Introduction

This report outlines and analyses an implementation of a parallel algorithm for the following problem: Given a grid, where each square has a positive integer as its cost, find the shortest 8 connected path from the top left most square to the bottom right most square, where the distance of a path is defined as the sum of costs of all squares on the path.

The algorithm I've devised is a modified version of the delta stepping algorithm from Mayer and Sanders [1], described in the algorithm 1 with its functions in algorithm 2.

Note that $\textsc{Cost}(x, y)$ returns the cost of the cell on x_{th} row and y_{th}. Furthermore, while the cost takes a long time to calculate for the first time. This function caches the cost in a 2D array so subsequence call returns instantly. This has the additional effect of calculating the cost of each cell upfront in a parallel fashion during the FindDelta procedure. Not only does this allow delta to be calculated at the optimal value. It also allows the load of calculating the cost of each cell to be spread evenly among all processors, taking advantage of parallelization for the most expensive part of the algorithm.

Furthermore, to ensure that *relaxed* and *removed* can be queried in constant time as well as being made emptied in constant time, I implemented them as 2D int arrays where (x, y) is in the set if $set[x][y] = trueValue$. Here $trueValue$ is an integer. By incrementing $trueValue$ we can empty the set very quickly.

The implementation of the algorithm is less than ideal. Because I choose to use vector in the C++ standard library, which isn't thread safe, I have to heavily used locking and OpenMP critical region to ensure data races does not happen. This means the implementation of the algorithm isn't as parallel as it should be and further improvement on data structures and hence performance can be made.

Algorithm 1

$x_{end} \leftarrow x_{size} - 1$
$y_{end} \leftarrow y_{size} - 1$
$delta \leftarrow$ FindDelta()
for $x \leftarrow 0$ **to** x_{end} **do in parallel**
    for $y \leftarrow 0$ **to** y_{end} **do in parallel**
        $distance[x][y] \leftarrow \infty$
        $bmap[x][y] \leftarrow -1$ ▷ bmap map (x, y) to the which b[i] set they're in, hence speeding up the Remove.
    end for
end for
$distance[0][0] \leftarrow$ Cost$(0, 0)$
Insert(0, 0)
$i_{dest} \leftarrow \infty$
$removed \leftarrow \emptyset$
$relaxed \leftarrow \emptyset$
$i \leftarrow 0$
while $i < i_{dest}$ **do**
    while b[i] is not empty **do**
        Relax(b[i], relaxed, removed)
        $b[i] \leftarrow \emptyset$
        for each$(x, y) \in relaxed$ **do in parallel**
            Insert(x, y)
            if $x_{end} = x \wedge y_{end} = y$ **then**
                $i_{dest} \leftarrow i$
            end if
        end for
    end while
    $relaxed \leftarrow \emptyset$
    $removed \leftarrow \emptyset$
end while
$x \leftarrow x_{end}$ ▷ Reconstruct the path from distance.
$y \leftarrow y_{end}$
$path \leftarrow []$
while $x \neq 0 \wedge y \neq 0$ **do**
    $x_{pred} \leftarrow x$
    $y_{pred} \leftarrow y$
    for $dx \leftarrow -1, 0, 1$ **do**
        for $dy \leftarrow -1, 0, 1$ **do**
            $x_{new} \leftarrow x + dx$
            $y_{new} \leftarrow y + dy$
            if (x_{new}, y_{new}) is valid coordinate $\wedge$ $(x_{new}, y_{new}) \neq (x, y)$ **then**
                if $distance[x_{new}][y_{new}] < distance[x_{pred}][y_{pred}]$ **then**
                $x_{pred} \leftarrow x_{new}$
                $y_{pred} \leftarrow y_{new}$
                end if
            end if
        end for
    end for
    $pathprepend(x_{pred}, y_{pred})$
end while
return path

Algorithm 2 Functions of Algorithm 1

```
1: function FindDelta( )
2:     for x ← 0 to x_end do in parallel
3:         for y ← 0 to y_end do in parallel
4:             cost_max ← Max(cost_max, Cost(x, y))
5:         end for
6:     end for
7:     return cost_max/8
8: end function
9: function Remove(x, y)
10:    if bmap[x][y] ≠ −1 then
11:        b[bmap[x][y]] ← b[bmap[x][y]] \ (x, y)
12:    end if
13: end function
14: function Insert(x, y)
15:    i ← ⌊distance[x][y]/delta⌋
16:    bmap[x][y] ← i
17:    b[i] ← b[i] ∪ (x, y)
18: end function
19: function Relax(to-relax, relaxed, removed)        ▷ pass NULL as removed will cause the function to only Relax heavy edges. Pass a set as removed will cause the function to call Remove for each coordinate in to-relax and add them to removed.
20:    for each (x, y) ∈ to-relax do in parallel
21:        if removed ≠ NULL then
22:            Remove(x, y)
23:            removed ← removed ∩ (x, y)
24:        end if
25:        for dx ← −1, 0, 1 do
26:            for dy ← −1, 0, 1 do
27:                x_next ← x + dx
28:                y_next ← y + dy
29:                if (x_next, y_next) is valid coordinate ∧ (x_next, y_next) ≠ (x, y) then
30:                    if (Cost(x_next, y_next) ≤ delta) ⊕ (removed = NULL) then    ▷ If removed is null, only relax heavy edges, otherwise only relax light edges.
31:                        distance_new ← distance[x][y] + Cost(x_next, y_next)
32:                        if distance_new < distance[x_next][y_next] then
33:                            distance[x_next][y_next] ← distance_new
34:                            relaxed ← relaxed ∩ (x_next, y_next)
35:                        end if
36:                    end if
37:                end if
38:            end for
39:        end for
40:    end for
41: end function
```

Methodology

To confirm and measure the performance of my algorithm, I've run the algorithm's implementation on spartan with 1, 2, 4, 8, 16, 32 and 72 threads respectively. The input samples I used are 10x10edge, 10x10expensive, 10x10fast, 10x10faster, 10x10zigzag, 4x4edge and 4x4Z. Among those input samples, 10x10expensive, 10x10fast and 10x10faster are three relatively similar samples with their only significant difference between each other being the cost of calculating the cost of a cell. These three samples will provide us with insight over how the overall runtime changes with different cost of calculating the cost of cells. I've also run a sequential Dijkstra's algorithm implementation with the same hardware to act as baseline to further evaluate the performance of my delta stepping algorithm.

I have collected the following metrics of performance:

- The overall time taken to run the algorithm.
- The overall time taken to calculate the delta, which should be approximately the overall time taken to calculate all costs of cells.
- The amount of CPU time used to run the whole algorithm. This should be a good approximation of the amount of work done by the algorithm.
- The amount of CPU time used to calculate all costs of cells. This should be a good approximation of the amount of work done to calculate the costs of cells.

Experiments

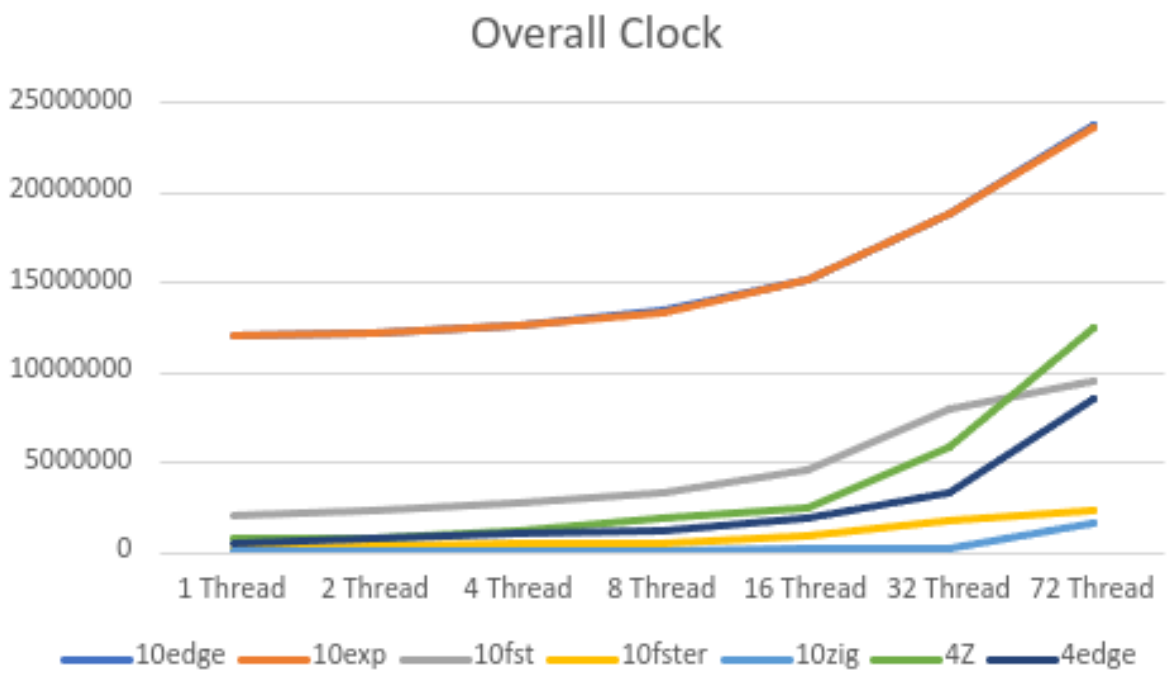

Figure 1: Clock Ticked Overall

Figure 1 and Table 1 shows the amount of CPU time taken by each sample inputs for each thread counts. CPU time excludes the time taken for IO and the time any core spent idling.

Figure 2 and Table 2 shows the amount of real world time that has elapsed since the the algorithm started running til the algorithm has finished printing out all the results. This includes the time taking to print the shortest path to stdout but does not include the time taken to read the input in.

Figure 3 and Table 3 shows the amount of CPU time taken to calculate the cost of all cells. This time

Threads	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	72 Thread
10edge	12078822	12277090	12626906	13439279	15240944	18808929	23802727
10exp	12079417	12281059	12643391	13406686	15185691	18888205	23635012
10fst	2111496	2314699	2719620	3339569	4636244	7993472	9472809
10fster	276932	322116	524912	518381	869857	1724042	2309900
10zig	12860	16124	28251	27568	253934	290537	1590605
4Z	729875	814800	1215310	1944857	2511456	5848036	12481201
4edge	571492	773687	1051905	1258741	1919442	3313304	8553222

Table 1: Clock Ticked Overall

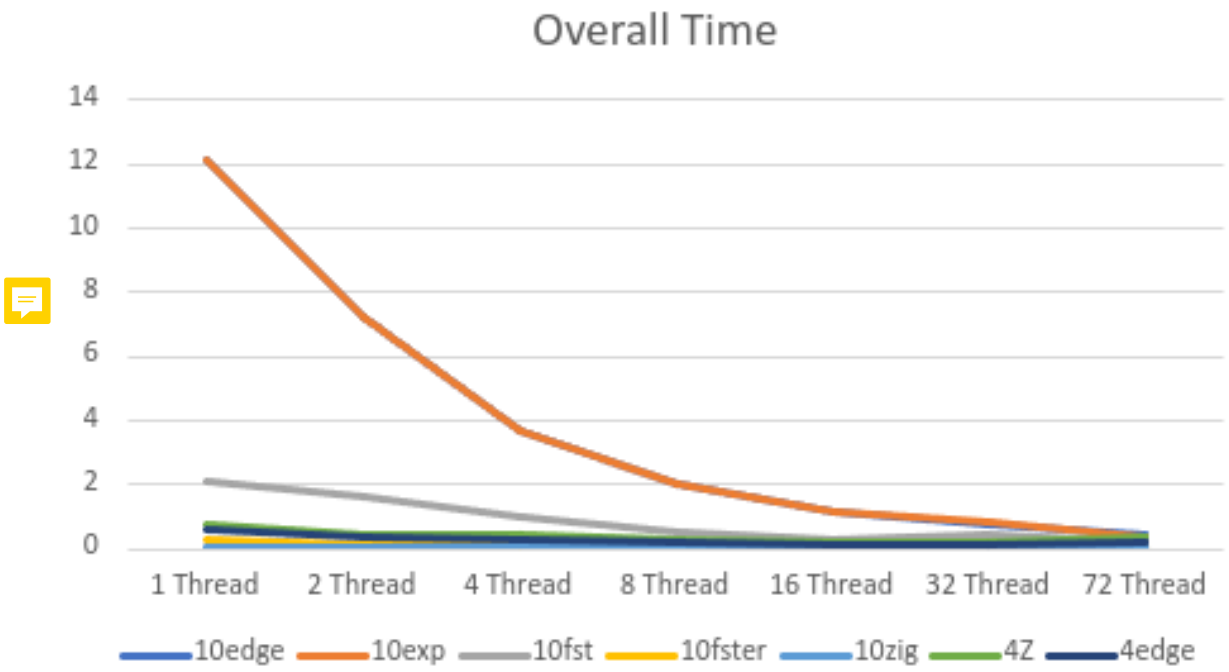

Figure 2: Time Taken Overall

Threads	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	72 Thread
10edge	12.14191	7.215103	3.674875	2.030348	1.162641	0.759203	0.457252
10exp	12.146206	7.201691	3.677784	2.045231	1.159449	0.8069	0.395972
10fst	2.120565	1.654374	1.026866	0.54627	0.320378	0.471045	0.198928
10fster	0.278852	0.16332	0.140824	0.070513	0.066769	0.078457	0.072619
10zig	0.013412	0.009532	0.017911	0.007888	0.036478	0.019915	0.052021
4Z	0.733462	0.411218	0.423022	0.311593	0.189334	0.209137	0.354891
4edge	0.584149	0.39121	0.266031	0.166482	0.153303	0.151062	0.166345

Table 2: Time Taken Overall

also includes the time taken to calculate the delta. However, because calculating the delta is just one single division and maximum reduce, the time of calculating the cost should dominate the time measured and the extra time for calculating the delta should be minimum.

Figure 4 and Table 4 shows the amount of real world time that has elapsed while all cell costs are being calculated. This is measured at the same point as the CPU time for cell cost.

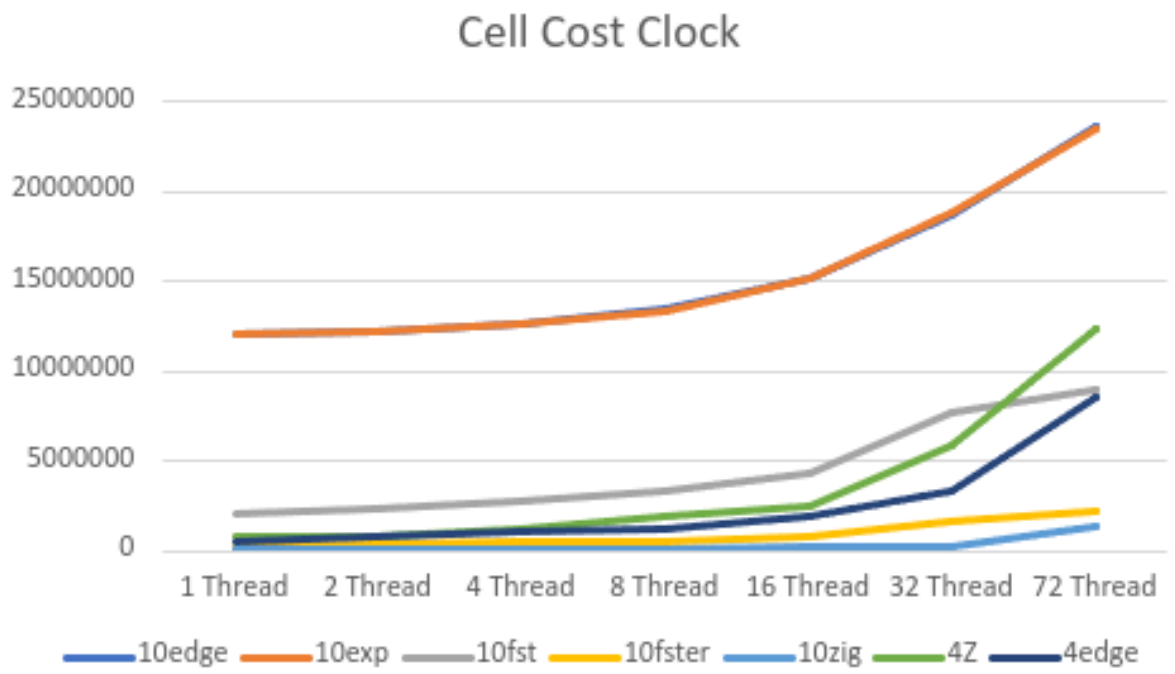

Figure 3: Clock Ticked Calculating Cell Cost

Threads	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	72 Thread
10edge	12078599	12276744	12624881	13436181	15123617	18747048	23717296
10exp	12079261	12280644	12641469	13399624	15177628	18846984	23580315
10fst	2111349	2312870	2715970	3333391	4384763	7728126	8973042
10fster	276772	321312	520643	511140	843950	1664345	2134147
10zig	12710	14498	14782	17840	205042	243652	1348440
4Z	729758	814045	1213116	1941143	2493236	5824255	12361348
4edge	571360	772923	1049339	1254804	1909493	3271299	8488262

Table 3: Clock Ticked Calculating Cell Cost

Threads	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	72 Thread
10edge	12.141602	7.214861	3.673924	2.029927	1.154142	0.757123	0.455895
10exp	12.145618	7.201427	3.676888	2.04422	1.158842	0.805415	0.395176
10fst	2.12045	1.652942	1.025374	0.545406	0.30344	0.460225	0.190933
10fster	0.278386	0.162702	0.139585	0.069563	0.064977	0.076181	0.069631
10zig	0.012993	0.008508	0.013746	0.00643	0.031127	0.018265	0.047379
4Z	0.733073	0.410653	0.42216	0.311068	0.187978	0.208346	0.352871
4edge	0.583667	0.39064	0.265484	0.166054	0.152565	0.149284	0.165284

Table 4: Time Taken Calculating Cell Cost

Figure 5 and Table 5 shows the difference between the overall clock time taken and the time taken to overall. The original intention of this data is to analyze the time taken by the actual path finding algorithm. However, it appears to be dominated by noise.

Figure 6 and Table 6 shows the difference between the elapsed real world time taken by the whole algorithm and the elapsed real world time taken by the cell costs evaluation. These data again appears

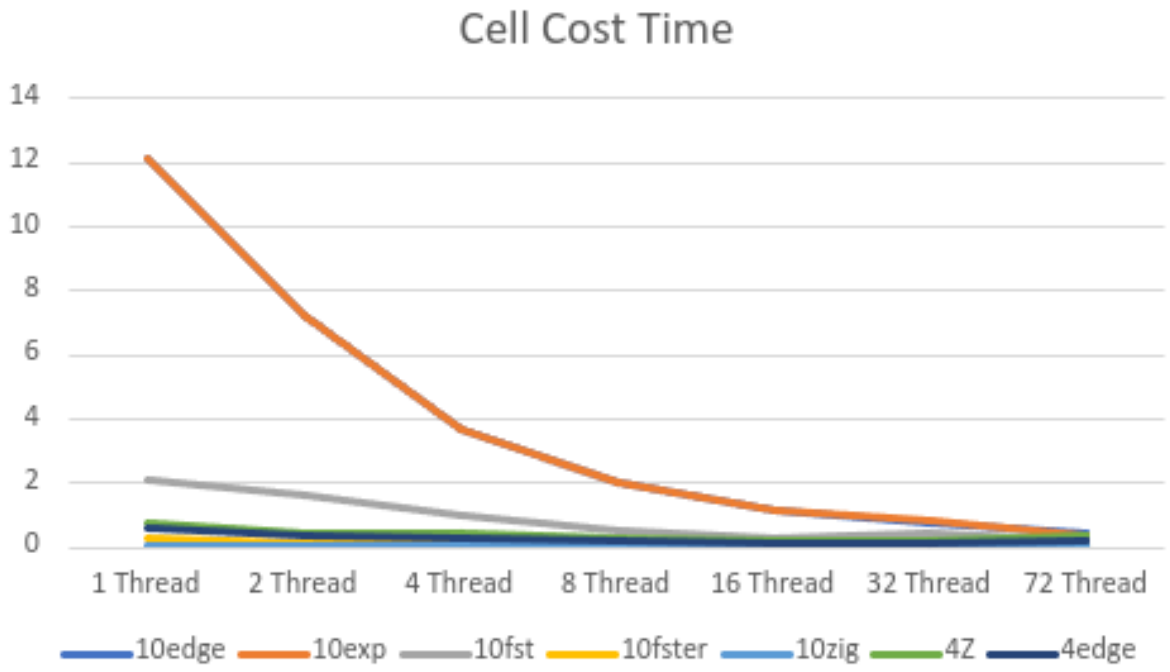

Figure 4: Time Taken Calculating Cell Cost

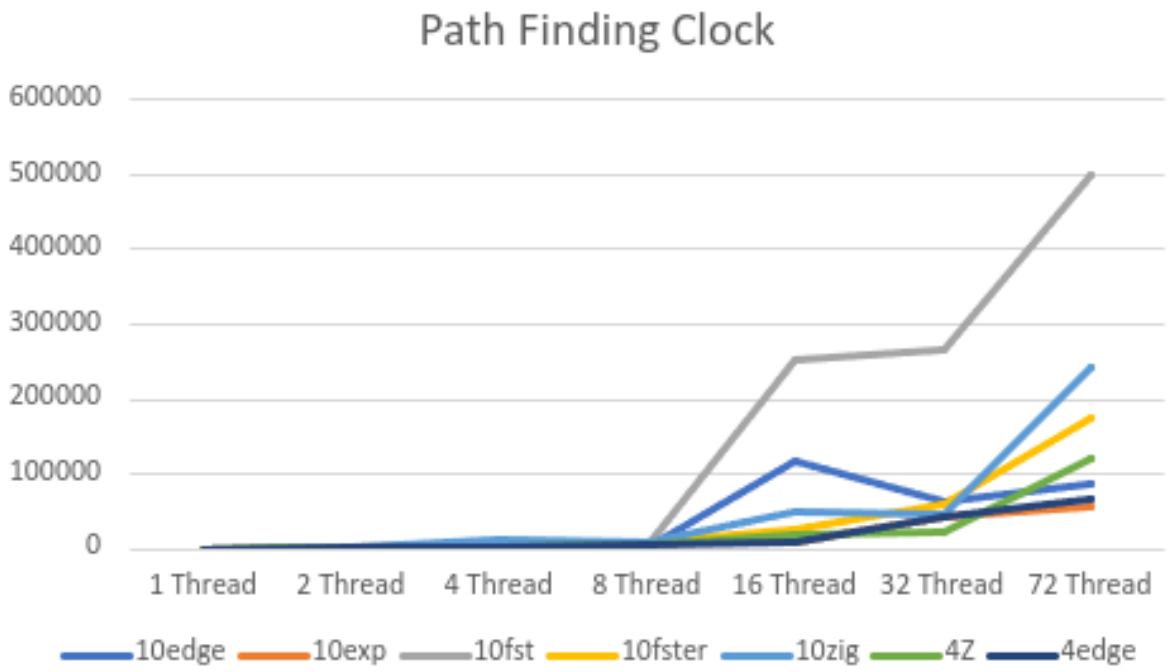

Figure 5: Time Taken Overall Minus Time Taken Calculating Cell Cost

to be dominated by noise.

Table 7 shows both the elapsed real time taken and the amount of CPU time taken by the sequential Dijkstra's algorithm. Those two measurements are somewhat duplicated because the sequential algorithm only uses one core, so that core is busy most of the time and the CPU time ends up being roughly

Threads	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	72 Thread
10edge	223	346	2025	3098	117327	61881	85431
10exp	156	415	1922	7062	8063	41221	54697
10fst	147	1829	3650	6178	251481	265346	499767
10fster	160	804	4269	7241	25907	59697	175753
10zig	150	1626	13469	9728	48892	46885	242165
4Z	117	755	2194	3714	18220	23781	119853
4edge	132	764	2566	3937	9949	42005	64960

Table 5: Time Taken Overall Minus Time Taken Calculating Cell Cost

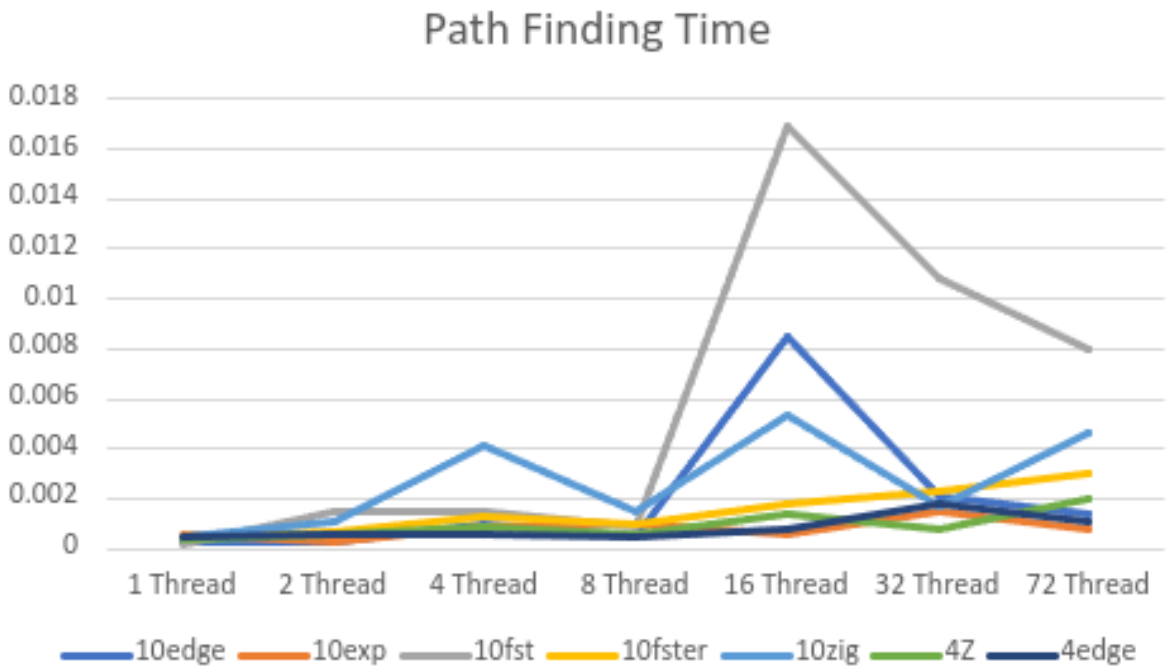

Figure 6: Clock Ticked Overall Minus Clock Ticked Calculating Cell Cost

Threads	1 Thread	2 Thread	4 Thread	8 Thread	16 Thread	32 Thread	72 Thread
10edge	0.000308	0.000242	0.000951	0.000421	0.008499	0.00208	0.001357
10exp	0.000588	0.000264	0.000896	0.001011	0.000607	0.001485	0.000796
10fst	0.000115	0.001432	0.001492	0.000864	0.016938	0.01082	0.007995
10fster	0.000466	0.000618	0.001239	0.00095	0.001792	0.002276	0.002988
10zig	0.000419	0.001024	0.004165	0.001458	0.005351	0.00165	0.004642
4Z	0.000389	0.000565	0.000862	0.000525	0.001356	0.000791	0.00202
4edge	0.000482	0.00057	0.000547	0.000428	0.000738	0.001778	0.001061

Table 6: Clock Ticked Overall Minus Clock Ticked Calculating Cell Cost

proportional to each other.

	Dijkstra time	Dijkstra clock
10edge	7.296148	7224625
10exp	6.746439	6697340
10fst	1.422838	1412013
10fster	0.591794	588129
10zig	0.038056	36405
4Z	2.523785	2509178
4edge	1.198375	1191219

Table 7: Time and Clock tick taken by the sequential Dijkstra's algorithm

Discussion

Compared the overall time taken by my delta stepping algorithm 2 with a single thread, and the overall time taken by the Dijkstra's algorithm 7, we can see that for 10x10edge, 10x10expensive and 10x10fast, my delta stepping algorithm took roughly twice as long as the Dijkstra algorithm. I believe this is due to the fact that the Dijkstra's algorithm only calculate the cell costs needed to find the shortest path while my delta stepping algorithm always calculate all the cell costs upfront. However, we can also see that for 10x10faster, 10x10zigzag, 4x4Z and 4x4edge, my delta stepping algorithm performed significantly better than the Dijkstra's algorithm. I believe this is due to the fact that the Dijkstra's algorithm calculate the cell costs from scratch every time it needs to query it while my delta stepping algorithm only calculate each cell costs once.

Nonetheless, when we look at my delta stepping algorithm with more than 2 threads 2, we can see that it ran significantly faster than the Dijkstra's algorithm. We can gain more insight into the speed-up by looking into Table 6 and Figure 6. Here we can see that the vast majority of the time taken by the delta stepping algorithm are spent calculating the cell costs. This make sense because cell costs are made to be intentionally expensive while the graph itself is relatively small with only 100 nodes and less than 400 edges. As a result, the speed-up we've obtained does not serve as a good source of analysis for generic parallel shortest path algorithm because our result is dominated by an easily parallizable step. For future work, we can test our algorithm without expensive cell cost calculation and with significantly larger graphs.

When we look at Figure 2 and Table 2, we can see that in the case of 10x10edge and 10x10expensive, the elapsed time roughly halves every time the number of thread doubles. This make sense as those two input sample's calculation are heavily dominated by calculating the cell costs. We can also see that for other input samples, the overall elapsed time in fact increases when going from 32 threads to 72 threads and for some of them even from 16 threads to 32 threads. I believe this is because for the node the algorithm running on are made up by 4 18 cores CPUs. As the number of threads increase beyond 18, the algorithm can no longer run on a single CPU and therefore the application starts incurring the communication costs between CPUs. If the algorithm doesn't have a lot of easily parallizable task to do, it will spend most of its time on the communication costs hence increasing the overall time taken by the algorithm.

When we look at Figure 1 and Table 1, we can see that the total CPU time increases as the number of threads increases. This is especially prevalent on small input sample such as 4x4Z and 4x4edge where the CPU time more than doubled going from 32 thread to 72 threads. This again matches with our earlier analysis of communication overhead between CPUs making the algorithms slower instead of faster.

If we try to ignore the cell costs time and attempt to view the time taken to calculate the path itself, we find very noisy data in Figure 6 and Table 6. This is potentially due to our flawed methodology of only doing one run per thread count per input sample instead of averaging over multiple runs. If we try to

ignore the noise in the data and attempt to view the trend, we can see that the time of everything except cell costs calculation actually increases significantly as the number of thread increases. I believe this is due to the fact that there isn't a lot of work to be done with path finding, and, as mentioned above, that the communication overhead between CPUs across sockets starts dominate the total running time. This communication overhead can also explain the large amount of variation between each run. It could be that our algorithm are sometimes being assigned to cores that are spread out among the sockets and sometimes assigned to cores that are on the same sockets.

Speculating, if we increase the core count of the hardware the algorithm is running on as well and the thread count of the algorithm to match the core count, the running time of the algorithm for input 10x10expensive and 10x10edge would not see significant improvement after the core count exceed 100, as there are only 100 cell costs to calculate. If we increase the size of the graph well beyond the core count and have cell costs calculation remain the dominating cost of the algorithm, we will see a close to inversely proportional relationship between the running time and the core count with some overhead as the core count increases due to communication overhead. However, if we make calculating the cost of each cell very cheap, my algorithm might not perform well on hardware of high core count. I believe this because the heavy use of locking in my algorithm to protect many non thread safe data structures that I used, and the fact that we saw our path finding time rise as the number of thread increased.

From a theoretical standpoint, if we discard the dominating factor that is the cell cost calculation, my delta stepping algorithm should have a time complexity of $O((m+n)log^2(m*n))$ where m and n are the width and height of the grid. This result is a specialization of the more general result that is proven by Meyer and Sander in [1] when the delta is chosen to be $\Theta(1/d)$ and the edge weights are uniformly distributed in $[0, 1]$. However, my implementation of the algorithm did not take advantage many of the parallizable data structure presented by Meyer and Sander [1]. In particular, my buckets are implemented as C++ vectors which are not thread safe and have to be surrounded by locks to ensure their correctness. When viewing the problem as a path finding problem rather than a cell cost calculating problem, there are many improvements to be made to my algorithm.

References

[1] U. Meyer and P. Sanders, ``Δ-stepping: A parallelizable shortest path algorithm,'' *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 1998, 1998 European Symposium on Algorithms, ISSN: 0196-6774. DOI: https://doi.org/10.1016/S0196-6774(03)00076-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403000762.