We test the parallel quicksort program with different number of processers, and recorded their performance. Then we presented the results in Fig. 6 and Fig.7 (Note, here we take simulated parallel quick sort as example). Obviously, with the number of processors increasing, the graph performance first improved, but decreased latter. This may be caused by the number of processors in classroom platform is limited or some other reason.

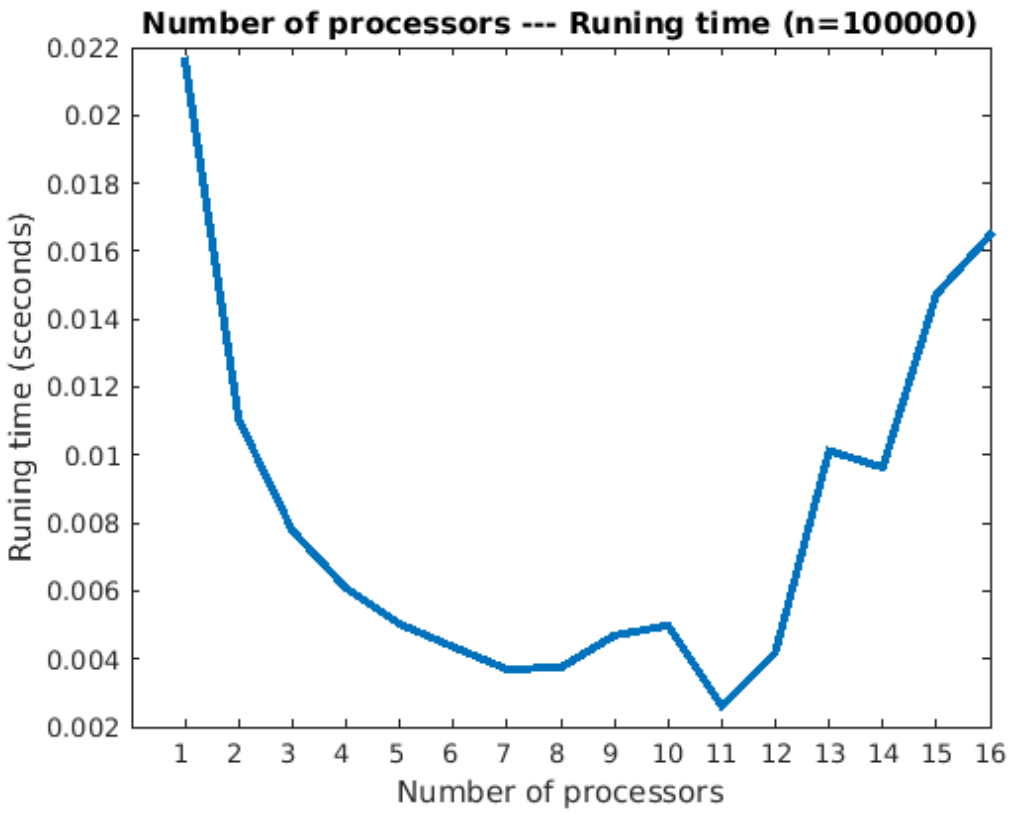

Fig.6 the running time over different number of processors for parallel quicksort

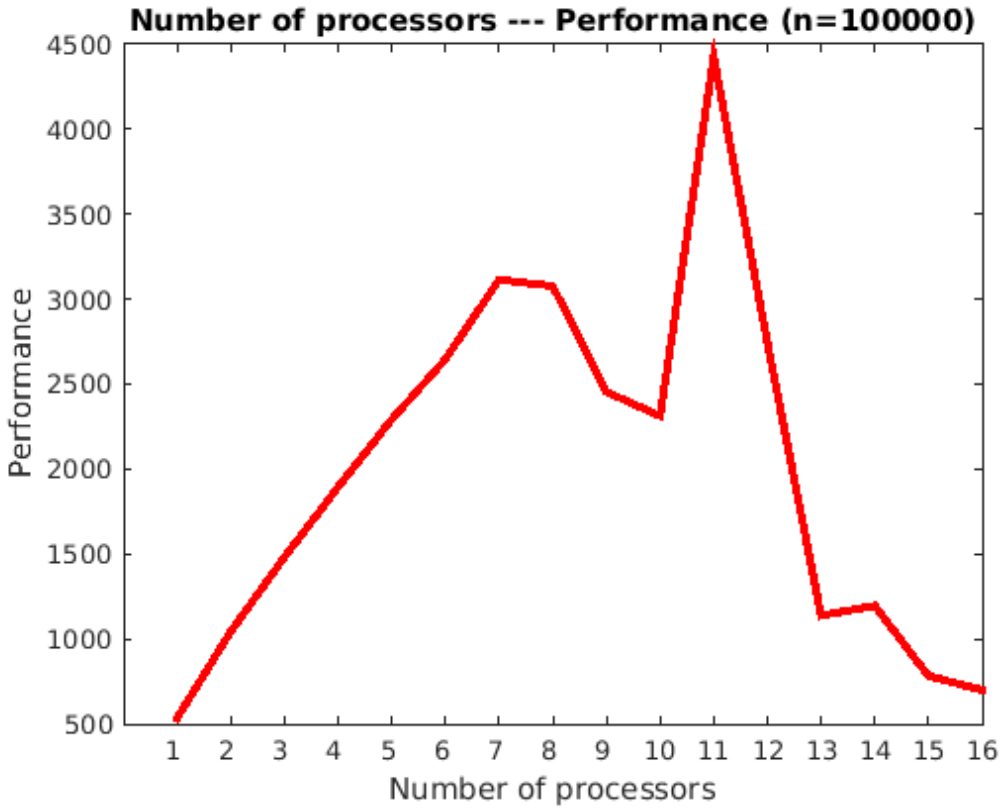

Fig.7 the graph performance over different number of processors for parallel quicksort

3.3.2 Problem Size

We test the parallel quicksort program with different problem size, and recorded their performance. Then we presented the results in Fig.8 and Fig.9. Obviously, with problem size increasing, the cost time goes up, and the graph performance first increased, then keep in a balance line.

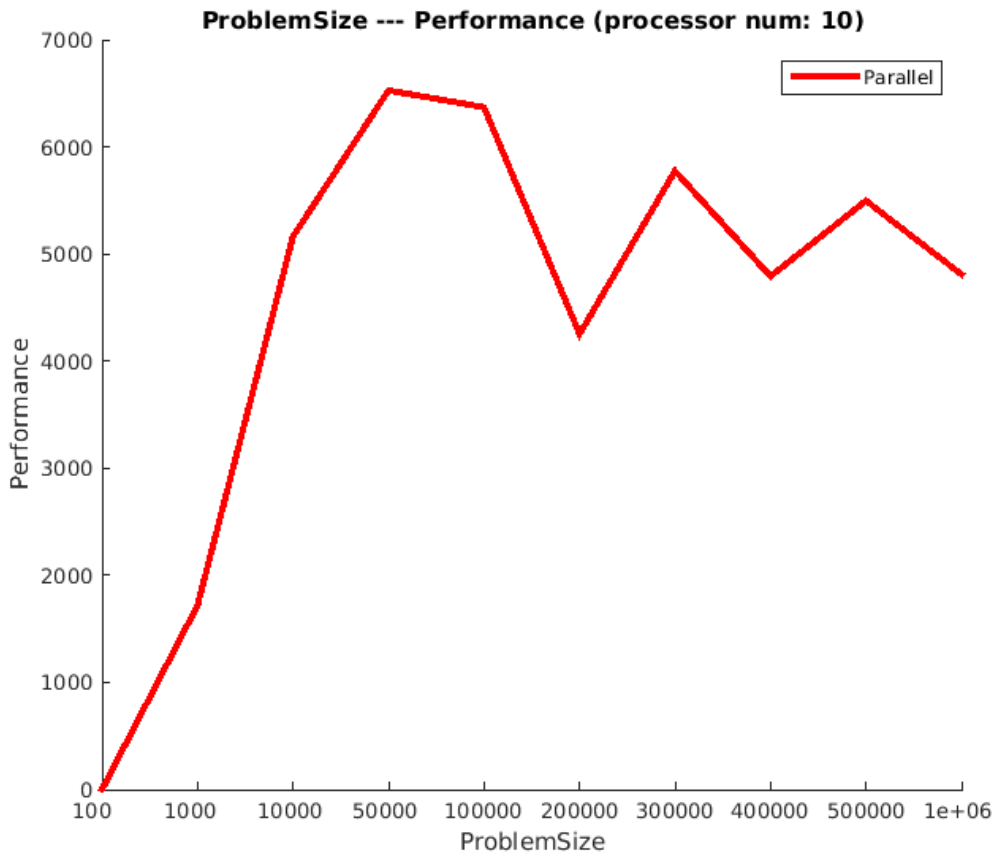

Fig.8 the performance over different problem size for parallel quicksort

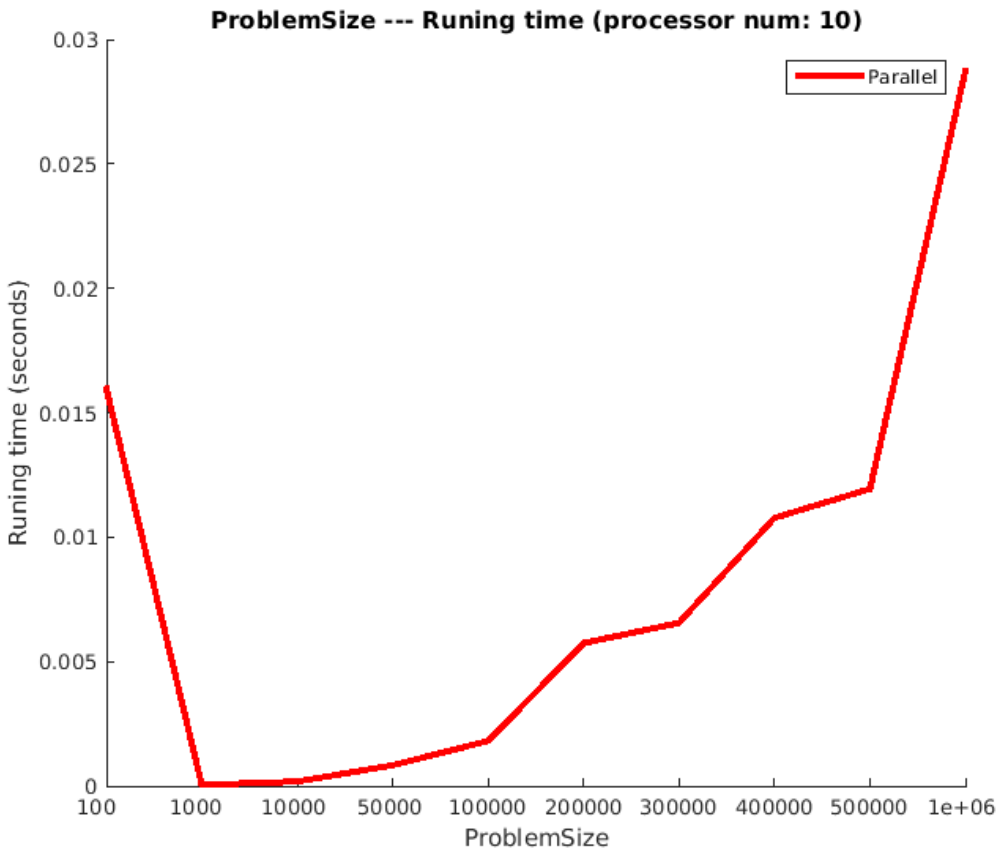

Fig.9 the running time over different problem size for parallel quicksort

3.3.3 Comparison with Sequential QuickSort

We compared the results of parallel quicksort with sequential quicksort. The result is showed in Fig.10. When the problem size is too small (<=100), the sequential sort is better, while if the problem size is larger, then parallel sort is much better than sequential sort.

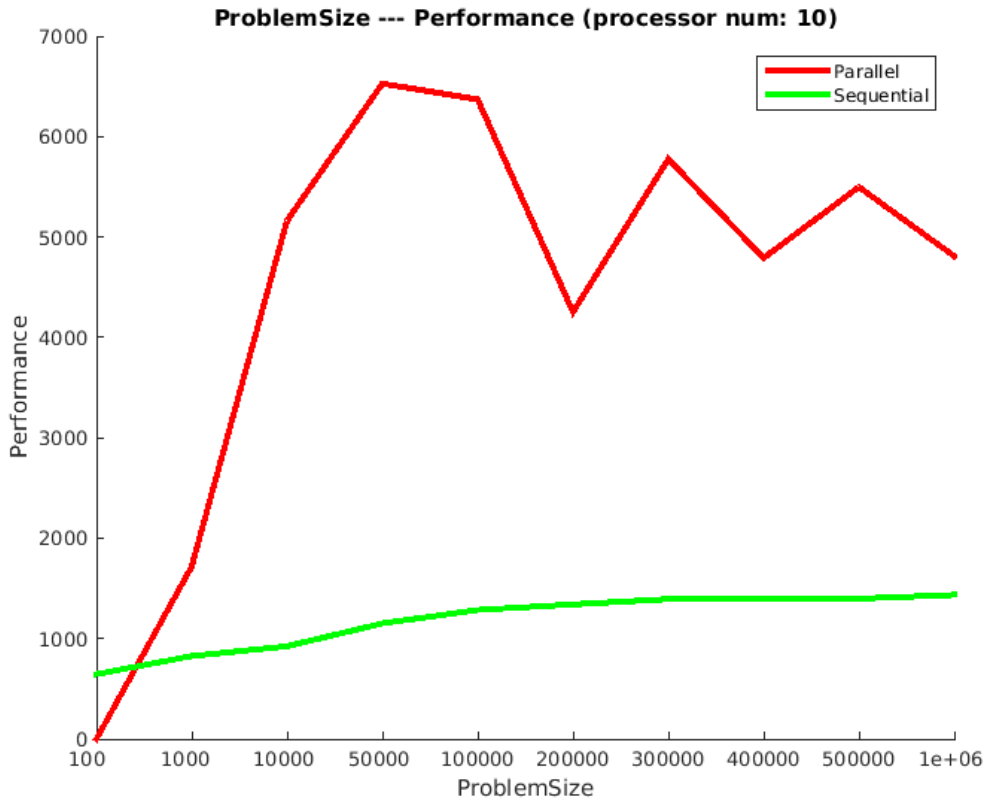

Fig.10 the graph performance: parallel quicksort vs. sequential quicksort

3.4 Performance bottlenecks

There are several factors which limit the speedup of parallel quicksort.

1. The first one is number of processors (threads), there are cost to make the program parallel, because they may produce wait time for different processors, so we have to design the program with great caution. The second one is cache locality, if there are more than one threads, the probability to produce cache miss is much higher.
2. The second one is about problem size.
3. Just as showed in section 3.2 and 3.3, the same program running on different platform (bass and classroom), the performance is largely different. So when we implement parallel programs, we should know the platform to compile and run our programs.