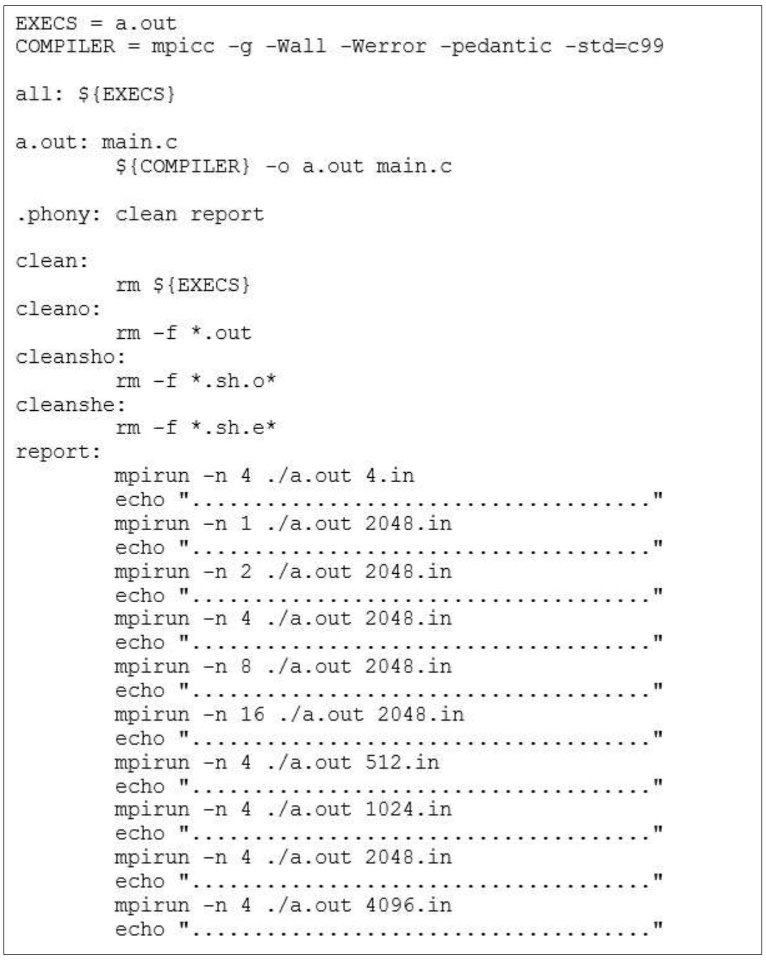
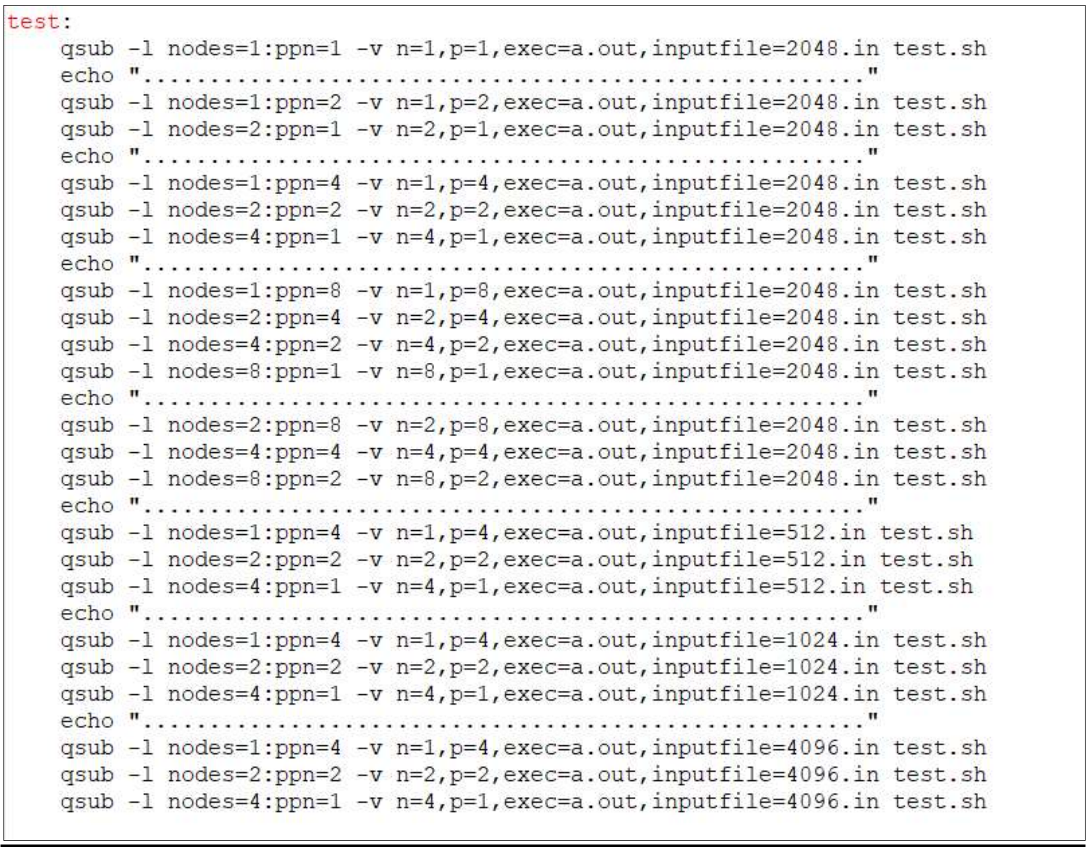
How to compile and run:

1. Open Terminal
2. Compile with command $mpicc -std=c99 finalfloyd.c or make
   a. make report



```
EXECS = a.out
COMPILER = mpicc -g -Wall -Werror -pedantic -std=c99

all: ${EXECS}

a.out: main.c
        ${COMPILER} -o a.out main.c

.phony: clean report

clean:
        rm ${EXECS}
cleano:
        rm -f *.out
cleansho:
        rm -f *.sh.o*
cleanshe:
        rm -f *.sh.e*
report:
        mpirun -n 4 ./a.out 4.in
        echo "....................................."
        mpirun -n 1 ./a.out 2048.in
        echo "....................................."
        mpirun -n 2 ./a.out 2048.in
        echo "....................................."
        mpirun -n 4 ./a.out 2048.in
        echo "....................................."
        mpirun -n 8 ./a.out 2048.in
        echo "....................................."
        mpirun -n 16 ./a.out 2048.in
        echo "....................................."
        mpirun -n 4 ./a.out 512.in
        echo "....................................."
        mpirun -n 4 ./a.out 1024.in
        echo "....................................."
        mpirun -n 4 ./a.out 2048.in
        echo "....................................."
        mpirun -n 4 ./a.out 4096.in
        echo "....................................."
```

   b. make test



```
test:
    qsub -l nodes=1:ppn=1 -v n=1,p=1,exec=a.out,inputfile=2048.in test.sh
    echo "....................................................."
    qsub -l nodes=1:ppn=2 -v n=1,p=2,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=2:ppn=1 -v n=2,p=1,exec=a.out,inputfile=2048.in test.sh
    echo "....................................................."
    qsub -l nodes=1:ppn=4 -v n=1,p=4,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=2:ppn=2 -v n=2,p=2,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=4:ppn=1 -v n=4,p=1,exec=a.out,inputfile=2048.in test.sh
    echo "....................................................."
    qsub -l nodes=1:ppn=8 -v n=1,p=8,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=2:ppn=4 -v n=2,p=4,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=4:ppn=2 -v n=4,p=2,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=8:ppn=1 -v n=8,p=1,exec=a.out,inputfile=2048.in test.sh
    echo "....................................................."
    qsub -l nodes=2:ppn=8 -v n=2,p=8,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=4:ppn=4 -v n=4,p=4,exec=a.out,inputfile=2048.in test.sh
    qsub -l nodes=8:ppn=2 -v n=8,p=2,exec=a.out,inputfile=2048.in test.sh
    echo "....................................................."
    qsub -l nodes=1:ppn=4 -v n=1,p=4,exec=a.out,inputfile=512.in test.sh
    qsub -l nodes=2:ppn=2 -v n=2,p=2,exec=a.out,inputfile=512.in test.sh
    qsub -l nodes=4:ppn=1 -v n=4,p=1,exec=a.out,inputfile=512.in test.sh
    echo "....................................................."
    qsub -l nodes=1:ppn=4 -v n=1,p=4,exec=a.out,inputfile=1024.in test.sh
    qsub -l nodes=2:ppn=2 -v n=2,p=2,exec=a.out,inputfile=1024.in test.sh
    qsub -l nodes=4:ppn=1 -v n=4,p=1,exec=a.out,inputfile=1024.in test.sh
    echo "....................................................."
    qsub -l nodes=1:ppn=4 -v n=1,p=4,exec=a.out,inputfile=4096.in test.sh
    qsub -l nodes=2:ppn=2 -v n=2,p=2,exec=a.out,inputfile=4096.in test.sh
    qsub -l nodes=4:ppn=1 -v n=4,p=1,exec=a.out,inputfile=4096.in test.sh
```

README

Included in the zipped files submitted, we have

- finalfloyd.c (mpi program)
- makefile (containing all the commands to run finalfloyd.c in the cluster using qsub automatically)
- test.sh (is required while using the makefile in the cluster)
- Final report (this report)

N.B. - .out files will only be generated using mpirun -n <no of processors> ./a.out <.in file>. Hence, we decided to print the final results out in the qsub generated files (.o<job_id>).

Introduction

The goal of this project is to implement parallel algorithms to solve the all-pairs-shortest path problem for large graphs. Specifically, we used the Floyd Warshall algorithm to solve the all-pairs-shortest path problem while using the C programming language with the Message Passing Interface (MPI) to write parallelized code. We have also reported our observations on how speedup is affected by changing the size of the graph (512 to 4096 vertices) and the total number of processors used (1 to 16). This is shown in **test #1 –** Speed up for graphs of **2048 vertices** using 1, 2, 4, 8, 16 processors and **test #2** – Speed up for 4 processors against graphs of 256, 512, 1024, 2048, 4096 vertices.

Approach – How Data is Partitioned

We initially read the file and store the matrix elements (excluding the first element which is the number of vertices – nv) in a one-dimensional array. The one-dimensional array is then divided among the number of processors (np) specified. For example, if we have four vertices (i.e. 4 rows and 16 elements) and two processors, two rows are allocated to each processor.

We then broadcast the number of vertices from the root processor to all other processors. Then, using MPI scatter function, we scattered the partitioned data (referred to as 'block of rows' = number of vertices/number of processors) to all other processors.

For example, if we have 4 vertices (16 elements) and 2 processors, a block of rows will contain 8 elements ((nv*nv)/np). Each processor will get a different block of rows. Now, all data has been partitioned among all processors and each process contains their own data.

Approach - How Data is Processed

Using the Floyd Warshall algorithm, we performed three for-loops to calculate the all-pairs-shortest path. We did not perform parallelism for the outermost loop (denoted by k) and instead, implemented parallelism only for the inner two loops (denoted by i and j).

In every iteration of a k, the common row (i-th row == iteration k) which will be different every time, is then broadcasted to all other processes. Now, all processors have their own data and the common row data to perform the Floyd Warshall algorithm for one iteration. Hence, the load is balanced on every processor and they are doing productive work equally.

Now, we repeat the above step nv times. Eventually, the program generates the all-pairs-shortest path matrix.

Test #1 - Speed up for graphs of 2048 vertices using 1, 2, 4, 8, 16 processors

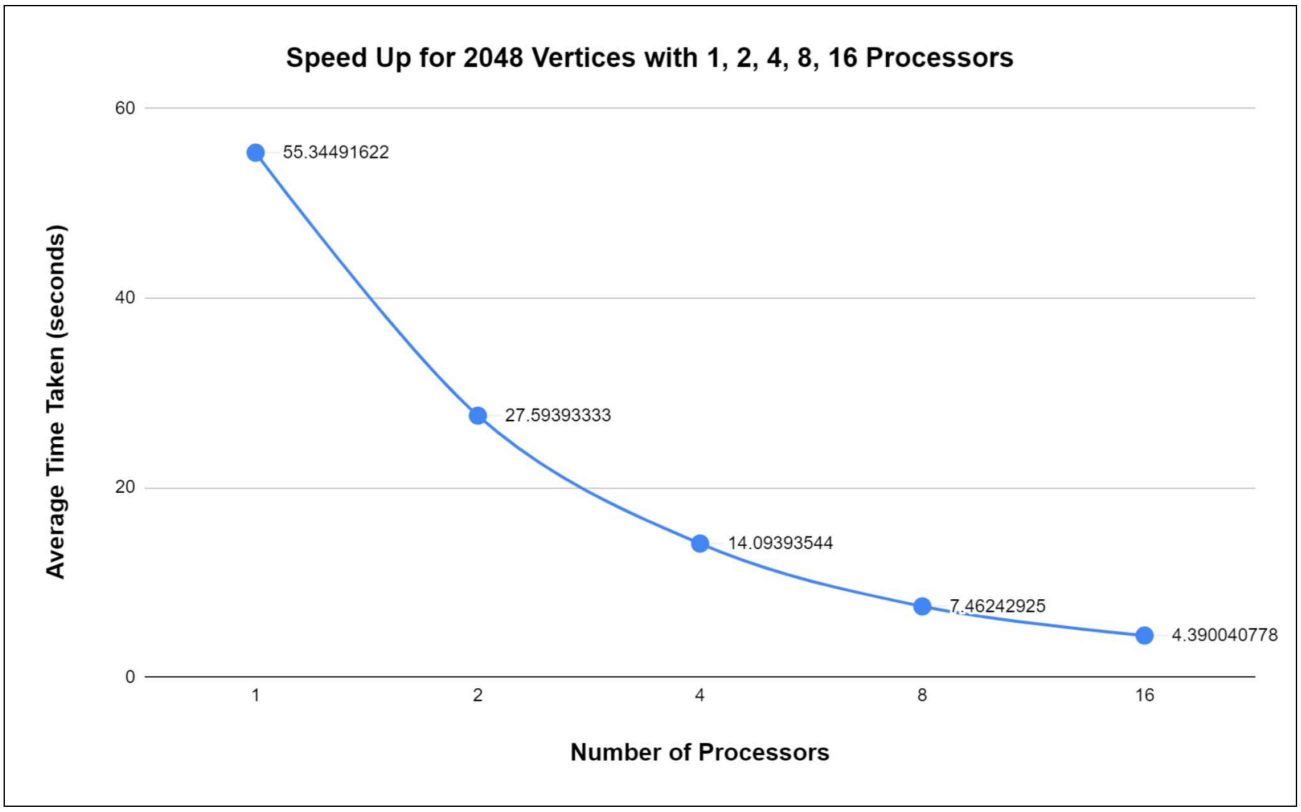
Our program was tested on various combinations of different nodes and processors per node (nodes:ppn). We took the average of 3 tests to check the accuracy of the results.

Test Summary

2048 vertices using 1, 2, 4, 8, 16 processors						
Processors	**nodes:ppn**	**Time taken 1 (seconds)**	**Time taken 2 (seconds)**	**Time taken 3 (seconds)**	**Average (seconds)**	**Total Average (seconds)**
1	1:1	55.031041	55.674722	55.279095	55.328286	55.328286
2	1:2	27.841553	27.799805	27.779714	27.807024	
	2:1	27.290658	27.288379	27.509382	27.36280633	27.59393333
4	1:4	13.785299	14.104659	13.76991	13.88662267	
	4:1	13.966101	13.92319	14.571509	14.1536	
	2:2	14.100609	14.31938	14.304762	14.24158367	14.09393544
8	1:8	7.209876	7.239678	7.204012	7.217855333	
	8:1	7.254781	7.410303	7.303688	7.322924	
	2:4	7.5777	7.575457	7.574447	7.575868	
	4:2	7.742146	7.634385	7.822678	7.733069667	7.46242925
16	2:8	4.363225	4.36349	4.368586	4.365100333	
	8:2	4.312155	4.324376	4.257564	4.298031667	
	4:4	4.473234	4.436393	4.611344	4.506990333	4.390040778

Our Findings and Conclusion

As shown above in the table above and in the graph below, as the number of processors increases, the average time taken to compute Floyd Warshall Shortest path decreases. By implementing parallelism and dividing the work up to multiple processors, we have successfully achieved speed up.


Speed Up for 2048 Vertices with 1, 2, 4, 8, 16 Processors
Average Time Taken (seconds)
Number of Processors