DS 221 - Introduction to Scalable Systems
Parallel Merge Sort using MPI

Shriram R.
M Tech (CDS)
06-02-01-10-51-18-1-15763

November 30, 2018

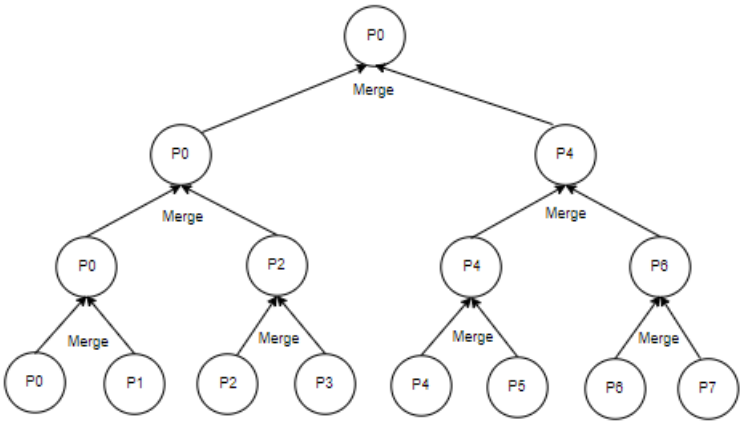
1 Introduction

Parallel Merge Sorting algorithm has been implemented using MPI and the speedup for different number of processes with respect to sequential version were observed through experiments. The following sections cover the methodology, experimental setup and results in detail.

2 Methodology

The parallel version of the algorithm performs the following steps:

1. The input data is read in from the file by process 0. File I/O has been used to read input instead of shell redirection to improve performance
2. Process 0 divides the input equally among the processes and sends the respective chunks to other processes. If input is not divisible exactly, last process would get some additional data items
3. Each process receives its chunk of data and stores it in a local array. The length of data is also captured in a variable
4. Each process performs a local quick sort on its chunk of data. Quick sort has been implemented as a seperate function in the program for this purpose.
5. The merging of data happens in a tree-like fashion as given below for the specific case of 8 processes,



The final sorted data will be available in process 0 and is written to stdout. The above methodology is illustrated in a code snippet in the appendix.

3 Experimental Setup

Experiments were run on multiple compute nodes having 8 core AMD Opteron 3380 processor in the Turing compute cluster. Batch jobs were created in the form of a shell script and executed using PBS. The no. of nodes used varied depending on the no. of processes required for the experiment.

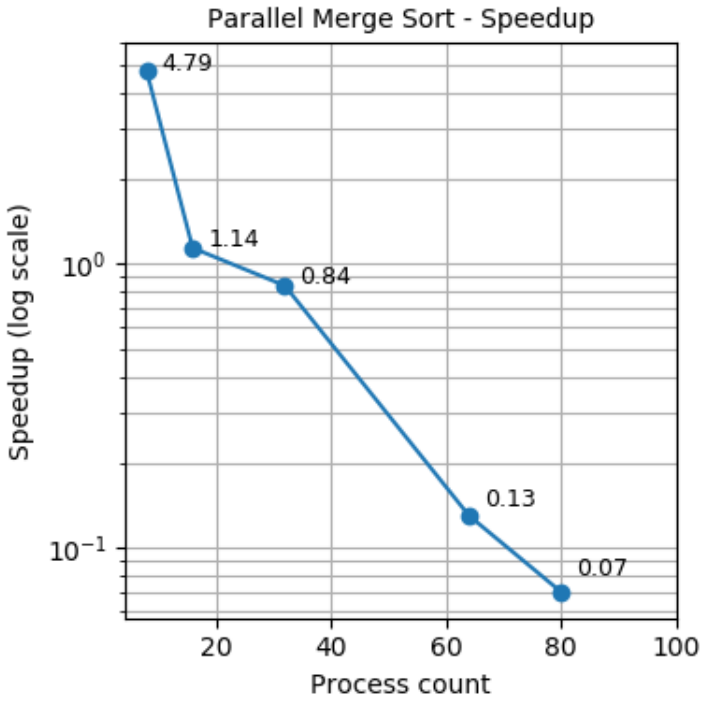
The time taken to run the parallel sort was measured using C library functions and was printed to stdout along with specified output. Each experiment was repeated 20 times consecutively and then the average was computed for the plots and analysis.

In each experiment, both sequential and parallel versions of code were executed so as to remove bias from any system load.

4 Results

The experimental results are tablulated and plotted below,

Processes	**Sequential (Avg.) (s)**	**Parallel (Avg.) (s)**	**Speedup**
8	0.0311	0.0065	4.79
16	0.0313	0.0274	1.14
32	0.0312	0.0370	0.84
64	0.0315	0.2434	0.13
80	0.0311	0.4567	0.07



5 Observations

It can be observed that the speedup is maximum for 8 processes which corresponds to 8 cores in a single node and then decreases with increase in process count.

As the process count is increased, the processes are distributed across multiple nodes. This leads to more delay in data transfer and synchronization due to communication across different nodes through the network. Therefore, the parallel version performs worser than the sequential code for higher process count.

The speedup is also heavily dependent on the CPU load in the nodes. At the time of experiments, only few nodes had a very low CPU load and all the other nodes were overutilized with high load averages close to 30. This could have contributed to decrease in speedup as well.

6 References

1. https://en.cppreference.com/w/c
2. DS 221 Course lecture notes
3. http://www.arc.ox.ac.uk/content/how-run-mpi-applications
4. https://www.open-mpi.org/faq/?category=running
5. https://www.open-mpi.org/doc/v2.1/man1/mpiexec.1.php

7 Appendix - Code

```
// Send and Receive Input
if (rank == 0) {
    for (int i = 1; i < procs; i++) {
        if (i < procs - 1)
            MPI_Send(arr + i * (SIZE / procs), SIZE / procs, MPI_INT, i, 0, comm);
        else
            MPI_Send(arr + i * (SIZE / procs), SIZE - i * (SIZE / procs), MPI_INT,
                                                                    i, 0, comm);
    }
    len = SIZE / procs;
} else {
    MPI_Recv(arr, SIZE, MPI_INT, 0, 0, comm, &status);
    MPI_Get_count(&status, MPI_INT, &len);
}

// Perform local quick sort
qusort(arr, 0, len - 1);
```

```
// Perform Merge
step = 0;
while (1 << step < procs) {
    if ((rank + 1) % (2 << step) == 1 && rank + (1 << step) < procs) {
        MPI_Recv(arr2, SIZE, MPI_INT, rank + (1 << step), step, comm, &status);
        MPI_Get_count(&status, MPI_INT, &len2);
        merge(arr, len, arr2, len2, temp);
        p = arr;
        arr = temp;
        temp = p;
        len += len2;
    } else if ((rank + 1) % (2 << step) != 1) {
        // printf("Send %d %d \n", step, rank);
        MPI_Send(arr, len, MPI_INT, rank - (1 << step), step, comm);
        break;
    }
    step++;
}

MPI_Barrier(comm);
```

————————End of Report————————