High Performance Computing Project 2

All Pairs Shortest Path using MPI

Ethan Chin : 22248878 Tomas Mijat : 21721589

Project GitHub : https://github.com/BolognesеBandit/HPCassignment2

Introduction

The project laid out was to utilize Message Passing Interface (MPI) communication protocol, to calculate the all pair shortest path matrix for any given input matrix of a standardised form. Using MPI functions, the aim was to increase the efficiency of the algorithm, by exploiting potential parallelism.

Implementation

To produce the all pair shortest path of a given graph, the team used Dijkstra's shortest path algorithm. This was selected, as it was easy to see the potential for independent processes to different parts of the all pairs shortest path matrix minimizing the communication each node must complete within the communicator. All processes are given the input matrix (root matrix), then allocates each process to perform shortest path sequentially on different sets of vertices. Each process solves one vertex at a time independently of others.

The elements from the root matrix are divided among the processes evenly.
1. Distance matrix is initialized to hold the shortest paths
2. Visited matrix is initialized to hold which vertices have been visited
3. The minimum distance on the vertex (row) that has not been visited is found
4. Every distance value on the vertex is iterated through and updated
5. Steps 3,4 are repeated for the set of vertices that are assigned to the process

To test the efficiency of MPI communications, we implemented 3 different communication patterns. These implementations are single process read/write, independent read/write and collective read/write.

Single Process Reads/Writes File

In this implementation, the root process reads the file and broadcasts the dimensions of the matrix and the data of the input matrix to all other processes. This can be slow as there is lots of communication occurring between the nodes with large chunks of data being delivered. The file is being read sequentially while the other nodes are idle and waiting for the root to finish processing the data and broadcast all of the information to each process before computations to determine the all pair shortest matrix can begin. This is inefficient as most processes sit idly waiting for details to be broadcast.
Additionally, when writing this means that all data must then be gathered back to a singular process (root process), which then writes all of the data at the same time. Again, this is a waste of all other processes computational power, as they do not contribute to the writing of data to the file, and sit idle, waiting for the program to terminate.

All Processes Read/Write - Independently

This aims to reduce the idle time of processes, by allowing each process to read a segment of the input file, utilizing MPI's 'File_read_at' function. Once read data is exchanged using the MPI collective communication function 'GatherAll'. The idea of this function is to collect data segments from each process to a main process, then broadcast the whole data to each process, to ensure that each process shares the information. This can be seen as a gather functions, which is then broadcast to all processes in the communicator, however MPI handles the heavy lifting to increase speed and efficiency.

Once all processes have this data, they can then process the graph and produce the result. The final step is, similar to the reading phase, each process writes a segment of data to the output file. This reduces the idle time of each process as each process contributes equally to the writing of the output file, rather than letting one process do all of the work.

All Processes Read/Write - Collectively

The idea behind this implementation is to reduce the number of I/O calls made to files, as this is one of the most time consuming processes. The way that the MPI speeds up the process, is rather than each process call happening independent of each other, MPI collects the requests into bigger write requests and processes these larger requests as a single request. This utilizes the MPI_File_read_at_all and MPI_File_write_at_all functions, to specify which segment the process would like to write to. Once again, MPI handles the 'heavy lifting' of the functions, however the motivation is to reduce the time each process is suspended while waiting for their I/O request to become available.

Testing & Results

Dijkstra's Algorithm in Parallel

The results below show that as the number of processes increase, the time of performing the Dijkstra's all-shortest paths algorithm takes less time. This performance increase is due to the number of compute nodes working in parallel to compute the results. There is no communication between the nodes during this time.

Processes	1	2	4	8	16
For File: 2048.in	112.160042	58.291113	29.898026	14.223249	7.921596
		58.375163	29.998674	14.559669	7.654571
			30.030882	14.318787	8.084132
			29.921147	14.429701	7.773776
				14.435858	7.894236
				14.2773	7.844755
				14.398825	7.953185
				14.346748	8.033285
					7.68737
					7.962904
					7.914159
					7.663845
					7.987526
					7.820603
					7.937479
					7.906144
Max	112.160042	58.375163	30.030882	14.559669	8.084132

Figure 1.1 - Table showing the total execution time of the shortest paths operation.

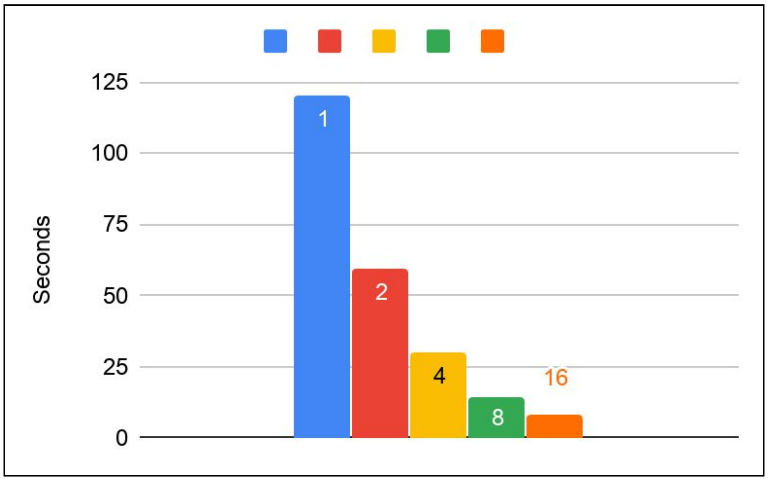


Figure 1.2 - Graph showing speed up as number of processes increase, from Figure 1.1 data.

Single Process	256	512	1024	2048	4096
For 1 Process	0.187944	1.409998	12.552498	110.764301	962.357609

Figure 2- Single Processor execution times on larger graphs

Graph Size	256	512	1024	2048	4096
For 4 processes	0.050498	0.345792	2.917082	26.706075	270.833256
	0.06296	0.336516	3.237163	26.799712	276.105319
	0.058494	0.34746	3.240987	26.781905	275.460425
	0.057602	0.33793	3.255726	26.783608	275.822867
Max	0.06296	0.34746	3.255726	26.799712	276.105319

Figure 3 - Quad processes on larger graphs

The above tables (Figure 2 & 3) show that there is significant speed-up of using 4 nodes in parallel than opposed to one. This is due to the increase of the number of compute nodes that are being utilised to perform the algorithm in parallel.

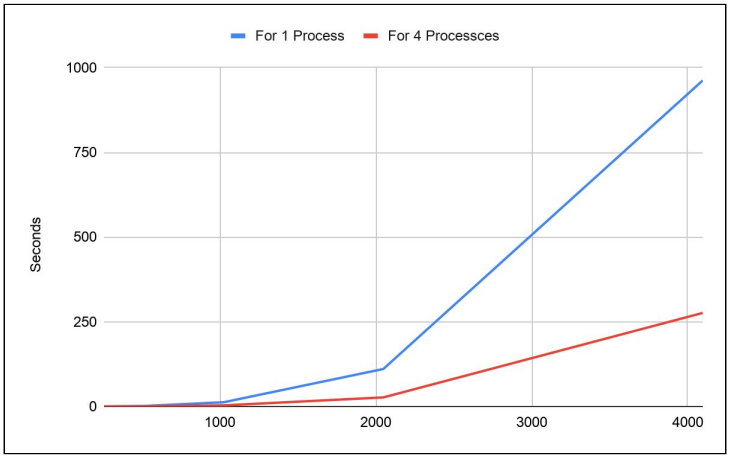
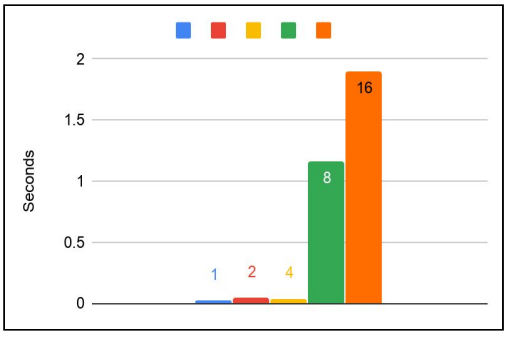


Figure 5 - Single Process vs Quad Processes w.r.t input graph size.

File Read

Figure 6&7 - Independent read time per node & for number of nodes, respectively (below)



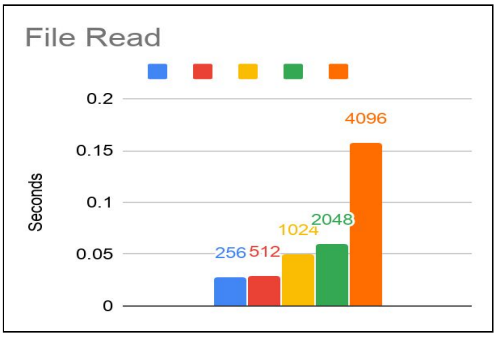
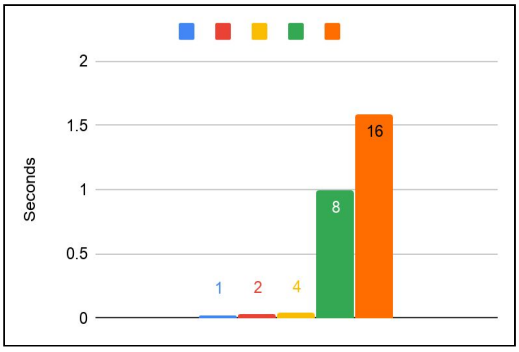
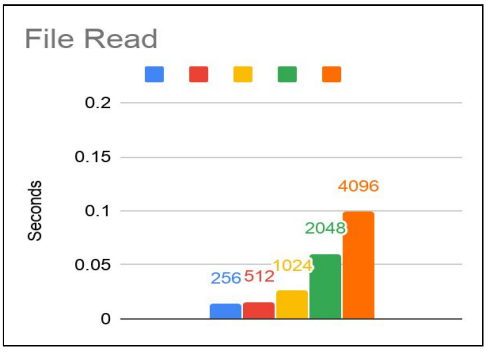


Figure 8&9 - Collective read time per node & for number of nodes, respectively (below)





The above graphs (Figures 6 - 9) shows that as the number of processes or the size of the file increases, the independent file reading is slower than the collective read.

File Write

Figure 10 - Independent write time w.r.t processes

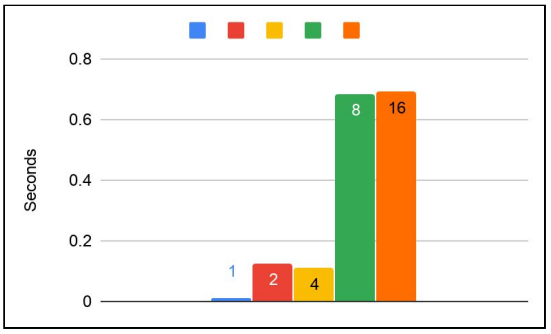
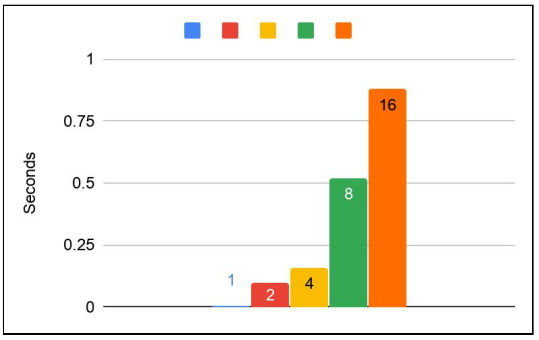


Figure 11 - Collective read time w.r.t processes



File Write	256	512	1024	2048	4096
	0.005506	0.256144	1.012714	0.00498	91.636684
	0.033426	0.036883	1.045689	5.179452	0.146203
	0.023597	0.281974	0.965305	5.319746	90.692288
	0.031042	0.307881	0.044594	5.311	90.962081
Max	0.033426	0.307881	1.045689	5.319746	91.636684

Figure 12 - Independent File write time w.r.t size of input graph

File Write	256	512	1024	2048	4096
	0.000727	0.026285	1.296671	0.007697	0.033428
	0.002203	0.008365	1.237099	0.151337	56.17883
	0.007844	0.001756	0.000995	0.093195	55.925186
	0.000747	0.007133	1.205247	0.265461	54.929533
Max	0.007844	0.026285	1.296671	0.265461	56.17883

Figure 13 - Collective File write time w.r.t size of input graph

The above data (Figures 10-13) shows that as the size of the file increases, collective write will take less time than using independent writing. However, as the number of processes increases, the time to write will increase. This may be due to an increase of processes waiting for other processes to finish writing.

Discussion & Analysis

From the analysis above we are able to see the difference in execution time with both increasing number of processes used, and collective I/O operations , relative to independent I/O. This is especially prevalent when analysing larger graphs, as the time spent waiting by a given process, for a different process to access a file and read/write data. This reduces the overall time taken to run the entire program, however this does not decrease the time for the shortest path phase of the program. This is because changing the speed of I/O will not increase the throughput of Dijkstra's algorithm, rather, the increase I/O speed allows to algorithm to begin faster, and then how quickly data is outputted once the shortest path matrix has been produced.

However, overall execution time of the Shortest Path algorithm was decreased with increasing number of processes, for a given graph input size. This also makes sense, as the large number of processes, the smaller each allocation of nodes to each process, meaning each process has less work to do to complete their section. In this case, overall time refers to the largest time value for each process, however the computational time (sum of each processes execution time) remains relatively constant or even increases, in comparison to a lower number of processes. An explanation for the increase in total computational time, may be, that as the workload is split, the initial overhead of allocating memory and copying data into memory, now must occur on each process.

Conclusion

In conclusion, the program produced increased performance when utilizing collective I/O functionality , provided by MPI. This improvement in speed, is easily observed as the size of the input graph increases. Additionally, the overall execution time (from program start to program termination) increases with larger number of processes being made available to the program, this increases the amount of overhead work that must be completed when considering all processes.

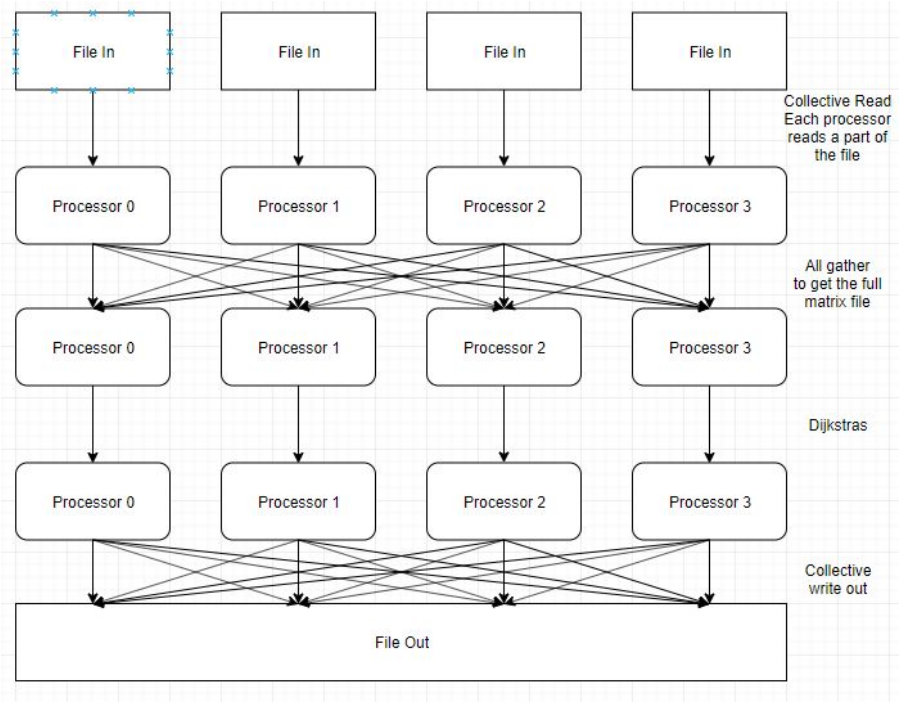


Figure 14 - Diagram of communications with collective read/write

References

- Parallel Graph Algorithms, Mikael Rännar after material by Robert Granat & Isak Jonsson och Erik Elmroth; https://www8.cs.umu.se/kurser/5DV050/VT10/handouts/F10.pdf
- https://www.codingame.com/playgrounds/349/introduction-to-mpi/broadcasting
- https://www.codingame.com/playgrounds/349/introduction-to-mpi/scattering-and-gathering---exercise
- https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
- http://teaching.csse.uwa.edu.au/units/CITS3402/lectures/3Collective.pdf

Documents:

- Time analysis of the implementation
- Diagram of the parallelism

Terminology

Node - each computer

Process(es) - each separate execution thread

Vertices - Graph node

Edge - Graph line (connection)