Introduction to Parallel System

HW 03

Floyd’s Shortest Path Algorithm

Submitted By

Ravisutha Sakrepatna Srinivasamurthy

1. Introduction

The HW3 is about parallel implementation of sequential Floyd's algorithm. The zip file Ravisutha_sakrepatna_srinivasamurthy.tar.gz contains

- SEQ – All files related to sequential implementation
  - Makefile
    - To generate print_graph.exe: make print_graph.exe
    - To generate make_graph.exe: make make_graph.exe
    - To generate floyd_serial.exe: make floyd_serial.exe
  - Source Codes
    - floyd_serial.c, graph.c, make_graph.c, print_graph.c, seq_floyd.c
  - Header Files
    - floyd_seq.h, graph.h, seq_floyd.h
- PARALLEL – All file related to parallel implementation
  - Makefile
    - To generate floyd.exe: make
  - Source Codes
    - main.c, my_mpi.c, parallel_floyd1.c
  - Header Files
    - floyd_parallel.h, MyMPI.h

In this assignment, Floyd's Shortest Path Algorithm is implemented by using 2D Checkerboard Decomposition. The execution time for sequential and parallel implementation is analyzed in the following sections.

2. Sequential Algorithm Implementation

In this part, the shortest path is determined by using sequential Floyd's algorithm. Figure 2.1 shows the sample output for 4 node input.

```
rsakrep@user001:~/ece473/hw3/Floyd/SEQ/WORKING
[rsakrep@user001 WORKING]$ ./print_graph.exe file.dat
          |       0        1        2        3
-----------------------------------------------------
0         |       0        21       36       89
1         |       34       0        2        -1
2         |       32       46       0        60
3         |       64       -1       75       0
[rsakrep@user001 WORKING]$
```

Figure 2.1 Input Adjacency Matrix to Floyd's Algorithm

```
rsakrep@user001:~/ece473/hw3/Floyd/SEQ/WORKING
[rsakrep@user001 WORKING]$ ./print_graph.exe file.seq
            |       0         1         2         3
-----------------------------------------------------
0           |       0         21        23        83
1           |       34        0         2         62
2           |       32        46        0         60
3           |       64        85        75        0
[rsakrep@user001 WORKING]$
```

Figure 2.2 Output of Sequential Floyd's Algorithm

3. Parallel Algorithm Implementation

In this part, we parallelize the Floyd's algorithm. The first step is to divide the matrix among the process. For this, we make use of Cartesian topology. Once we create a communicator associated with this topology, the input file.dat is read and the data is distributed among the processes. The row and column communicators are created for each row and column. Finally, the parallel Floyd's algorithm is called. In the parallel Floyd's implementation, the owner of kth row and column broadcasts the data and the processes belonging to same row and column will also call Broadcast to receive the data. Hence communication happens through Broadcast. Once all processes compute its respective part, print function is called, which prints the output back to console.

The shortest path for the same input shown in Figure 2.1 is computed using Floyd's algorithm (Figure 3.1).

```
rsakrep@user001:~/ece473/hw3/Floyd/PARALLEL/WORKING
Matrix size is 4*4
Before Floyd's algorithm
     0     21     36     89
    34      0      2     -1
    32     46      0     60
    64     -1     75      0

Time taken to compute = 0.000042
Time taken overall = 0.004122
After Floyd's algorithm
     0     21     23     83
    34      0      2     62
    32     46      0     60
    64     85     75      0

Before Floyd's algorithm
After Floyd's algorithm
Before Floyd's algorithm
After Floyd's algorithm
Before Floyd's algorithm
After Floyd's algorithm


+------------------------------------------+
| PALMETTO CLUSTER PBS RESOURCES REQUESTED |
+------------------------------------------+

mem=4gb,ncpus=16,walltime=00:10:00


+-------------------------------------+
| PALMETTO CLUSTER PBS RESOURCES USED |
+-------------------------------------+

cpupercent=0,cput=00:00:00,mem=636kb,ncpus=16,vmem=12612kb,walltime=00:00:01
```

Figure 3.1 Shortest Path computed using parallel Floyd's implementation

4. Data Collection

The data is collected for both sequential as well as parallel implementation. The following table gives the details of the data collection.

Number of processors → Grid size ↓	1 (Seq. Execution)	4	16	64
1000	22.736388	4.712171	1.160726	0.352820
2000	161.460472	49.304814	10.846890	2.320309
3000	546.204504	117.970095	32.818548	7.949958

Table 4.1 Time taken to compute shortest path

Number of processors → Grid size ↓	1 (Seq. Execution)	4	16	64
1000	22.736388	4.795285	1.185418	0.479398
2000	161.460472	49.465345	11.016309	2.528621
3000	546.204504	118.272500	33.142528	8.291699

Table 4.2 Overall Time taken

5. Analysis

- **Execution Time**

Execution time is computed in two fronts. In the first case, only the shortest path computation time is considered. In the next case, overall computation time is considered.

The table 4.1 contains only the shortest path computation time and table 4.2 contains overall execution time. The corresponding graph between number of processors and the execution time for various nodes (1000, 2000, 3000) are as shown in Figure 5.1 and 5.2.

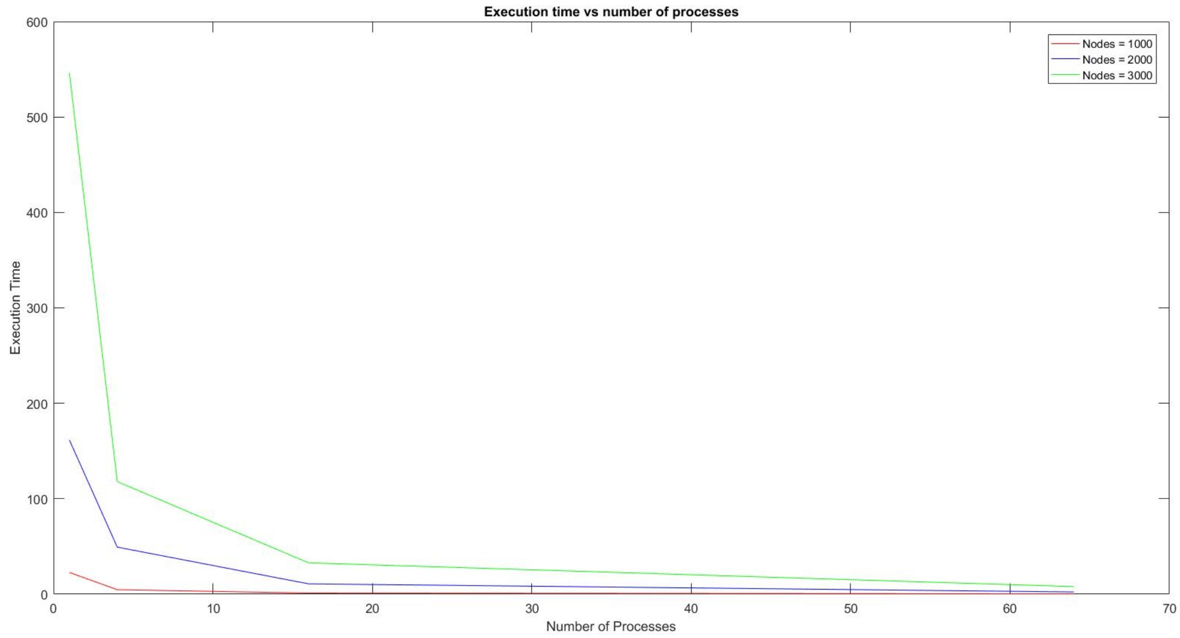

Figure 5.1 Plot of Computation time vs number of processors

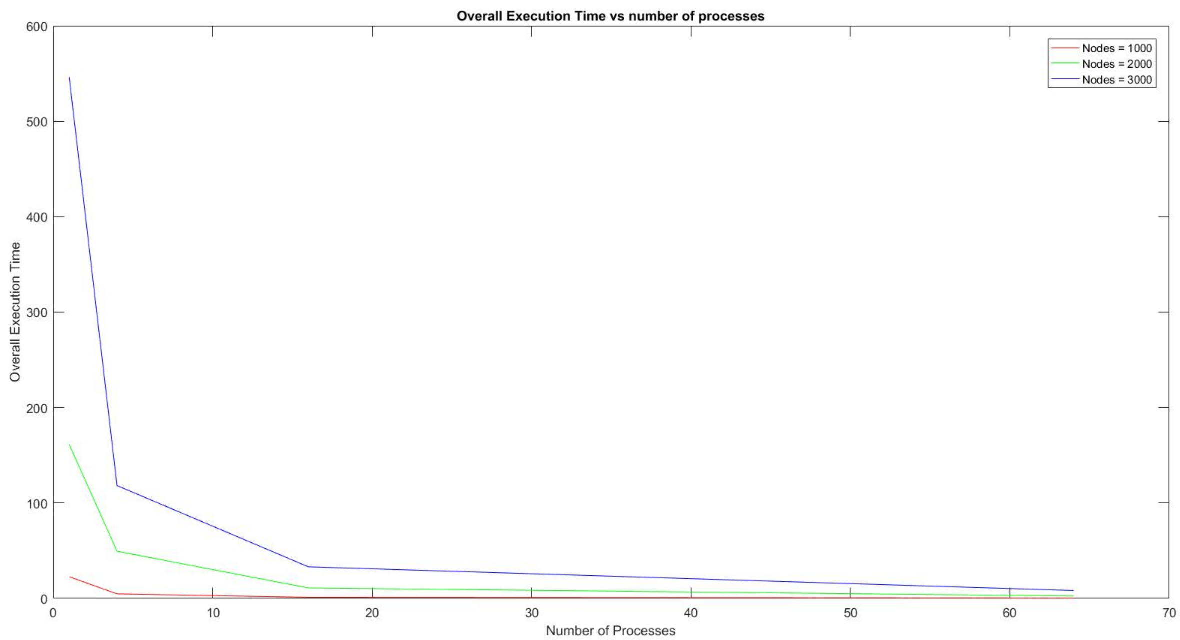

Figure 5.2 Plot of Overall computation time vs number of processors

- Observation
    - As the number of processors increases, the execution time decreases in an exponential fashion.

- **Speedup**

The formula for computing speedup is as shown below. Tables 5.1 and 5.2 contains the speedup for the corresponding execution times.

$$speedup = \frac{sequential\ execution\ time}{parallel\ execution\ time}$$

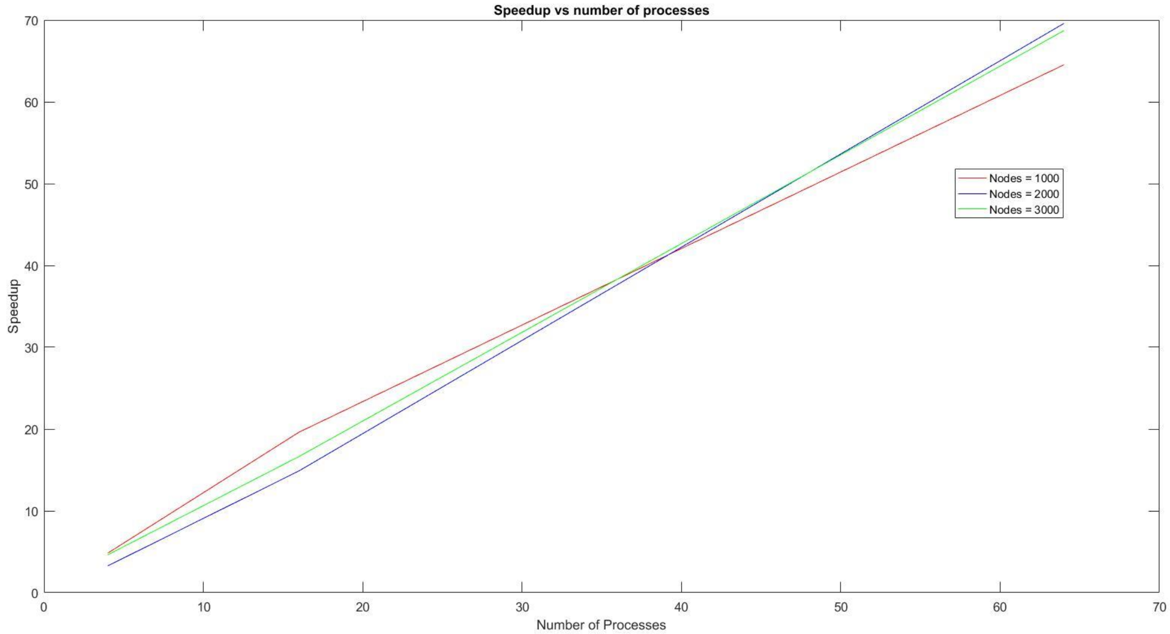

Figure 5.3 Plot of speedup vs number of processes

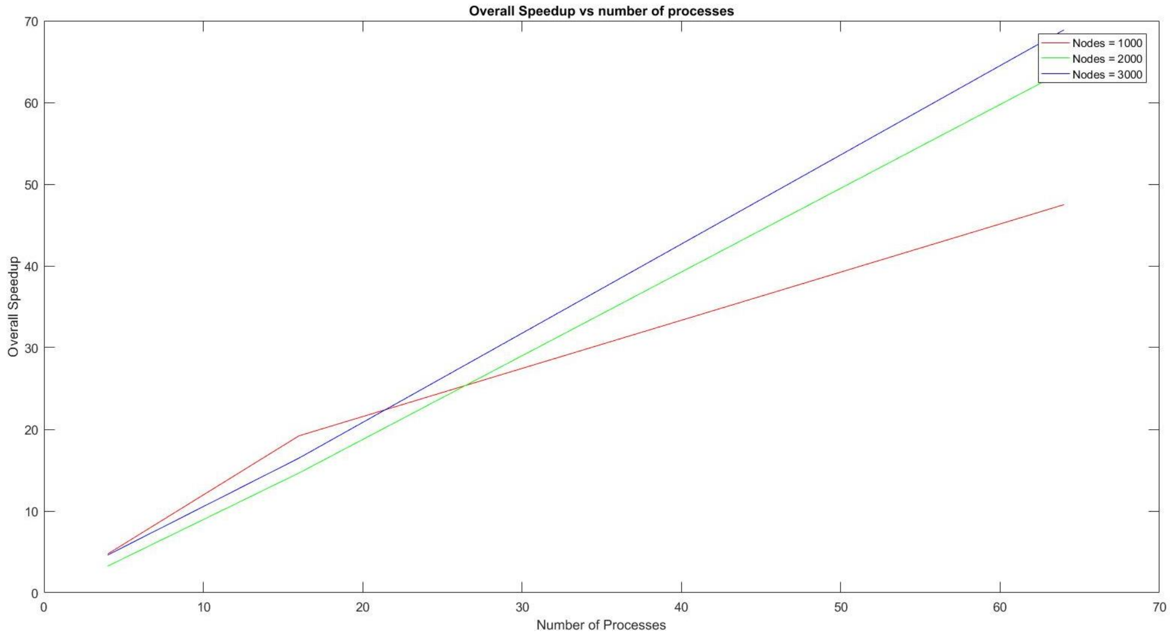

Figure 5.4 Plot of overall speedup vs number of processes

Number of processors → Grid size ↓	4	16	64
1000	4.8314	19.6144	64.5306
2000	3.27470	14.88541	69.5858
3000	4.63002	16.64316	68.7053

Table 5.1 Speedup for computation part only

Number of processors → Grid size ↓	4	16	64
1000	4.7477	19.2054	47.4896
2000	3.2641	14.6565	63.8532
3000	4.6182	16.4805	68.8737

Table 5.2 Overall Speedup

- Observation
  - As the number of processors increases, the speedup also increases.

- **Efficiency**

$$efficiency = \frac{speedup}{processors}$$

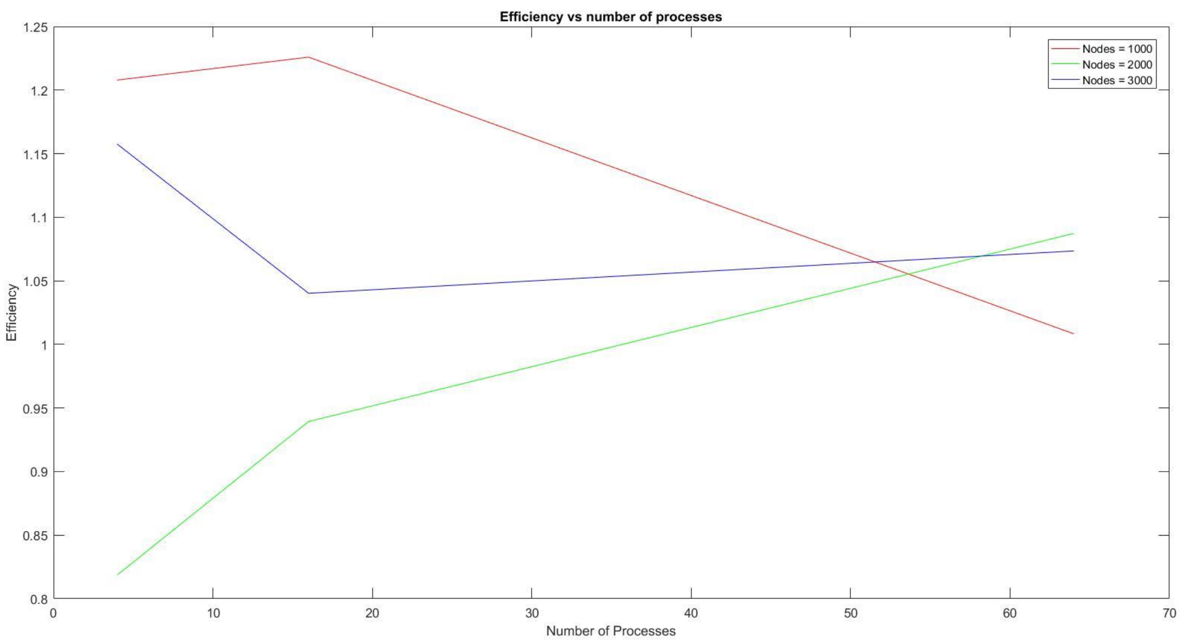

Figure 5.5 Plot of efficiency vs number of processors

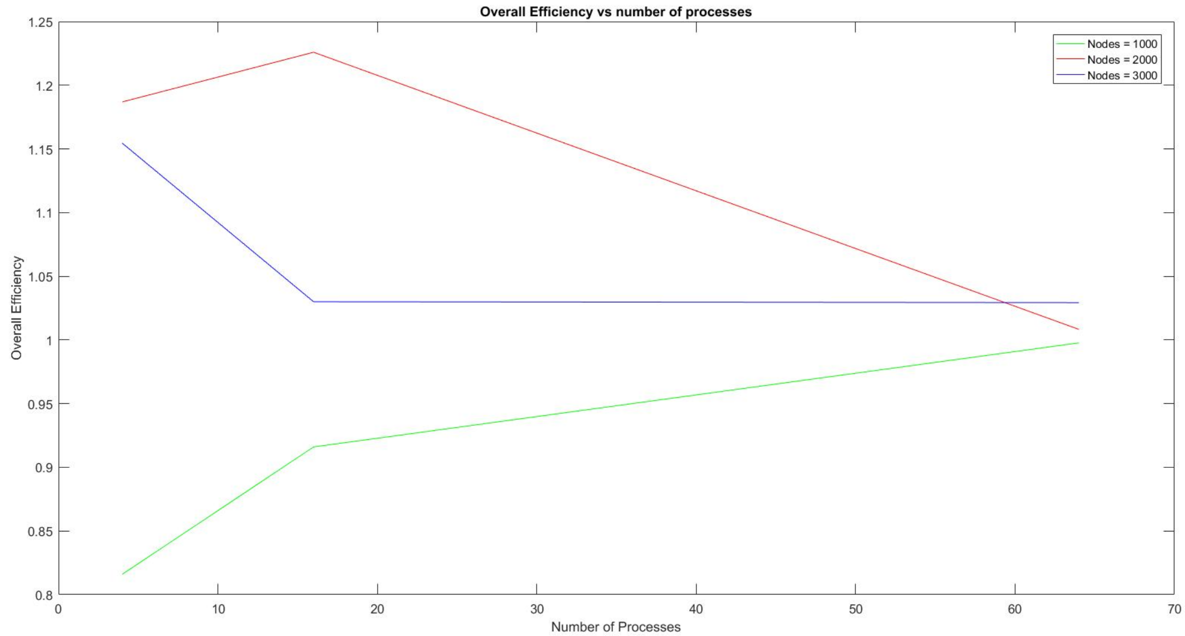

Figure 5.6 Plot of overall efficiency and number of processors

Number of processors → Grid size ↓	4	16	64
1000	1.2079	1.2259	1.0083
2000	0.81867	0.93033	1.0873
3000	1.15750	1.04019	1.0735

Table 5.3 Efficiency for computation part only

Number of processors → Grid size ↓	4	16	64
1000	1.1869	1.2003	0.7420
2000	0.8160	0.9160	0.9977
3000	1.1545	1.0300	1.0293

Table 5.3 Overall Efficiency

- Observation
  - Efficiency initially reduces as number of processors increases.
  - But later, efficiency increases as the number of nodes increases (observe row 2000 and 3000). This is because the overhead caused due to communication will fade out as the number of node increases.

Compiling and observing the output

Compiling

- Makefile: SEQ
    - To generate print_graph.exe:
        - make print_graph.exe
    - To generate make_graph.exe:
        - make make_graph.exe
    - To generate floyd_serial.exe:
        - make floyd_serial.exe
- Makefile: PARALLEL
    - To generate floyd_parallel.exe:
        - Make

Executing

- SEQ
    - ./print_graph.exe file.dat
    - ./make_graph.exe -n 100 -r 100 -p 150 -o file.dat
    - ./floyd_serial.exe file.dat file.seq or qsub rsakrep.hw.pbs
- PARALLEL
    - qsub rsakrep.hw.pbs