Faiq Raza and Matt Robinson

Part I Processes

The basic idea behind implementing this algorithm with processes and shared memory was to create a single contiguous array and index into it as if it was a two-dimensional array. If we would normally index with g[i][j], then we could index into the single array with g[j*n + i], where g was the graph, n was the number of vertices (making the matrix n^2 in size), and i and j were the current nodes we were working on. The work was then split up over multiple processes by simply dividing up the rows (or columns depending on how you looked at the graph).

Here were running times for the first test case in milliseconds:

689.058899
686.071899
693.277466
717.533081
715.632202
692.510193
674.961792
699.416260
676.624695
695.861877

Part II Threads

We implemented the threaded part of the assignment with a different algorithm than our processed implementation. The way we set this algorithm up is figuring out the maximum work size each thread is going to be doing. We do this by doing the following calculation

max_work_size = vertices/threads

if there was a remainder the max_work_size was incremented by 1.

We then do a similar operation as we did in the processes implementation. Each thread gets a work_size amount of rows to operate on. After all the threads are finished, the k is then incremented.

Some times that we got to calculate the transitive closure on test case 1 from the homework file are

132.979980
177.298889
175.863922
154.878830
465.796631
151.326019

138.648697
212.632629
206.949051
157.492676
199.921997
238.268265
167.819992

There was one anomaly where the amount of time skyrocketed to 465 milliseconds