2008

ECE566_Puneet_Kataria_Project
Introduction to Parallel and Distributed Computing

PARALLEL QUICKSORT IMPLEMENTATION USING MPI AND PTHREADS

This report describes the approach, implementation and experiments done for parallelizing sorting application using multiprocessors on cluster by message passing tool (MPI) and by using POSIX multithreading (Pthreads). Benchmarking using 1MB sort and Minute sort were done .Pivot optimization techniques are also discussed for worst case scenarios in quicksort.

SUBMITTED BY:

PUNEET KATARIA
RUID – 117004233
puneet25685@gmail.com

10 Dec, 2008

TABLE OF CONTENTS:

1. Project Motivation and goals

Sorting is one of the fundamental problems of computer science, and parallel algorithms for sorting have been studied since the beginning of parallel computing. Batcher's -depth bitonic sorting network [3] was one of the first methods proposed. Since then many different parallel sorting algorithms have been proposed.

Some of the most common applications of sorting have been Sort a list of names, Organize an MP3 library, Display Google PageRank results. Some other applications like finding the median, finding the closest pair, binary search in a database, finding duplicates in a mailing list could also make use of efficient sorting algorithms to improve the performance.

Sorting is an important part of high-performance multiprocessing. It is also a core utility for database systems in organizing and indexing data. Sorting may be requested explicitly by users through the use of Order By clause.

Dramatic improvements in sorting have been made in the last few years, largely due to increased attention to interactions with multiprocessor computer architecture [2]. Improved results for sorting using multiprocessors shown by researchers [Sort Benchmark Home Page] was one of the main reason for choosing a project topic in the field of parallel sorting.

Parallel quicksort and Parallel Soritng by Regular Sampling (PSRS) are two of the most common techniques adopted for parallel sorting. Because of its low memory requirements, parallel Quicksort can sort data sets twice the size that sample sort could under the same system memory restrictions [1], hence I chose quicksort as the technique for sorting.

The goal of the project as described in the project proposal is benchmarking the parallel implementation of quicksort using MPI and Pthreads and optimizing pivot selection.

2. Background Material

Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that, on average, makes O(nlogn) comparisons to sort n items.

The implementation details for a simple quicksort as explained by Anany Levitin in his book Design and Analysis of Algorithms [4] are:

1. Choose a pivot element, usually by just picking the last element out of the sorting area.

2. Iterate through the elements to be sorted, moving numbers smaller than the pivot to a position on its left, and numbers larger than the pivot to a position on its right, by swapping elements. After that the sorting area is divided into two subareas: the left one contains all

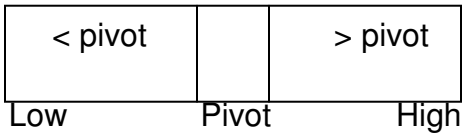
numbers smaller than the pivot element and the right one contains all numbers larger than the pivot element. The pivot is now place in its sorted position

3. Go to step 1 for the left and right subareas (if there is more than one element left in that area)

The key thing to note is that this implementation is nothing but a divide and conquer strategy where a problem is divided into subproblems that are of the same form as the larger problem. Each subproblem can be recursively solved using the same technique.

Once partititon is done, different sections of the list can be sorted in parallel. If we have p processors, we can divide a list of n elements into p sublists in Θ(n) average time, then sort each of these in Θ ((n/p)log(n/p)) average time. A pseudocode and diagram explaining the recursion are shown below:

```
quicksort( void *a, int low, int high )
  {
  int pivot;
  /* Termination condition! */
  if ( high > low )
    {
    pivot = partition( a, low, high );
    quicksort( a, low, pivot-1 );
    quicksort( a, pivot+1, high );
    }
  }
```



Initial Step - First Partition

Recursive steps

Figure 1. Basic Quicksort diagrammatic representation

3. Approach taken for MPI and Pthreads:

Two different approaches were taken for implementation in MPI and Pthreads optimizing the parallel implementation keeping in mind that MPI uses message passing and Pthreads use shared memory.
For MPI the goal was to reduce message passing making sure that communication was kept to minimum and hence computation to communication ratio would remain high. For Pthreads, since memory is shared the goal was to have less critical sections and barriers in the implementation.

3.1 Approach for MPI

Each process is given a section of the sequence of data to be sorted. This is done by performing a scatter operation by the master process. Each process works on its subsequence individually using the quicksort algorithm explained in section 2 and sends its sorted sub sequence for merge.

The major advantage of this technique is that communication time is kept low by sending only a subsequence to each process. But this is not an optimized solution. The disadvantages of such a technique are as follows:

--- Load balancing is not achieved since a particular process may finish its sorting process and wait for merge operation while other processes are still sorting their subsequences. This leads to idle process time and is not efficient.

--- Merge operations performed for each sorted subsequence is also computationally expensive as explained in section 4.1.1

Hence a different approach is taken for pthreads where shared memory provides us with an alternative of avoiding the merge operations.

3.2 Approach for Pthreads

While each MPI process selects first element of its subsequence as pivot and does sorting on its subsequence, in Pthreads a global pivot is selected and each thread compares its values with this global pivot and finds values in its subsequence less than this pivot. Once this is done, total values less than the global pivot are found and hence
the global pivot is sent to its sorted position.

The major advantage here is that load balancing is obtained since the threads are working on the same sequence simultaneously. Merge operation is also not required like in MPI but disadvantage is

----- Finding total values less than the global pivot and updating it becomes a critical section in the code and hence mutex locks are required which slow the process. The swapping of global pivot to its original position is also a critical section and hence synchronizing operation barrier is required in both cases.

4. Implementation

All implementations and experiments are done by sorting integer values generated by a random function. Once sorting process finishes sorted values are saved in a result file to check for correctness of code. Size of integer is considered as 4 bytes since a sizeof (int) operation reveals that int is 4 bytes on the cluster.

To sort n numbers, random numbers are created and saved in an array as follows:

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

4.1 Implementation of MPI code

Size of array that each slave has to work on is calculated by master process and this size is Broadcast to slaves. Each slave allocates memory enough to accept data of the specified size. A scatter operation is then executed by master and each slave receives data equal to its assigned size. Code snippet shown below shows how this is done

```
MPI_Bcast(&s,1,MPI_INT,0,MPI_COMM_WORLD);     /* --- Broadcast size of
array that each
                                                    slave has to sort -
-- */
chunk = (int *)malloc(s*sizeof(int));
MPI_Scatter(data,s,MPI_INT,chunk,s,MPI_INT,0,MPI_COMM_WORLD);    /* ---
Scatter the values
                                                          from array to
the slaves --- */

elapsedtime(id,"Scattered data:");
myqsort(chunk,0,s-1);                          /* --- call to sort
function  --- */
elapsedtime(id,"Sorted data:");
```

Here elapsedtime is a function that displays time. In the above case time spent for data scatter and for sorting are found out.

4.1.1 Merge function

Consider that v1 and v2 are 2 sorted subsequences with number of values n1 and n2 respectively. Then the merge function compares values one by one and saves the lower value in an array result[ ]

Here the best case for merge would be when all values of one sequence are lower than all the vales of the other. The time complexity would be Θ(n1 +1) or Θ(n2+1) since after a pass through one array it will just copy the values of the other array and append to the first.

The worst case performance would be when alternating values of the 2 subarrays are lower. i.e v1[1] < v2[1], v2[1] < v1[2], v1[2] < v2[2] and so on. In this case the time complexity would be Θ(n1+ n2).

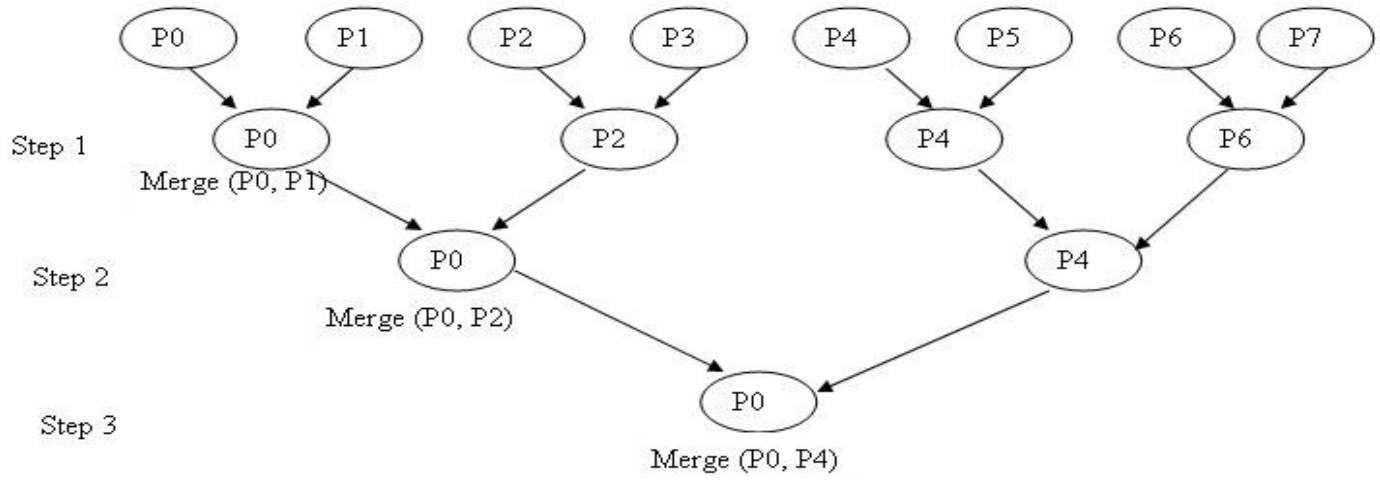
```
while(i<n1 && j<n2)
            if(v1[i]<v2[j])
            {
                  result[k] = v1[i];
                  i++; k++;
            }
            else
            {
                  result[k] = v2[j];
                  j++; k++;
            }
      if(i==n1)
            while(j<n2)
            {
                  result[k] = v2[j];
                  j++; k++;
            }
      else
            while(i<n1)
            {
                  result[k] = v1[i];
                  i++; k++;
             }
```

Figure 2. Merge function code

4.1.2 Tree Based Merge Implementation:

A simple method of merging of all sorted subsequences would have been, a simple gather implementation where the master gathers the sorted subsequences and merges them. But this would degrade the performance. If there are n elements to sort and p processes, then each slave sorts (n/p) values and master gathers the sorted subsequences. Master now has p sorted subsequences and it will have to call merge operation (p-1) times. Hence a time complexity of **Θ ((p-1) (n1+n2)).**

But with the tree based merge, each process sends its sorted subsequence to its neighbor and a merge operation is performed at each step. This reduces the time complexity to **Θ((log p)(n1+n2))** as shown in the diagram below. Code snippet for this tree implementation is also attached.



```
step = 1;
      while(step<p)
      {
            if(id%(2*step)==0)
            {
                  if(id+step<p)
                  {
                        MPI_Recv(&m,1,MPI_INT,id+step,0,MPI_COMM_WORLD,&status);
                        other = (int *)malloc(m*sizeof(int));
                        MPI_Recv(other,m,MPI_INT,id+step,0,MPI_COMM_WORLD,&status);
                        elapsedtime(id,"Got merge data:");
                        chunk = merge(chunk,s,other,m);
                        elapsedtime(id,"Merged data:");
                        s = s+m;
                  }
            }
            else
            {
                  int near = id-step;
                  MPI_Send(&s,1,MPI_INT,near,0,MPI_COMM_WORLD);
                  MPI_Send(chunk,s,MPI_INT,near,0,MPI_COMM_WORLD);
                  elapsedtime(id,"Sent merge data:");
                  break;
            }
            step = step*2;
      }
```

Figure 4. Code for tree based merge implementation

4.1.3 MPI Setup Time

For timing analysis, MPI set up time is also calculated as follows

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
MPI_Comm_size(MPI_COMM_WORLD,&p);

elapsedtime(id,"MPI setup complete:");
```

Once MPI is initialize time is calculated. For smaller number of processes, this time is negligible but as number of processes increases this time becomes substantial part of execution time.

4.2 Implementation of Pthreads

Each thread finds the number of values less than and greater than the global pivot within its section of the sequence. Then using these values a total count of values less than the pivot is made. This is implemented as follows

```
for (i=my_start_index; i<my_start_index + my_length; i++)
      {
            if (a[i] < head -> pivot)
                  less ++;
            else
                  greater ++;
      }

pthread_mutex_lock(&(head->sum_lock));
head -> less_than += less;                /* ---  Critical section  -
                                           Each thread updates the
                                          global value less_than  --- */
pthread_mutex_unlock(&(head->sum_lock));

barrier(&(head -> barrier2), head -> numthreads);  /* -- Barrier call
                                                     to wait for all
                                                    threads to update
                                                  value less_than ---*/
```

Total count of values less than pivot is made by adding each threads value of less. Since it is a critical section and simultaneous access by multiple threads should not be allowed, hence it is put under mutex locks. Once a Total count of values less than pivot is made this value is used to put pivot in appropriate position. This action should not take place until all threads have updated the value less_than, hence a barrier is used here which ensures that only updated value of less_than is not used for further operations.

4.2.1 Pthread Setup Time

Similar to MPI, Pthread set up time is also calculated

```
for (i=0; i<NUM_THREADS; i++)
{
    pthread_create(&sort_threads[i], NULL, sort_thread,(void *)
&workspace[i]);
}
void *sort_thread(void *ptr)
{
barrier(&(head -> barrier1), head -> numthreads);
(void)pthread_once(&once_ctr, once_rtn);
```

Here once the main program calls pthread _create all threads start executing the routine sort_thread. Here a barrier call ensures all threads have been created and started executing routine sort_thread. At this time pthread_once function is called which allows only one thread to call routine once_rtn. The only task of this once_rtn routine is to display time. Hence pthread set up time will be obtained. This part was only included for time analysis.

5. Experiments and results:

8.1 Benchmark Experiment: (1 MB sort)

Initial goal of the project was to have a benchmark with 1TB sort but I later realized dealing with such huge numbers was not feasible on the cluster.

To sort 1 MB, N (the number of integers to sort) is set to 250000. [Since size of 1 int is 4 bytes] and the number of processes were varied for MPI and number of threads were varied for pthreads. Code used for this experiment are seq_qsort.c, mpi_qsort.c and pthreads_qsort.c. For each execution a result file is generated which stores the sorted values. The file can be used to check for correctness of the programs. Values obtained for the experiments are as follows.
Execution time for sequential program is 0.24 seconds.

Number of processes	Pthread execution time (sec)	MPI execution time (sec)
5	0.23	0.11
7	0.19	0.1
9	0.18	0.08
11	0.16	0.06
13	0.16	0.1
15	0.16	0.15

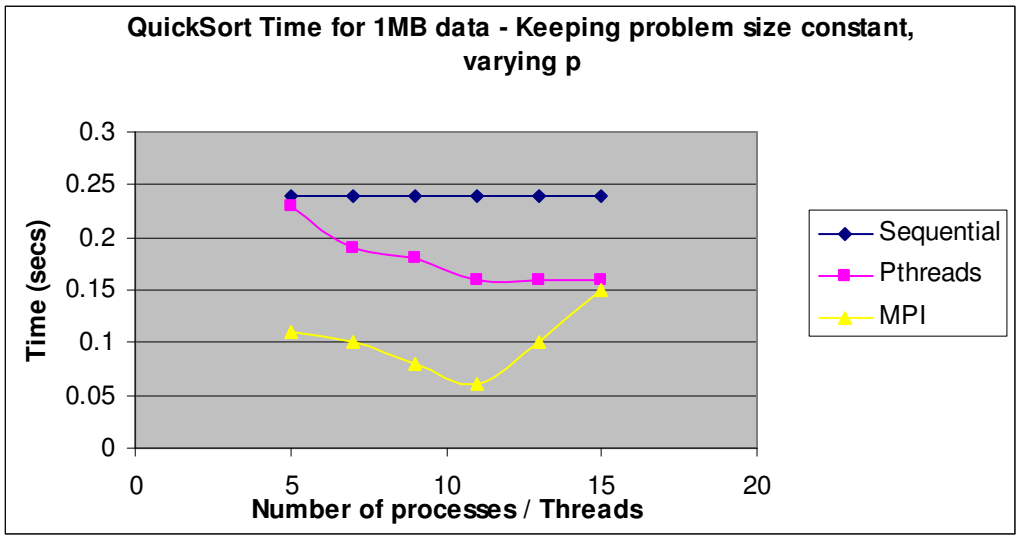


Figure 5. Experiment readings and graph for 1MB quicksort using MPI, Pthreads and Sequential code

Here minimum execution time for MPI and Pthreads is obtained at approximately 10 processes / threads after which increasing processes increases execution time. This is because as the number of processes increases communication time increases and hence computation to communication ratio begins to decrease.

5.2 Speedup analysis

$$\text{Speedup} = \frac{\text{Execution time using best sequential algorithm}}{\text{Execution time using a multiprocessor using p processes}}$$

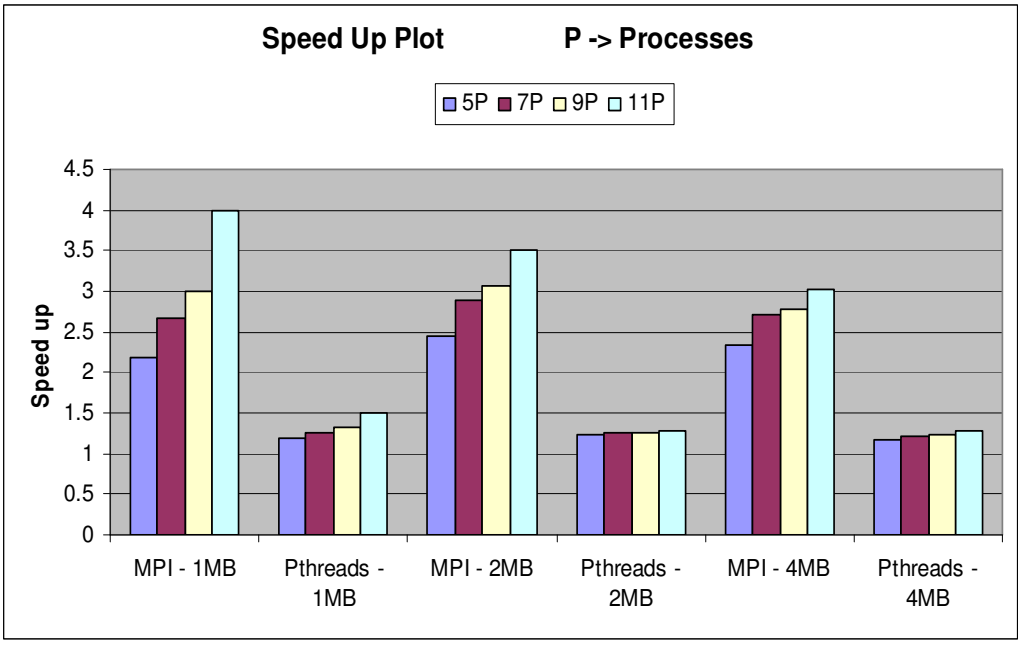


Figure 6. Speedup plot for varying problem size and varying number of processes

As mentioned in the previous analysis, minimum execution time is obtained at approximately 10 processes and hence speedup is maximum at that value.

5.3 Execution time split up into different phases of the program

To analyse the time spent in different phases of the program, I executed the MPI program for 5, 10 and 15 processes and time spent in each phase of the program execution was found out.

	5 processes	10 processes	15 processes
MPI Set up Time	0.01	0.07	0.11
Generate random numbers	0.04	0.03	0.03
Communication time	0.02	0.03	0.08
Sorting time / Computation time	0.07	0.04	0.03

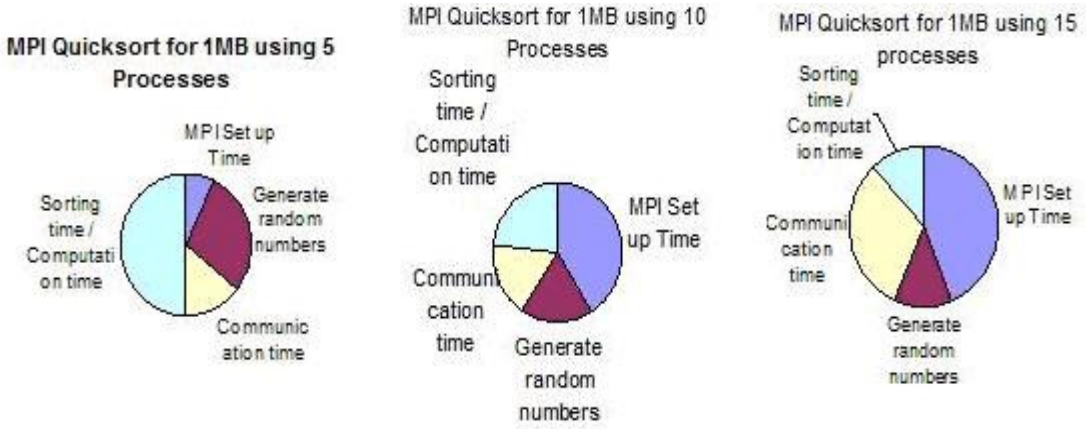


Figure 7. Plot for time spent in each phase of the program

From the above plots, we can see that for 5 processes time sent for sorting or computation time is maximum while for 15 processes time spent for communication is more. One important thing to note is that as processes are increased MPI set up time also increases drastically. Hence to have any substantial speedup, it is important that problem size increases with number of processes.

6 Optimizations

6.1 Benchmark Minute Sort - Optimizing chunk size

One of the most standard benchmarks for sorting is Minute Sort (number of bytes sorted in 1 minute)

For implementing this, I created chunks of data. For experiment 1, chunk size of 100KB was selected. The sequential code sorts this 100KB, stores the sorted values in result file and then checks if 1 minute has elapsed. If execution time has passed 1 minute execution stops, if not then another chunk of 100 KB is sorted. In MPI at the completion of sorting of each chunk, the master process checks if 1 minute has elapsed, and broadcasts a termination flag to all slaves if 1 minute has elapsed, else program execution continues. Similar experiment was done for chunk size of 500KB and 1 MB. Codes attached for minute sort are seq_minute.c and mpi_minute.c .

Sequential execution - 118.9 MB sorted in 1 minute with chunk size of 100KB
113 MB sorted in 1 minute with chunk size of 500KB
114 MB sorted in 1 minute with chunk size of 1 MB

Number of processes	Chunk size - 100KB	Chunk size - 500KB	Chunk size - 1MB
5	147.1	148	149
10	152	155	153
15	152.3	159	154
20	153.9	162	155
25	154.5	164.5	157
30	156	154	155
35	149	152	155
40	146.6	150.2	153

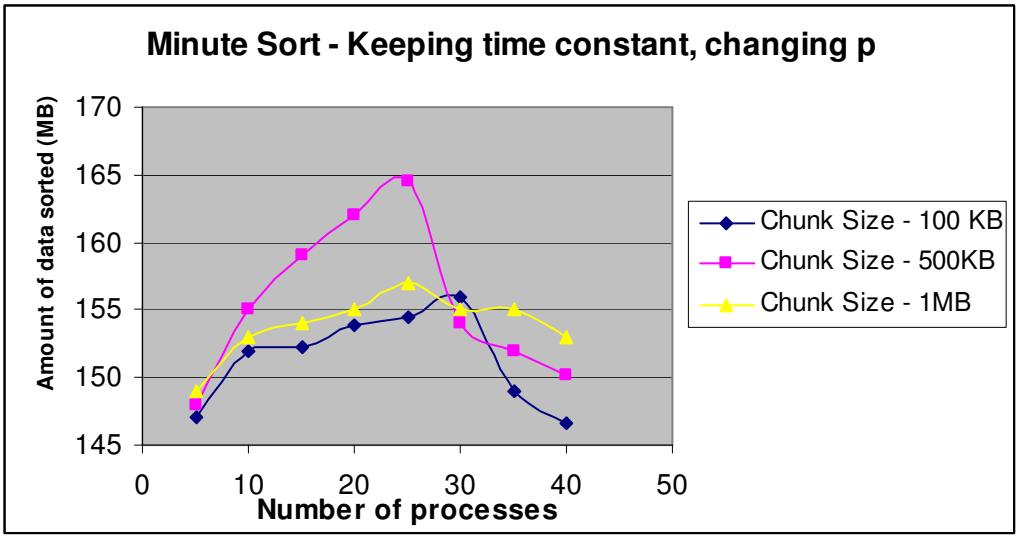


Figure 7. Minute Sort plot – Verifying optimum chunk size

From the above plot, we can see that maximum data can be sorted with chunk size of 500 KB and 25 processes (165 MB).

6.2 Pivot Optimization

Average case efficiency for quicksort is Θ(nlogn) but worst case efficiency is Θ(n^2) [4]. The worst case condition for quicksort is when the sequence is already sorted. Techniques to improve performance in this worst case have been suggested by many researchers:
One possible solution that is adopted for my project is to choose the median value as pivot to improve the worst case performance. I conducted three experiments and compared the outcomes.

Experiment 1 – First element as pivot and input to quicksort is a random sequence
Experiment 2 – First element as pivot and input to quicksort is a sorted sequence
Experiment 3 – Middle element as pivot and input to quicksort is a sorted sequence

The results are as shown:

Number of processes	Random Input - First element as pivot	Sorted input - First element as pivot	Sorted input - Middle element as pivot
5	0.07	14.16	0.45
10	0.08	6.15	0.25
15	0.08	2.77	0.21
20	0.07	1.52	0.19
25	0.16	1	0.18
30	0.17	0.74	0.18

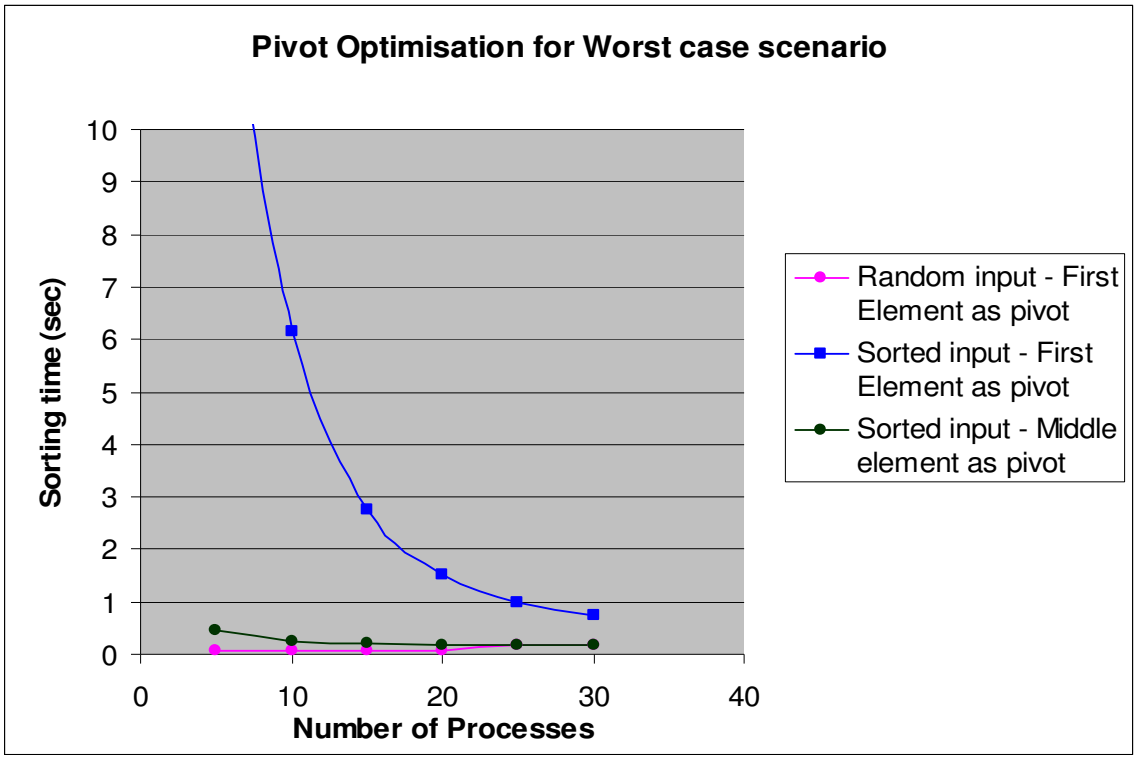


Figure 8. Pivot optimization plot for Worst case scenario for quicksort

Clearly, we can see that selecting middle element as pivot improves worst case performance considerably. Some other pivot optimization techniques like median of 3 pivot select, randomized pivot select can also be tested.

7. Conclusion

I successfully implemented the parallel quicksort program using MPI and Pthreads and performed benchmarking like Megabyte sort and minute sort on the implementation. The project gave me a chance to gain insight into some of the techniques that can be used for message passing cluster architectures and shared memory architectures. The speedup obtained using parallel implementation suggests that such parallel methods should be incorporated into real world applications which require sorting.

References:

[1] Philippas Tsigas and Yi Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise
10000

[2] Hongzhang Shan and Jaswinder Pal Singh. Parallel Sorting on Cache-coherent DSM Multiprocessors

[3] K. E. Batcher. Sorting networks and their applications. In AFIPS Springer Joing Computer Conference, pages 307–314, Arlington,VA, April 1968

[4] Anany V. Levitin, Villanova University Introduction to the Design and Analysis of Algorithms 2003

[5] Barry Wilkinson and Michael – Parallel Programming, Techniques and Applications using Networked Workstations and Parallel Computers