CITS3402: Shortest Paths

Parallelised Floyd Warshall

Nicholas Cannon 22241579

Paul O'Sullivan 21492328

Unit: High Performance Computing

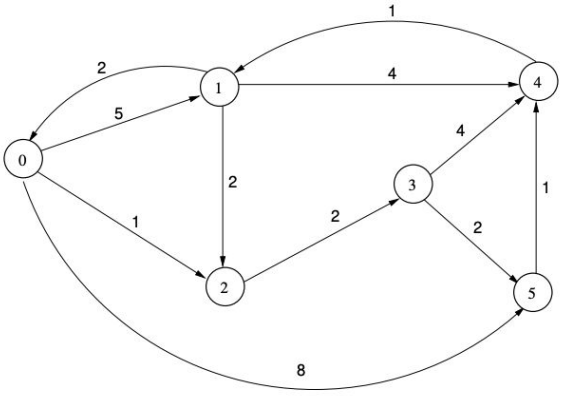
Unit Code: CITS3402

Due Date: 25/10/2019

Word Count: 1152

All Pairs Shortest Path

The all-pairs shortest path problem asks for the shortest path from every possible source to every possible destination in a graph. In other words, the shortest path between every pair of vertices in the graph. We can imagine a graph as a collection of locations, for example, churches within a single city where each church represents a node. For this problem, we assume that the graph is directed and contains no negative edge cycles. Each edge in this graph is weighted and represents the distance or cost between the two nodes. Nodes with no path between them have a weight of infinity or an arbitrarily large number. Any paths from a node to itself are removed and replaced with a distance of 0.



	0	1	2	3	4	5
0	0	5	1	∞	∞	8
1	2	0	2	∞	4	∞
2	∞	∞	0	2	∞	∞
3	∞	∞	∞	0	4	2
4	∞	1	∞	∞	0	∞
5	∞	∞	∞	∞	1	0

Floyd Warshall Algorithm

A number of algorithms such as Dijkstra's, Johnson and Floyd Warshall's all pairs, shortest path algorithms solve. Floyd Warshall works on graphs with both positive and negative edge weights (but no negative edge cycles). Floyd Warshall's algorithm was implemented for the purpose of this report. The algorithm consists of a triple nested loop and operates with time complexity of $O(v^3)$ and a space complexity of $O(v^2)$ where V represents the number of vertices in our graph. The algorithm operates by considering the distance between nodes as shown in the above adjacency matrix ($G[i, j]$) and tests to see if there exists a shorter distance between nodes through an intermediary node k such that the distance between $G[i, j] > G[i, k] + G[k, j]$. For example, the distance between nodes 3 and 4 in the above graph ($G[3, 4]$) is 4. However, the intermediary node 5 allows for a shorter distance to node 4 because $G[3, 4] > G[3, 5] + G[5, 4]$ and therefore $G[3, 4] = G[3, 5] + G[5, 4] = 3$.

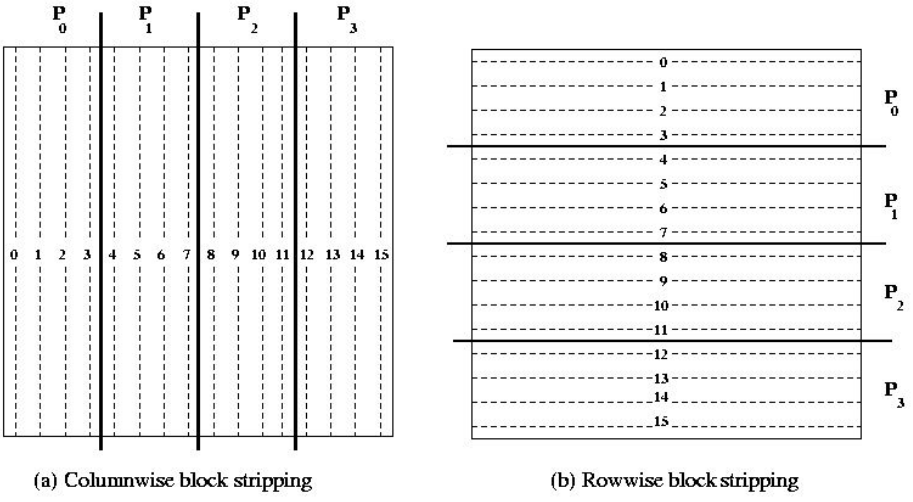
Implementation with MPI

Partitioning

Partitioning is required to separate work amongst processes and is the first step after reading in the matrix. Work can be partitioned in two ways, the first is domain decomposition in which the problem space is divided and worked on separately and the second, functional decomposition divides a problem into smaller subproblems which can be worked on independently. For the case of partitioning Floyd Warshall, we decided to use domain decomposition. Domain decomposition is incorporated because the algorithm essentially assigns a single portion of the matrix to a single process. A task can be made up for each of the matrix elements A[i, j] and can then be updated it's independently with the shortest path. It's not always the case that we have V^2 processors and hence agglomeration is required.

Agglomeration

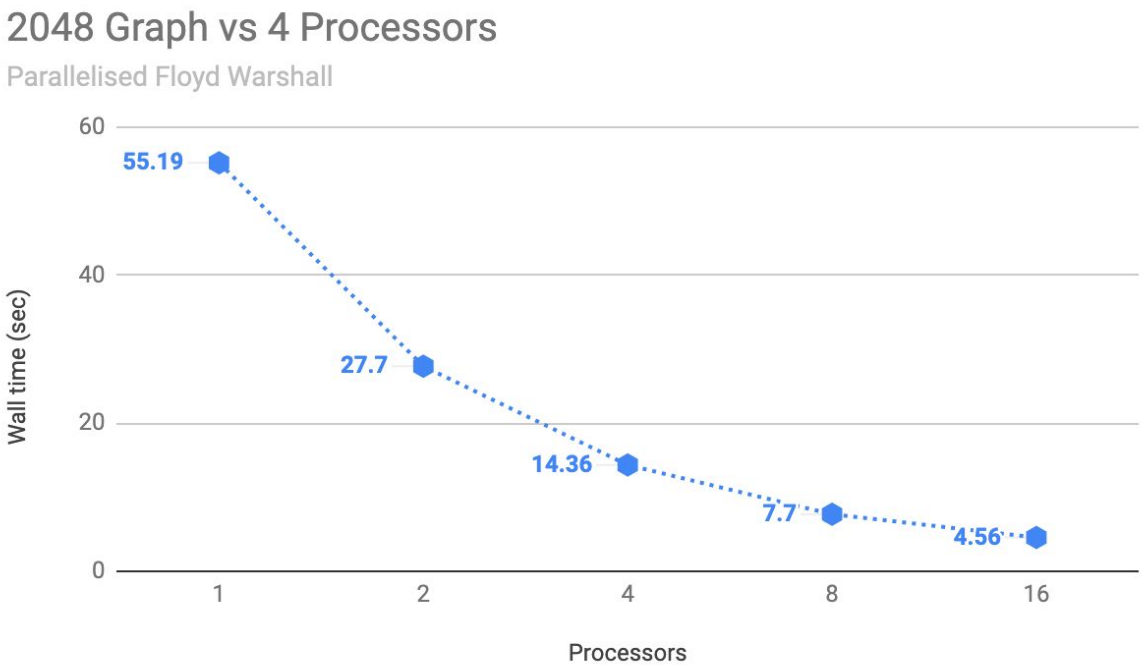
Agglomeration multiple tasks for each processor which will allow us to reduce communication costs. Multiple tasks will be created for each physical processor and one or more matrix rows will be assigned to it, thus becoming an MPI process. There are a number of ways to agglomerate tasks. Agglomeration through adjacent rows, columns or sub-matrices are all acceptable as seen below. We have chosen to agglomerate these tasks in adjacent rows. Adjacent rows are assigned to each of our tasks, broadcasting the elements to the row of tasks is no longer required as they are stored in contiguous memory. However, we still require the operations to broadcast the elements to each column. Agglomeration helps in the instance where the number of rows does not divide evenly into the number of processes. Each task gets n/p adjacent rows if the number of rows divides evenly into the number of processes p. If the number of rows does not divide evenly into the number of processes then some tasks will receive fewer tasks than others.



(a) Columnwise block stripping (b) Rowwise block stripping

Results for 2048 vs 1, 2, 4, 8 and 16 processors

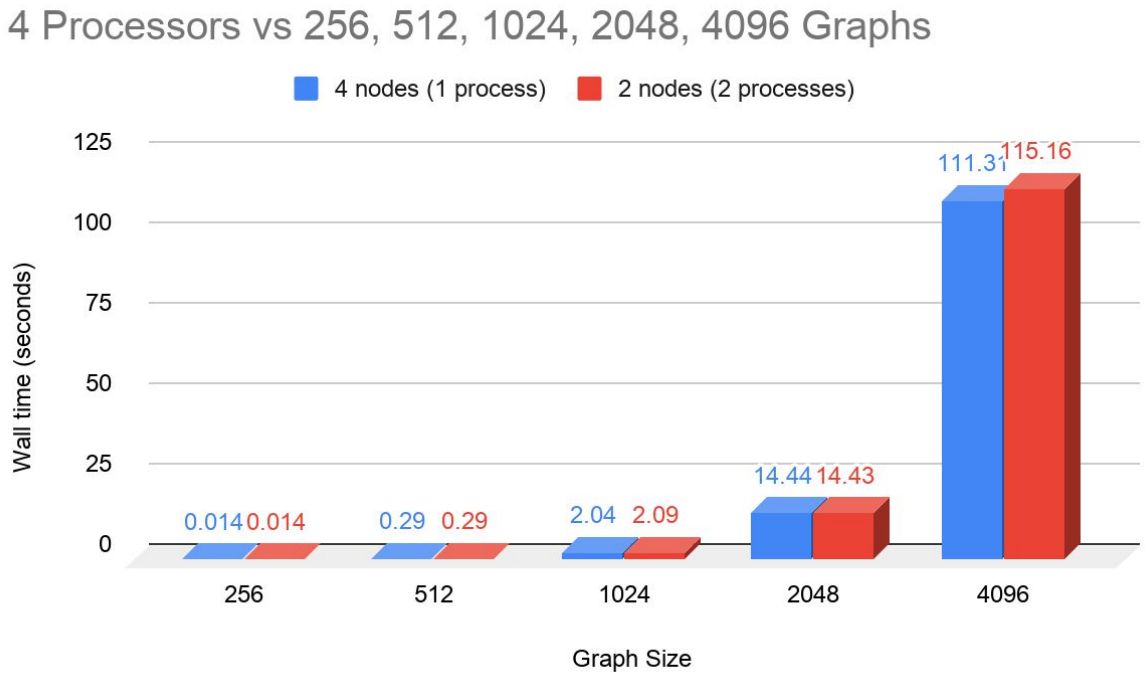
Graph / Processors	1	2	4	8	16
2048 Vertices	55.19 sec	27.7 sec	14.36 sec	7.7 sec	4.56 sec



From the graph above we can see a downward exponential curve as the number of processors increase with a fixed graph size. This is to be expected as more work is being done in parallel and the communication and management cost of multiple processors is minuscule compared to the benefits of parallelism.

Results for 4 processors vs 512, 1024, 2048, 4096 graphs

# Nodes / Graph	256	512	1024	2048	4096
4 nodes (1 process)	0.014 sec	0.29 sec	2.04 sec	14.44 sec	111.31 sec
2 nodes (2 processes)	0.014sec	0.29sec	2.09 sec	14.43 sec	115.16 sec



Looking above we can see that an increase in graph size has an exponential increase in wall time with a fixed number of processors. This is expected as the work is increasing and the amount of parallelisation is fixed. This means nodes are doing more work because they are receiving larger partitions of the matrix. We've also measured the same number of processes on a different number of nodes (each node can have up to 12 processes as each node has 12 cores). Looking at the comparison between 4 and 2 nodes (both with the same number of processes) there is not much difference in wall execution time.

Analysis

For this analysis we have decided to remove the time taken to perform I/O operations such as reading in the adjacency matrix and writing to the log file. The adjacency matrix is read into memory by the master process (process rank 0) and is then partitioned among the other processes as described above.

For this analysis, we focus on the parallelised Floyd Warshall function. This function performs a number of operations. The outer loop runs $O(V)$ time where V is the number of vertices in the graph. Inside this loop it then checks if this process is the owner of the current row which is done in linear time. If the process is the owner of the current row then this row needs to be broadcast to all other processes. It is copied to a temporary array of the same size which is done in $O(V)$ linear time. This temporary array must then be broadcast to every other process. We assume broadcasting messages is proportional to the size of the message being broadcast (that is the transfer time dominates any latency). Assuming we have p processes and the message size of V must be sent to $p - 1$ processes, this will produce an expected broadcast time of $O(Vp)$. After the broadcast is complete two for loops are run with two linear-time operations inside. These for loops run with an expected time of $O(V^2/p)$, as the outer loop runs through a processes local rows which are defined by V/p and the inner most loop runs V times. Combining this all together we get an expected run time of:

$$O(V \times (V + np + V^2/p)) = O(V^2p + V^3/p)$$