CITS3402 Project 2 Report

Henry Hollingworth / 21471423 & Ekin Bukulmez / 22412569 23rd/09/2019

Operating System

This submission was developed on Windows 10 (version 10.0.18362) using the Microsoft Visual Studio 2019. The C code was compiled with the MSVC toolkit included with Visual Studio. The MS-MPI implementation of MPI was used. An installation of Python 3.7 with the parameterized (wolever, 2019) and matplotlib (Hunter & Droettboom, 2019) was used to run a collection of tests and performance test script.

Overview

The project required the implementation of a parallelized version of either Dijkstra's or the Floyd-Warshall all-pairs shortest-path algorithm (APSP). Plan optimization and navigation are two examples of real world APSP. The input to an APSP problem is a weighted directed graph and the answer is the collection of shortest paths from any node to any other node.

We decided to implement the Floyd-Warshall algorithm because we hadn't implemented it before, and it was in our opinion more conceptually elegant than Dijkstra's algorithm. Python scripts were written to test the implementation and gather performance metrics.

Parallelization Strategy

The non-parallelized version of the Floyd-Warshall (The Algorithm) is typically implemented with a dynamic programming approach which applies a series of relaxations to an adjacency matrix which contains estimates of the minimum distance between nodes. On successive iterations this estimate is relaxed until it is optimal.

Dynamic Programming Reduction

The problem of finding the shortest path in a weighted directed graph G with n vertices V between two vertices A and B can be reduced to choosing the shorter of:

a) A path from A to B which does not visit the k-th node.
b) A path from A to the k-th node, and then from the k-th node to B.

By arbitrarily ordering our vertices, we can successively ask this question for $k = 1$ up to and including $k = n$. Intuitively this is asking for increasing sets of vertices if there is a shorted path from A to B using the current (larger) set of nodes, or the previous (smaller) set of nodes.

Task Subdivision

The algorithm can be parallelized by distributing vertex pairs between the available processes. Conceptually this is the equivalent of re-writing the implementation in *Figure 1: Floyd-Warshall Algorithm* by copy-pasting the two inner loop (over i's and j's) where

each copy of the loop traverses a unique portion of the vertices.

```
def floyd_warshall_solve(adjacency_matrix):
    """Solve in-place the all-pairs shortest path for a 2D adjacency matrix."""
    n_vertices = len(adjacency_matrix)
    for k in range(n_vertices):
        for i in range(n_vertices):
            for j in range(n_vertices):
                if adjacency_matrix[i][j] > adjacency_matrix[i][k] + adjacency_matrix[k][j]:
                    adjacency_matrix[i][j] = adjacency_matrix[i][k] + adjacency_matrix[k][j]
    return adjacency_matrix
```

Figure 1: Floyd-Warshall Algorithm

Data Dependencies

Using the beforementioned task subdivision strategy results in each process operating over a rectangular 'tile' of the adjacency matrix. For a process to update its tile, it may require information from neighbouring tiles along the x and y axis of the tile grid it is situated in. To better visualize and understand the dependencies we constructed *Figure 2: Tile Size Communication* Trade-off

Figure 3: Data Dependency Diagram which depicts the data dependencies of a 9x9 matrix for the *k*=1 iteration. The diagram depicts the tiles (assigned 1:1 to processes) with different colors. Each tile has a specific letter, and wherever that letter appears is a cell which is required to update the tile.

Performance

The program was compiled in release mode and invoked by a Python script completing 25 trials for each combination of matrix size and thread count category.

The mean and standard deviation were calculated and recorded in *Table 1: Performance Results*. Generally, there was a significant performance improvement of between 20-63% per category as the number of threads increased. This trend stopped however at the 16-thread category – this is simply an artefact of the tests being executed on a 4-core CPU with Hyperthreading enabled exposing 8 logical processors.

In analysing the performance characteristics, the following variables must be considered: matrix size (*m*), tile size (*t*), number of available processors (*p)* and the inter-processes communication overhead (*L(x)*).

At a high level the heuristic to apply when deciding the size of the tiles is – what is the *maximum* tile size *t* I can make given that I can use at most *p* square tiles to cover m^2 adjacency values.

For small matrix sizes in particular you must consider if the communication overhead for each process of *L(2t)* (each process must broadcast/receive two *t*

sized segments) outweighs the parallelization benefit. The relationship between tile size and communication overhead is presented graphically in *Figure 2: Tile Size Communication Trade-off*. Notice that only at a certain tile size is the communication overhead L(x) less than the parallelization benefit which scales with the number of adjacency values (t^2) in the tile.

It should be noted that this overhead consideration is also relevant at *large* matrix sizes if *vast* numbers of processors are available because then the tile size is driven down. In the extreme case having m^2 processes each cover a 1x1 tile would be less efficient than having 1 process cover the whole adjacency matrix (refer to *Figure 2*).

A more detailed analysis utilizing Amdahl's Law (Wikipedia, 2019) could be used to determine the precise matrix and tile sizes for a given computer environment (communication overhead and processor count) for which parallelization confers a benefit. For this project the empirical results gathered show that at least for matrices of size greater than or equal to 256 that parallelization results in a speedup.

Figure 2: Tile Size Communication Trade-off

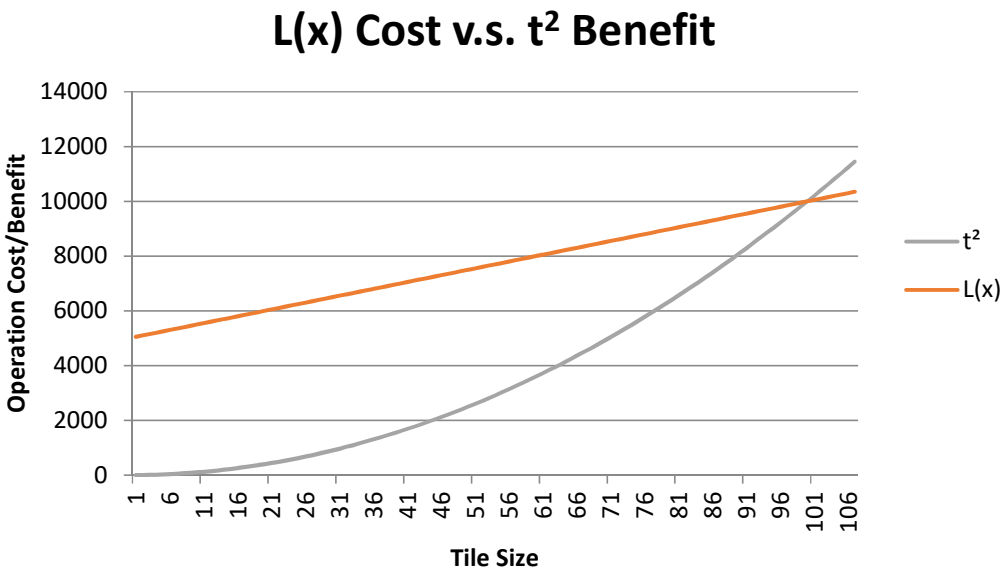

Figure 3: Data Dependency Diagram

	0	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
0	F	XMF	F	M	M	M	X	X	X
1	FWC	XMFWC	FWC	IMP	IMP	IMP	XKO	XKO	XKO
2	F	XMF	F	M	M	M	X	X	X
3	C	COI	C	I	I	I	O	O	O
4	C	COI	C	I	I	I	O	O	O
5	C	COI	C	I	I	I	O	O	O
6	W	WKP	W	P	P	P	K	K	K
7	W	WKP	W	P	P	P	K	K	K
8	W	WKP	W	P	P	P	K	K	K

Table 1: Performance Results

	1 Threads	*2 Threads*	*4 Threads*	*8 Threads*	*16 Threads*
256x256	μ=15.88	μ=6.52 (-59%)	μ=7.92 (+21%)	μ=4.08 (-48%)	μ=10.48 (157%)
	σ=0.73	σ=7.67 (+957%)	σ=1.22 (-84%)	σ=4.08 (+234%)	σ=0.92 (-78%)
512x512	μ=121.68	μ=86.08 (-29%)	μ=51.68 (-40%)	μ=25.08 (-51%)	μ=48.48 (+93%)
	σ=1.46	σ=54.18 (3600%)	σ=9.38 (-82%)	σ=24.53 (+161%)	σ=2.29 (-91%)
1024x1024	μ=975.6	μ=638.8 (-34%)	μ=297.68 (-53%)	μ=186.72 (-37%)	μ=410.84 (+120%)
	σ=11.13	σ=445.35 (+3900%)	σ=36.17 (-92%)	σ=141.81 (+292%)	σ=21.11 (-85%)
2048x2048	μ=8534.84	μ=6756.64 (-21%)	μ=3904.4 (-42%)	μ=1411.32 (-63%)	μ=2825.84 (+100%)
	σ=182.14	σ=3005.75 (+1505%)	σ=42.82 (-99%)	σ=1908.50 (+4357%)	σ=31.43 (-98%)
4096x4096	μ=70491.08	μ=51873.8 (-26%)	μ=42018.96 (-20%)	μ=25274.4 (-40%)	μ=50302.96 (+100%)
	σ=2722.78	σ=26470.79 (+872%)	σ=279.29 (-99%)	σ=21033.82 (+7431%)	σ=261.52 (-99%)

μ = mean, σ = std. deviation. The % changes are with reference to the previous thread count category for the same size matrix. A negative percentage represents a decrease in time, whilst a positive percentage is an increase.

References

Hunter, D. J., & Droettboom, M. (2019, 10 22). *matplotlib*. Retrieved from pypi: https://pypi.org/project/matplotlib/

libcheck. (2019, 09 23). *Home*. Retrieved from Check: https://libcheck.github.io/check/

Wikipedia. (2019, 10 22). *Amdahl's Law*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Amdahl%27s_law

wolever. (2019, 10 22). *parameterized*. Retrieved from pypi: https://pypi.org/project/parameterized/

zeehio. (2019, 09 23). *parmap*. Retrieved from pypi: https://pypi.org/project/parmap/