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 following pseudocode:

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 find_ delta 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] = true-value. Here true value is an int. By incrementing true-value by 1 we can empty the set very quickly.

Methodology

To confirm and measure the performance of my algorithm, I've

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: 0196-6774. DOI: https://doi.org/10.1016/S0196-6774(03)00076-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403000762.

$x_{end} \leftarrow x_{size} - 1$
$y_{end} \leftarrow y_{size} - 1$
$delta \leftarrow$ 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 process.
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
▷ Reconstruct the path from distance.
$x \leftarrow x_{end}$
$y \leftarrow y_{end}$
$path \leftarrow []$
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

```
function FindDelta( )
    for x ← 0 to x_end do in parallel
        for y ← 0 to y_end do in parallel
            cost_max ← Max(cost_max, Cost(x, y))
        end for
    end for
    return cost_max/8
end function
function Remove(x, y)
    if bmap[x][y] ≠ −1 then
        b[bmap[x][y]] ← b[bmap[x][y]] \ (x, y)
    end if
end function
function Insert(x, y)
    i ← ⌊distance[x][y]/delta⌋
    bmap[x][y] ← i
    b[i] ← b[i] ∪ (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) ∈ to-relax do in parallel
        if removed ≠ NULL then
            Remove(x, y)
            removed ← removed ∩ (x, y)
        end if
        for dx ← −1, 0, 1 do
            for dy ← −1, 0, 1 do
                x_next ← x + dx
                y_next ← y + dy
                if (x_next, y_next) is valid coordinate ∧ (x_next, y_next) ≠ (x, y) then
                    if (Cost(x_next, y_next) ≤ delta) ⊕ (removed = NULL) then    ▷ If removed is null, only relax heavy edges, otherwise only relax light edges.
                        distance_new ← distance[x][y] + Cost(x_next, y_next)
                        if distance_new < distance[x_next][y_next] then
                            distance[x_next][y_next] ← distance_new
                            relaxed ← relaxed ∩ (x_next, y_next)
                        end if
                    end if
                end if
            end for
        end for
    end for
end function
```