



NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
(KARACHI CAMPUS)

Department of Computer Science

Fall 2022



Project: [Parallelized Quicksort, Binary search,
Optimized Dense Matrix Manipulation]

Group Members:

[Syed Aun Ali] - [20K-0286]

[Ammar Amin] - [20K-0285]

[Muhammad Anas] - [20K-0179]

Objective:

Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quick Sort that pick pivot in different ways. In this project, the algorithm will be implemented in parallel environment and then its computing time will be compared to the conventional one [that is OpenMP implementation].

Binary Search is a divide and conquer searching algorithm, which only works on a sorted array. The approach is to repeatedly divide an array into half until the element required is found.

Dense Matrix Manipulation involves optimizing the conventional practices relating matrix in Message Passing Environment.

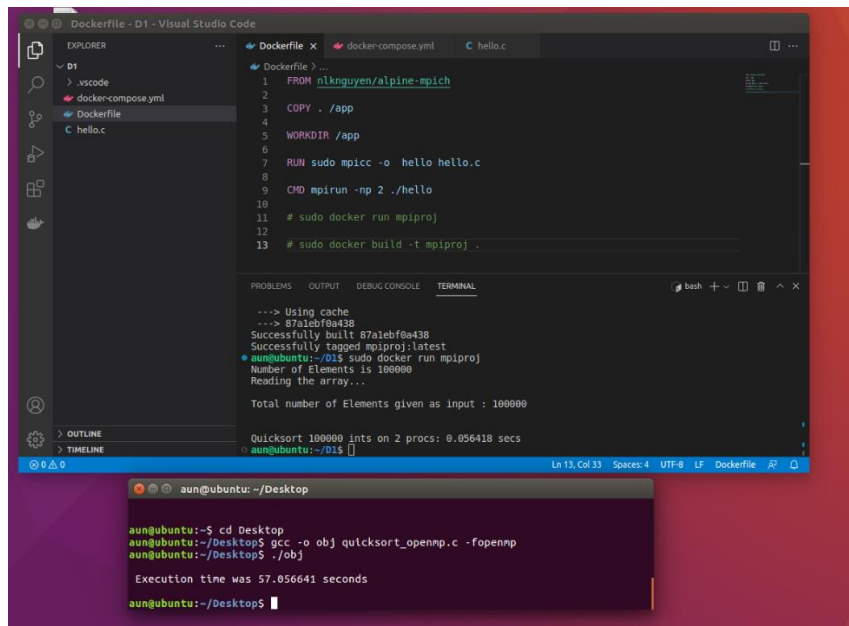
Methodology:

Quicksort is implemented via MPI with message passing kept in mind. The goal was to reduce this message passing making sure that communication was kept to minimum and hence computation to communication ratio would remain high. Firstly, the variables used throughout the process and the MPI itself is initialized. Then the master process scatters the data and allocates chunk size of the data to each process. This actually determines the size of each process that needs to work on the data. The chunk size for each process is calculated by dividing the total number of elements of the data and the total number of processes after comparing their mod. Then with respect to chunk size and number of processes, the actual data itself is dynamically calculated for each process and values are assigned to it (that are random values in range from 0 to 1000). The data will consist of 100,000 unsorted random integers. Zeroes are assigned to particular indexes in order to separate the data for each process inside the same data array. After this, MPI barrier ensures that all processes are blocked until they reach this particular line of the code. The timer is started after this instruction. Now the size is broadcasted to all processes from the root process. The total size of all chunks is calculated and size of each chunk is scattered to all processes. Since the data is distributed, we will dynamically 'free' it and nullify it (this is to ensure that there are no conflicts since the data is being worked on parallelly). The chunk, along with its own size, is sent to the quicksort function. All of the processes do this. Then a loop, along with a condition, sends data to even processes first (since they have no data dependencies) then decrements and makes sure all processes receive their respective part. For the part where the chunk size themselves are received by the processes, firstly it is compared with number of elements. If it exceeds then those chunks are associated with even process, else odd. Then finally the respective chunk received is sent to merge function and 'free' ed. Here the timer stops and the sorted data is shown along with the time taken to sort the unsorted data. Finally, MPI finalize finishes the actual MPI process.

Binary Search is implemented using both Open MP and MPI separately. First, we created and allocated data to an array, then initialize the key (element of the array we want to find). Next step was to divide the array into chunks and have different available processes work on it. Each process will further divide the chunk of array and check if the middle value is equal to the key? If yes, the process will print a message letting the user know the element has been found along with the time the process took to complete the search, otherwise it will keep dividing the array until the top value also becomes the bottom value.

For **Dense Matrix Manipulation**, at first, the matrix dimensions will be broadcast via MPI_Bcast to the workers. The size of the matrix is fixed. Afterwards, the 2-Dim matrix is converted into a 1-Dim matrix in order to distribute the matrix data. Secondly, the matrix data is broadcasted to the workers. Each slave computes its own matrix area with the mpi rank. Lastly, the data is collected via MPI_Gather. At the end, the master presents the result matrix.

Results:

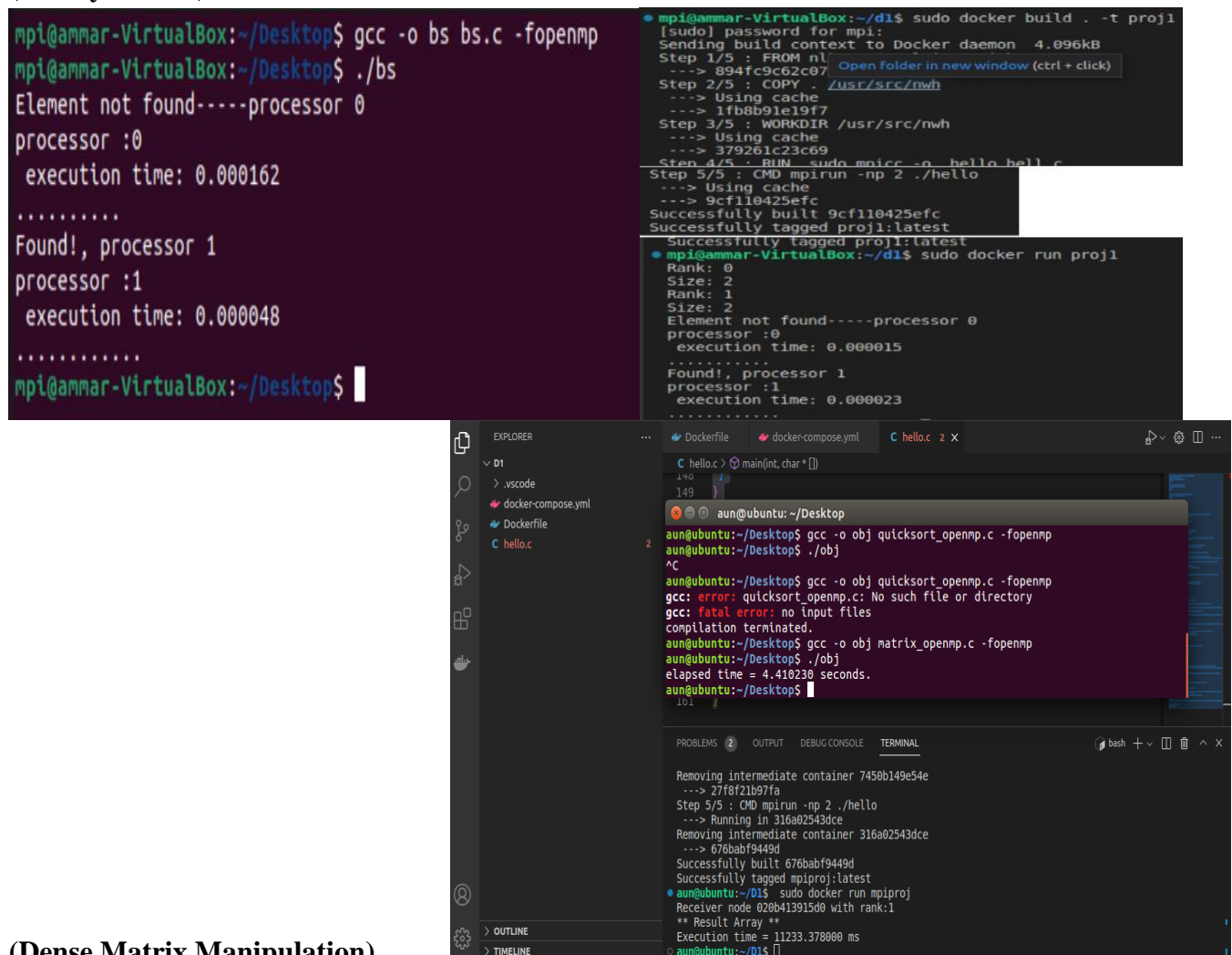


```
Dockerfile
1 FROM niknuyen/alpine-mpich
2
3 COPY . /app
4
5 WORKDIR /app
6
7 RUN sudo mpicc -o hello hello.c
8
9 CMD mpirun -np 2 ./hello
10
11 # sudo docker run mpiproj
12
13 # sudo docker build -t mpiproj .
```

```
---> Using cache
---> 87a1ebf0a438
Successfully built 87a1ebf0a438
Successfully tagged mpiproj:latest
aun@ubuntu:~/D1$ sudo docker run mpiproj
Number of Elements is 100000
Reading the array...
Total number of Elements given as input : 100000
Quicksort 100000 ints on 2 procs: 0.056410 secs
aun@ubuntu:~/D1$
```

(Quicksort)

(Binary Search)



```
mpi@ammar-VirtualBox:~/Desktop$ gcc -o bs bs.c -fopenmp
mpi@ammar-VirtualBox:~/Desktop$ ./bs
Element not found-----processor 0
processor :0
execution time: 0.000162
.....
Found!, processor 1
processor :1
execution time: 0.000048
.....
mpi@ammar-VirtualBox:~/Desktop$
```

```
mpi@ammar-VirtualBox:~/D1$ sudo docker build . -t proj1
[sudo] password for mpi:
Sending build context to Docker daemon 4.096KB
Step 1/5 : FROM mpi
---> 894fc9c62c07
Step 2/5 : COPY . /usr/src/nwh
---> Using cache
---> 1fb8b91e19f7
Step 3/5 : WORKDIR /usr/src/nwh
---> Using cache
---> 379261c23c69
Step 4/5 : RUN sudo mpicc -o hello hello.c
---> Using cache
---> 9cf110425efc
Successfully built 9cf110425efc
Successfully tagged proj1:latest
mpi@ammar-VirtualBox:~/D1$ sudo docker run proj1
Rank: 0
Size: 2
Rank: 1
Size: 2
Element not found-----processor 0
processor :0
execution time: 0.000015
.....
Found!, processor 1
processor :1
execution time: 0.000023
.....
```

```
EXPLORER
D1
.vscode
docker-compose.yml
Dockerfile
hello.c
```

```
hello.c
140 int main(int, char*)
141 {
142     // ...
143     return 0;
144 }
```

```
aun@ubuntu:~/Desktop$ gcc -o obj quicksort_openmp.c -fopenmp
aun@ubuntu:~/Desktop$ ./obj
^C
aun@ubuntu:~/Desktop$ gcc -o obj quicksort_openmp.c -fopenmp
gcc: error: quicksort_openmp.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
aun@ubuntu:~/Desktop$ gcc -o obj matrix_openmp.c -fopenmp
aun@ubuntu:~/Desktop$ ./obj
elapsed time = 4.410230 seconds.
aun@ubuntu:~/Desktop$
```

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
Removing intermediate container 7450b149e54e
---> 27f8f21b97fa
Step 5/5 : CMD mpirun -np 2 ./hello
---> Running in 316a02543dce
Removing intermediate container 316a02543dce
---> 676babf9449d
Successfully built 676babf9449d
Successfully tagged mpiproj:latest
aun@ubuntu:~/D1$ sudo docker run mpiproj
Receiver node 020b413915d0 with rank:1
** Result Array **
Execution time = 11233.378000 ms
aun@ubuntu:~/D1$
```

(Dense Matrix Manipulation)

Graphs (Quicksort):

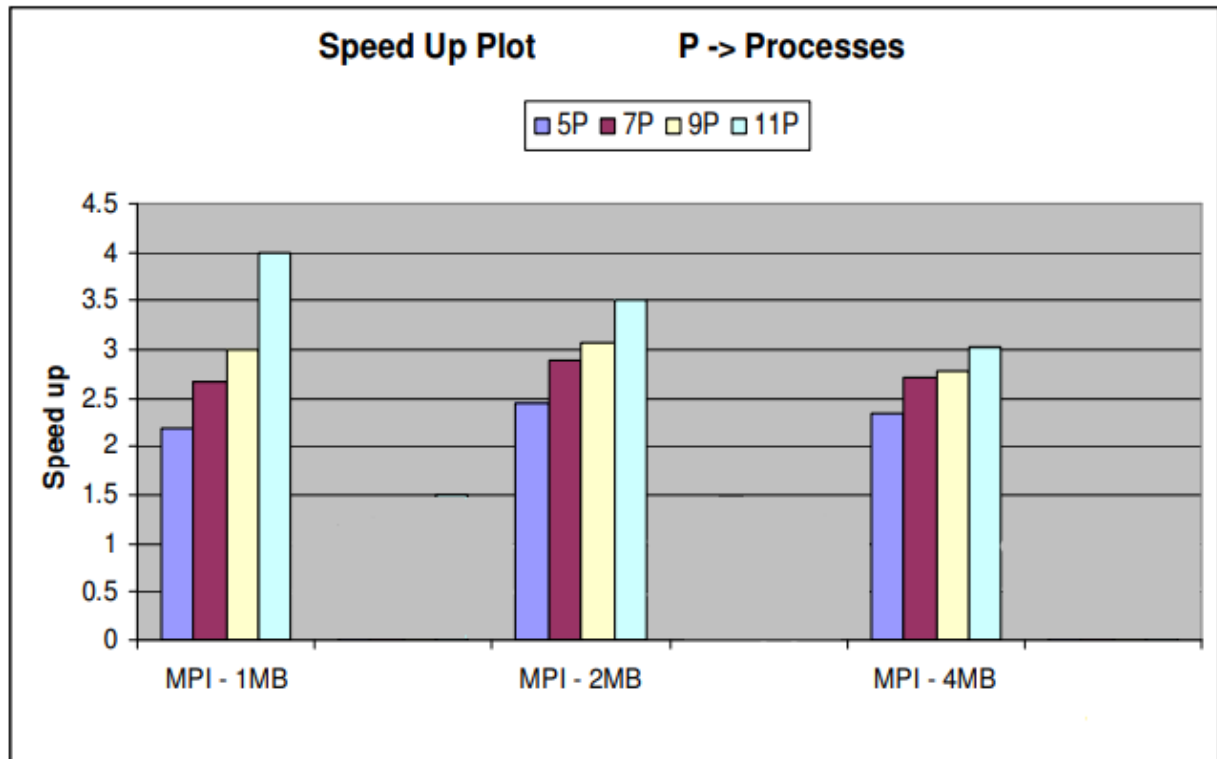


Figure 6. Speedup plot for varying problem size and varying number of processes

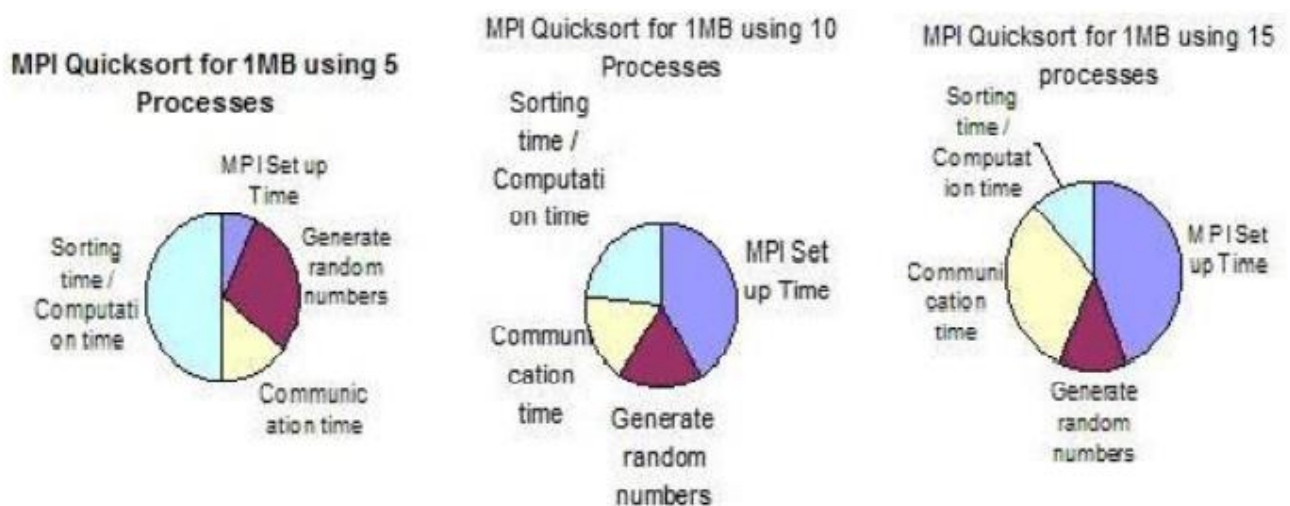


Figure 7. Plot for time spent in each phase of the program

Number of processes
5
7
9
11
13
15

MPI execution time (sec)
0.11
0.1
0.08
0.06
0.1
0.15

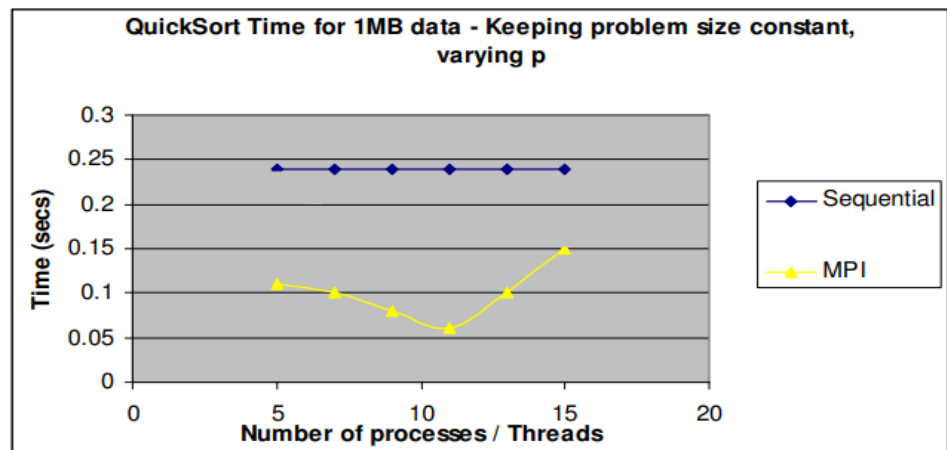


Figure 5. Experiment readings and graph for 1MB quicksort using MPI, Pthreads and Sequential code

Conclusion:

Our Docker Algorithms are accurate and efficient. They give the required output via the required manner that is parallelization of algorithms in a distributed environment along with conventional OpenMP implementation.

Thank You!