ASSIGNMENT REPORT

On

PARALLEL PROGRAMMING

Title**:**

Parallel Implementation and Evaluation of QuickSort using Open MPI

By

First name: Soulemane

Last name: Moumie

Submission Date: 21st January 2016

1 Contents

1. Objective:

Sorting is used in human activities and devices like personal computers, smart phones, and it continues to play a crucial role in the development of recent technologies. The QuickSort algorithm has been known as one of the fastest and most efficient sorting algorithm. It was invented by C.A.R Hoare in 1961and is using the divide-and-conquer strategy for solving problems [3]. Its partitioning aspects make QuickSort amenable to parallelization using task parallelism. MPI is a message passing interface library allowing parallel computing by sending codes to multiple processors, and can therefore be easily used on most multi-core computers available today. The main aim of this study is to implement the QuickSort algorithm using the Open MPI library and therefore compare the sequential with the parallel implementation. The entire study will be divided in many sections: related works, theory of experiment, algorithm and implementation of Quicksort using open MPI, the experimental setup and results, problems followed by the conclusion and future works.

2 Related Works

Many studies have been done on parallel implementation of Quicksort algorithm using either Posix threads (Pthreads), OpenMP or message passing interface (OpenMPI). For instance the time profit obtained by the parallelization of Quicksort Algorithm using OpenMP is presented in [9]. Here authors have taken advantage of the rich library offers by OpenMP and its simplicity to considerably reduce the execution time of Quicksort in parallel compared to its sequential version. In this study the unpredictability of the execution time in parallel version appears to be the main problem. Another similar experiment [7] on parallel Quicksort is carried out using multiprocessors on clusters by OpenMPI and Pthreads highlighting different benchmarking and optimization techniques. The same implementation of the algorithm done in [7] is used in [8] this time focusing merely on performance analysis. In short [7, 8] present almost a common implementation of parallel Quicksort and MPI, after the sorting their respective chunks of data, a tree based merge is used for the collection of different chunks in order to get the final sorted data. However in our study a completely different approach has been taken. In our implementation, we have simply used the MPI collective routine and an additional sorting step to replace the merge functionality. Initially for each version of QuickSort (iterative and recursive) algorithm we have made a sequential implementation using the C programming language. Then we have ported the two sequential implementations to MPI in order to run them in parallel.

3 Theory of Experiment

Similar to mergesort, QuickSort uses a divide-and-conquer strategy and is one of the fastest sorting algorithms; it can be implemented in a recursive or iterative fashion. The divide and conquer is a general algorithm design paradigm and key steps of this strategy can be summarized as follows:

- Divide**:** divide the input data set S into disjoint subsets S_1, S_2, $S_3 \ldots S_{k.}$
- Recursion**:** solve the sub-problems associated with S_1, S_2, $S_3 \ldots S_k$
- Conquer**:** combine the solutions for S_1, S_2, $S_3 \ldots S_{k.}$ into a solution for S.
- Base case**:** the base case for the recursion is generally sub-problems of size 0 or 1.

Many studies [2] have revealed that in order to sort N items; it will take the QuickSort an average running time of O(NlogN). The worst-case running time for QuickSort will occur when the pivot is a unique minimum or maximum element, and as stated in [2] the worst-case running time for QuickSort on N items is $O(N^2)$. These different running times can be influenced by the inputs distribution (uniform, sorted or semi-sorted, unsorted, duplicates) and the choice of the pivot element. Here is a simple Pseudocode of the QuickSort algorithm adapted from Wikipedia [1].

```
function quicksort(array)
    less, equal, greater := three empty arrays
    if length(array) > 1
        pivot := select any element of array
        for each x in array
            if x < pivot then add x to less
            if x = pivot then add x to equal
            if x > pivot then add x to greater
        quicksort(less)
        quicksort(greater)
        array := concatenate(less, equal, greater)
```

As shown in the above Pseudocode, the following is the basic working mechanism of Quicksort:

- Choose any element of the array to be the pivot
- Divide all other elements (except the pivot) into two partitions
  - All elements less than the pivot must be in the first partition
  - All elements greater than the pivot must be in the second partition
- Use recursion to sort both partitions
- Join the first sorted partition, the pivot and the second partition
- The output is a sorted array.

We have made use of Open MPI as the backbone library for parallelizing the QuickSort algorithm. In fact learning message passing interface (MPI) allows us to strengthen our fundamentals knowledge on parallel programming, given that MPI is lower level than equivalent libraries (OpenMP). As simple as its name means, the basic idea behind MPI is that messages can be passed or exchanged among different processes in order to perform a given task. An illustration can be a communication and coordination by a master process which split a huge task into chunks and share them to its slave processes. Open MPI is developed and maintained by a consortium of academic, research and industry partners; it combines the expertise, technologies and resources all across the high performance computing community [11]. As elaborated in [4], MPI has two types of communication routines: point-to-point communication routines and collective communication routines. Collective routines as explained in the implementation section have been used in this study.

4 Algorithm and Implementation of QuickSort with MPI

A detailed explanation of how we have ported the QuickSort algorithm to MPI is given in the following sections:

4.1 Proposed Algorithm

In general the overall algorithm used here to perform QuickSort with MPI works as followed:

i. Start and initialize MPI.

ii. Under the root process ***MASTER,*** get inputs:

   a. Read the list of numbers ***L*** from an input file.

   b. Initialize the main array ***globaldata*** with **L**.

   c. Start the timer.

iii. Divide the input size ***SIZE*** by the number of participating processes ***npes*** to get each chunk size ***localsize.***

iv. Distribute ***globaldata*** proportionally to all processes:

   a. From ***MASTER*** scatter ***globaldata*** to all processes.

   b. Each process receives in a sub data ***localdata.***

v. Each process locally sorts its ***localdata*** of size ***localsize.***

vi. Master gathers all sorted ***localdata*** by other processes in ***globaldata***.

   a. Gather each sorted ***localdata.***

   b. Free ***localdata.***

vii. Under MASTER perform a final sort of ***globaldata.***

   a. Final sort of ***globaldata.***

   b. Stop the timer.

   c. Write the output to file.

   d. Sequentially check that ***globaldata*** is properly and correctly sorted.

   e. Free ***globaldata.***

viii. Finalize MPI.

4.2 Implementation

In order to implement the above algorithm using C programming, we have made use of a few MPI collective routine operations. Therefore after initializing MPI with **MPI_Init,** the size and rank are obtained respectively using **MPI_Comm_size** and **MPI_Comm_rank.** The beginning wall time is received from **MPI_Wtime** and the array containing inputs is distributed proportionally to the size to all participating processes with **MPI_Scatter** by the root process which collect them again after they are sorted using **MPI_Gather.** Finally the ending wall time is retrieved again from **MPI_Wtime** and the MPI terminate calling **MPI_Finalize.** In the following part, each section of our proposed algorithm is illustrated with a code snippet taken from the implementation source code.

i. Start and initialize MPI.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

ii. Under the root process ***MASTER***, get inputs:

```
if (rank == MASTER) {
  printf("SZIE IS  %lld", SIZE);
  globaldata = malloc (SIZE * sizeof(long long) );
  if (globaldata == NULL) {
    printf ("\n\n globaldata Memory Allocation Failed !   \n\n ");
    exit(EXIT_FAILURE);
  }
  inp = fopen (argv[1],"r");          /* Open file for reading         */
  if (inp == NULL) {
    printf ("\n\n inp Memory Allocation Failed !  \n\n ");
    exit(EXIT_FAILURE);
  }
  printf ("\n\nInput Data \n\n ");
  for (i=0; i<SIZE; i++) {
    retscan = fscanf (inp, "%lld \t", &tmp);
    globaldata[i] = tmp;
  }
```

Starting the timer.

```
if (rank == MASTER)
{
  t_start = MPI_Wtime ();
}
```

iii. Divide the input size ***SIZE*** by the number of participating processes ***npes*** to get each chunk size ***localsize***.

```
/*Getting the size to be used by each process                              */
if (SIZE < npes) {
  printf ("\n\n SIZE is less than the number of process!  \n\n ");
  exit (EXIT_FAILURE);
}
localsize = SIZE/npes;
```

iv. Distribute ***globaldata*** proportionally to all processes:

```
/*Scatter the integers to each number of processes (npes)              */
MPI_Scatter (globaldata, localsize, MPI_LONG_LONG, localdata,
                  localsize, MPI_LONG_LONG, MASTER, MPI_COMM_WORLD);
```

v. Each process locally sorts its ***localdata*** of size ***localsize***.

```
/* Perform local sort on each sub data by each process                 */
quickSortRecursive (localdata,0, localsize-1);
```

vi. ***Master*** gathers all sorted ***localdata*** by other processes in ***globaldata***.

```
/* Merge locally sorted data of each process by MASTER to globaldata  */
MPI_Gather (localdata, localsize, MPI_LONG_LONG , globaldata,
                  localsize, MPI_LONG_LONG, MASTER, MPI_COMM_WORLD);
free (localdata);
```

vii. Under ***MASTER*** perform a final sort of ***globaldata***.

```
if (rank == MASTER) {
  /* Final sorting                                                    */
  quickSortRecursive (globaldata, 0, SIZE-1);
```

Stop the timer

```
/* End wall  time                                                    */
t_end = MPI_Wtime ();
```

Write information to in the output file

```
/* Opening output file to write sorted data                          */
out = fopen (argv[2], "w");
if (out == NULL) {
  printf ("\n\n out Memory Allocation Failed !  \n\n ");
  exit (EXIT_FAILURE);
}
/* Write information to output file                                  */
fprintf (out, "Recursively Sorted  Data : ");
fprintf (out, "\n\nInput size : %lld\t", SIZE);
fprintf (out, "\n\nNber processes : %d\t", npes);
fprintf (out, "\n\nWall time      : %7.4f\t", t_end - t_start);
printf ("\n\nWall time      : %7.4f\t", t_end - t_start);
fprintf (out, "\n\n");
for (i = 0; i<SIZE; i++) {
  fprintf (out, " %lld \t", globaldata[i]);
}
fclose (out);                    /* closing the file                 */
```

Checking that the final globaldata array is properly sorted.

```
/* checking if the final globaldata content is properly sorted      */
sortCheckers ( SIZE, globaldata );
```

Free the allocated memory

```
if (rank == MASTER) {
  free (globaldata);                 /* free the allocated memory    */
}
```

ix. Finalize MPI.

```
/* MPI_Finalize Terminates MPI execution environment                 */
MPI_Finalize ();
```

The two versions of QuickSort algorithm have been implemented; however even though they have almost the same implementation using similar functions, the recursion in the recursive part has been replaced by a stack in order to make it iterative. Their function signatures are presented in the following part:

- Recursive QuickSort

```
void quickSortRecursive (long long [], long long, long long);
long long partition (long long [], long long , long long );
void swap (long long [], long long , long long );
void sortCheckers (long long, long long []);
long long getSize ( char str[] );
```

- Iterative QuickSort

```
void quickSortIterative (long long [], long long, long long);
long long partition (long long [], long long , long long );
void swap (long long [], long long , long long );
void sortCheckers (long long, long long []);
long long getSize ( char str[] );
```

The input file necessary to run this program can be generated using an input generator where we specify the size and the filename (***input_generator.c***), then compile and run with the following instructions:

5 Experimental Setup and Results

This section presents in a more elaborated way the results obtained by applying real data to different input parameters on parallel QuickSort. It comprises the following sections: hardware specifications, software specifications, the installation, the results, visualization and analysis.

5.1 Hardware specifications

- Computer : ProBook-4440s
- Memory : 3,8 GiB
- Processor : Intel® Core™ i5-3210M CPU @ 2.50GHz × 4

- Graphics : Intel® Ivybridge Mobile x86/MMX/SSE2
- OS type : 32-bit
- Disk : 40,1 GB

5.2 Software specifications

- Operating system : Ubuntu 14.04 LTS
- Open MPI : 1.10.1 released
- Compilers : gcc version 4.8.4
- Microsoft Excel 2007 for plotting

5.3 Installation

The installation procedure of open MPI is not always straightforward, and varies depending on the intended environment. Initially we have installed open MPI on Microsoft windows 7 operating systems using Cygwin. CygWin is a Unix-like environment and command-line interface for Microsoft Windows. It provides native integration of Windows-based application, data, and other system resources with applications, software tools, and data of the Unix-like environment [10]. The full installation procedure of CygWin on Microsoft Windows can be found in [5]. After updating CygWin we faced an unresolved issue and critical error as explained in the problems section of this report. We therefore opted for installing a virtual machine with Linux as a guest operating system in which MPI is installed. Again due to some limitations explained in the problem section, we finally installed CygWin on Ubuntu normally installed as host operating system. Ubuntu like other Linux flavored operating systems provides a suitable environment for the installation and execution of Open MPI packages. Basic installation steps can be found in [6]. All the following results obtained in this study are from our Ubuntu-based installation. However Open MPI can also be installed in some other operating systems not mentioned here.

5.4 Results

The following table presents the different recorded data. In the first column we have the experiment number (No.); the second column is the number of participating processes (# process), the third column is the input data size applied to QuickSort. Finally the last two columns represent respective the execution wall time of the iterative and recursive version of parallel QuickSort.

No.	# process	Input size	Iterative wall time	Recursive wall time
1	1	20	0.0000	0.0000
	2		0.0001	0.0000
	5		0.0002	0.0004
	10		0.0004	0.0005
	20		0.0013	0.0032
2	1	100	0.0000	0.0000
	2		0.0001	0.0001
	5		0.0002	0.0004
	10		0.0003	0.0005
	20		0.0016	0.0020
3	1	1000	0.0011	0.0012
	2		0.0003	0.0003
	5		0.0004	0.0004
	10		0.0007	0.0008
	20		0.0014	0.0016
4	1	10000	0.0849	0.0860
	2		0.0030	0.0030
	5		0.0031	0.0030
	10		0.0038	0.0035
	20		0.0035	0.0043
5	1	100000	8.2165	8.5484
	2		0.0393	0.0383
	5		0.0333	0.0325
	10		0.0418	0.0488
	20		0.0446	0.0475
6	1	1000000	835.8316	2098.7
	2		0.4786	0.4471
	5		0.3718	0.3590
	10		0.3646	0.3445
	20		0.4104	0.3751

5.5 Visualization

Different charts comparing the iterative QuickSort and the recursive QuickSort for different number of processes and various input sizes are presented in this section. The X-axis represents the number of processes and the Y-axis represents the execution wall time in seconds.

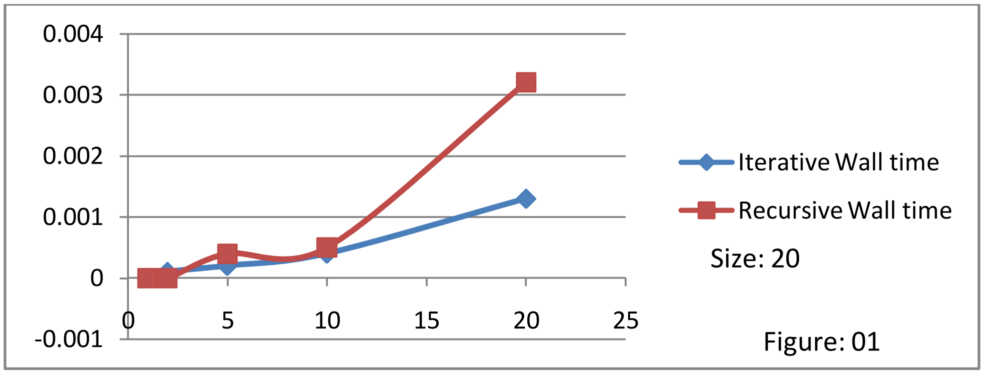


Figure: 01

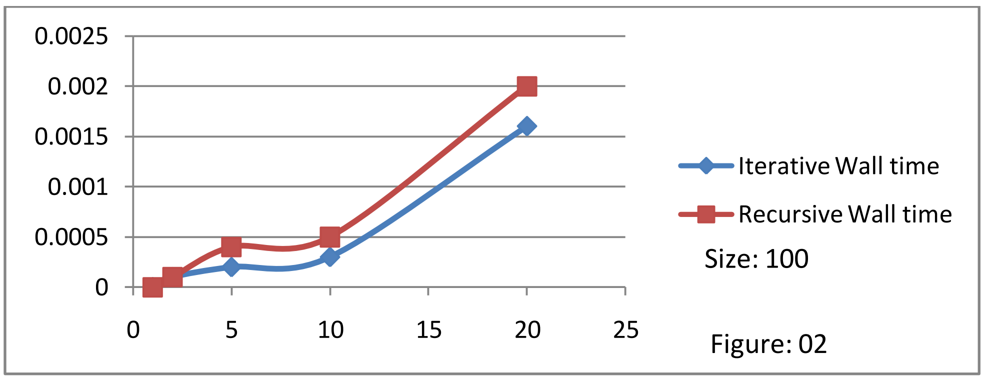


Figure: 02

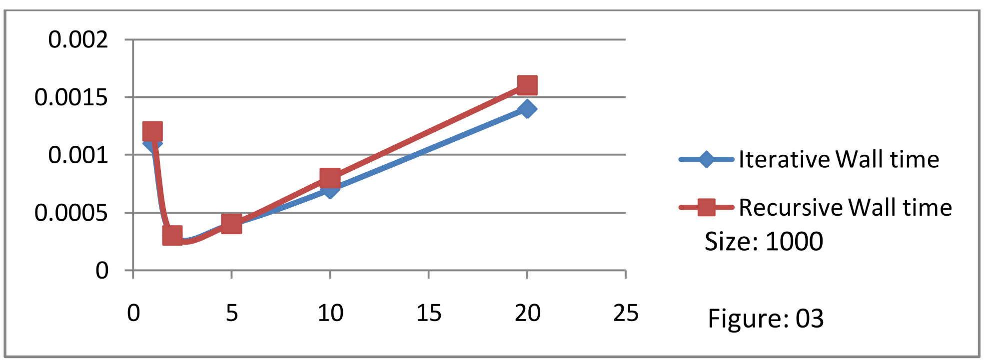


Figure: 03

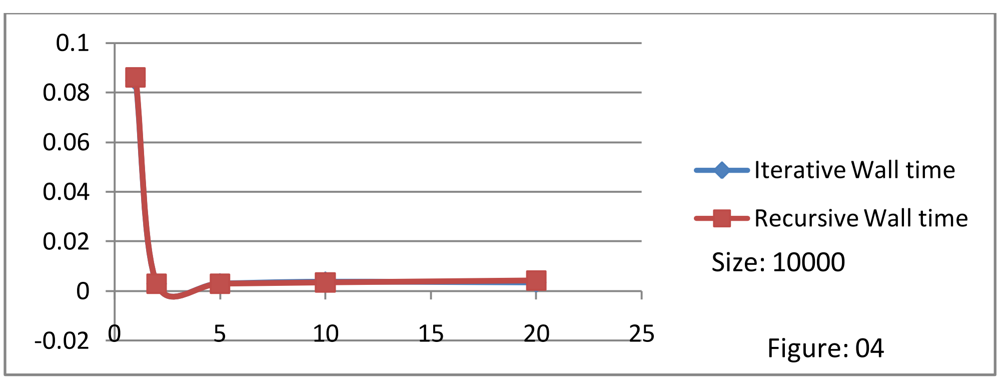


Figure: 04

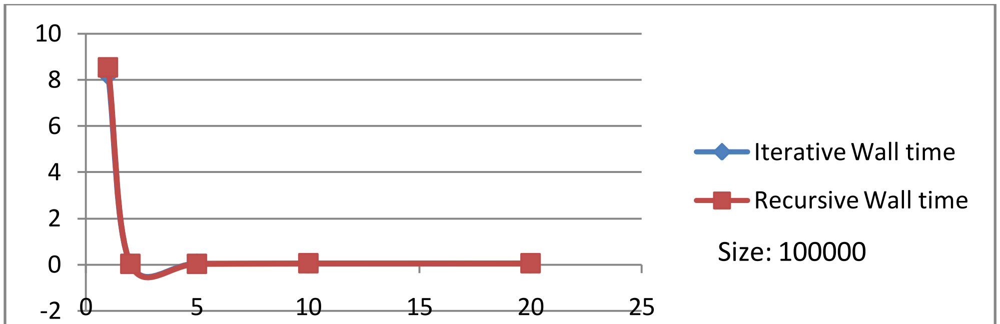


Figure: 05

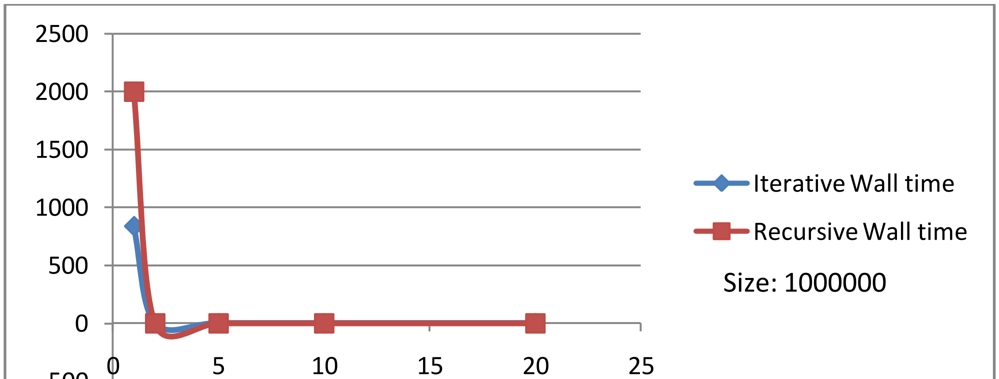


Figure: 06

Here is a sample output file format we have used along this experiment.

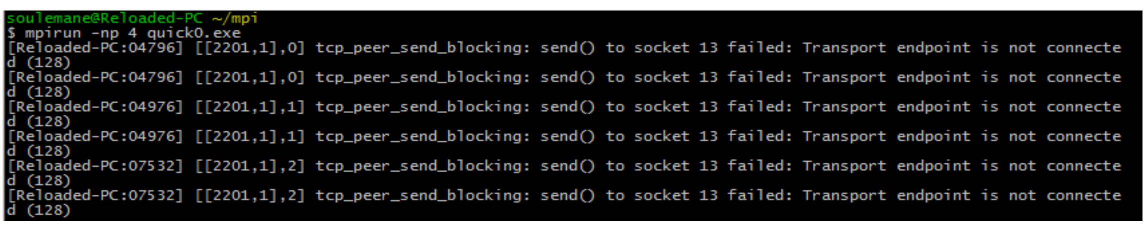
```
Iteratively Sorted  Data :

Input size : 100

Nber processes : 10

Wall time       :  0.0003

 2   3   5   5   8   11    11    12    13    13
```

5.6 Analysis

In Figure 01 and Figure 02, the sorting is faster with a single process and keeps slowing as long as the number of processes increases. However in Figure 03 the execution time drastically dropped from a single process to its fastest execution time where the two processes are running in parallel, then it starts slowing again when we increases the number of processes. Finally in Figure 04 and Figure 05 the iterative and recursive implementations have almost the same execution time, the sorting becomes faster with more processes, with less variations. Figure 06 having one million numbers as input size. Here on single process, the sorting is the slowest of this experiment and even stop on recursion. The sorting becomes again faster with the number of process increases. On these different charts we can clearly observe that in general the iterative QuickSort is faster than the recursion version. On sequential execution with a single process, the sorting is faster with small input and slows down as long as the input size goes up. However on parallel execution with more than one process, the execution time decreases when the input size and the number of process increases. We noticed sometimes an unusual behavior of MPI execution time that keeps changing after each execution. We have taken only the first execution time to minimize this variation and obtain a more consistent execution time.

6 Problems

This project was initially set to be done on Microsoft windows using Cygwin; however after facing some problems following the update on Cygwin we opted for a virtual machine. A sample screenshot of error message is shown below.



We then install CygWin under Ubuntu installed as a guest operating system on Oracle virtual box. The entire system appeared to be less-user friendly and sometimes extremely slow. Due to the limited hardware configuration, we had only the possibility of using 4 processes with 3 processes being recommended. It became better after we installed CygWin on Ubuntu, with Ubuntu a normal operating system.

Regarding our current implementation of QuickSort using open MPI, there exist certain undesirable features and limitations presented in the future works section as possible future research directions.

7 Conclusion

To conclude this project, we have successfully implemented the sequential QuickSort algorithm both the recursive and iterative version using the C programming language. Then we have done a parallel implementation with the help of Open MPI library. Through this project we have explored different aspects of MPI and the QuickSort algorithm as well as their respective limitations. This study has revealed that in general the sequential QuickSort is faster on small inputs while the parallel QuickSort excels on large inputs. The study also shows that in general the iterative version of QuickSort perform slightly better than the recursive version. The study reveals some unusual behavior of the execution time that unexpectedly varies. However due to the limited time we could not overcome all the difficulties and limitations, the next section opens an eventual scope for future improvements.

8 Future Works

This parallel implementation of QuickSort using MPI can be expanded or explored by addressing its current limitations in the following ways:

- Instead of the input size being always divisible by the number of processes, it can be made more dynamic.
- Collective communication routines can be replaced by point-to-point communication routines.
- Gathering locally sorted data and performing a final sort can just be replaced with a merge function.
- Minimizing some variation observed in the execution time by calculating the average running time.
- This proposed algorithm can be re-implemented with other programming language supporting Open MPI like C++, Java or Fortran.

9 References

[1] Rosetta code, available at http://rosettacode.org/wiki/Sorting_algorithms/Quicksort , December 2015.

[2] P. Tsigas and Y. Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP'03)*, 2003, IEEE.

[3] HOARE, C. A. R. 1961. Algorithm 64: Quicksort.Commun. ACM 4, 7, 321.

[4] Blaise Barney, *Lawrence Livermore National Laboratory*, available at https://computing.llnl.gov/tutorials/mpi/, December 2015

[5] Rer. nat. Siegmar Groß, available at http://www2.hsfulda.de/~gross/cygwin/cygwin_2.0/inst_conf_cygwin_2.0.x_x11r7.x.pdf , December 2015

[6] D. G. Martinez, S. R. Lumley, Installation of Open MPI parallel and distributed programming.

[7] P. Kataria, Parallel quicksort implementation using MPI and Pthreads, December 2008

[8] P. Perera, Parallel Quicksort using MPI and Performance Analysis

[9] V. Prifti, R. Bala, I. Tafa, D. Saatciu and J. Fejzaj. The Time Profit Obtained by Parallelization of Quicksort Algorithm Used for Numerical Sorting. In *Science and Information Conference 2015, June 28-30, 2015,* IEEE.

[10] V. Gite, *UNIX Command Line Tools For MS-Windows XP / Vista / 7 Operating Systems*, available at http://www.cyberciti.biz/faq/unix-command-line-utilities-for-windows/ , December 2015

[11] *Open MPI: Open Source High Performance Computing*, available at https://www.open-mpi.org/, December 2015