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.

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

Discussion

Conclusion

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-

Algorithm 1

```
1: $x_{end} \leftarrow x_{size} - 1$
2: $y_{end} \leftarrow y_{size} - 1$
3: $delta \leftarrow$ FINDDELTA()
4: for $x \leftarrow 0$ to $x_{end}$ do in parallel
5:     for $y \leftarrow 0$ to $y_{end}$ do in parallel
6:         $distance[x][y] \leftarrow \infty$
7:         $bmap[x][y] \leftarrow -1$   ▷ bmap map (x, y) to the which b[i] set they're in, hence speeding up the REMOVE.
8:     end for
9: end for
10: $distance[0][0] \leftarrow$ COST$(0, 0)$
11: INSERT(0, 0)
12: $i_{dest} \leftarrow \infty$
13: $removed \leftarrow \emptyset$
14: $relaxed \leftarrow \emptyset$
15: $i \leftarrow 0$
16: while $i < i_{dest}$ do
17:     while b[i] is not empty do
18:         RELAX(b[i], relaxed, removed)
19:         $b[i] \leftarrow \emptyset$
20:         for each$(x, y) \in relaxed$ do in parallel
21:             INSERT(x, y)
22:             if $x_{end} = x \land y_{end} = y$ then
23:                 $i_{dest} \leftarrow i$
24:             end if
25:         end for
26:     end while
27:     $relaxed \leftarrow \emptyset$
28:     $removed \leftarrow \emptyset$
29: end while
30: $x \leftarrow x_{end}$   ▷ Reconstruct the path from distance.
31: $y \leftarrow y_{end}$
32: $path \leftarrow []$
33: while $x \neq 0 \land y \neq 0$ do
34:     $x_{pred} \leftarrow x$
35:     $y_{pred} \leftarrow y$
36:     for $dx \leftarrow -1, 0, 1$ do
37:         for $dy \leftarrow -1, 0, 1$ do
38:             $x_{new} \leftarrow x + dx$
39:             $y_{new} \leftarrow y + dy$
40:             if $(x_{new}, y_{new})$ is valid coordinate $\land$ $(x_{new}, y_{new}) \neq (x, y)$ then
41:                 if $distance[x_{new}][y_{new}] < distance[x_{pred}][y_{pred}]$ then
42:                     $x_{pred} \leftarrow x_{new}$
43:                     $y_{pred} \leftarrow y_{new}$
44:                 end if
45:             end if
46:         end for
47:     end for
48:     $pathprepend(x_{pred}, y_{pred})$
49: end while
50: return path
```

Algorithm 2 Functions of Algorithm 1

```
function FindDelta( )
    for $x \leftarrow 0$ to $x_{end}$ do in parallel
        for $y \leftarrow 0$ to $y_{end}$ do in parallel
            $cost_{max} \leftarrow \textsc{Max}(cost_{max}, \textsc{Cost}(x, y))$
        end for
    end for
    return $cost_{max}/8$
end function
function Remove(x, y)
    if $bmap[x][y] \neq -1$ then
        $b[bmap[x][y]] \leftarrow b[bmap[x][y]] \setminus (x, y)$
    end if
end function
function Insert(x, y)
    $i \leftarrow \lfloor distance[x][y]/delta \rfloor$
    $bmap[x][y] \leftarrow i$
    $b[i] \leftarrow b[i] \cup (x, y)$
end function
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.
    for each $(x, y) \in$ to-relax do in parallel
        if $removed \neq NULL$ then
            Remove(x, y)
            $removed \leftarrow removed \cap (x, y)$
        end if
        for $dx \leftarrow -1, 0, 1$ do
            for $dy \leftarrow -1, 0, 1$ do
                $x_{next} \leftarrow x + dx$
                $y_{next} \leftarrow y + dy$
                if $(x_{next}, y_{next})$ is valid coordinate $\wedge$ $(x_{next}, y_{next}) \neq (x, y)$ then
                    if $(\textsc{Cost}(x_{next}, y_{next}) \leq delta) \oplus (removed = NULL)$ then    ▷ If removed is null, only relax heavy edges, otherwise only relax light edges.
                        $distance_{new} \leftarrow distance[x][y] + \textsc{Cost}(x_{next}, y_{next})$
                        if $distance_{new} < distance[x_{next}][y_{next}]$ then
                            $distance[x_{next}][y_{next}] \leftarrow distance_{new}$
                            $relaxed \leftarrow relaxed \cap (x_{next}, y_{next})$
                        end if
                    end if
                end if
            end for
        end for
    end for
end function
```

6774. DOI: https://doi.org/10.1016/S0196-6774(03)00076-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403000762.