**ALL-PAIRS SHORTEST PATH USING FLOYD-WARSHALL ALGORITHM**

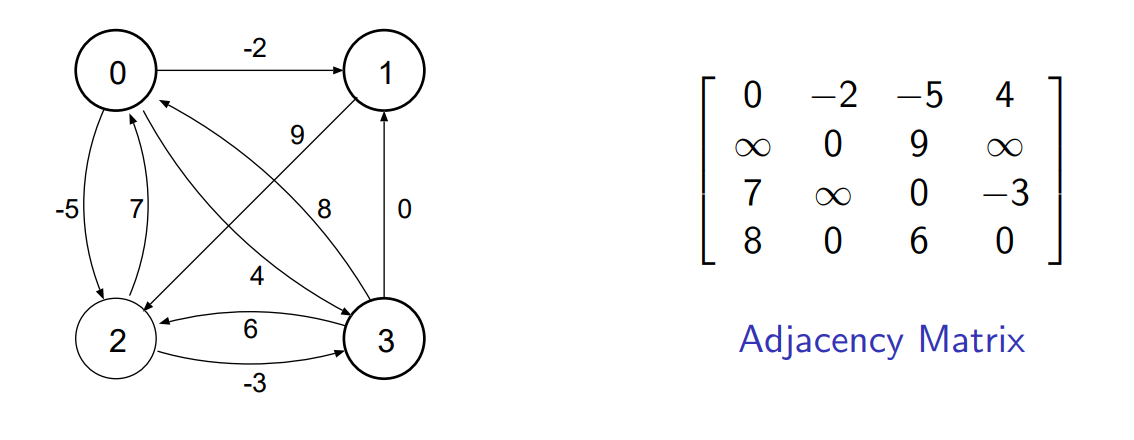
Alexander Rohl – 22233158

Farruh Mavlonov – 22252282

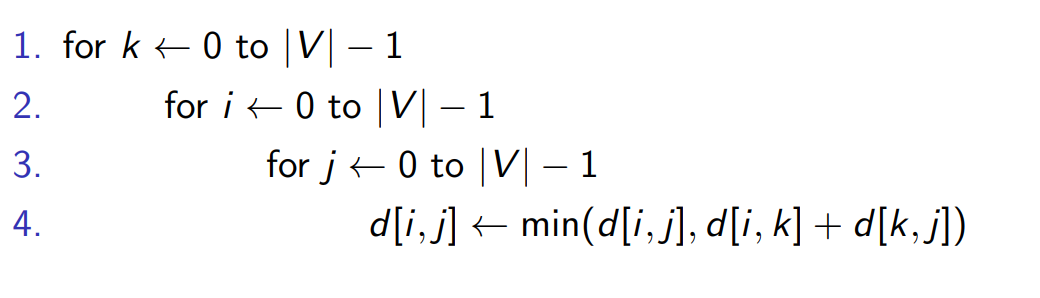
**SHORTEST PATHS**

Given a positively weighted directed graph, determine the shortest path between any two nodes in the graph.

The weighted directed graph below has 4 vertices and the corresponding adjacency matrix of size 4 x 4 is shown on the right. Given no self-looping, all the diagonal entries of the adjacency matrix are 0. If there is no directed edge from one vertex to another, the value in the corresponding adjacency matrix is .

In the code we defined infinity as #DEF INFINITY 100000000.

**FLOYD-WARSHALL ALGORITHM**

Recursive solution based on *intermediate* vertices with the following sequential sudo code:

Sequential complexity:

**PARALLEL ALGORITHMS USING MPI**

The most common way to write a parallel program is to use a sequential language and a subroutine library. For our project, we will be using C to write the bodies of processes and calling the MPI library for process creation, process management, communication and synchronization.

**RUNNING MPI CODE ON THE CLUSTER**

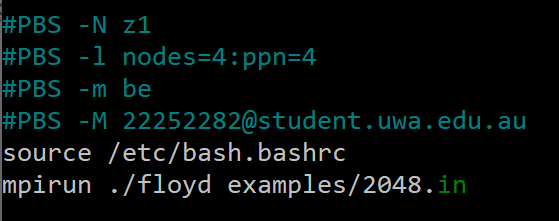
For this project, we ran the sequential code locally, and the MPI code on the cluster provided. The cluster had 29 nodes with 12 cores each so each node could run a maximum of 12 processes. We performed timing tests on the performance of our MPI code on the cluster by (1) varying the number of processes used on a graph with 2048 vertices, and (2) by varying the number of vertices in the graph whilst using 4 processes.

The steps taken to achieve to run the MPI code on the cluster:

1. SSH into the home node with our student number as the username and PHEME password. ssh [22252282@ecm-ubl-006.uniwa.uwa.edu.au](mailto:22252282@ecm-ubl-006.uniwa.uwa.edu.au)
2. Copy the required files from our git repo to the head node of the cluster using SCP.
3. Compile the code using mpicc to create an executable.

mpicc -o floyd FastMPIFloyd.c

1. syncCluster to synchronise all the clusters. This will send the files from the home node (node 0) to all other nodes on the cluster
2. Create a shell script with run.sh specifying the number of nodes (max 29) and processes per node (max 12) required. Edit this run.sh to vary the number of processes and the file input required for speedup testing.



1. Submit the job to the queue on the cluster using qsub run.sh. When execution is complete, 2 files will be added to the current directory z1.o123 and z1.e123, and a file containing the final adjacency matrix will be printed to the outputs/ directory.
2. The timing of the program will be output to the z1.o123 so we can run cat z1.o123

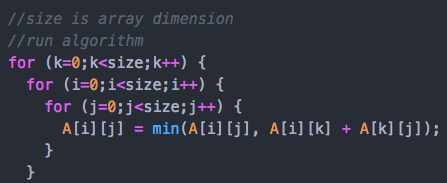
**TIME ANALYSIS OF THE SEQUENTIAL ALGORITHM**

We first parse the matrix sequentially. Since there are elements to be parsed, we expect this to grow . Over 5 runs for each matrix size, we notice the following time complexity.

Figure 1: Sequential Parsing Time Complexity Plot

The plot above shows an approximate quadratic fit. We can confirm this by considering a log-log plot of the data. By doing so, we see below a gradient of 1.8352, which is reasonably close to 2 and hence implies a quadratic relationship.

Figure 2: Sequential Parsing Time Complexity Log-Log Plot

We now test the sequential Floyd-Warshall algorithm. Since we have 3 nested for loops each having size number of iterations, we expect the time complexity to grow .

Unsurprisingly, the tests closely follow a cubic trend. To confirm this, we consider a log-log plot. By doing so, we see below a gradient of 2.8633 which is reasonably close to 3 and confirms a grow rate of .

Figure 3 - Floyd-Warshall Algorithm Time Complexity Log-Log Plot

Figure 4: Floyd-Warshall Algorithm Time Complexity Plot

**TIMING ANALYSIS OF SLOW MPI CODE**

*Filename*: SlowMPIFloyd.c

This approach is based on partitioning the matrix among processors and broadcasting the entire matrix for each iteration k iteration to all the subarrays. *Refer to the SUDO CODE below*.

NOTE: The actual code SlowMPIFloyd.c is modified to consider cases where the number of processors does not evenly divide the number of elements in the matrix. It does this by giving the left-over elements to the root process.

np <- number of processes

***--- MASTER PROCESS ---***

read matrix size

size <- matrix size

num\_local\_elements <- local number of elements = size/np

lo <- remaining elements after splitting = size - num\*np

read matrix

convert zeros to infinity

convert diagonals to zeroes

***----------------------***

MPI\_Bcast size

MPI\_Bcast num\_local\_elements

MPI\_Bcast lo

global\_index = np\*pid

SCATTER matrix to sub\_array

MPI\_Bcast matrix

for iteration (k) in range(size):

for local\_index in range(num\_local\_elements):

i (global row) <- (global\_index + local\_index)/size

j (global col) <- (global\_index + local\_index)%size

sub\_array[local\_index] = min(sub\_array[local\_index], matrix[i\*size+k] + matrix[k\*size+j])

GATHER sub\_array into matrix

MPI\_Bcast matrix

print matrix

Running our code on the cluster we see the following times:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **MATRIX SIZE** | | | | |
|  | **32** | **512** | **1024** | **2048** | **4096** |
| 1 | 0.001078 | 1.920015 | 13.93651 | 188.09429 |  |
| 2 | 0.001123 | 1.928698 | 13.74084 | 185.030222 |  |
| 3 | 0.00119 | 1.928318 | 13.743335 | 190.333041 |  |
| 4 | 0.001121 | 1.912096 | 13.533887 | 183.23194 |  |
| 5 | 0.001059 | 1.909035 | 13.471012 | 186.294095 |  |
| **Average** | 0.0011142 | 1.9196324 | 13.6851168 | 186.5967176 |  |

Table 1 - Slow MPI timing data using 4 processes

We had difficulties running the algorithm for matrices of size 4096. Assuming a time complexity growth rate of , we can estimate the time this computation would take to be approximately . Since this is well beyond the cluster wall-time limit of 600 seconds, we were unable to test this.

Comparing theses slow MPI code times to the sequential code times, we get the following speedup table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of vertices** | **Sequential time (s)** | **Parallel time (s)** | **Speedup** |
| 32 | 0.00048 | 0.00111 | 0.4265 |
| 512 | 0.97625 | 1.91963 | 0.5086 |
| 1024 | 8.12674 | 13.68512 | 0.5938 |
| 2048 | 72.69271 | 186.59672 | 0.3896 |

Table 2 - Speedup Table for slow MPI using 4 processes

This data shows that the SlowMPIFloyd.c MPI code actually runs about twice as slow compared to the sequential code. The reason for this is because the entire matrix is copied and queried for each process.

**TIMING ANALYSIS OF FAST MPI CODE**

*Filename*: FastMPIFloyd.c

To improve on the MPI code above, we notice that in each iteration k only the *k*th row and *k*th column are required. Using this, we came up with a more optimal algorithm that only shares this information with each process rather than the full matrix.

In order to preserve the functionality of being able to effectively run the algorithm such that the number of processes don’t necessarily divide the total number of matrix elements, we consider the following method:

1. Broadcast th row and th column.
2. Collectively build the next ()th row and ()th column while the th iteration is being computed.
3. Repeat.

SUDO CODE: (items crossed out are from the SLOW approach)

**~~MPI\_Bcast matrix~~**

MPI\_Bcast k\_row <- 0row

MPI\_Bcast k\_col <- 0col

for iteration (k) in range(size):

for local\_index in range(num\_local\_elements):

i (global row) <- (global\_index + local\_index)/size

j (global col) <- (global\_index + local\_index)% size

sub\_array[local\_index] = min(sub\_array[local\_index], **~~matrix[i\*size+k]~~** k\_col[i] + **~~matrix[k\*size+j]~~** k\_row[j])

if i==k+1:

(k+1)row element found

if j==k+1:

(k+1)col element found

GATHER sub\_array into matrix

**~~MPI\_Bcast matrix~~**

MPI\_Bcast k\_row <- (k+1)row

MPI\_Bcast k\_col <- (k+1)col

We see the performance of this algorithm on the cluster as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **MPI ALGORITHM TIMING (with 2048 vertices)** | | | | |
| NUMBER OF NODES | 1 | 2 | 4 | 4 | 4 |
| NUMBER OF PROCESSES | 1 | 2 | 4 | 8 | 16 |
| 1 | 158.751591 | 80.36369 | 40.630325 | 21.450497 | 12.993871 |
| 2 | 159.708206 | 80.388465 | 40.348857 | 21.464516 | 12.95157 |
| 3 | 158.738431 | 80.275108 | 40.351639 | 21.47298 | 13.008058 |
| 4 | 158.64898 | 80.453915 | 40.299123 | 21.454593 | 13.23492 |
| 5 | 158.582564 | 80.29967 | 40.301254 | 21.452171 | 12.893623 |
| **Average time** | 158.8859544 | 80.3561696 | 40.3862396 | 21.4589514 | 13.0164084 |
|  |  |  |  |  |  |
|  | **MPI ALGORITHM TIMING (with 1 ppn on 4 nodes)** | | | | |
| MATRIX SIZE | 32 | 512 | 1024 | 2048 | 4096 |
| 1 | 0.000793 | 0.767922 | 5.392001 | 40.630325 | 321.570263 |
| 2 | 0.000856 | 0.769462 | 5.384408 | 40.348857 | 320.174032 |
| 3 | 0.000892 | 0.770977 | 5.372458 | 40.351639 | 334.600912 |
| 4 | 0.000878 | 0.770193 | 5.384641 | 40.299123 | 333.497945 |
| 5 | 0.000882 | 0.770851 | 5.380796 | 40.301254 | 326.014467 |
| **Average time** | 0.0008602 | 0.769881 | 5.3828608 | 40.3862396 | 327.1715238 |

Table 3 - Timing data for Fast MPI code

More interestingly, we can plot this and compare against the previous implementations.

In the above plots, the log-log plot tells the story. By studying the gradients we see that the ‘Fast MPI’ algorithm clearly surpasses the previous two algorithms. Furthermore, although at the 4096 node scale the ‘Slow MPI’ algorithm was impossible to run, the log-log plot suggests a marginally superior time complexity for larger graphs (gradient of 2.83 for SlowMPI which is smaller than 2.87 for Sequential).

We also compare speedup with respect to the number of processors implemented:

The faster algorithm follows a clear power trend as we increase the number of processors at our disposal. This is reasonable as an efficient parallel program should increase the code speed *n*-fold for *n* processors. However, note the contrast to the slower algorithm, which from the data we have does not follow a clear trend. This is consistent with our intuition that this algorithm is not an effective parallelisation. **PERFORMANCE ANALYSIS OF MPI COLLECTIVE OPERATIONS**

Previous studies of application usage show that the performance of collective communications are critical for high performance computing (HPC) and are often overlooked when compared to the point-to-point performance. It is essential for MPI implementations to provide high-performance collective operations. The optimal implementation of a collective for a given system depends on many factors, including physical topology of the system, number of processes involved and message sizes.

**SPEEDUP TABLES**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of processes** | **Time (s)** | **Speedup** | **Efficiency (%)** |
| 1 | 158.8860 | 0.4575 | 45.75 |
| 2 | 80.3562 | 0.9046 | 45.23 |
| 4 | 40.3862 | 1.7999 | 45.00 |
| 8 | 21.4590 | 3.3875 | 42.34 |
| 16 | 13.0164 | 5.5847 | 34.90 |

Table 4 - Speedup for graphs of 2048 vertices vs 1, 2, 4, 8, 16 processors (sequential execution time 72.6927)

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of vertices** | **Sequential time (s)** | **Parallel time (s)** | **Speedup** |
| 32 | 0.00048 | 0.00086 | 0.5524 |
| 512 | 0.97625 | 0.76988 | 1.2681 |
| 1024 | 8.12674 | 5.38286 | 1.5097 |
| 2048 | 72.69271 | 40.38624 | 1.7999 |
| 4096 | 524.43165 | 327.17152 | 1.6029 |

Table 5 - Speedup for 4 processors vs graphs of 32, 512, 1024, 2048, 4096

**DISCUSSION OF SPEEDUP DATA**

Let be the size of the matrix and be the number of processes.

* Computational complexity:
  + Innermost loop:
  + Middle loop: + leftovers (negligible)
  + Outer loop:
  + Overall complexity:
* Communication complexity:
  + No communication in innermost loop
  + No communication in middle loop
  + Broadcast in outer loop:
  + Number of broadcasts: (one per outer loop iteration)
  + Overall complexity:

Therefore, the overall complexity of the FastMPIFloyd.c code is compared to the of the sequential code. For large matrices, the term will dominate and the time complexity will simplify down to For extremely large matrices and a fixed number of processors, the complexity will be approach , which is the complexity of the sequential code.

From *Table 4,* the speedup only occurs when the number of processes exceeds 4 when dealing with a matrix of size 2048. This is because the communication complexity outweighs the improved computational complexity for a small number of processes.

From *Table 5,* speedup only occurs for the larger matrices and generally improves with larger matrix size. For large , the computational complexity dominates compared to the communication overhead, leading to better speedup.