Parallel Quicksort using MPI
&
Performance Analysis

Prasad Perera
prasadcse@gmail.com

Table of Contents

Abstract

This is an individual effort on parallelizing the quicksort algorithm using MPI (Message Passing Interface) to sort data by sharing the partitions generated from regular sampling. The basic idea was to avoid the initial partitioning of data and merging step of sorted partitions in different processes. And finally to conduct a performance evaluation for the implementation. This evaluation was based on comparing sorting times with an algorithm which uses initial partitioned set between processes and to the simple sequential sorting algorithm.

1. Introduction

Quicksort is a well known algorithm used in data sorting scenarios developed by C. A. R. Hoare. It has the time complexity of O (n log n) on average case run and O (n^2) on worst case scenario. But quicksort is generally considered to be faster than some of sorting algorithm which possesses a time complexity of O (n log n) in average case.

The fundamental of quicksort is choosing a value and partitioning the input data set to two subsets which one contains input data smaller in size than the chosen value and the other contains input data greater than the chosen value. This chosen value is called as the pivot value. And in each step these divided data sets are sub-divided choosing pivots from each set. Quicksort implementations are recursive and stop conditions are met when there is no sub division is possible.

In this attempt, the main idea was to implement a parallelized quicksort to run on a multi-core environment and conduct a performance evaluation. This parallelization is obtained by using MPI API functionalities to share the sorting data set among multi processes.

1.1 Steps followed:

- Implement an optimized algorithm using MPI to sort data.
  This algorithm will be a parallelized implementation of the quicksort algorithm and it will avoid merging step by dividing input set through regular sampling.
- Performance evaluation
  - Perform sorting sample data sets against sequential quicksort algorithm.
  - Perform sorting sample data sets against an algorithm with initial portioning and merging.
  - Produce statistical evaluation through result forecasting.

- Evaluate its performance based on logical time complexities.
- Conclusion.

1.2 Quicksort algorithm

```
 int partition(int  *arr, int left, int right) {
 int i = left, j = right;
 int tmp;
 int pivot = arr[(left + right) / 2];
 /* partition */

 while (i <= j) {
     while (arr[i] < pivot)
          i++;
     while (arr[j] > pivot)
          j--;
          if (i <= j) {
             tmp = arr[i];
             arr[i] = arr[j];
             arr[j] = tmp;
             i++;
             j--;
            }
     }
     return j;
}

void quick_sort(int* arr, int left, int right){
 /* recursion */
 int part_index = partition(arr, left, right);
     if (left < part_index)
          quickSort(arr, left, part_index);
     if (part_index + 1 < right)
          quickSort(arr, part_index + 1, right);
}
```

2. **Parallel Quicksort**

Not only quicksort has considered being a better performing sorting algorithm but it's also considered to be one of reliable algorithms which can adapt to parallelization. With

quicksort, partitions can be sorted in parallel and combined with operations like merge to assemble the outputs. The conventional methods of parallelizing quicksort based on dividing the initial input to set of sub arrays and distributes them among the available set of processes to be sorted sequentially and later gathered using merge steps. Other than that there has been many researches done on parallelizing quicksort by optimizing pivot selections and various partitioning strategies.

My attempt was to avoid initial scattering of data and distribute data sets among processes using initial partitioning steps. So this will avoid the merging steps as the final output is produced only by gathering the sorted set of data from each process.

2.1 Basic implementation steps:

- Perform an initial partitioning of data until all the available processes were given a subset to sort sequentially
- Sort the received data set by each process in parallel.
- Gather all data, corresponding to the exact partitioned offsets without performing any merging.

2.2 Initial data partitioning and sub array allocations

The scheme of initial data partitioning contains following steps:

- Each process performs an initial partitioning of data using pivots regarding if there's any process available to share its data set.
- Send one part of data to the sharing process and start partitioning the remaining data set if there's more processes available.
- Else perform a sequential quicksort on the data set.
- Send locally sorted data set to its original sender.

This scheme requires a proper allocation of processes in each partitioning and data sharing steps. The scheme I used is to allocate data considering its rank and a calculation schema which determines which processes will share the data set.

2.3 Process Allocation scheme

Process	Sharing set	Sharing process rank calculation	Process set in each step
0	1	$0 + 2^0$	0
0 1	1,2 3	$0 + 2^0$, $0 + 2^1$ $1 + 2^1$	1
0 1 2 3	1,2,4 3,5 6 7	$0 + 2^0$, $0 + 2^1$, $0 + 2^2$ $1 + 2^1$, $1 + 2^2$ $2 + 2^2$ $3 + 2^2$	2
0 1 2 3 4 5 6 7	1,2,8 3,5,9 6,10 7,11 12 13 14 15	$0 + 2^0$ $0 + 2^1$ $0 +2^2$ $0 +2^3$ $1 + 2^1$ $1 + 2^2$ $1 + 2^3$ $2 + 2^2$ $2 + 2^3$ $3 + 2^2$ $3 + 2^3$ $4 + 2^3$ $5 + 2^3$ $6 + 2^3$ $7 + 2^3$	3

Table 1: Process sharing scheme

So according to the above scheme, processes are allocated using following equation:
Process p with the rank r will share its partition with the process rank,

- **Rank = r + 2^n** where n satisfies the condition**:** **$2^{n-1} \leq r < 2^n$**

With this calculated rank, each process will try to find its data partition sharing process. If the process is there, then the partition is sub divided and shared with the process which possesses that particular rank. If there is no such process, then the partition is sequentially sorted and sent back to the parent process who originally shared the sorted set. Each of these calls is implemented to run recursively.

Choosing which partition to share

In most cases, partitions produced by regular sampling may differ in size. So when any partitioning is performed to share the data set, the largest sub partition is kept within the lead process and the smaller set is sent to the receiving process. There are two reasons to send the smaller partition. They are,

1. It reduces the amount of communication overhead by transferring the smaller data set.
2. In process allocation schema, the process that is responsible in partitioning will be the process next in line to share the data set once again before the receiving process. So if any more processes are in the pool, the sending process has more chance in sharing its data set than the receiving process.

Ex: For a data set to be sorted using three processes, 0^{th} process will send the smaller data set to 1^{st} partition. Then the remaining process no. 2 will be assigned to process no. 0 to share its data set. So before 1^{st} process get a data sharing process, the process no. 0 who created the partitions has the chance of having another process to share its data set.

The performance gain was checked using a slightly changed parallel quicksort implementation where the lead process always keeping the first half of the array. This altered version ran against the smaller partition sharing quicksort implementation. Sorting time tends to be smaller on smaller partition sharing implementation than the first half of array sharing implementation.

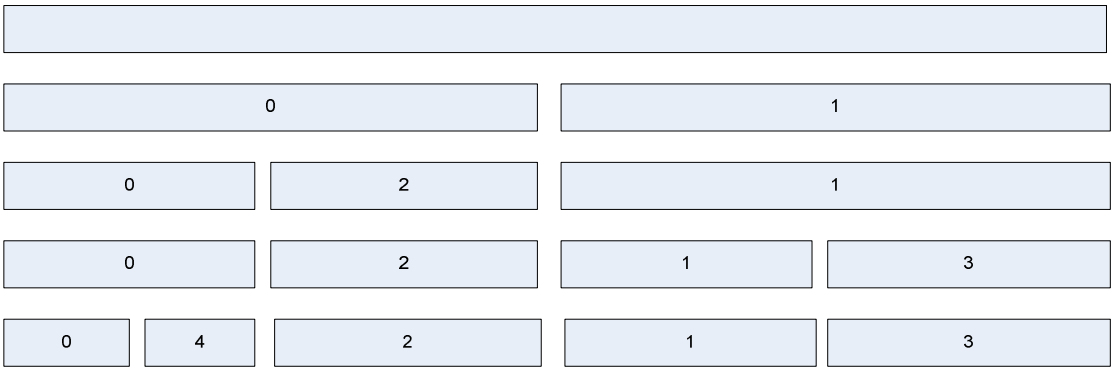


Diagram 1: Partition sharing scheme

Diagram 2: Variable partition sharing scheme

3. MPI implementation

Parallel quicksort was implemented using OpenMPI, one of open source MPI implementations available. And the program is written in C language.

- Function sort_recursive

This function is called recursively by each process in order to perform the initial partitioning when a sharing process is available or to sequentially sort the data set. Finally it sends the sorted data set to its sharing process.

The pivot for each step is chosen as the element indexed at [size / 2], which takes the middle value of the array. Also a specific index value is used to keep the number of increments while calculating the rank of next process in line to share the data set recursively.

```
int sort_recursive(int* arr,  int size, int pr_rank, int
max_rank, int rank_index){
 MPI_Status dtIn;
 int share_pr  = pr_rank + pow(2, rank_index);   /* Calculate the
                                          rank of sharing process*/
 rank_index ++;                          /*Increment the count
                                         index*/

 if(share_pr  > max_rank){               /*If no process to share
     sort_rec_seq(arr, size);
                                               sequentially*/
     return 0;
     }
  int pivot = arr[size/2];               /* Select the pivot */

  int partition_pt = sequential_quicksort(arr, pivot, size,
(size/2) -1);  /* partition array */
  int offset = partition_pt  + 1;



/* Send partition based on size, sort the remaining partitions,
receive sorted partition */

 if (offset > size - offset){
     MPI_Send((arr + offset), size - offset, MPI::INT, share_pr
     , offset, MPI_COMM_WORLD);
     sort_recursive (arr, offset, pr_rank,  max_rank,
     rank_index);

     MPI_Recv((arr + offset), size - offset, MPI::INT, share_pr,
     MPI_ANY_TAG, MPI_COMM_WORLD, &dtIn);
  }
  else{
     MPI_Send(arr, offset, MPI::INT, ch_pr , tag,
     MPI_COMM_WORLD);
      sort_recursive ((arr + offset), size - offset, pr_rank,
     max_rank, rank_index);

     MPI_Recv(arr, offset, MPI::INT, ch_pr, MPI_ANY_TAG,
     MPI_COMM_WORLD, &dtIn);
  }
```

Except the leading process, all other processes will execute the following lines of code

```
int* subarray = NULL;
MPI_Status msgSt, dtIn;
int sub_arr_size = 0;
int index_count = 0;
int pr_source = 0;

while(pow(2, index_count) <=  rank)     /* calculate the
      index_count ++;                    index_count as
                                         2 n-1 ≤ rank < 2n
                                        n = index_count */

MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&msgSt);
MPI_Get_count(&msgSt, MPI::INT, &sub_arr_size);
     pr_source = msgSt.MPI_SOURCE;          /* Get the
                                         sending process rank
                                         */

subarray = (int*)malloc(sub_arr_size * sizeof(int));
MPI_Recv(subarray, sub_arr_size, MPI::INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &dtIn);

int pivot = subarray[(sub_arr_size / 2)];          /* Find
                                                the pivot */
sort_rec(subarray, sub_arr_size, rank, size_pool -1,
rec_count);  /* sort recursively */
 /*   send sorted sub array */
MPI_Send(subarray, sub_arr_size, MPI::INT, pr_source,
 tag, MPI_COMM_WORLD);
free(subarray);
```

4. Experimental Results

Quicksort implementation was benchmarked with a parallel quicksort implementation with merge and with sequential quicksort implementation letting them to sort same set of data in varying sizes. These test results were gathered by running a batch of sorting tasks on each test case and averaging all the obtained results.

- All these tests were performed on a Linux cluster of 7-core Intel Xeon E5420 processors with 16 GB memory.

The benchmarks were run for following scenarios

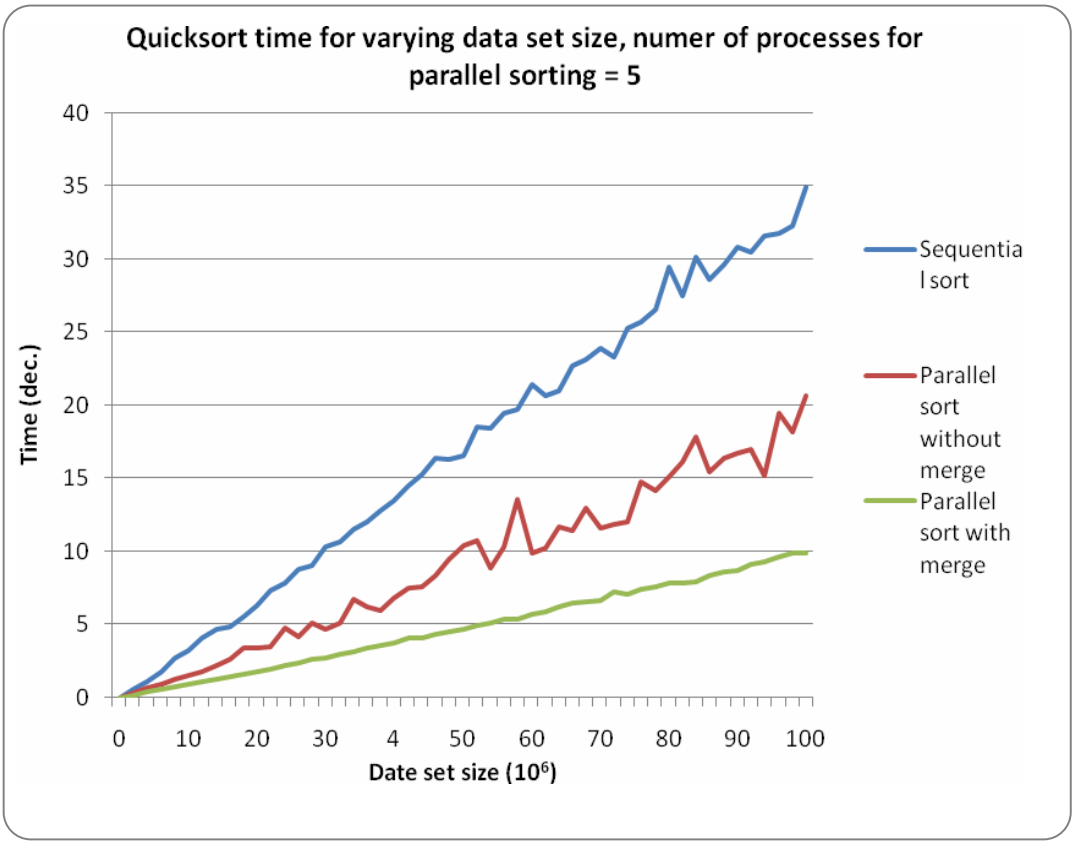
- Sorting data sets of 5 – 100 M with five processes for each implementation.
- Sort data set of 10 M for a process range 1 - 10 , 1 M for a process range 5 – 70

4.1 Sequential implementation

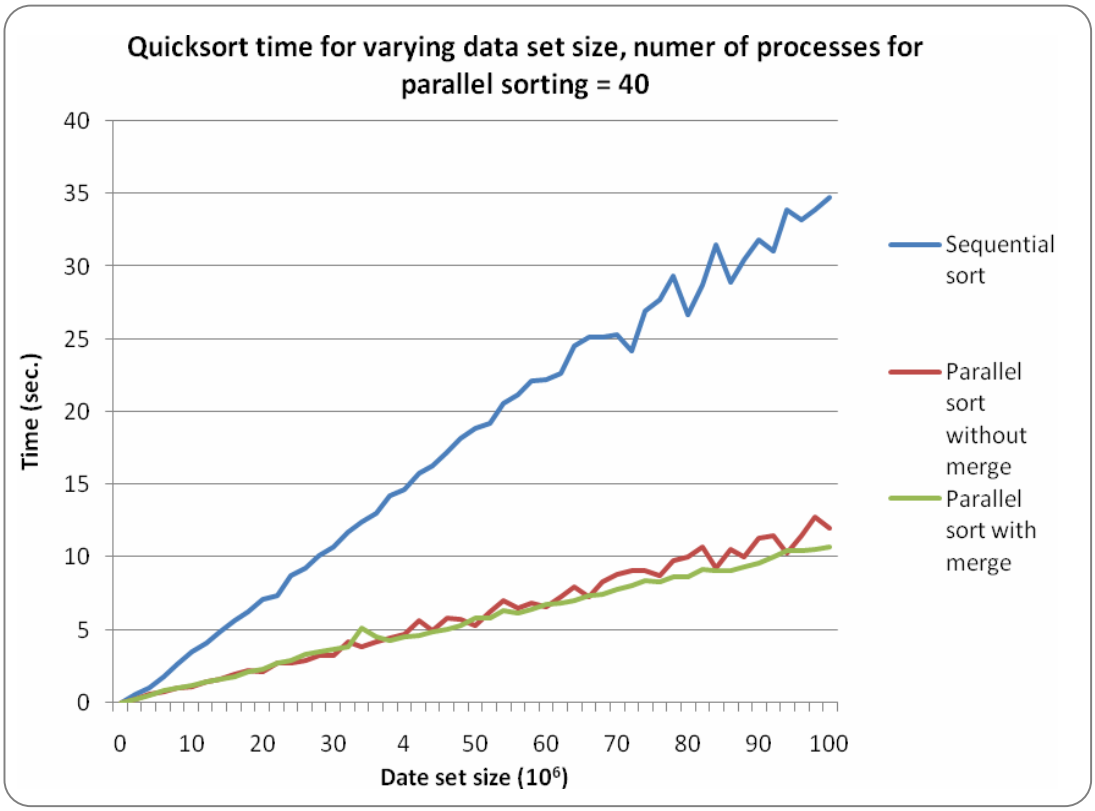
Sequential implementation was the simple quicksort algorithm, runs with a single process. Each data set was first ran with the sequential implementation and then with two parallel implementations.

4.2 Parallel quicksort with merge

This implementation belongs to Puneet C Kataria's, [Parallel Quciksort Implementation Using MPI and Pthreads] a project aimed to implement a parallelized quicksort with minimum cost of merging by using tree structured merging. He is currently a graduate student and more details about his implementation can be found in his personal page, http://www.winlab.rutgers.edu/~pkataria/



Graph 1: Data plot for the sorting times for varying data set sizes.



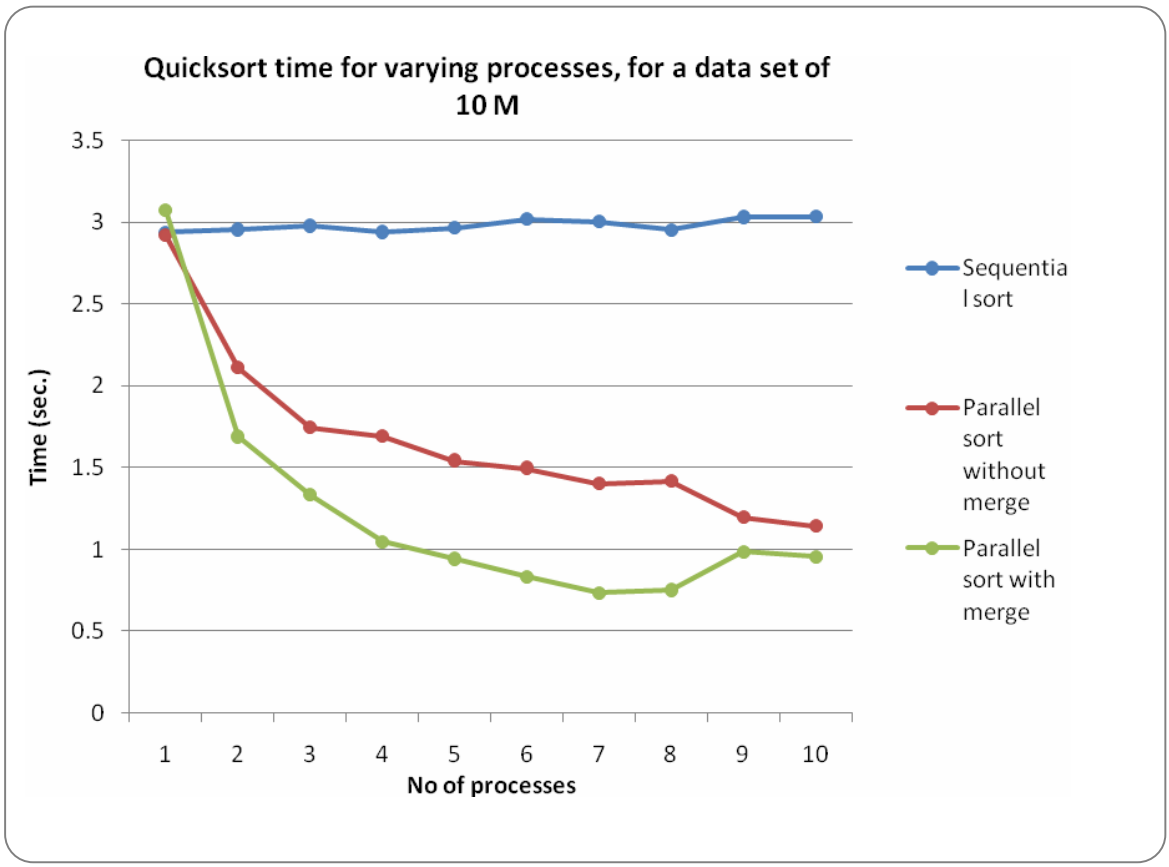
Graph 2: Data plot for the sorting times for varying data set sizes.

These graphs show the results of sorting data with a range of 1-100 M. Each data set was averaged with multiple runs and the two figures are corresponding to the results of running parallel quicksort without merge and with merge implementations, parallelized by 5 to 40 processes.

Graph1 indicates a lowest running time for parallel quicksort with merge and the time increment tend to be very smooth and linear. And the parallel quicksort without merge tends to show fluctuations with the increasing size of data set yet linear.

Graph2 indicates that the two implementations tend show close results, yet the parallel quicksort without merge implementation shows much irregularity than the other parallel implementation.

These results show that overall parallel implementation with initial partitioning and merging outperforms the parallel quicksort implementation without merge. Also the fixed partitioning shows much smoother increment of time with increasing data size on less number of processes. But the parallel quicksort without merge shows the opposite behavior as it gets smoother with the higher number of processes.

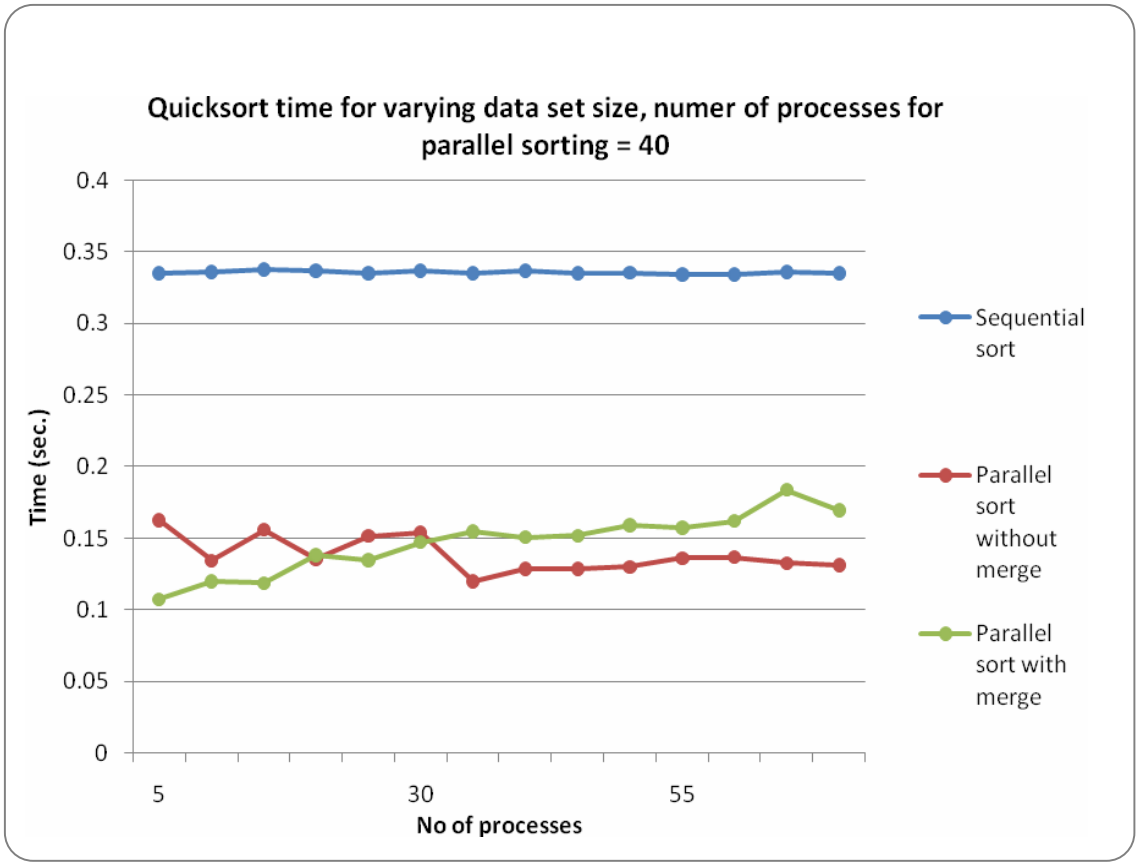


Graph 3: Data plot for the varying number of processes [1 – 10], data set size 10^7 [10 M]

This diagram indicates the results after running two parallel implementations with 1 – 10 processes for a random data set of 10^7 (10 M). Parallel quicksort with initial fixed size partitioning has again showed better performances while increasing the number of processes and lowest running time was when the number of processes equals to 7. On the

other hand, the parallel quicksort implementation without initial fixed partitioning also seems to show parallelism when number of processes were increased but it underperformed the first one.

This quite explains that fixed partition implementation seems to outperform in small number of processes comparing to the parallel quicksort without initial fixed partitioning. When the number of processes was small, the fixed partitioning implementation seems to be more efficient with small number of merging steps but the parallel implementation without initial partitioning seems to be not efficient regarding the unbalanced partitioning and communication overheads.



Graph 4: Data plot for the varying number of processes [5 – 70], data set size 10^6 [1 M]

Above figure shows the results of sorting 1M data set with increasing the number of processes from 5 to 70.

Though without merge implementation outperform with merge implementation at some value of processes, the lowest time was shown by the with merge implementation.

4.7 Speedup analysis

Speedup = Running time for best sequential algorithm

Running time for parallel algorithm

Speedup was estimated based on the results of sorting 10 M data set. [Graph 3]

No of processes	With fixed partition	Partition by regular sampling
2	1.752	1.4
3	2.236	1.71
4	2.813	1.73
5	3.148	1.923
6	3.637	2.021
7	4.095	2.141
8	3.938	2.084
9	3.084	2.536
10	3.186	2.656

According to speedup analysis, fixed partition outperformed the partition by regular sampling and the highest gained by fixed partitioning and partitioning by regular sampling implementations were 4.05 and 2.655.

5. Time complexity Analysis

With the general assumption of time complexity for sequential quicksort = n log n,
Let's take the best case sort analysis for the above implementation,
So assuming that each partitioning might create two equal sections,

For a two process running,
T (n) = (n/2) log (n/2)
For three process running,
T (n) = max ((n/2) log (n/2) + (n/4) log (n/4))
= (n/2) log (n /2)
According to table 1, after 4 processes running,
T (n) = (n/4) log (n/4)

Hence for p number of processes,
T (n) = (n / k) log (n / k) where k satisfies = $2^{k-1} < p <= 2^k$

But since the implementation runs as sharing the smallest partition strategy and variable partitioning sizes, this time analysis can't prove the average running time for the parallel quicksort without merging. Also adding up the overhead of communication is not taken to account. Also the experimental results also proven that this time complexity isn't preserved for the average run of the implementation.

6. Conclusion

With the performance analysis, it's evident that the new implementation seems to become more efficient with a higher number of processes.

Also it could outperform with merge implementation in certain number of processes. But with the performance basis, the lowest running time was shown by the with merge implementation for a less number of processes. Without merge implementation performs less efficiently with fewer number of processes.

As for the conclusion below points can be highlighted.

- Initial partitioning can be very crucial in parallel quicksort with MPI.
  After implementing parallel quicksort without initial fixed partitioning and running it against a counter-implementation, it's evident that the time efficiency gains from initial fixed petitioning is important. If not, a better way to perform initial partitioning through regular sampling is vital in order to gain higher performance.
- With MPI, two parallel implementations tend to show less performance on higher number of processes since MPI communication overheads. This overcome has to compensate by limiting the amount of running processes according to the data set size.
- Also partitioning through regular sampling, can cause very irregular running time due to variable partitioning in each step. Also this may leads to a worst case of highly unbalanced data distributions which might lead to inefficient time performances.
- With variable partitioning in each step considering the available processes, some processes may not be efficiently used. In fact for the worst case with a higher number of processes for a small data set, some processes may leave out without any data to sort sequentially which shows inefficient use of processes.

So this analysis emphasized the point of initial partitioning combined with regular sampling that can partition input data set to equal subsets is important in order to improve performance in parallel quicksort. Also more parallelized merging and regular sampling can improve performance.

References

[1]. Puneet C Kataria, Parallel quicksort implementation using MPI and Pthreads

[2]. Hanmao Shi Jonathan Schaeffer, Parallel Sorting by Regular Sampling

[3]. Philippas Tsigas and Yi Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise10000