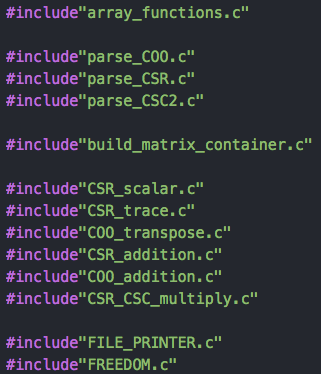
***Program Architecture and OS***

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

We compile our code in terminal by running:

/usr/local/opt/llvm/bin/clang -fopenmp -L/usr/local/opt/llvm/lib main.c -o mysolution.exe

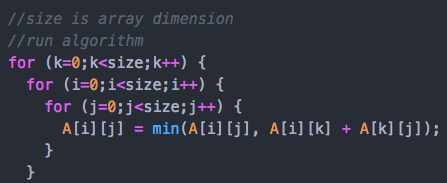
We measure program speed using wall-clock time with *omp\_get\_wtime()*. The regular c clock function takes the sum of all runtimes for all threads and therefore doesn’t result in an accurate reading of speed up via threading.

Our code consists of the files implemented on the right in *main.c*. The *parse* files are responsible for

We execute our code in terminal by running:

./mysolution.exe --mm -f INPUTS/int128.in INPUTS/int128.in -t 16

***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 : 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 : 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 : 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 : Floyd-Warshall Algorithm Time Complexity Log-Log Plot

***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.