Faculty of Science

Parallelization of the cost distance algorithm

Master Thesis

Ewout van der Velde - 6509223

Applied Data Science

Supervisors

Prof. dr. Derek Karssenberg
Physical Geography

Dr. Kor de Jong
Physical Geography

Dr. Oliver Schmitz
Physical Geography

07-07-2023

Abstract

Cost distance tool are embedded in various geographic information systems (GIS) giving insights into spatial relationships. Most GIS software use a serial cost distance algorithm. Cost distance calculations have a strong sequential nature due to the order we access cells in the raster. This limits the development of a parallel algorithm, hindering the usage of multiple processes for faster computation. This paper proposes a parallelization framework, accounting for the sequential nature while running in parallel. By dynamically distributing partitions to different processes, we ensure full workload distribution and minimising idle times for processes. Relative strong and weak scaling efficiencies drop below 80% when run with more than 3 and 2 workers respectively. We notice that this is partly caused by the fact that the size of the input data and the amount of work a worker has to perform, do not scale linearly. When scaling to more workers, we expect to run into a performance bottleneck caused by input output operations of the root node. Recommendations are made for future research to limit the amount of input output operations by statically assigning partitions to workers.

Contents

1 Introduction

Distance plays a critical role in various geographic systems (Eastman, 1987), offering insights into spatial relationships and enabling optimized decision-making. Understanding distances between objects such as schools and houses is essential for tasks such as site selection for new schools or finding the nearest school to a house. By minimizing distances, we can enhance efficiency, identify optimal locations, and streamline transportation networks (Douglas, 1994). In practice, we can represent the world as a raster (Olsson, 1985) and perform calculations on this raster to determine the shortest path between points. Cost distance analysis, also known as cost path analysis, is a tool embedded in most geographic information systems (GIS) (ArcGIS, 2023; GRASS, 2023; QGIS Development Team, 2009; Karssenberg et al., 2010) that enables the determination of optimal routes or paths based on cost considerations. Building on the distance calculations in geographic systems, cost distance analysis also considers factors next to the physical distance between locations. In cost distance analysis, these additional factors are in the form of a cost raster where we can assign different costs to cells in a raster. These costs can be derived from landscape characteristics such as slope, land cover types, or other attributes relevant to an analysis. By considering these costs together with the distances between cells, we can make a quantification of the total cost required to travel between locations on a raster.

While multiple algorithms exist for calculating cost distance (Eastman, 1987), many GIS platforms rely on the approach proposed by Douglas (1994). Algorithms like that of Eastman (1987) treat the cost raster as a network problem, restricting movement between cells to predefined paths. These movements are orthogonal, diagonal, and sometimes a knight move (GRASS, 2023). In contrast, Douglas' algorithm treats the raster as a continuous field, allowing movement in continuous space. By embracing the continuous field approach, we can obtain more realistic results for various applications, as real-world space is continuous as well. It is worth noting that working with a continuous field introduces longer computation times compared to discrete field approaches.
However, one major limitation of these existing cost distance algorithms is their sequential nature. Like the Dijkstra shortest path algorithm, these algorithms calculate the minimal accumulated cost distance to a cell in order from lowest possible cost to highest possible cost. This dependency to know the global minimum of non-calculated cells can pose a problem when parallelizing the algorithm. When parallelizing an algorithm, the aim is to speed up computation time by distributing the workload over multiple processes. The challenge for us is that processes do not share resources and we thus cannot know the global minimum accumulated cost of non-calculated cells.

To the best knowledge of the author, only one proposal for a parallel cost distance algorithm has been made in the past. Wang et al. (2013) divides the raster into p partitions and distributes them over p processes. This method has multiple drawbacks, for instance when a partition is made up of no-data values, it does not do any work and is idling, waiting for calculation in other processes to finish.
We propose and evaluate a new parallel algorithm that dynamically distributes partitions to different processes, making sure no process is idling unnecessarily.
In this paper we aim to answer the following questions: How can a scalable parallel cost distance algorithm be developed with no processes idling unnecessarily? How does this algorithm scale? What are the potential bottlenecks when scaling the algorithm?
We developed a parallelization framework that dynamically distributes partitions to different processes, and is able to use existing serial algorithms in a parallel environment. We find that the algorithm does not have good weak relative scalability for multiple worker processes. The poor weak relative scalability is mainly caused by the fact that the workload per worker does not scale linearly with the raster size. Depending on partition size, the strong relative scalability is above or around 80%, for up to 4 worker processes. A potential bottleneck when scaling further will be the dependency on I/O operations by the root node, used for storing intermediate states of the accumulated cost raster.

2 Methods

2.1 Cost Distance Algorithm

Distance

Cost distance builds on the concept of distance calculations. When computing distances on a computer, we can project the world onto a raster and perform distance calculations on this raster. We have multiple methods to calculate distances on a raster. We will use one of the simpler methods where we can only move one cell at a time, either orthogonal or diagonal.

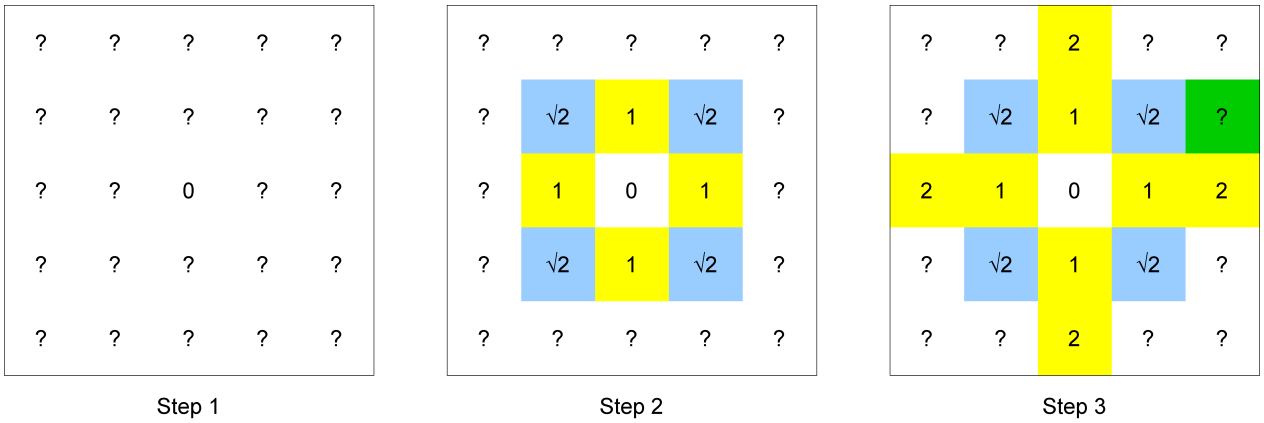


Figure 1: Example of distance calculations on a raster.

If the raster cells are the size of a unit cell, moving to raster points directly next to each other, the distance traversed will be 1. If we were to move diagonally, according to the Pythagorean theorem the distance would be $\sqrt{2}$. Using this information, we can, for each cell in the grid, determine how far away they are from a designated source cell by accumulating the distances. To do this we start at a source cell. If we want to know the total distance to travel to this cell, the answer is simply 0, since we are already at this cell (see figure 1). For its direct neighbours (yellow), the total distance to move to the source will be 1. For its diagonal neighbours (blue), the total shortest distance to the source cell will be $\sqrt{2}$. For the green cell it gets more complicated as there are multiple paths to get here. From the source we can go diagonal and then right once, right once and diagonal, or right twice and then up. For the total accumulated distance we then get $\sqrt{2}+1$, $1+\sqrt{2}$ or $1+1+1$ respectively. We see that, although the first two paths have the same total distance, the minimum distance from the source cell to the green cell equals $1+\sqrt{2}$. With this method we can eventually calculate the minimum distance from a source cell to all other cells.

Cost distance

If we were to use these distance calculations in the real word, we quickly find that the results are not always useful. Although we may know the minimum distance, we do not know if the path to get there is actually good. This path might try to traverse untraversable terrain. In cost distance analysis we encapsulate extra information in our calculation in the form of an additional raster. This raster could be a height map, terrain map or any other user defined feature map. Each cell in this raster represents the "cost" to be in that cell. If we add this cost raster, we can calculate the cost distance (CD) to travel between neighbouring cells. The cost to move between cell i to j can be calculated with

$$\mathrm{CD}_{i,j} = \frac{\mathrm{cost}_i + \mathrm{cost}_j}{2} \times d_{i,j}$$

where $cost_i$ the cost of cell i, $cost_j$ the cost of cell i and $d_{i,j}$ the distance between cell i and j.
Since we are interested in the minimum accumulated cost, and not just the cost distance between two cells, we calculate the minimum of the cost distances to move between each neighbouring cell added to their

accumulated cost distance. We refer to this as the accumulative cost distance ACD and can in practice be calculated with

$$\text{ACD}_i = min\left\{CD_{i,j} + ACD_j\right\} | j \in N$$

where ACD_i the accumulated cost distance at cell i, ACD_j the accumulated cost distance at cell j, and N the set of all neighbouring cells.
The same way as with the distance calculations we can than calculate the ACD for each cell.

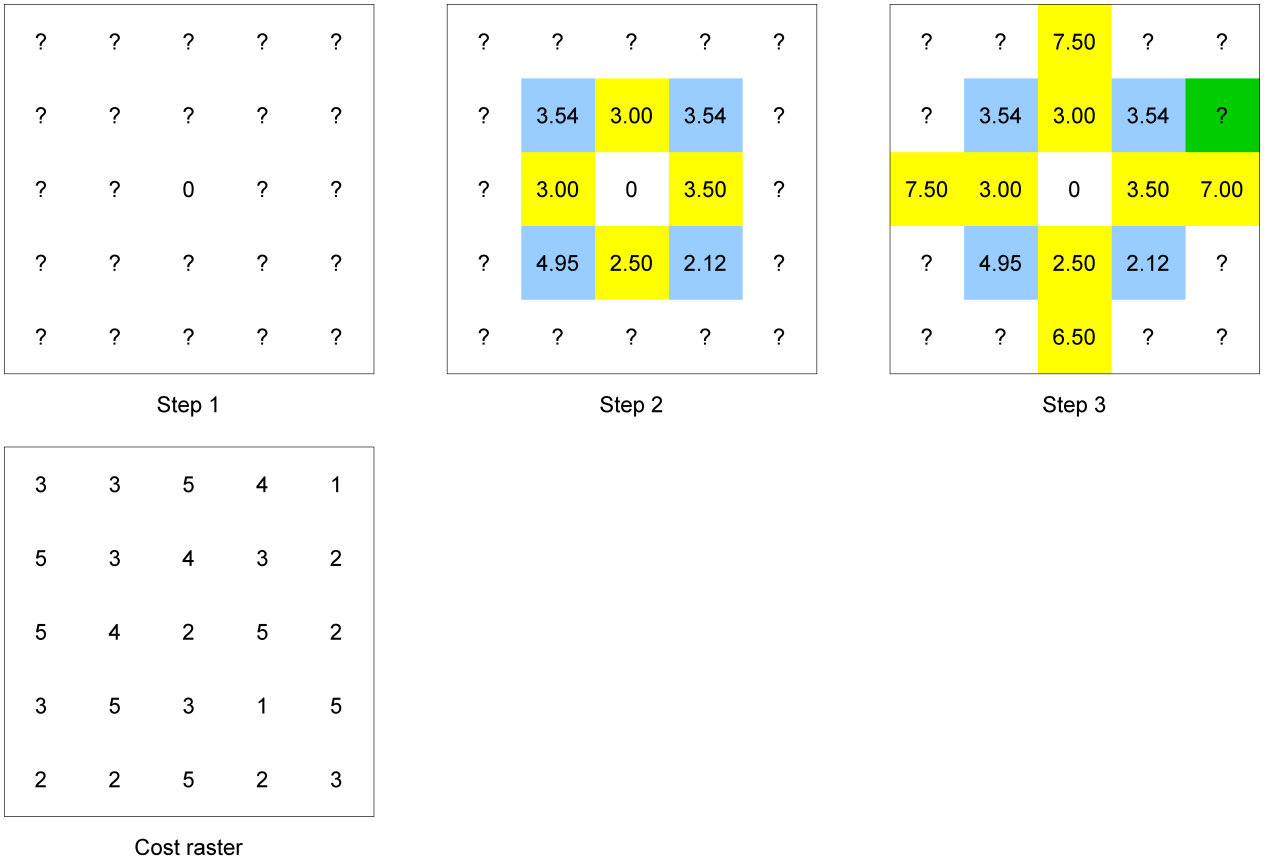


Figure 2: Example of cost distance calculations on a raster.

We see that in figure 2 step two is already different than in figure 1. This is caused by the addition of a cost raster. We are interested in the green cell, so we calculate the ACD it would take to get there from its neighbours. From the source cell to the green cell entering from the neighbour on its left it would cost $\frac{3+2}{2} \times 1 + 3.54 = 6.04$, from the bottom neighbour $\frac{2+2}{2} \times 1 + 7.00 = 9$ and from diagonal $\frac{5+2}{2} \times \sqrt{2} + 3.50 = 8.45$. We see that the cheapest route would be to enter from the left, giving us an ACD of 6.04 for the green cell.

2.2 Parallelization

Sequential or serial programs run on a single process on a single core. Serial algorithms work step by step in a sequential fashion, enabling the usage of results obtained in calculations made before the current calculation. The main drawback with serial algorithms is that they do not use the full computational power of modern machines which have multiple cores. To use the full computational power, we need to run an algorithm in parallel. This way we can use all cores of the processor to perform computations, theoretically decreasing the run time of an algorithm.

Currently there are multiple ways to calculate the ACD in a relatively efficient way for the entire raster (Eastman, 1987; Douglas, 1994). The most popular one used in GIS software is a serial algorithm based on

that of Douglas (1994). To run efficiently, these algorithms make smart use of data structures such as heaps and priority queues to keep track of the order cells need to be processed in. Similar to the Dijkstra Path finding algorithm, we start by assigning an ACD of 0 at the source locations, and an ACD of infinity to all points in the raster. For the cell with the lowest ACD, we calculate the ACD for its neighbouring cells. If the calculated ACD is lower than the already stored value in the ACD raster, we update the raster to the new value. Since we know that the ACD can only go up when moving between cells, and we know that the current cell already had the lowest ACD, we know there is no cheaper way to get to the current cell. Therefor we can mark this cell as checked and we do not have to visit this cell again. We now move to the next lowest ACD and do the process over again. Eventually we have checked all the cells and know that we have found the global solution for the minimum ACD. In practice this means that we access the raster in a non-structured, random-access pattern. This means point by point where points do not have to be close to each other.

Standard parallelization techniques usually involve dividing the raster into structured partitions and distributing these partitions to dedicated processes. If we were to take his route, we are posed with the problem that the Dijkstra like algorithms does not work anymore. Each process can only access its own partition, and we cannot be certain if the next lowest ACD in the partition is the lowest possible ACD for this cell, as there might be a cheaper route possible when moving from another partition. This means that the local solution of a partition is not necessarily the global solution for that partition. This gives us the challenge that we cannot calculate the local solution for each partition and assume we have a global solution and stop calculations. If we want a global solution we have to make corrections to the partitions after we made calculations for the partitions.

A current proposal for a parallel algorithm for cost distance analysis is that of Wang et al. (2013). Wang divides the raster into P partitions, where P the number of processes, and assigns these to the processes to calculate local ACD and makes correction in a final pass through. The drawback of this method is that it may be possible for a process to wait until a neighbouring partition is done with an initial calculation before it can start doing work on its own partition. This could happen when source cells are unevenly distributed over the raster or when we have a lot of no-data values in parts of the raster. Assigning processes to partitions with only no-data values results in processes doing no calculations at all.

This paper will take an approach based on that of Wang et al. (2013). We will develop a parallelization framework, allowing the user to use regular serial algorithms in a parallel setting. To work around the drawback of the algorithm by Wang et al. (2013), we chose to divide our raster in $> P$ partitions, and dynamically assign partitions to processes whenever they can be calculated.
As mentioned before, in the serial algorithm it was relatively easy to know when we found a global solution for our ACD raster. To keep track of the status of the calculations, we use a root node. This root node registers which processes are busy and have found a local solution for a partition. This root node is also in charge of I/O operation on the accumulated cost raster to prevent concurrent writing of the raster by multiple processes. From here on, we will refer to all processes that are not the root node, as workers.
We can only start the ACD calculations at source cells since it is the only place where we know the accumulated cost of. Each worker searches predefined partitions of the source raster for source cells. If a partition contains at least one source cell, we send the location of the partition to the root node to store in a list, now referred to as list k.
After all workers have gone through the predefined partitions, the workers can request a partition location and the corresponding partition of the ACD raster from the root node.
The worker reads the source raster and cost raster from file for its own partition including a buffer to calculate costs on the boundary of the partition.
We run a serial ACD algorithm on the partition. If we find that there is a cell update of the ACD on the edge of the partition, we keep track of which side of partition this update was on. When the serial algorithm is done, we add the neighbouring partitions from the edges that had updates to a list. We send this list to the root node together with the ACD raster.
The root node will add the received neighbouring partitions to list k if they are not already in the list. If list k is empty and all workers are waiting for a new partition, we have found the global solution and terminate the program. If not, we hand out partitions of list k to workers waiting for a new partition.

One major drawback of dynamically handing out partitions is that we need to save results intermediately. This is because we do not know which process will (re)calculate which partitions, making it not possible to store intermediate results in the memory of a process. This does mean that we probably spend much time with I/O and sending data from one process to the root and back for the I/O operation.

Pseudocode

```
Define padding size
Define partition size
Load metadata from raster
Create a list with all partition locations

Root node:
    Initialize:
        Make cumulative cost raster same shape as cost raster with infinite costs
        Wait for all other workers to send partition locations with source
          cell(s) and add these to a list of partitions to update
        Create an empty list with partitions working on
        Create queue with available workers

    In main loop:
        If all workers are in the queue and there are no partitions to update:
            Send message to all worker to terminate
        In handout loop:
            If worker available queue empty:
                Break handout loop
            Valid partitions is partitions to check - partitions working on
            If no valid partitions:
                Break handout loop

            Get worker from queue
            Send partition from valid partition to worker
            Add partition to the list with partitions working on

        Wait worker to send message
        Save received partition to main raster
        Remove partition from set with partition currently worked on
        Add worker back to worker available queue
        If message has new partition locations:
            add received partition to partitions to update

Worker node:
    Initialize:
        Read a predefined partitions of the raster to find if they contain source cells
        Send partition location where there are source cells to the root node

    In main loop:
        Wait for partition from the root node to work on
        If partition location is termination command:
            Terminate process
        Read in partition location with padding from source and cost raster
        If partition location is on edge of global raster:
            Pad the partition to be a square
  Do serial cost distance on the partition
```

```
        If the cumulative raster has changed:
            Add neighbours of the partition to a list
    Send the cumulative raster with neighbour list to the root node
```

A flowchart representation of the algorithm can be found in figure 3.

2.3 Implementation

The algorithm is coded in Python and Message Passing Library (MPI) (Message Passing Interface Forum, 2021) is used to facilitate communication between processes. We setup one root node and multiple workers. We have chosen to divide the raster into square partitions. This was due to the practical reason that squares are easier to work in code with than rectangles. The algorithm should also work on rectangular partitions.
If the cell is on a boundary from the global raster, we pad the boundary of the partition with infinite costs, this is again to make sure we are only working with squares.
The serial algorithm used by the workers is inspired on the Dijkstra shortest path algorithm. The Python code of the parallel algorithm can be found on the authors' GitHub (Velde, 2023).

2.4 Evaluation

CPU	Intel Core I7-8750H
Cores	6
Logical processors	12
CPU base frequency	$2.2GHz$
CPU load frequency	$\sim 4GHz$
RAM	$16GiB$
OS	Windows 11 Home 22H2
Python	3.10.9
MPI	mpi4py

Table 1: Hardware and software used for the experiments.

We perform experiments to characterise the performance and scalability of the parallel cost distance algorithm. Testing is done on a laptop with properties described in table 1. There are two major limitations of this setup.

1. We cannot use all of the CPU's capabilities since Windows is running background processes next to our tests.
2. The CPU has a variable clock speed. The system will dynamically change its clock speed according to the load it is given. At a high load it goes to a stable high of $\sim 4GHz$, but when the load is low it may throttle back to the base speed of $2.2GHz$.

The goal of the scalability experiment is to determine how well the algorithm is able to make good use of additional resources. In our case the additional resources are in the form of adding more processes.

We evaluate two kinds of scalability: strong scalability and weak scalability. Strong scalability is relevant if you are interested in speeding up an existing model. A model that has a high strong scalability can make good use of additional resources to make it finish faster (Jong et al., 2022).
An effective way to evaluate strong scalability is with the relative strong scalability efficiency (RSE_{strong})

$$RSE_{strong} = \frac{T_{S,1}}{P \times T_{S,P}} \times 100\% \tag{2.1}$$

where P the number of processes, $T_{S,1}$ the latency when using one process and $T_{S,P}$ the latency when ran with P processes while keeping the problem size constant.

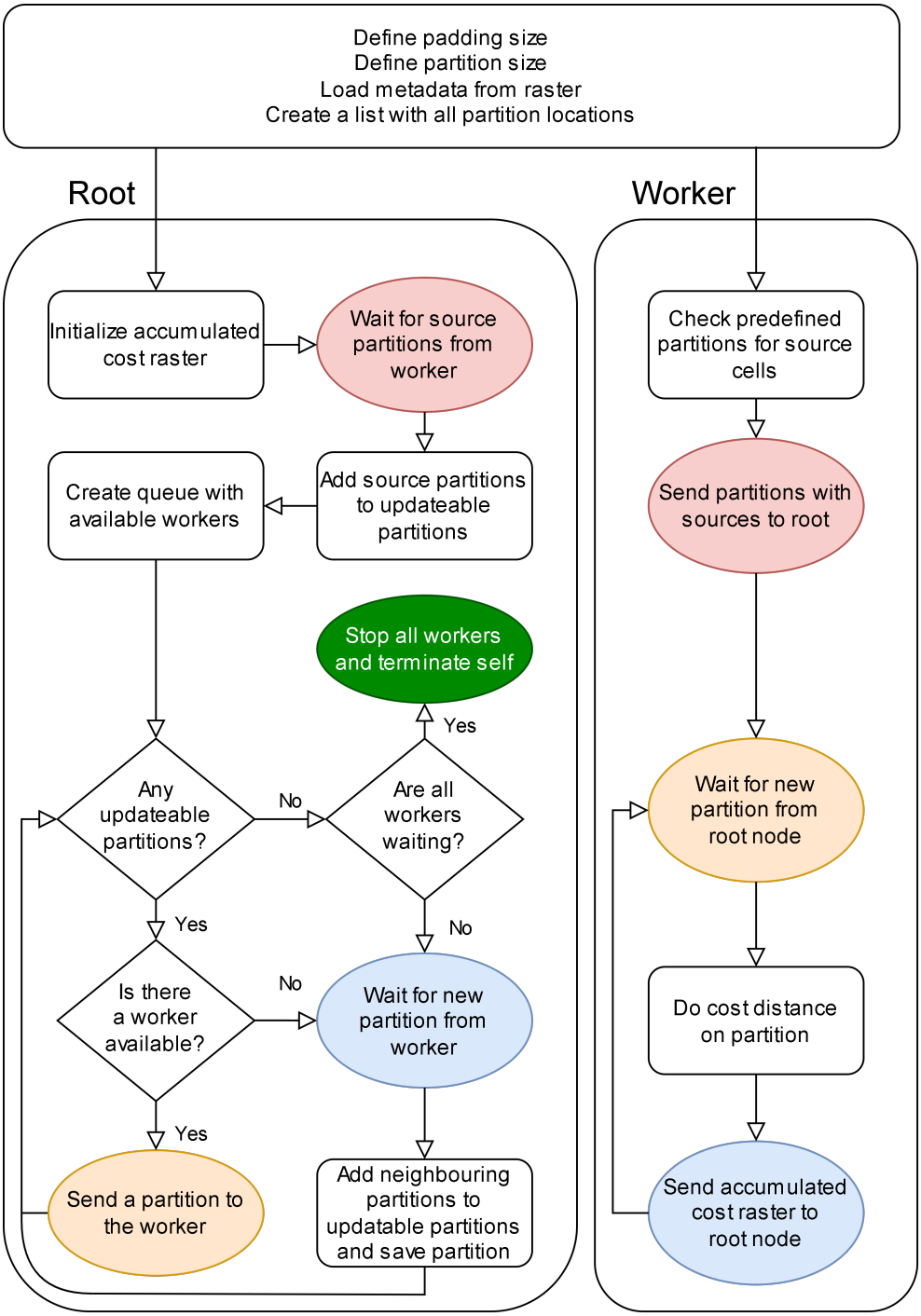


Figure 3: Flowchart of the proposed algorithm. Coloured ellipses are communication points between the root node and the workers.

With this definition we run into a problem. If we were to use only one process, we would have only a root node or only a worker and the algorithm would never finish. For now we will disregard the root node as a process, as it does no calculations on any partition, and calculate RSE_{strong} with the number of workers instead the number of processes. The fact that we need a minimum of two processes is a drawback of the algorithm, and we will reflect on the choice to ignore the root node as a process in the discussion.
For the strong scalability evaluation we use a raster of 3000x3000 cells. The cost raster is randomly generated, and the source raster contains 20 source cells. We evaluate RSE_{strong} for three different scenarios.

1. Uniform distribution of costs and uniform distribution of source cells.
2. Non-uniform distribution of costs and uniform distribution of source cells. Costs in lower half of the raster are significantly higher than in top half.
3. Uniform distribution of costs and non-uniform distribution of source cells. All source cells are in the lowest ten percent of rows.

We evaluate with partition sizes of 250, 500 and 1000 as we expect partition size to have influence on the scalability.
To mitigate the variance in the latency caused by the variable clock speed of machine where we run the tests on, we run each experiment five times and use the average latency.

By measuring weak scalability, we characterise how well the algorithm manages a higher total workload with additional processes.
To measure the weak scalability, we scale the overall raster size with the number of processes we run the algorithm with. If we were to double the number of processes, we would double the number of raster cells as well. To calculate the weak scalability efficiency, we use

$$RSE_{weak} = \frac{T_{W,1}}{T_{W,P}} \times 100\% \tag{2.2}$$

where $T_{W,1}$ the latency to run the algorithm with only one process, and $T_{W,P}$ the latency when ran with P processes on a raster that is P times as large as the one used for $T_{W,1}$.
We run into the same problem as with RSE_{strong} where one process gives no result. We again will disregard the root node as a process and focus on the number of workers.
For the weak scalability we start with a raster of 1000x1000 with 20 randomly generated source cells. For each worker added, we add 1000 rows and 20 sources to the raster. We evaluate with the same three different raster scenarios as with RSE_{strong}.

1. Uniform distribution of costs and uniform distribution of source cells.
2. Non-uniform distribution of costs and uniform distribution of source cells. Costs in lower half of the raster are significantly higher than in top half.
3. Uniform distribution of costs and non-uniform distribution of source cells. All source cells are in the lowest ten percent of rows.

Again, we do this five times and use the average latency.
Both the strong and weak scalability experiments will be run with a minimum of two processes and a maximum of nine. Two processes are the minimum number of processes for the algorithm to run, and during preliminary testing we found nine to be the maximum to not over saturate the CPU when ran combined with Windows background processes.
Although it is normal to exclude time in I/O operations from the scalability testing, it is included in these tests, as I/O plays a vital role in transferring information between processes and expect this to be a limitation of the algorithm itself.
One assumption that we make in the weak scalability test is when we increase our raster size linearly with the number of workers, we keep the workload per worker equal. Since our parallelization method can call for updates on partitions that are already calculated, we expect that this assumption will not hold. To obtain

an insight in how much this may affect weak scalability, we track the number of partitions (re)calculated per worker. To quantify these findings, we calculate the relative workload (RW) with

$$RW_P = \frac{N_1 \times P}{\sum_{i=1}^{P} N_i} \tag{2.3}$$

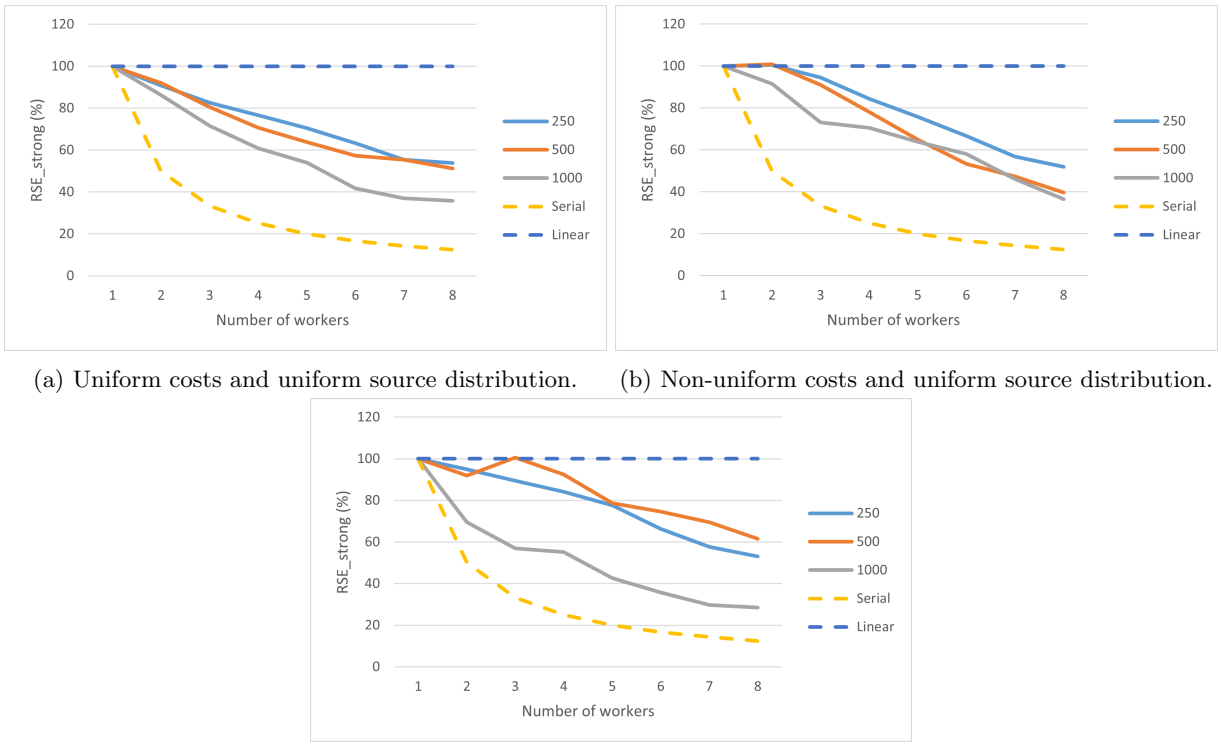
where P the number of workers, RW_P the relative workload for P workers, N_1 the workload for worker 1 when run with a total of 1 worker, and N_i the workload of worker i when run with P workers. If the workload scales linearly we would expect RW to be equal to 1, if the workload per worker becomes higher, we expect that $RW > 1$. Additionally, we will also track how many times a cell needs to be (re)calculated by the algorithm. Both the RW and number of (re)calculations of cell will be calculated with a partition size of 100 for 3 different scenarios.

1. Uniform distribution of costs and uniform distribution of source cells.
2. Non-uniform distribution of costs and uniform distribution of source cells. Costs in lower half of the raster are significantly higher than in top half.
3. Uniform distribution of costs and non-uniform distribution of source cells. All source cells are in the lowest ten percent of rows.

We stated before that the root node of the algorithm will not be counted as a process, as it does not perform any computations on the raster itself. The work that it does, for the limited number of processes used, is expected to be negligible compared to the workload of the other processes. To check if the impact of the claim that the work done by the root node is negligible, we profile the root node to reflect on the validity of our RSE results in the discussion.

3 Results

For the evaluation of RSE_{strong} for variable number of worker processes at different raster arrangements, we find the result in figure 4. For uniform cost and uniform source cell distribution we see that for partition sizes 250 and 500, the RSE_{strong} is high at 80% for up to three workers, while a partition size of 1000 only has a high efficiency at two workers. For more workers, we see that the RSE_{strong} declines. The scalability increases with non-uniform costs and uniform source cell distribution for partition sizes 250 and 500 and remains high when ran with up to 4 workers. For the non-uniform distribution of source cells, we see that a partition size of 1000 has poor scalability in general, while for 250 and 500 it has high scalability for up to 5 and 6 workers.
The average latency for each number of workers for RSE_{strong} can be found in appendix table 2.



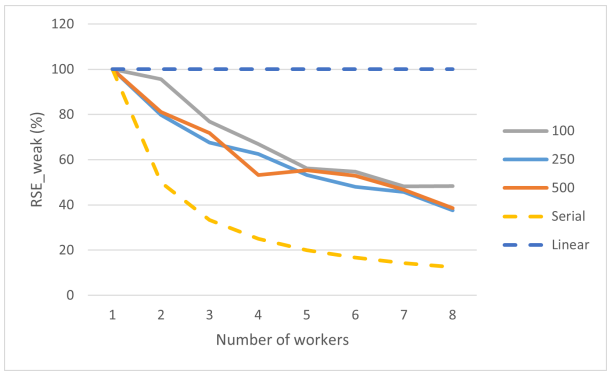
(a) Uniform costs and uniform source distribution.

(b) Non-uniform costs and uniform source distribution.

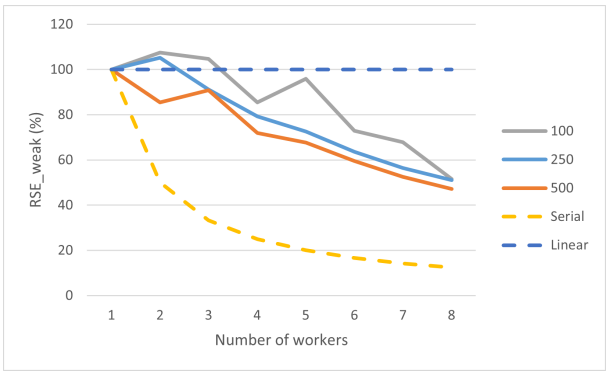
(c) Uniform costs and non-uniform source distribution.

Figure 4: Strong relative scaling efficiency per number of workers when run at a raster size of 3000x3000 for different partition sizes. Dotted lines are linear scaling and serial scaling added for reference.

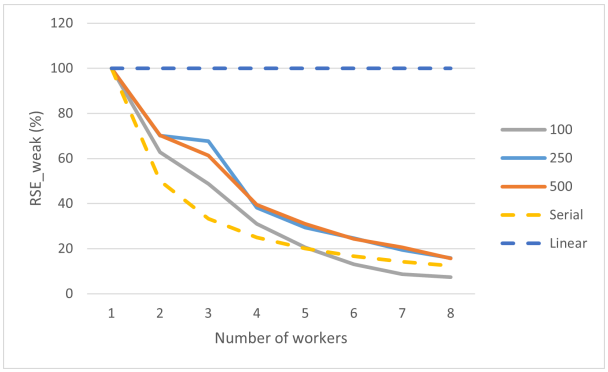
For the evaluation of RSE_{weak} for variable number of worker processes at different raster arrangements, we find the result in figure 5. For uniform cost and uniform source distribution, we notice that RSE_{weak} for all partition sizes rapidly declines. We see that the scalability is high at 80% for two workers. After this the scalability drops for all partitions. For non-uniform costs and a uniform source cell distribution we find that RSE_{weak} gets better compared to the other scenarios. With uniform costs and non-uniform source cell distribution, we find that the scalability is worse and has near serial like scalability.



(a) Uniform costs and uniform source distribution.



(b) Non-uniform costs and uniform source distribution.



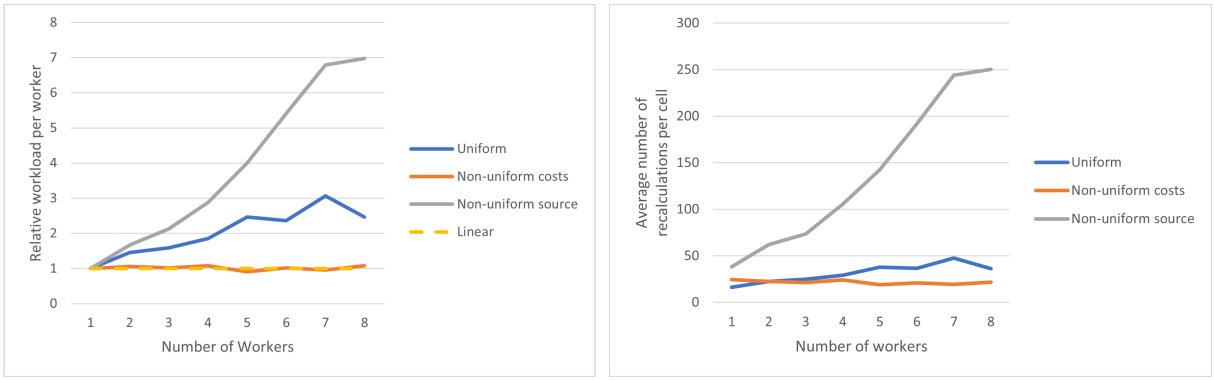
(c) Uniform costs and non-uniform source distribution.

Figure 5: Weak relative scaling efficiency per number of workers when run with a raster size linearly scaled to the number of workers, keeping the workload per worker equal in theory. Dotted lines are linear scaling and serial scaling added for reference.

The average latency for each number of workers for RSE_{weak} can be found in appendix table 3.

To check whether the workload increases linearly with the raster size, we see in figure 6a the RW. Raster size is scaled linearly with the number of workers. If the workload were to scale linearly with the number of workers, we would see that the RW stays constant. We find the RW increases when increasing the raster size for both uniform costs and uniform source cell distribution, as well as for uniform costs and non-uniform source cell distribution. The RW scales linearly when using a non-uniform costs raster with a uniform source cell distribution. Figure 6b shows similar results, as the number of cells (re)calculated is closely related to the number of partitions (re)calculated. Both results used for figure 6a and 6b can be found in appendix table 4 and 5 respectively.

To argue whether the root node can be handled as a non-process, we profiled the root node. Results can be seen in figure 7. We see that a lot of time is spend in the MPI recv function. This time also includes the time that the function is waiting for a MPI send from a worker. We estimate the time it took to actually receive data to be equal to the MPI send of the root, as these communicate nearly an identical amount of data. We see when adding more workers, relatively more time of the root node is spend not waiting and an increasing amount of time is spent in GDAL functions. The rasterIO and Flushcache are both used in reading and writing rasters to files.



(a) Relative workload per worker as calculated with equation 2.3. Dotted line is linear scaling added for reference.

(b) Average number of recalculated cells.

Figure 6: Increase in workload per worker during RSE_{weak} evaluations at a partition size of 100.

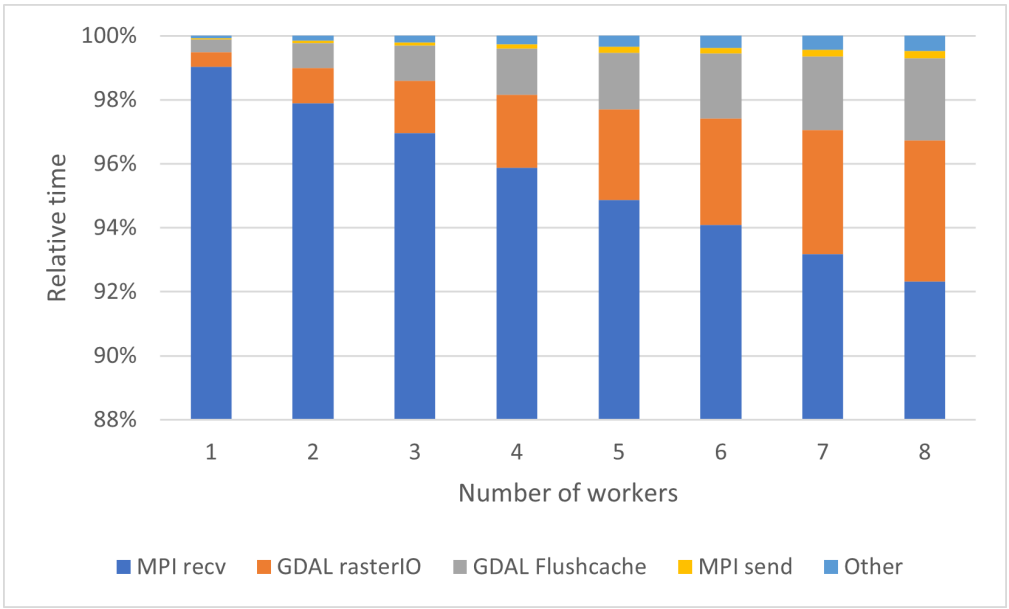


Figure 7: Relative time in function calls by the root node. Note the y-scale does not start from 0 to make it possible to read of the small fractions of time spend in functions at lower numbers of workers.

4 Discussion

As seen in figure 4, RSE_{strong} for a uniform cost distribution and uniform source cell distribution drops below 80% when having more than 4 workers. We note that the decline is stronger for a partition size of 1000. This is to be expected for our raster, as when having a partition size of 1000 on a 3000x3000 raster grid, means that we can have a maximum of 9 partitions and workers. This has the effect that workers will spend a lot of time idling waiting for neighbouring partitions. The assumption that we make from these results, is that a partition size of 1000 is too large for the relatively small size of our raster. For further research it would be a good idea to test the partition sizes on a larger raster, to see if this effect decreases.
For a non-uniformly distributed cost raster with a uniform distribution of source cells, we find that RSE_{strong} goes up for all partition sizes. This is to be expected as we have little chance of costs travelling in the raster from the high-cost part to the low-cost part, causing recalculations of cells.
The results for the uniform costs and non-uniform source cell distribution, affirm our assumption that a partition size of 1000 is too large for a 3000x3000 raster. We see that RSE_{strong} is plainly bad for this partition size. This is to be expected as all source cells are in the lowest 10 percent of rows, the 300 bottom rows. Meaning that in the first iteration of our algorithm, only a maximum of 3 workers can start with calculations, instead of all workers. This causes all other workers to wait for the first iteration to finish, before we can go on to the next iteration.

The results in figure 5 shows that we have in general poor RSE_{weak} for both uniform costs and source cell distribution, as well as for uniform costs and non-uniform source cell distribution. In comparison, the uniform source cell distribution with non-uniform costs yields good RSE_{weak} up to 3, 4 and 5 workers for partition sizes of 500, 250 and 100 respectively. It is understandable that RSE_{weak} decreases when given more workers and a larger raster size when the sources are non-uniformly distributed, as all sources are clustered and chances are higher that we need to do more recalculations of cells when moving further away from the source cells. As we get further away the accumulated cost becomes more uniform, causing more recalculations due to neighbours that have slightly lower accumulated costs. This is confirmed in figure 6b, where we see more recalculations are needed when source cells are clustered. In contrast, non-uniform cost distribution with uniform sources achieves better RSE_{weak}. This is also to be expected as we will have less recalculations, caused by the same reason as why the scalability was bad for non-uniform sources. The accumulated costs also become less uniform and thus the chance at recalculations of cells becomes smaller.

We expected in section 2.4 if we were to increase the raster size, the total workload would increase not at the same, but a higher rate, increasing the workload per worker. We can clearly see that this is the case in figure 6a. This result is caused by a fundamental part of the algorithm in its current form where we randomly hand out partitions to workers and thus cannot be mitigated by tuning input parameters of the algorithm. In future work, one could try to implement a sorting structure where we store the maximum ACD of a partition. If we hand out partitions with lowest maximum ACD first, we decrease the chance that an updated partition has to spread to far back into already calculated raster partitions. This method has resemblance with how we currently calculate the ACD in the serial part of the algorithm. Using the fact that the ACD can only increase when moving between cells, this method lowers the number of recalculations per partition, therefore lower the overall workload and therefore lower workload per worker. A downside of this approach is that we give the root node more tasks, which in turn means that it cannot use the time spend on this new task to communicate with workers, possibly letting workers idle unnecessarily.

The fact that we treated the root node as a non-process seems to be acceptable for our weak and strong scalability tests, given the results from figure 7. The root node spends most time waiting for other processes, meaning that the effect off the root node being busy, making a worker wait, has no significant effect on the overall performance of the algorithm. However, the claim does not work in general. We see that the amount of time spent doing I/O operations increases quite a bit when adding more workers. This is a clear indication that the root node will become a bottleneck when adding more workers, since it starts using more time doing things that are not communicating between processes. Therefore, one could easily argue that for this reason only we should always see the root node as a worker as well. As stated before, currently this would mean that we cannot calculate $T_{S,1}$ and $T_{W,1}$ from eq.(2.1) and eq.(2.2) respectively, as the algorithm would never finish with only a root node or worker node. A relatively easy way to solve this problem is to give the root node

the capability to also carry out cost distance calculations. This could be done in multiple ways, where the simplest would be to check the total amount of processes spawned, and if this is only one, solve everything in the owe process, and if more than one, use the algorithm in the current form. This would however mean that when using 1 or 2 processes, we would get no performance gain since we do the exact same amount of work with the same number of processes doing actual calculation. A better way would be to use multi-threading in the root node and let it both calculate ACD and be open to do communication with other workers. Both methods would make the calculations of the relative scalability efficiency fairer, as we are not excluding a process from the total number of processes.

For future research we recommend creating an algorithm that does not dynamically hand out partitions to workers. As discussed in the method section we chose to dynamically hand out partition to keep the downtime per process to a minimum. This however created a problem that we cannot store an in between state of the partition in memory, as we do not know to which worker we (re)send a partition to for (re)calculations. We also cannot store the total raster in memory of a single process, as the size could be too large to fit in memory. This meant we that had to introduce a way to communicate intermediate results, and the path chosen was to use I/O. In hindsight, the performance gain of keeping all processes busy probably does not weigh up to the increased amount of time spend in I/O operations. For the number of workers used in testing for this paper, I/O operations did not become a bottleneck, but when adding more workers, this bottleneck will become noticeable in performance. If we were to use statically assigned partitions, we can store a local copy of the partition in memory of that process, reducing the I/O operations to a minimum where we need to only read once and write once. Another benefit is that we do not have to send entire partitions between processes, but only the boundaries. When using smaller partitions as done in this paper, this is not the most important problem to solve. However, scaling to larger partitions, the amount of memory to store/send the raster increases exponentially and this will likely become a new bottleneck if we were to send entire partitions. For reference, the maximum partition size in this paper is 1000x1000 with is around $0.07GiB$, whereas a 20000x20000 raster is around $3GiB$. Sending only the boundary in the latter case means we only send 20000*4 = 80.000 cells between the process, compared to 20000*20000 = 400.000.000, which is a decrease of 0.02% of data to be send.
To conclude, we have shown that we could develop a parallel cost distance algorithm that minimized idling time of processes. This was done by dynamically distributing partitions. RSE_{strong} of the algorithm drops below 80% when run with more than three workers, while RSE_{weak} drops below 80% after when run with more than two workers. The low RSE_{weak} is to be expected since the workload per worker does not scale linearly when increasing the raster size. A potential bottleneck when scaling to more workers will be the amount of I/O by the root node, as was shown that relative time spend in I/O increases drastically when adding more workers.

References

ArcGIS (Feb. 2023). *Distance accumulation (spatial analyst)*. URL: https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/distance-accumulation.htm.

Douglas, David H. (1994). "Least-cost Path in GIS Using an Accumulated Cost Surface and Slopelines". In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 31.3, pp. 37–51. DOI: 10.3138/D327-0323-2JUT-016M. eprint: https://doi.org/10.3138/D327-0323-2JUT-016M. URL: https://doi.org/10.3138/D327-0323-2JUT-016M.

Eastman, J Ronald (1987). "Graduate School of Geography Clark University". In: *Worcester, MA, is in the process of analysing recent satellite images and secondary growth in the southern Yucatan peninsular region*.

GRASS (Feb. 2023). *GRASS manual*. URL: https://grass.osgeo.org/grass82/manuals/r.cost.html.

Jong, Kor de et al. (2022). "Scalability and composability of flow accumulation algorithms based on asynchronous many-tasks". In: *Computers & Geosciences* 162, p. 105083. ISSN: 0098-3004. DOI: https://doi.org/10.1016/j.cageo.2022.105083. URL: https://www.sciencedirect.com/science/article/pii/S0098300422000462.

Karssenberg, Derek et al. (2010). "A software framework for construction of process-based stochastic spatio-temporal models and data assimilation". In: *Environmental Modelling & Software* 25.4, pp. 489–502. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2009.10.004. URL: https://www.sciencedirect.com/science/article/pii/S1364815209002643.

Message Passing Interface Forum (June 2021). *MPI: A Message-Passing Interface Standard Version 4.0*. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

Olsson, Lennart (1985). "An Integrated Study of Desertification". English. Defence details Date: 1985-12-18 Time: 10:00 Place: Lund External reviewer(s) Name: Townshend, John Title: Reader Affiliation: University of Reading —. Doctoral dissertation.

QGIS Development Team (2009). *QGIS Geographic Information System*. Open Source Geospatial Foundation. URL: http://qgis.osgeo.org.

Velde, Ewout van der (June 2023). *Parallel cost distance algorithm*. Version 1.0.0. URL: https://github.com/EwoutvanderVelde/parallel_cost_distance.

Wang, Y et al. (May 2013). *Parallel Algorithm for Calculating Cost Distance of Raster Data*.

A Measurements

Size	Workers	Latency (s)
250	1	744.57
250	2	410.30
250	3	300.83
250	4	243.02
250	5	211.56
250	6	196.22
250	7	191.99
250	8	173.15
500	1	569.97
500	2	309.50
500	3	236.21
500	4	201.61
500	5	178.79
500	6	165.89
500	7	146.95
500	8	139.09
1000	1	407.48
1000	2	236.21
1000	3	189.85
1000	4	167.36
1000	5	150.89
1000	6	162.65
1000	7	157.03
1000	8	142.20

(a) uniform cost, uniform sources.

Size	Workers	Latency (s)
250	1	2049.24
250	2	1079.62
250	3	763.19
250	4	608.35
250	5	527.81
250	6	515.53
250	7	508.62
250	8	483.37
500	1	1057.32
500	2	575.63
500	3	350.61
500	4	286.05
500	5	268.53
500	6	236.05
500	7	217.47
500	8	214.90
1000	1	513.67
1000	2	370.08
1000	3	300.86
1000	4	233.12
1000	5	240.66
1000	6	239.86
1000	7	247.26
1000	8	255.39

(b) Uniform costs, non-uniform source.

Size	Workers	Latency (s)
250	1	830.50
250	2	413.31
250	3	292.67
250	4	246.14
250	5	219.11
250	6	207.74
250	7	209.10
250	8	200.16
500	1	560.92
500	2	278.31
500	3	205.50
500	4	179.63
500	5	172.92
500	6	175.27
500	7	169.36
500	8	176.72
1000	1	517.44
1000	2	282.51
1000	3	235.83
1000	4	183.62
1000	5	162.21
1000	6	148.67
1000	7	160.21
1000	8	177.26

(c) Non-uniform cost, uniform sources.

Table 2: Average latency of 5 runs per configuration at 3 different partition sizes for the RSE_{strong} evaluation. Total raster dimension are 3000x3000.

Size	Workers	Latency (s)
100	1	78.15
100	2	96.38
100	3	108.93
100	4	146.96
100	5	141.21
100	6	147.68
100	7	167.39
100	8	202.68
250	1	54.04
250	2	56.48
250	3	70.29
250	4	80.68
250	5	96.28
250	6	98.90
250	7	112.20
250	8	111.73
500	1	40.28
500	2	50.50
500	3	59.55
500	4	64.48
500	5	75.55
500	6	83.99
500	7	88.19
500	8	107.22

(a) uniform cost, uniform sources.

Size	Workers	Latency (s)
100	1	188.78
100	2	300.96
100	3	387.02
100	4	608.19
100	5	915.26
100	6	1446.53
100	7	2172.71
100	8	2568.36
250	1	95.76
250	2	136.69
250	3	141.51
250	4	250.69
250	5	325.17
250	6	387.55
250	7	493.18
250	8	605.74
500	1	51.52
500	2	73.29
500	3	83.89
500	4	130.40
500	5	165.87
500	6	211.90
500	7	250.49
500	8	327.66

(b) Uniform costs, non-uniform source.

Size	Workers	Latency (s)
100	1	115.5
100	2	107.56
100	3	110.42
100	4	135.30
100	5	120.52
100	6	158.53
100	7	170.42
100	8	224.08
250	1	65.96
250	2	62.66
250	3	72.39
250	4	83.22
250	5	90.88
250	6	103.60
250	7	117.02
250	8	129.10
500	1	54.00
500	2	63.22
500	3	59.50
500	4	75.07
500	5	79.83
500	6	90.60
500	7	102.70
500	8	114.60

(c) Non-uniform cost, uniform sources.

Table 3: Average latency of 5 runs per configuration at 3 different partition sizes for the RSE_{weak} evaluation.

Cells	Partitions (re)calculated	Average per worker
1000000	889	889
2000000	2580	1290
3000000	4244	1415
4000000	6598	1650
5000000	10934	2187
6000000	12578	2096
7000000	19093	2728
8000000	17529	2191

(a) uniform cost, uniform sources.

Cells	Partitions (re)calculated	Average per worker
1000000	4532	4532
2000000	15146	7573
3000000	28920	9640
4000000	52326	13082
5000000	90790	18158
6000000	146970	24495
7000000	215556	30794
8000000	252744	31593

(b) Uniform costs, non-uniform source.

Cells	Partitions (re)calculated	Average per worker
1000000	2076	2076
2000000	4402	2201
3000000	6346	2115
4000000	8944	2236
5000000	9462	1892
6000000	12668	2111
7000000	13964	1995
8000000	17841	2230

(c) Non-uniform cost, uniform sources.

Table 4: Number of partitions (re)calculated with average amount that a partitions (re)calculated per worker for 3 raster scenarios. A partition size of 100 was used.

Cells	Cells (re)calculated	Average times cell (re)calculated
1000000	16431764	16.4
2000000	44761597	22.4
3000000	74694459	24.9
4000000	116103352	29.0
5000000	188503596	37.7
6000000	219841605	36.6
7000000	333523638	47.6
8000000	290172547	36.3

(a) uniform cost, uniform sources.

Cells	Cells (re)calculated	Average times cell (re)calculated
1000000	38375386	38.8
2000000	124345481	62.2
3000000	220682171	73.6
4000000	421990035	105.5
5000000	712969080	142.6
6000000	1151352169	191.9
7000000	1708295630	244.0
8000000	2002923358	250.4

(b) Uniform costs, non-uniform source.

Cells	Cells (re)calculated	Average times cell (re)calculated
1000000	24578897	24.6
2000000	44986914	22.5
3000000	64250221	21.4
4000000	95544459	23.9
5000000	94453387	18.9
6000000	124934772	20.8
7000000	136375453	19.5
8000000	174017569	21.8

(c) Non-uniform cost, uniform sources.

Table 5: Number of ACD calculations with average amount that a cell has been (re)calculated for 3 raster scenarios. A partition size of 100 was used.