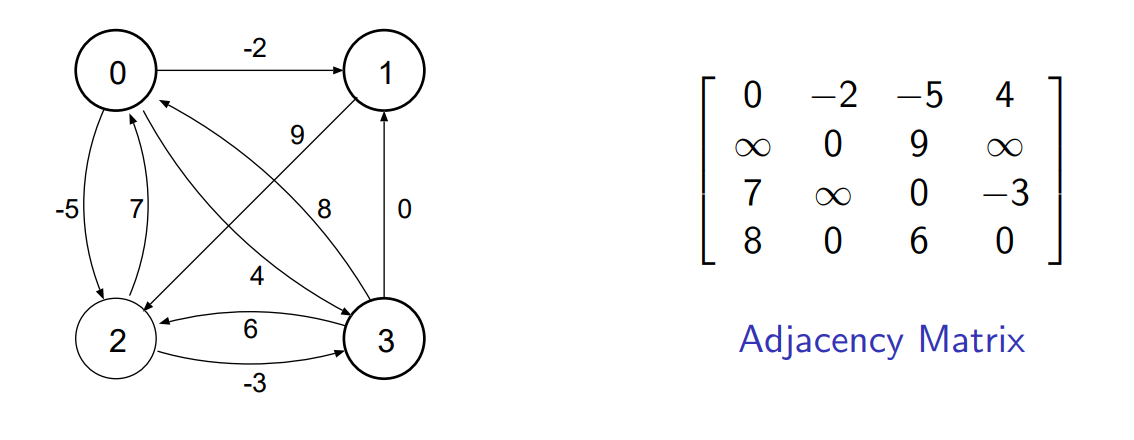
**ALL-PAIRS SHORTEST PATH USING FLOYD-WARSHALL ALGORITHM**

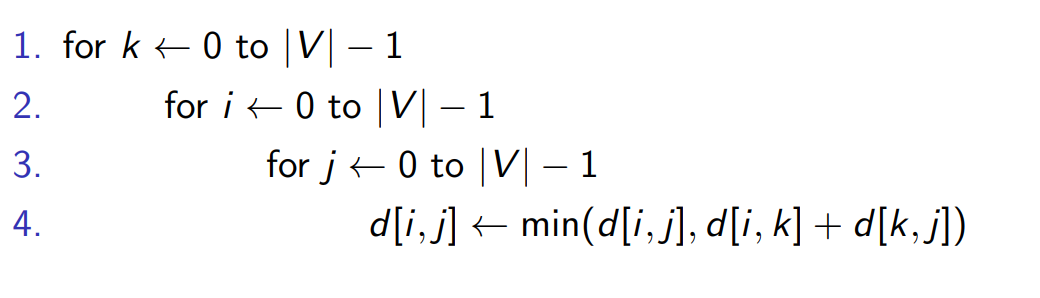
Alexander Rohl – 22233158

Farruh Mavlonov – 22252282

**Shortest Paths**

Given a weighted (positive only) directed graph, determine the shortest path between any two nodes in the graph. 

**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.

­­

**Program Architecture and OS**

For this project, we are running on MacOS High Sierra version 10.13.6

We compile our code in terminal by running:

mpicc main\_with\_matrix.c -o mysolution.exe

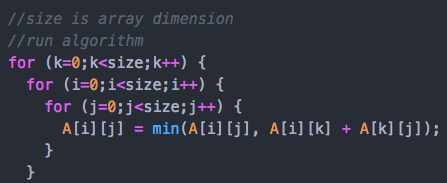
We measure program speed using wall-clock time with *MPI\_Wtime().* The regular c clock function takes the sum of all runtimes for all threads and therefore doesn’t return an accurate reading of speed up via threading.

Our code consists of the files implemented on the right in *main.c*. *parse\_input* is responsible for reading the binary matrix input and *helper\_functions* consists of any subfunctions required by main.

We execute our code in terminal by running:

 mpirun -n 20 --hostfile hostfile ./mysolution.exe examples/512.in

**Sequential Approach**



We first parse the matrix sequentially. Since there are N^2 elements to be parsed, we expect this to grow O(N^2). 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. However, 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 O(N^3).

Figure 3: Floyd-Warshall Algorithm Time Complexity Plot

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 O(N^3).

Figure 4: Floyd-Warshall Algorithm Time Complexity Log-Log Plot

**Farruh.c parallel code – when I get it working**

* Let be the time to compute an iteration
* Sequential execution time:
* Computational complexity:
  + Innermost loop:
  + Middle loop: at most
  + 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:
* Broadcast time:
  + Each broadcast has steps
  + is the message latency
  + is the bandwidth
  + Each broadcast sends bytes
* Overall parallel complexity:
* Expected parralel execution time:
* Deadlock: process waiting for a condition that will never become true
  + 2 processes both receive before send
  + Send tag doesn’t match receive tag
  + Process sends message to wrong destination process
* Strategy
  + Agglomerate tasks to minimise communication
  + Create one combined task per MPI process
* Row-wise block striped decomposition
  + Broadcast within rows eliminated
  + Reading matrix from file simpler because elements in C are stored in row major order
* A master process reads all the rows and send the data to the appropriate process

***Naïve Parallel Approach***

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 = lo + np\*pid

SCATTER matrix to sub\_array

--- MASTER PROCESS ---

get leftovers <- matrix[0:lo]

----------------------

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])

--- MASTER PROCESS ---

for local\_index in range(lo):

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

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

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

----------------------

GATHER sub\_array into matrix

matrix append leftovers to front

MPI\_Bcast matrix

print matrix

***Better Parallel Approach***

Why the method described above is “naïve” is that the entire matrix is copied and queried for each process. However, we notice that in each iteration (*k*) only the *k*th row and *k*th column is actually required. Hence, are more optimal algorithm should only share this information with each process rather than the full matrix. There are two methods we can do this…

1. Merge as before then extract kth row and kth column
2. Build the next (k+1)th row and (k+1)th column while the kth iteration is being computed.

~~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])

GATHER sub\_array into matrix

matrix append leftovers to front

~~MPI\_Bcast matrix~~

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

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

print matrix

**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. There are many parallel communicational models that predict performance of any given collective operation based on standardized system parameters.

**Speedup Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **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 - 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 - Speedup for 4 processors vs graphs of 32, 512, 1024, 2048, 4096