A MINI PROJECT REPORT ON

Evaluate Performance enhancement of Parallel Quicksort Algorithm using MPI.

SUBMITTED TO THE SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE
IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF THE DEGREE
OF

BACHELOR OF ENGINEERING (COMPUTER ENGINEERING)

SUBMITTED BY

YASH DHUMAL 14369

Dr. D. Y. PATIL
College of Engineering and Innovation
APPROVED BY AICTE, RECOGNIZED BY GOVT. OF MAHARASHTRA & AFFILIATED TO S. P. PUNE UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

Dr. D. Y. Patil College of Engineering and Innovation,

Varale, Talegaon, Pune

SAVITRIBAI PHULE PUNE UNIVERSITY

2024 - 2025

Introduction:-

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 algorithms. It was invented by C.A.R Hoare in 1961 and is using the divide-and-conquer strategy for solving problems. 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.

Theory

Like merge sort, 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 S1, S2, S3…Sk.
- **Recursion**: Solve the sub-problems associated with S1, S2, S3…Sk.
- **Conquer**: Combine the solutions for S1, S2, S3…Sk. into a solution for S.
- **Base case**: The base case for the recursion is generally sub-problems of size 0 or 1.

Many studies have revealed that 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, the worst- case running time for Quicksort on N items is O(N2). These different running times can be influenced by the input's distribution (uniform, sorted or semi-sorted, unsorted, duplicates) and the choice of the pivot element.

Algorithm and Implementation of Quick sort with MPI

1. Algorithm

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

1) Start and initialize MPI.
2) Under the root process MASTER, get inputs:
    a) Read the list of numbers L from an input file.
    b) Initialize the main array global data with L.
    c) Start the timer.
3) Divide the input size SIZE by the number of participating processes npes to get each chunk size local size.
4) Distribute global data proportionally to all processes:
    a) From MASTER scatter global data to all processes.
    b) Each process receives in a sub data loca ldata.
5) Each process locally sorts its local data of size local size.
6) Master gathers all sorted local data by other processes in global data.
    a) Gather each sorted local data.
    b) Free local data.
7) Under MASTER, perform a final sort of global data.
    a) Final sort of global data.
    b) Stop the timer.
    c) Write the output to file.
    d) Sequentially check that global data is properly and correctly sorted.
    e) Free global data. 8) Finalize MPI.

2. Implementation

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.

Experimental Setup

1. Hardware Specifications

- Memory: 8 GB
- Processor: AMD Ryzen 5 3500x 6-core
- Graphics: Nvidia GeForce GT 710
- OS type: 64-bit
- Disk: 40 GB

2. Software Specifications

- Operating system: Ubuntu 14.04 LTS
- Open MPI: 1.10.1 released
- Compilers: gcc version 4.8.4
- Microsoft Excel 2011for plotting

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. The full installation procedure of Cygwin on Microsoft Windows can be found in. 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 flavoured operating systems provides a suitable environment for the installation and execution of Open MPI packages. Basic installation steps can be found in. 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.

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.	# processor	Input size	Interactive 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

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.

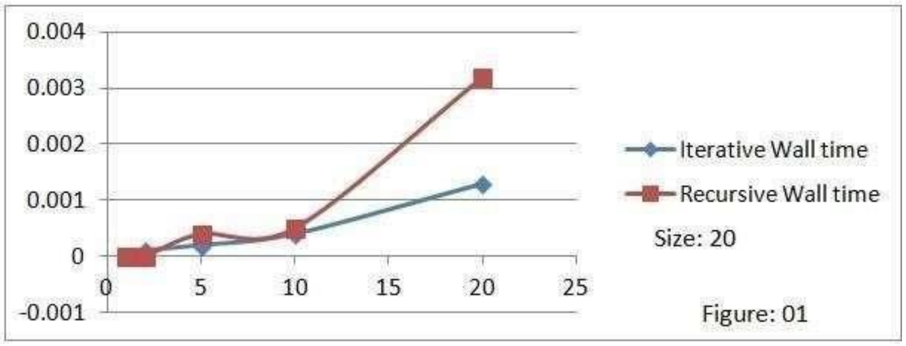

Figure: 01

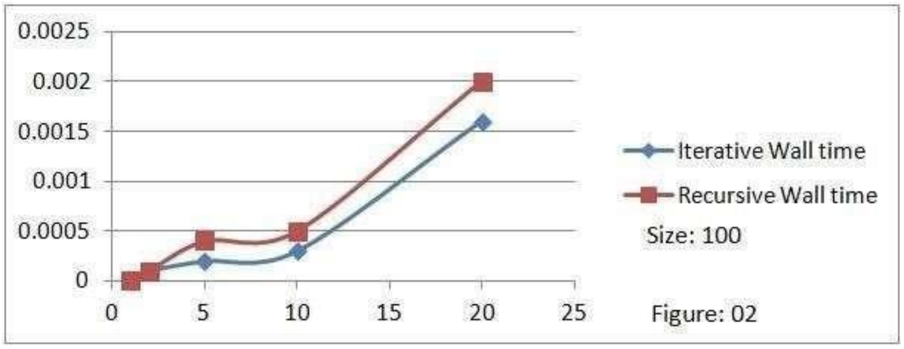

Figure: 02

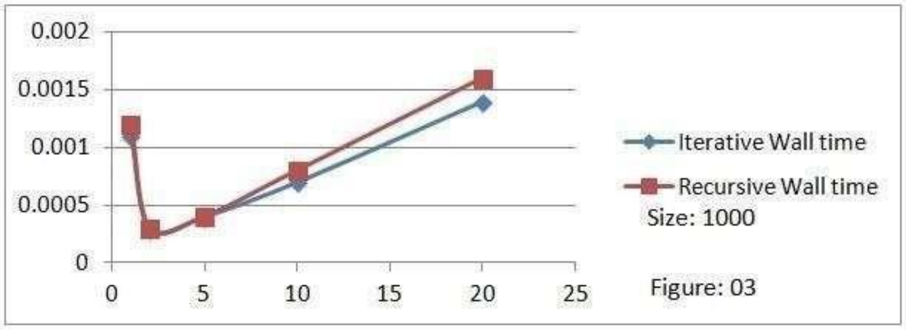

Figure: 03

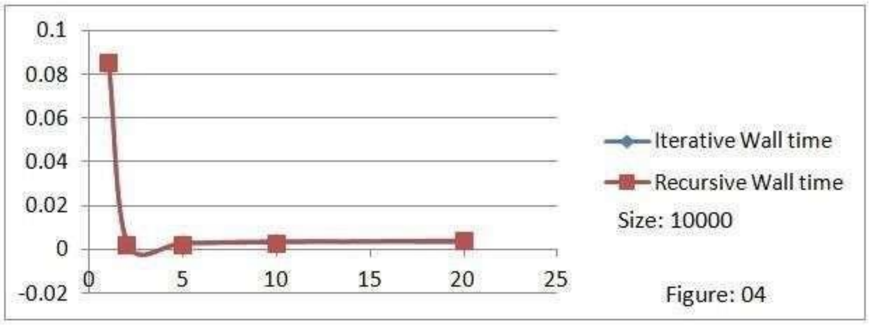

Figure: 04

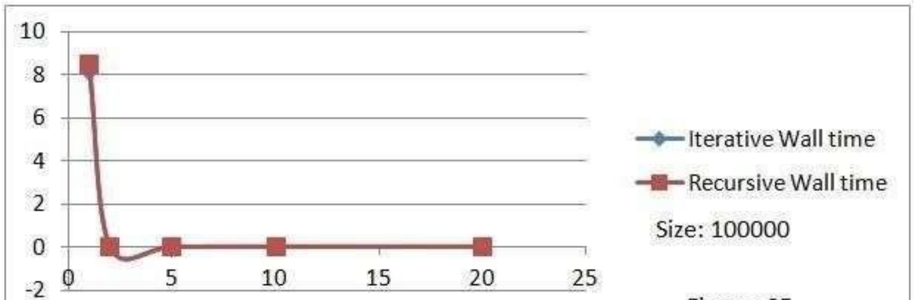

Figure: 05

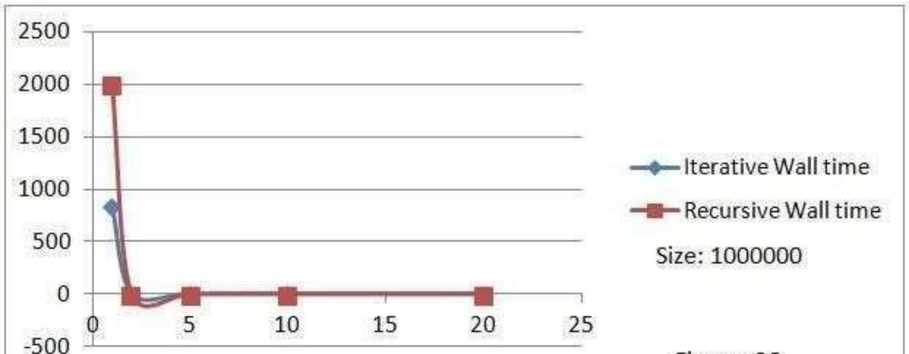

Figure: 06

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 increase 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 processes 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 if 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 processes increases. We noticed sometimes an unusual behaviour 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.

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 performs slightly better than the recursive version. The study reveals some unusual behaviour of the execution time that unexpectedly varies. However, due to limited time, we could not overcome all the difficulties and limitations, the next section opens an eventual scope for future improvements.

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.

Source Code

Input_generator.c

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 10

int main (void)
{
  static            long            long
  globaldata[SIZE];     long     long
  value; long long i = 0;
  long long tmp=0;

  FILE *inp;
  inp = fopen
  ("input_10.txt","w+"); for (i=0;
  i<SIZE; i++)
  {
    value = rand ()% SIZE; fprintf
    (inp, "%lld \t", value);
  }
  fclose (inp);

  inp  =  fopen  ("input_10.txt","r");
  for (i=0; i<SIZE; i++)
  {
    fscanf  (inp,  "%lld  \t",  &tmp);
    globaldata[i] = tmp;
    printf ("%lld \t", globaldata[i]);
  }
  fclose (inp);

  return 0;
}
```

Recursive.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MASTER 0

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[] ); int main (int argc, char **argv) {
  long long SIZE = 1;
  char   strsize[]="";
  int rank;
```

```
long long i; long long
retscan;   long   long
tmp;  double  t_start,
t_end;    long    long
test_size=5;
FILE
*out, *inp; int npes ;
long   long   *globaldata   =
NULL;  long  long  *localdata
=     NULL;     long     long
localsize; if (argc!=3) {
  printf ("\n Properly specify the executable, input , output files");
  printf  ("\nmpirun  -np  <process  nber>  %s  <input  file>  <output  file>\n",
  argv[0]); exit (1);
}
strcpy(strsize,argv[1]);
SIZE = getSize
(strsize);
MPI_Init(&argc,
&argv);
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD,
                &rank); 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"); 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;
  }
  printf   ("\n\n   End   Input   Data");
  fclose (inp);
  printf ("\n\nProcessor %d has data: ", rank);
  for ( i = 0; i<test_size; i++) {
    printf ("%lld \t", globaldata[i]);
  }
  printf ("\n");
}

if (rank == MASTER)
{
  t_start = MPI_Wtime ();
}
if (SIZE < npes) {
  printf ("\n\n SIZE is less than the number of process! \n\n "); exit
  (EXIT_FAILURE);
}
localsize = SIZE/npes;

localdata = (long long*) malloc (localsize* sizeof(long long));
```

```
  if (localdata == NULL) {  printf ("\n\n localdata Memory
    Allocation Failed ! \n\n "); exit (EXIT_FAILURE);
  }
  MPI_Scatter (globaldata, localsize, MPI_LONG_LONG, localdata,localsize,
  MPI_LONG_LONG, MASTER,
MPI_COMM_WORLD);
  quickSortRecursive (localdata,0, localsize-1);
  MPI_Gather (localdata, localsize, MPI_LONG_LONG , globaldata,localsize,
  MPI_LONG_LONG, MASTER,
MPI_COMM_WORLD);
  free   (localdata);   if
  (rank == MASTER) {
    quickSortRecursive (globaldata, 0, SIZE-
    1); t_end = MPI_Wtime (); out = fopen
    (argv[2], "w"); if (out == NULL) {
      printf ("\n\n out Memory Allocation Failed ! \n\n "); exit
      (EXIT_FAILURE);
    }
    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); printf
    ("\n\n");
    sortCheckers ( SIZE, globaldata ); printf
    ("\n\n");
  }
  if (rank == MASTER) {
    free (globaldata);
  }
  MPI_Finalize ();
  return
  EXIT_SUCCESS;
}
long long partition (long long x[], long long first, long long last)
{
  long long
  pivot; long
  long j, i;
  pivot
  = first; i =
  first; j = last;
  while (i
  < j) {

    while (x[i ] <= x[pivot] && i < last) {
      i++;
    }
while (x[j] > x[pivot])
      { j--;
    }
    if (i < j) {
      swap (x, i,
      j);
    }
```

```
 swap    (x,    pivot,    j);
 return j;
}
void swap (long long s[], long long m, long long n) {
 long long tmp;
 tmp   =   s[m];
 s[m]  =  s[n];
 s[n] = tmp;
}


void quickSortRecursive (long long x[], long long first, long long
last) {
 long long pivot;
 if (first < last) {
  pivot   =   partition   (x,   first,   last);
  quickSortRecursive  (x,  first,  pivot-
  1);  quickSortRecursive  (x,  pivot+1,
  last);
 }
}
void sortCheckers (long long SIZE, long long input[]) {
 long long i;
 for (i = 1; i<SIZE; i++) {
  if (input[i-1] > input[i]) {
   printf   ("\n\n%lld   --   %lld   \t",   input[i-1],
   input[i]); printf ("\n\nCheck failed. Array not
   sorted"); break;
  }
  }
  printf ("\n\nCheck successfully completed. Array Sorted");
 }

long long getSize ( char str[] ) {

 char * pch; long long
 count  =  0;  char  *  e;
 long long inpsize = 0;
 pch = strtok (str," ._");

 while ( pch != NULL )
 {
  if (count == 1 ) {


 inpsize = strtoll ( pch, &e, 10 );
  return inpsize;
}
 pch = strtok ( NULL, " .-" ); count
 ++;
 }
}
```