Faculty of Science

A parallel cost-distance algorithm & its run-time characteristics

Master Thesis

Mohamad Kobeissi (4983998)

MSc. Applied Data science

Supervisors:

Prof. Dr. Derek Karssenberg,
Faculty of Geo-sciences

Dr. Kor de Jong,
Faculty of Geo-sciences

Dr. Oliver Schmitz,
Faculty of Geo-sciences

07.2023

Contents

Abstract

The cost-distance algorithm (CDA) is a widely used analysis that provides researchers with profound information about distances and locations. The required cumulative cost operations makes the algorithm serial since it depends on prior steps for information to be passed on to update the cumulative cost. Hence, the algorithm results in extensive run-times and therefore becomes computationally inefficient. This limitation of the CDA motivates this research to explore a potential approach to parallelize the cost-distance algorithm to improve run-time. The approach of parallelising the cost-distance algorithm is dividing the input raster into partitions so that on each partition the cost-distance algorithm can be run simultaneously resulting in improved run-time. The parallel computing technique is using distributed memory and the message passing interface framework (MPI). This parallel technique allows the algorithm to run its partitions on numerous workers simultaneously, where each worker has its own local memory making this technique set better for scalability purposes. MPI was used to create the communication structure between processes whilst taking overhead communication into account. The communication structure was set-up to have one process that acts as the handler and focuses on the communication and coordination of the workers. The remaining workers focus on running the CDA on their assigned partitions. The parallel algorithm did improve the run-time and the speed-up ratio resulted 2.6 times faster than the serial one. However, as more and more workers were added the run-time started to increase indicating a potential bottleneck. Therefore, scalability tests were conducted which revealed poor scalability performances in terms of relative efficiency, so as more workers were added the lower the efficiency became. Primarily, the issue was caused by the overhead communication between the handler and the workers. The issue in communication was that as the workers were processing their partitions the handler is waiting for longer periods of time to receive the information back making the handlers role inefficient. Additionally, other potential indicators of low efficiency that were identified are the sequential fragments and the checking of the partition boundaries. To overcome the major overhead communication bottleneck the handler can be assigned additional tasks that could improve run-time. Meaning, during the time the handler is waiting to receive the information back from the worker, it can undergo additional tasks by implementing asynchronous communication. Overall, the research did identify a potential solution that parallelizes the cost-distance algorithm to improve run-time, however there are remaining overhead communication issues that cause poor scalability measurements. Once these issues are addressed a complete scalable parallel cost-distance algorithm can be successfully implemented.

1 Introduction

In recent years the increasing number of satellites orbiting earth or drones capturing aerial footage enabled researchers around the globe to collect vast amount of Geo-data. This wealth of data has provided valuable insights and opportunities for various fields. One commonly used approach of projecting Geo-data for analysis is using raster grids, which reflects and aligns real life Geo-objects onto digital display. Raster data offers versatility and flexibility, making it beneficial for researcher to conduct their analysis on (*What is Raster Data?* 2023). One widely employed analysis technique on raster grids is the cost-distance algorithm (CDA). This algorithm addresses the challenge of accurately representing distance in a raster by considering the actual effort required to traverse the terrain (*Understanding Cost Distance Analysis* 2023).

Imagine a scenario where a hiker is faced with the task of reaching point B, with the option to start from either point A or point C. Utilizing a raster map for guidance, the hiker initially notices that the grid indicates a shorter distance from point A to B compared to point C to B. However, upon embarking on the journey, the hiker realizes that reaching point B from point A entails traversing challenging mountain ranges. In contrast, if the hiker had chosen point C as the starting point, although the actual distance may be longer, it would have resulted in energy savings. This discrepancy highlights how the raster underestimated the true distance from point A to B. To address this limitation and provide a more accurate representation of distance, the concept of cost-distance algorithm was developed and introduced in the 1960's (Greenberg et al., 2011). The concept of cost-distance analysis (CDA) has been used since its initial introduction across many industries and disciplines such as Geo-information system (GIS), network analysis, logistical optimisation problem etc. It certainly has proceeded in GIS and has been a essential tool for numerous cases ever since (Murekatete and Shirabe, 2018).

Despite the algorithms diverse application ability, it inherently introduces inefficiencies that many researchers tried overcome such as Lu et al. (2011). The cost distance algorithm encounters computational challenges due to the cumulative cost calculation by requiring to repeatedly check and update neighboring cells to find the minimum cost since it is reliant on prior steps. This becomes particularly clear as the data-set size increases, demanding extensive processing time and memory resources to complete the computation. Efficient approaches such as involving algorithmic enhancements and parallel computing techniques are crucial for addressing these computational challenges to ensure the algorithm's effectiveness in large-scale raster analysis.

The current computational implementation of the algorithm is sequential. This means given its inherent nature of being a serial algorithm it requires time to complete each step because each step is dependent on the prior one, only then the calculations can proceed. As currently constructed the algorithm does not utilize the computational advancement of cores in current computers and therefore has an extensive run-time (Chen et al., 2009). The CDA is challenging to parallelize because of the required need of prior steps in cumulative cost calculations, hence the title of being a serial algorithm. Developments in computational hardware has been one of the paradigm shifts that allows software to run efficiently. However, often the CPU cores that are available are being underutilized, and therefore face performance bottlenecks e.g. the application of the CDA on large raster data (Laccetti, Lapegna, and Mele, 2016). Therefore, one potential solution is to use parallel techniques to parallelise the CDA by utilizing the available cores (Chen et al., 2009). The application of parallel computing techniques on CDA are yet to be fully explored.

The inherent limitation of the CDA being a serial algorithm motivates this research to investigate a potential approach on how to use parallel computing techniques to parallelize the algorithm to improve run-time. Additionally, certain parameters need to be investigated to examine the influence these have on the parallel algorithms performance and scalability. There is a possibility where a parallel cost-distance algorithm can be developed that overcomes bottlenecks, exhibits scalability, and improves run-time characteristics for enhanced usability. Once an efficient sequential cost-distance algorithm is constructed, the parallel computing techniques can be implemented, which will determine the speed-up ratio and the distinction in run-time characteristics of the two approaches can be displayed (Kotov and Sergienko, 2009). Essentially, the aim for the parallel algorithm is to improve run-time taking speed-up parameters into account by utilizing the CPU cores using distributed memory. Additionally, after acquiring the necessary baseline parameters the scalability performance of the parallel algorithm will be studied, tested and evaluated. There are potential

results that can seem surprising especially considering the parameters that could affect the algorithms ability to improve run-time and its ability to scale.

2 Literature review

2.1 The cost-distance algorithm

The cost-distance algorithm aims to find the least-cost path by calculating the cumulative cost between a source node(s) and a destination(s) [1]. The algorithm calculates the cost of moving from one cell to another and accumulates the cost by checking and updating the cost of the previous step. The cumulative cost process of visiting new cells and updating their costs, is time intensive and therefore computational inefficient. This is especially highlighted with larger raster data sets. The cost-distance algorithm fundamentally consists of sequential fractions since it relies on previous calculations to calculate and update the cumulative cost (*Understanding Cost Distance Analysis* 2023).

The paper by Lu et al. (2011) provides extensive research in making these sequential fractions more efficient, which can act as a proficient baseline for the parallel algorithm (Kotov and Sergienko, 2009). There are many variants that were created of the CDA that aim to improve the efficiency and accuracy of the initial algorithm. The paper by Lu et al. (2011) display and evaluate various cost-distance algorithms that are currently used (Lu et al., 2011). The goal of the evaluation is to find the most efficient (in terms of run-time) variant of the cost-distance algorithm for raster data. Eventually, the paper derives that the traditional algorithm with minimum cost heap (TAMH) was the most efficient out of all of the ones that were selected for the study (Lu et al., 2011). The main-takeaways reflect that a proper data structure such as the heap enables the algorithm to efficiently update by keeping the lowest cost at the top of the array.

It is important for this study to look at prior research where the cost-distance algorithm has been optimised as Lu et al. (2011) has done, so that the sequential fractions can run effectively, this can be referred to as serial performance tuning before the implementation of the parallel fractions (Schürhoff, 2021). In this case the integration of the heap structure seems to be an essential and significant factor of making the sequential fractions much more effective, resulting in better run-time characteristics.

2.2 Parallel computing

The use of parallel computing techniques on the cost-distance algorithm is a relatively unprovoked area that not many have tried or publicly published. A paper by Wang et al. (n.d.) attempted a basic blueprint but does not specify any detailed methodology infrastructure nor test results. It can be assumed that there will be some limitations of the algorithm where Amdahl's law applies, where only certain fractions of the algorithm can be conducted in parallel, despite the serial performance tuning (Shi, 1996). This limitation applies for the cost-distance as-well given the sequential cumulative cost required, which has motivated this research to be conducted. Consequently, this research will evaluate the possibilities of using modern parallel computing techniques to apply to the cost-distance algorithm to improve run-time.

2.2.1 Distributed-memory using message passing interface (MPI)

There are two main ways of applying parallel computing techniques on sequential algorithms, shared memory and distributed memory. In this paper the focus will be on using distributed-memory parallelism (DMP) to parallel the cost-distance algorithm. Distributed-memory computing is a parallel computing approach that employs multiple processes, each with its own individual memory space, that operate independently without sharing memory with one another. In contrast to shared-memory computing, where processes operate on a common memory pool, distributed-memory computing requires an interconnected network for bidirectional communication among the individual processes (Cornell University, 2023). This means that the nodes[2] must be able to communicate with each other through the network to coordinate their computation

[1] Source node, cell and point will refer to the same thing & will be used interchangeably.
[2] Nodes, workers & processes will be used interchangeably

and exchange data (Princeton research computing, 2023). To be able to do that the framework uses a message passing interface (MPI) that coordinates the processes and enables communication between them (Graham and Shipman, 2008).

Nowadays there are many platforms where distributed memory can be used with different programming languages. For example, in C++ 'OpenMPI' (*Open MPI: Open Source High Performance Computing* n.d.) is a popular framework and a Python derivative 'mpi4py' has also made its way (*MPI4py: Python bindings for MPI* n.d.). As the name of both libraries suggest the distributed memory framework revolves around MPI. The breakthrough that allowed MPI to become such as prominent approach is its scalability since its designed to function on many parallel computers (Graham and Shipman, 2008). MPI can be used to perform different tasks of the algorithm simultaneously by using multiple processes at the same time and achieving the desired output (Shoeb et al., 2012). The issue that lies with distributed memory and using MPI is the stemming overhead communication that could occur (Fink et al., 2021). Given the communication infrastructure that arises, high overhead might result in performance degradation which could make the algorithm redundant (Fink et al., 2021). This is a result of how MPI systems inherently are built. MPI in simple terms is the inter-communication between processes where messages are being sent from one process to another. These messages contain tags and information about the tasks, including from which source (process) it is expected from. This derives in a complex communication system, which if not handled correctly can cause high overhead communication, resulting in extensive run-times (Balaji et al., 2011).

The communication infrastructure of MPI is one of the main challenges in building MPI based program's. There are many possible reasons why overhead communication might occur in the MPI program. One of the main issues that occur in MPI overhead communication is overhead in tags and source matching (Balaji et al., 2011). In MPI communications there is a sender and a receiver. The sender sends information that contains tags which contains the purpose of the message, and the source information. Every message is tagged with this information before its sent to the receiver. On the other hand when the process is expecting to receive messages from the sender, it specifies the tags and source information in a receive request (Balaji et al., 2011). The receive request maintain a queue of the received messages, as it goes through the queue and the message is matched then the data is stored into a buffer (Balaji et al., 2011). The issue arises because most MPI applications use a single queue to handle all received requests, regardless of the information that is being sent. Meaning, all requests are stored in a single queue and as the messages increase and the queue grows, scalability issues of the parallel program might occur (Balaji et al., 2011). Another overhead issue that could occur is overheads in derived datatype processing (Balaji et al., 2011). A study conducted by Balaji et al. (2011) display how non-contiguous and contiguous data types differ in performance. MPI allows non-contiguous data types however implementing this is certainly a challenge that is difficult to overcome as the paper suggest. Lastly, there is the overhead caused by unexpected messages (Balaji et al., 2011). The sender can send multiple requests and messages at once but the receiver will look for the message it needs, so all the messages sent prior are seen as unexpected and are added to the queue (Balaji et al., 2011). The processes of queuing can cause major overhead given the back and forth of the receiver. In the case of parallelizing the cost-distance the overhead communication could occur between neighbouring partitions since the partitions need to calculate their costs and update them based on neighbouring values which could potentially increase the latency.

The challenge of parallelising serial algorithm occurs because of the required dependencies the steps in the algorithm have. The same goes for the cost-distance algorithm, where the cumulative cost is dependent on prior steps, which makes the parallelisation strategy challenging. However, there are possibilities where different input of the implementation can be parallelised and potentially resulting in improved run-time. One example of a parallel strategy is dividing the input into partitions and let the serial algorithm run on these partitions simultaneously as conducted and presented by Hoefler, Lorenzen, and Lumsdaine (2008). This is one approach that can be investigated and potentially used for the case of the cost-distance algorithm.

2.3 Measurements and performance of parallel program's

High-performance computations, such as distributed-memory, require validation through measurement. The first step in this process is to define the performance metric of the parallel algorithm. The

common performance metric for sequential programs is the number of arithmetic operations[3] per second. Normally for parallel programs it requires a more nuanced metric of efficiency, which compares the execution time of the parallel program to that of the serial program (Kotov and Sergienko, 2009). To ensure high efficiency of a parallel program, it is essential to optimize the initial sequential fractions of the algorithm as stated in section 2.1. This approach enables a more meaningful comparison of the relative performance of the sequential and parallel versions of the program also known as the speed-up ratio (see equation 2.1), and provides a basis for further optimization to achieve even better performances (Kotov and Sergienko, 2009). Despite speed-up being a prominent run-time indicator of measuring parallel algorithms it does not necessarily indicate how well was the parallel algorithm constructed. Therefore, for applications used in high-performance computing additional scalability measurements are needed (Skinner and Antypas, 2010). This relates back to the initial reasoning of the advantage of using distributed memory, which is to enable the parallel program to scale effectively across many computers for larger cases.

$$\text{Speed-up Ratio} = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}} \tag{2.1}$$

To properly measure the scalability of the parallel program two indicators will be used, the relative strong and weak scalability efficiencies (Skinner and Antypas, 2010). The difference between the two is that strong scalability has a fixed problem size as the number of processes increases. Whereas, the weak scalability indicator focuses more on growing the raster size concurrent to the number of processes (Skinner and Antypas, 2010). This will allow the parallel program to undergo scalability tests from different aspects to conclude its performance. There will be parameters that will affect the parallel program, therefore it is essential before conducting the scalability tests, to ensure that the ideal parameter configurations were selected LUE (2023).

$$RSE_{\text{strong}} = \frac{\text{latency using one worker}}{\text{number of processes} \times \text{latency of P-worker(s)}} \times 100 \tag{2.2}$$

$$RSE_{\text{weak}} = \frac{\text{latency with X-problem size with one worker}}{\text{latency with X-problem size per P-worker(s)}} \times 100 \tag{2.3}$$

The strong scalability equation (2.2) compares the latency of the one worker and will be divided by the latency of the added number of workers, keeping the problem size the same throughout the calculation. For the weak scalability equation (2.3) the main difference is that the problem size will increase linearly from worker one so that its concurrent with the number of processes added. This will allow to test how well each worker does when all of the workers have the same amount of increased workload (LUE, 2023) (Schürhoff, 2021).

[3]Operations are additions & multiplications

3 Research Methodology

3.1 Research question

How can a parallel cost-distance algorithm be constructed to improve run-time characteristics?

3.1.1 sub-research questions

1. How can distributed memory be used to parallel the cost-distance with minimum cost heap algorithm?
2. How does the partition size, the number of processes and the raster-size affect the run-time and scalability of the parallel algorithm?

3.2 Methodology

In this section the methodology of the research will be presented which takes the theoretical framework presented in chapter 2 into account. At first the cost-distance algorithm implementation will be presented that highlights the inherent sequential nature of the algorithm, thus is followed by the implementation method and strategy of the parallel algorithm, where the parallel infrastructure will be laid out. Lastly, the measurement and testing methods will be displayed.

3.2.1 The cost-distance algorithm

When discussing the cost-distance algorithm during literature review it was determined that its inherent nature of being a sequential algorithm makes this process challenging. Considering Amdahl's law there are additional limitations where the algorithm cannot run in parallel, given the need of prior cumulative cost calculations. Hence, it can be expected that the algorithm will always take some significant time to run. Nevertheless, certain pieces of the implementation can be parallelized to ensure that the run-time is reduced. However, before advancing to the parallel implementation, first the serial fractions of the CDA needs to be implemented efficiently as suggested in chapter 2.

To create a good baseline for the parallel algorithm the approach is to find and implement a relatively efficient working cost-distance algorithm that outputs the desired result. The inspiration for the sequential algorithm is from Lu et al. (2011) where the traditional cost-distance algorithm with minimum cost heap (TAMH) proved to be the most efficient as discussed in the literature review. The implementation of this algorithm revolves around the heap data structure, specifically the minimum heap (min-heap). This allows the methodology to focus solely on developing and optimizing the parallel algorithm. The min-heap structure in place allows the algorithm to efficiently establish a priority queue based on the needs of the cost-distance. Meaning, the min-heap data structure in this case will always store the processed cell(s) with the lowest cost at the top of the heap ensuring efficient retrieval of the lowest cost needed for cost accumulation. The start of the min-heap will always be the source cell(s).

The pseudo-code of the cost-distance with min-heap implementation can be found in appendix A. The code iterates over the input raster, where each cell compares its value with its eight neighboring cells and checks the accumulated cost (the pseudo-code also shown in Appendix A). The cell then moves to the neighbor with the lowest value which then becomes the new current cell. This process continues iteratively until there are no further cells to be checked. Regarding the heap structure, during each iteration, the code checks the heap queue to determine if it needs to update the list of active cells based on the values that are being checked. If the value of the current cell is smaller than the values in the queue, the cell is added to the top of the queue indicating change. The accumulated cost values are updated based on the costs of neighboring cells and the distance between them. This approach efficiently computes the cost distance by prioritizing cells using the heap with lower accumulated costs, to ensure that the shortest paths are processed first. Essentially, the algorithm makes use of a heap queue to maintain the active cells and efficiently identify the next cell to process.

3.2.2 Implementation of the parallel algorithm

Distributed memory will be used as the parallel technique. The parallel algorithm will use distributed memory to utilize all cores in a given system and make this applicable to all machines. Hence, scalability is a big factor that needs to be considered throughout the algorithms design and implementation. The initial problem when applying cost-distance is the extensive run-time required to calculate and update cumulative cost. Thus being serial makes the parallelisation approach a challenge since it needs to consider prior steps to calculate cumulative cost. Consequently, the parallelisation strategy is to breakdown the data set into unique partitions so that each process will work on its assigned unique partitions simultaneously, and eventually successfully consolidate them together into one cost-distance raster in a improved run-time.

The program needs to begin by initializing the message passing interface (MPI) distributed memory framework. Once the MPI variables are initialised these can be used to divide the input raster accordingly. In distributed memory optimally all processes should have the same workload, and make all processes equal to one another. Therefore, the data is equally divided into unique partitions across the size of the number of processes. The uniqueness of the partitions is an essential attribute to avoid overlapping of processes, so that each process works on a unique set of assigned partitions. The challenges of using MPI is the overhead communication as mentioned in chapter 2. A study conducted by Hoefler, Lorenzen, and Lumsdaine (2008) where a similar approach was conducted, faced the inherent issue of neighbouring partition cells needing to communicate and update with one another. Eventually, this must be also considered throughout the process and during testing.

3.2.2.1 Partition strategy and coordination

First step of is to find a suitable partitioning approach since it will be the foundation of the parallelisation strategy. Ideally, the partition size can be flexible and can have different inputs based on the need of the case study. It also has been discussed in the literature review section of the importance of testing the partition size (LUE, 2023). Once the partition size is given the dimensions of the partitions need to be determined. To do that the number of rows and the number of columns will be divided by the partitions size which will give the partition dimensions as displayed by equations 3.1 and 3.2. This will also ensure that each partition is equal in size and dimensions so that each worker will process same amount of workload.

$$Y_{\text{PARTITIONS}} = \left\lceil \frac{\text{rows}}{\text{Partition size}} \right\rceil \tag{3.1}$$

$$X_{\text{PARTITIONS}} = \left\lceil \frac{\text{columns}}{\text{Partition size}} \right\rceil \tag{3.2}$$

The partition size and dimensions have been constructed. The issue that now needs to be resolved is the coordination of these individual partitions ensuring that the partitions can coordinate properly in the raster. Meaning, when dividing the raster into sub-raster's there are multiple raster's that need to know their unique global coordinates for them to be able to be uniquely identified. Therefore each partition will be given global coordinates. Global because the partitions need to know their place in the complete input raster. The coordinates will be unique across partitions. This also ensures that the whole raster is covered. In the figure 1 is an example of how to conceptualize the final grid to be used. Noticeably, the number of partitions can also be deducted.

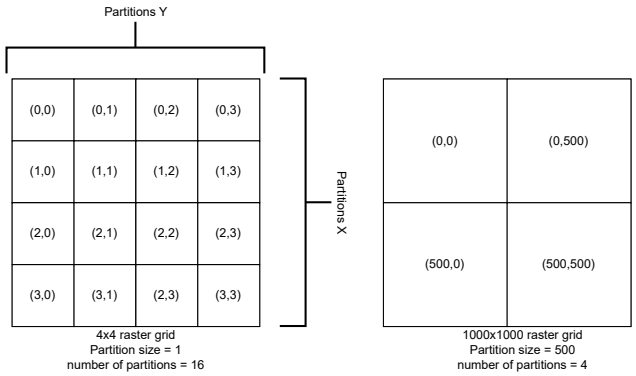


Figure 1: Example of partitioning & coordination

The partition size as displayed in figure 1 can vary and should vary. Initially there is no indication what will be the ideal partition size for the parallel algorithm. These are input variables that will need to be investigated at later stages during testing. In the current display the first raster grid is just an example how coordinates and partitions will look like based on the equation(s) mentioned prior. The 1000x1000 grid is a more likely scenario in which the raster will be tested with a partition size of 500. These two are just examples to conceptualise the beginning of the partition strategy of the parallel algorithm. The number of partitions will be a derived variable that will also be observed throughout testing.

3.2.2.2 Padding & global boundary

One of the challenges that needs to be conceptualised and implemented is taking neighbouring partitions cost values and updating them accordingly without overstepping the boundary. Additionally, partitions that lie in the outer edges of the input raster, need to be aware that they are on the edge.

To address the first challenge of neighbouring partitions sharing and updating cost values a padding method will be introduced which will create a layer around the local partition boundary of the size of one so only the direct neighbours are being accounted for. This layer will act as a buffer zone that will share the cost of the layer adjacent to the neighbouring partition(s). This will allow the cost-distance to be consistent and take the cost of neighbouring partitions into account and update the cost accordingly. In the figure 2, this is indicated by the red zones around each partition to show where each partition shares a boundary and updates costs on behalf of its neighbours.

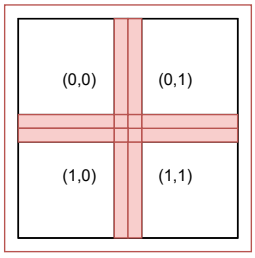


Figure 2: Padding & global boundary of for partitions

To overcome the challenge of edging partitions a global boundary can be added to the input raster. This means specific starting points along the y and x axis will be identified that correspond with the edges of the edging partitions. The global boundary will be added to indicate to the edging partitions that they are on the edge, as shown by the figure 2 by the red outer line. Its sole purpose is to inform the edging partitions that they are located on the global boundary of the raster. Once the calculations are finished it can be removed so that the output cumulative cost raster will have the same dimensions as the input raster.

3.2.2.3 Process communication: Handler and workers

When mentioning process communication its is referred to the inter-communication between processes. This is a very important aspect of distributed memory to avoid complex and redundant communication systems to accelerate run-time of the algorithm, which is also known as overhead communication as described in the literature review. Fundamentally, that is what MPI is all about to pass messages to enable distributed memory. For the cost-distance algorithm it is essential since neighbouring partitions need to communicate with one another to pass on calculated information and update based on the neighbouring partitions. The approach of constructing the ideal communication system is to make sure that the structure is consistent throughout between the workers and to keep the variables that need communication as clear as possible to avoid communication overhead. Additionally, each process should only be concerned with its own information within its partition boundaries. However, a root node should be selected to solely handle the communication and coordination of the processes. This is essential since the root node is the one that's initially distributes the partitions to the different processes and lastly gathers the partitions in the right order and merges the output raster back together. Essentially, there are two types of processes a handler and its workers. These types of processes and their tasks and communication structure is seen in the flow-chart in figure 3.

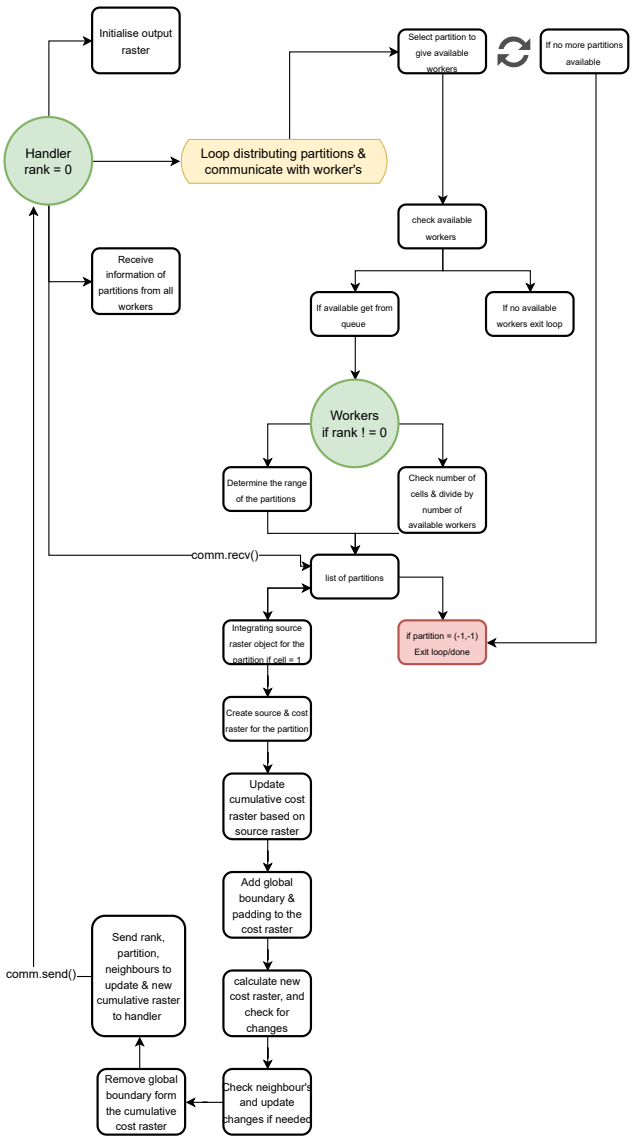


Figure 3: Flow chart of MPI communication between handler & workers

The strategy is to make the handler distribute the work to the workers accordingly. Before that the handler needs to be assigned the root node so it knows its the handler. The handler at the beginning already initialises the output raster format and the expectation to receive some sort of information from the workers as shown in the flow chart3. So the handler provides the ability to already construct the output raster framework. This is followed by the handler expecting to receive messages from other processes indicating that they are ready to work on the partitions or whether they are finished with their partitions. The handler will have queues. These queue consists of available workers and partitions, workers and partitions that are currently processing. This is needed to ensure that partitions are not reintroduced into the queue once they are processed, and to only give out work to workers that are available. So there will be a set of partitions to check and a queue of available workers to work on the partitions. Then the handler will assign available partitions to available workers with the corresponding data for the workers to undergo the cost-distance calculation on its assigned partition(s). The handler will need to keep track of partitions that are being processed and update them accordingly to avoid miscommunication and overlapping issues. Eventually, the handler will distribute the partitions equally across workers so they can work on their partitions simultaneously. Once all of the partitions have been assigned and processed the workers send the handler the information, in which the handler checks if there are any partitions left. If there are no more partitions left then all partitions have been processed and therefore the workers stop. Then the handler consolidates the partitions that were processed into the output raster in a coordinated way. The pseudo-code appendix B display how the handlers tasks specifically and how it receives the processed information.

The workers play a crucial role in performing the intensive computations of the cost-distance calculation. To ensure a consistent workflow and data consistency, certain initialization steps are necessary. The workers need to be aware of the partitions they are responsible for processing, which requires providing them with relevant information of these partitions. One important aspect is determining the amount of work each worker should handle by giving them the range of partitions that they need to work on as displayed in the flow-chart 3. Additionally, the total number of grid cells is divided by the available workers. Emphasizing the concept of "available workers" is important because it accounts for the fact that some processes may already be occupied with work, while others are available to take on new tasks. Dividing the work based on available workers helps avoids unnecessary queuing and ensures a balanced workload distribution. Hence, this could minimize the unexpected message overhead since the the order does not matter in which the partitions are worked on. It is essential to align the variables and parameters so that each worker and its corresponding partitions are aware of the number of cells they need to process, as well as the starting and ending positions. These partitions are used to create raster objects for further processing. Additionally, the workers need to identify the source cells within their assigned partitions. When a worker encounters a source cell, it immediately communicates this information to the handler. The amount of information each message has is constructed so to avoid overhead in matching information. The workers send their list of partitions to the handler, enabling coordination and collaboration between the processes. The workers continue their computations as long as the handler does not signal the program's completion, as discussed previously. The cost-distance algorithm incorporates padding and a global boundary, which need to be taken into account by the workers when working with their raster subsets. If a change is detected in the cumulative cost raster during the cost-distance calculation, the worker updates and adjusts the corresponding values to reflect the minimum cost appropriately. Finally, the worker communicates its work and results back to the handler. This involves sending several variables, including the process rank, the partition(s) it worked on, the list of neighboring partitions that require updates, and the updated cumulative cost raster subset. Overall, the workers' role encompasses handling the actual computations, coordinating with the handler, considering padding and global boundaries, and effectively communicating their work to contribute to the overall cost-distance calculation process. The pseudo-code of how the workers tasks are initialised and sent back to the handler can be found in appendix B.

3.2.3 Testing scalability and run-time measurements

To effectively justify the parallel algorithm's proper usage the performance and the scalability of the program needs to be tested, measured and evaluated. When referring to scalability of parallel program as mentioned in the literature review its about how well does the program do once additional processes or hardware such as additional CPU cores are added. Additionally, to properly asses the scalability of the

algorithm different parameters need to be tested to extract best parameter configuration that outputs the best run-time. The main variables to test and measure are the partition size, the number of processes and the number of cells. The partition size is a significant variable, as suggested prior that the potential inherent obstacle of the cost-distance algorithm re-occurs where each neighbouring partitions need to communicate with one another, so that the neighbouring cells can update accordingly. Consequently, to minimize this potential issue different partition sizes will be tested before conducting the scalability tests. The number of processes need to be measured and tested since its about distributed memory across number of processes. Lastly, in combination with the other two parameters the change across number of cells will also be measured and tested. Essentially, the best run-time output of these variable configurations will create the baseline for the speed-up ratio (equation 2.1) and the input for the scalability tests. Note that during the implementation it needs to be asserted that the number of partitions is greater than size of the cores to completely utilize all cores efficiently. Otherwise the use of multiple cores becomes redundant for the raster data at hand.

For this methodology the two types weak and strong scalability will be tested, the equations can be seen in the literature review section equations 2.2 and 2.3. The main aim is to find how efficient is the parallel algorithm when using additional processors. For the scalability measurements there must be preliminary testing conducted beforehand to find the ideal parameters and then eventually test the scalability of the algorithm. Hence, permutations of relevant parameters must be identified and tested. For the parallel cost-distance algorithm the following parameters will act as tuning parameters to create the best possible baseline for scalability measurements.

Table 1: Parameters for scalability measurements

Parameter	Description
Partition size	Dimension of each partition
Number of processes	Number of selected MPI processes to run
Number of cells	total number of cells of input raster
latency (output variable)	time needed for processes to execute

These parameters have been selected because of the assumption that these variables could potentially affect the scalability performance of the program. The partition size is essentially how big each partition is (partition dimensions), the more partitions there are (the smaller the partition size) the more the partitions the workers need to calculate resulting in more communication sending and receiving information. However, the bigger the partition size the longer the cost-distance calculation will take per partition. Hence, why the optimal balance needs to be found to find the ideal baseline to measure the speed-up ratio and scalability. The parameters will be used as input and the output will be the latency of the program, which will be used to determine the best possible parameter configuration reflecting the lowest latency. Once the data has been accumulated and explored the scalability will be measured for weak and strong scalability. In essence the first step is to find the best run-time with on P-partition size, ran on W-number of workers with N-number of cells. Once that is established the best performed permutation will undergo speed-up ratio which tells by how much the run-time was improved. This is followed by undergoing weak and strong scalability measurements, thus will eventually display how effective the parallel algorithm is once more processes are added. So how efficient is the parallel program using the additional processes.

Both scalability measures require different input as the equations 2.2 and 2.3 display and described in the literature review. The distinction between the two is that the weak scalability equation requires the change of the problem size concurrent with the number of processes so each process has the same amount of work due. For this additional raster sizes were created that are concurrent with the addition of number of nodes. In the table below the approach is displayed.

Table 2: Weak Scalability method

Size of Processes	Number of workers	Number of Cells
2	1	1,000,000
4	3	2,999,824
6	5	4,999696
8	7	6,996,025
10	9	9,000,000
12	11	10,995,856

Notably, the number of processes does not equal to the number of workers in this case. This is due to the fact that there is one handler constantly for coordination and communication that does not undergo the cost-distance calculation. Additionally, the number of cells are created using rows and columns dimensions hence the uneven outcomes. Nevertheless, this input is sufficient to test the relative weak scalability efficiency of the parallel algorithm since the number of cells is increasing concurrent with the number of workers. The increase of the size of the processes is referred to as a cluster of 2. This approach ensures that with each cluster increase, every worker in the cluster, has the same amount of work by increasing the number of cells.

4 Results

4.1 Testing environment

For transparency the testing environment is presented. All tests have been conducted on the same computer to ensure consistency throughout the research.

Table 3: Computer Specifications

Computer	Processor name	Number of hardware cores	Number of logical cores
iMac 2019	3,0 GHz 6-Core Intel Core i5 Processor	6	12

Table 4: Environment Specifications

Programming language	Package(s)
Python 3.11	mpi4py 3.1.4

4.2 Preliminary testing

In the literature review and methodology it was determined that to be able to properly test for scalability the right configuration of variables need to be identified. Therefore, numerous tests were conducted to find the best executed time with the different parameters. In the figures 4 and 5 the test results are displayed where the results of the different parameters can be identified. This was done for different raster sizes (number of cells), however there was no clear change in pattern as can be seen in appendix C in figures 16, 17 and 18, and therefore was determined not to be as significant to present for the main results. One other parameter was tested which is varying the number of source cells. There was some slight change but also was deemed to be not as significant, however more is analyzed under chapter 5 discussion and the results can be observed under appendix C figures 19 and 20. Hence, the permutations of using different number of processes and partition size was the focus.

4.2.1 Preliminary results

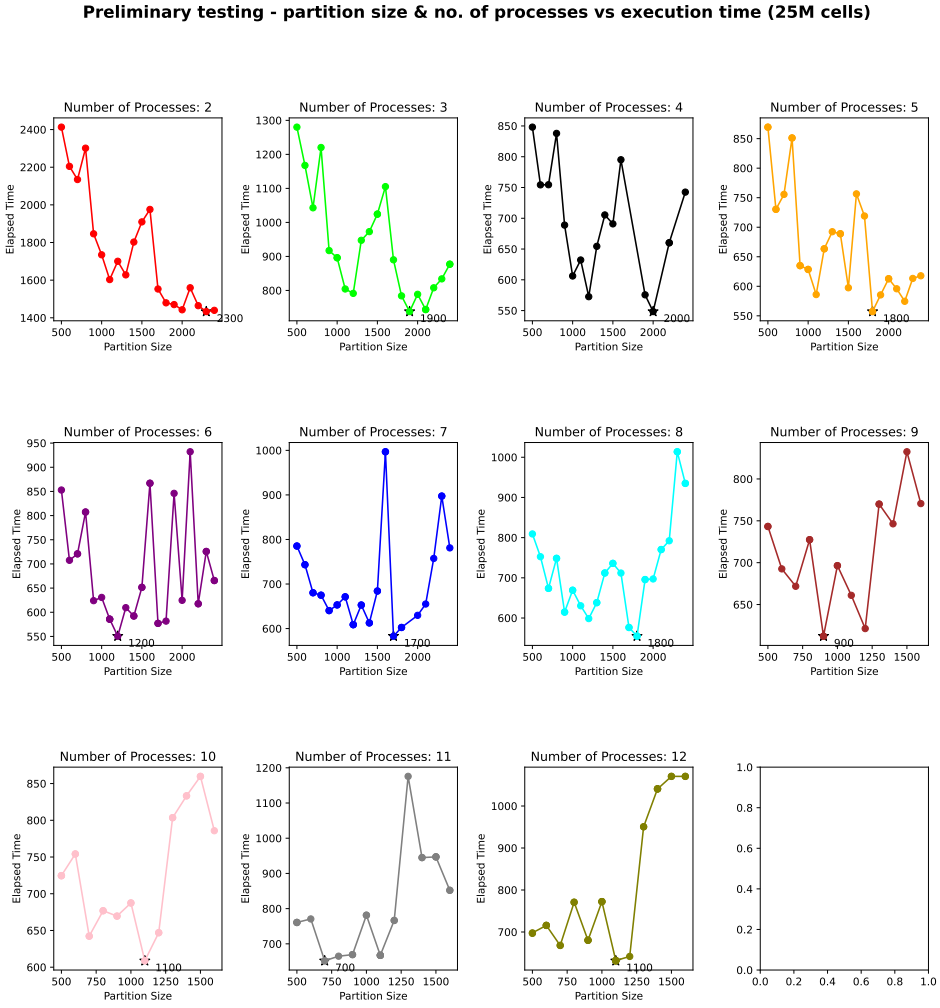


Figure 4: Sub-plots of across the number of processes & partition size with 25 million cells

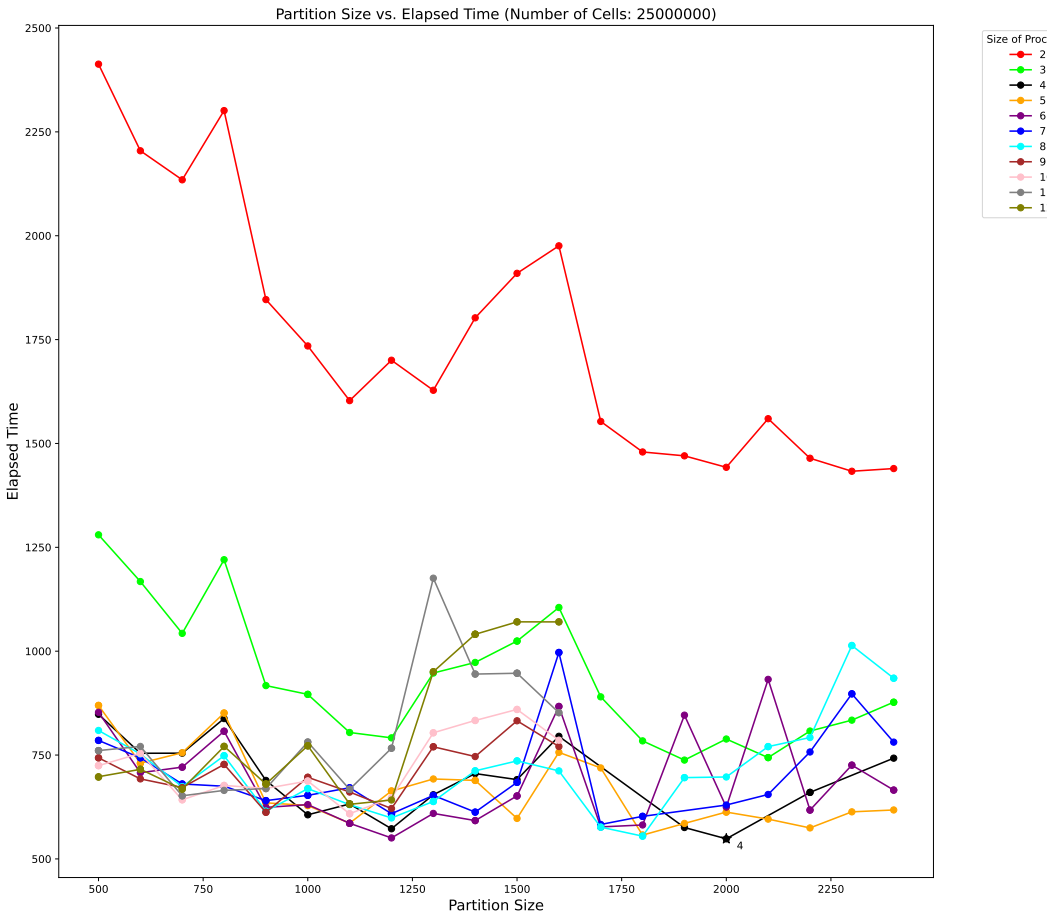


Figure 5: Partition size & no. of processes vs elapsed time graph

The figures 4 and 5 display the execution time of the parallel algorithm with using permutations of various number of processes and partition size. In the first figure 4 the focus is on displaying the change in latency behaviour as the size of partitions grow for different number of processes. In the second figure 5 is to show the relativity on one plot to observe how the number of processes execution time differs across the size of partitions. In both figures there is a star to indicate which combination of partition size and number of processes scored the best execution time. In figure 4 it can be seen for every subplot, for figure 5 there only should be one best performing one out of all permutations.

To get a better understanding of how good the number of processes did in term of execution time the average execution of each size can be calculated as can be seen in table 6. For better interpretation refer to figure 7 where the average time of each size has been visualized.

Number of Processes	Average Elapsed Time (s)
2	1754.76
3	923.94
4	689.84
5	674.74
6	682.79
7	700.66
8	717.23
9	711.46
10	721.86
11	789.43
12	798.63

Figure 6: Average Elapsed Time for each size

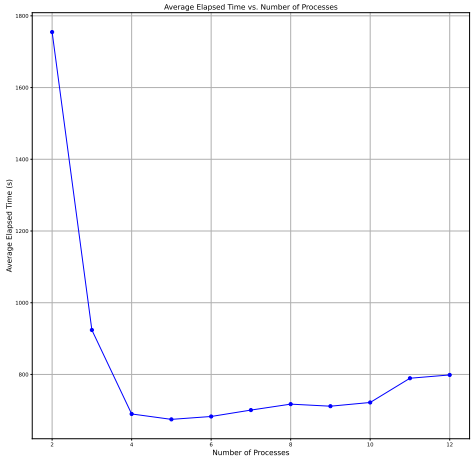


Figure 7: Average elapsed time for each size

4.2.2 Speed-up ratio

The speed-ratio determines the run-time difference between the serial and the parallel algorithm as stated by the equation 2.1 displayed in the literature review. In the equation 4.1 and 4.2 the speed-up ratio of the parallel algorithm can be observed. Equation 4.1 the quickest latency of average of the number of processes is taken which is 5, and the speed-up is 2.6. For equation 4.2 the average latency across all number of processes average latency is taken which also results in a speed-up ratio of 2.6.

$$\text{Speedup} = \frac{\text{Sequential run-time}}{\text{Parallel run-time}} = \frac{1754.76}{674.74} = 2.6 \tag{4.1}$$

$$\text{Speedup} = \frac{1754.76}{\frac{923.94+689.84+674.74+682.79+700.66+717.23+711.46+721.86+789.43+798.63}{10}} = 2.6 \tag{4.2}$$

4.3 Scalability measurements

During preliminary testing the ideal parameters were identified to create the best possible baseline to measure the scalability of the parallel program. The scalability measurements were discussed and displayed in the literature review and methodology. Essentially, there will be two aspects that will be measured weak and strong scalability. Where strong scalability will indicate how well the algorithm performs increasing the number of processes (size) and keeping the problem size the same. Weak scalability indicates how well the parallel algorithm performs by increasing the number of processes concurrent with the increase of the problem size.

4.3.1 Strong scalability

In the table 5 and figure 8 the strong scalability is measured according to the equations displayed prior. The measured output is the relative strong efficiency (RSE) across the cluster number of workers. The actual performance and the ideal scenario have been graphed.

Table 5: RSE Strong Scalability

Number of Processes	Latency	RSE
2	1700.3s	100.0%
4	572.6s	74.24%
6	550.5s	51.48%
8	598.6s	35.51%
10	646.9s	26.28%
12	641.8s	22.08%

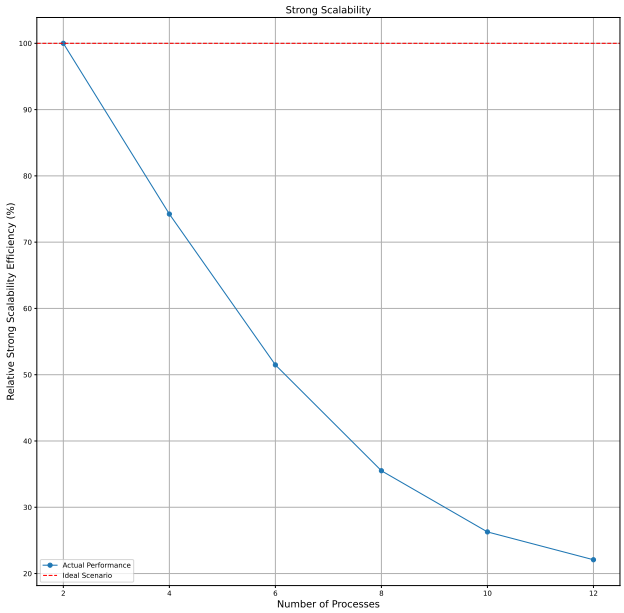


Figure 8: Strong scalability measurements for the parallel program of the raster size 25 million cells.

4.3.2 Weak scalability

For weak scalability different raster sizes had to be constructed to measure the efficiency of the parallel algorithm as the problem size increases concurrent to the number of processes. In the table 6 the selected data size and their corresponding parameters are displayed. The partition size was adjusted for each raster size which were determined in during preliminary testing. For each raster size the partition size was determined by the best performing configurations that had a similar size. These tests can be found in appendix C.

Table 6: RSE Weak Scalability

Cluster Size	Workers	Partition Size	Cells Count	Time (s)	RSE (%)
2	1	200	1M	101.8	100%
4	3	600	3M	72.3	140.80%
6	5	600	5M	162.7	62.57%
8	7	900	7M	234.6	43.39%
10	9	500	9M	242.0	42.07%
12	11	500	11M	264.1	38.55%

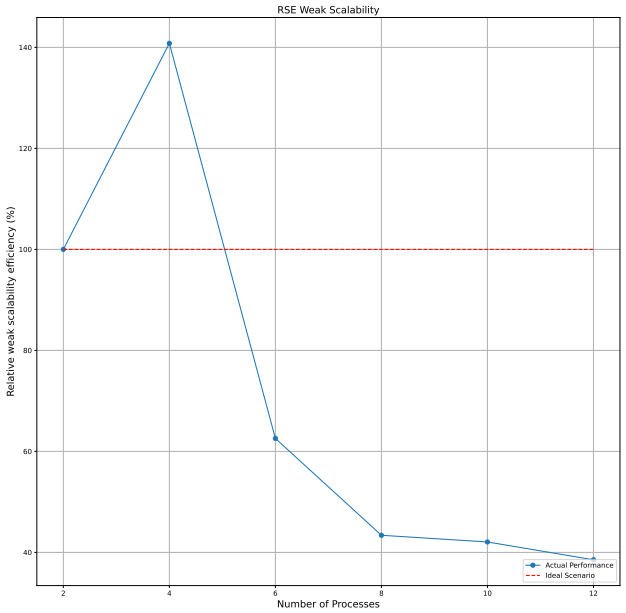


Figure 9: RSE weak scalability measurements for the parallel algorithm

5 Discussion

5.1 Parameter configurations results

The preliminary testing was needed to identify for different raster sizes and parameter configurations the ideal execution time[4] as presented in figure 4 and 5 in chapter 4. The figure displays the differences when the parameters are changed for the raster size 25 millions cells. This was done with more raster sizes that showed no significant differences where the overall pattern remains (which can be observed in the appendix C). Generally, across number of processes the overall pattern sort of remains the same. Nevertheless, when observing the latency across the number of processes there is an unsettling observation, which is as the number of processes increases it does not always improve the latency as it should (more on that under the following subsection run-time characteristics and scalability measurements).

The main take-away regarding different partition sizes is the different performances when varying the partition sizes. The pattern indicates that the ideal partition sizes that perform best on execution time lie somewhere in the middle range. The tested range of the partition sizes was determined by an assertion that ensures that the size of the processes is lower than the number of partitions indicating that each process has sufficient workload to work on. Notably there are some spikes and peaks at certain partition sizes. This could be explained by the remainders left on each partition that still need to be calculated. Meaning, since the partitions are divided equally across the processes there were always be some remainders that need to be processed after the first batch that was divided. This causes the execution time peaks. For example, in process number of 4 the ideal partition size is 2000 which results in 12.500 partitions. When using partition size 1600, the execution time spikes all of the sudden where the number of partitions is 15.625. Hence, 15.625 cannot be evenly distributed across the 4 processes given the way the partition dimensions are created. Resulting in more time spent on distributing the remainders again. Overall the partition size certainly does affect the performance of the parallel algorithm which seems to be a logical conclusion since the foundation of the parallel strategy is to divide the input raster into partitions.

There were other parameters that were tested that proved not to have as a significant affect as initially assumed. For the different number of cells it was expected that as it increases so does the run-time since as more cells are added the more cells that need to be processed. The initial idea to test this parameter is to observe if there is a change in performance when applying the different permutations on the different raster sizes. One notable observation is, when only 1 million cells were used the ideal partition size was at the lowest see appendix C figure 16. This is however not a distinct significant observation because other raster sizes behave the same, not necessarily at the lowest partition size but lower partition sizes tend to perform better than the higher ones, hence the middle range being the ideal sizes as discussed earlier. So this parameter did not necessarily affect the performance of the parallel algorithm significantly.

The second parameter is the number of source cells. When initially testing the different number of source cells with 1 million cells the run-time characteristics display no clear pattern as the chart in appendix C indicate. It was expected that as the number of source cells increases the longer the run-time, given how often the cells need to be recalculated and updated based on the number of source cells and their cost paths. However, there is no clear pattern when testing this on 1 million cells raster see appendix C figure 19. Whereas, when using a bigger raster with 4 million cells a pattern emerges. The lower the number of source cells the longer the latency needed as can be seen in figure 20 in appendix C. This was a surprising find. One potential explanation could be that as the number of source cells decreases the uniform distribution becomes less equally distributed, in which the source cells cumulative cost paths become longer. To get a better understanding the distribution of source cells need to be studied in more detail. In the current approach of creating these source cells the distribution was uniform and thus could be changed to find a clearer pattern of how the cells are recalculated based on the source cell distribution, rather than on the number of source cells. Note that the different raster sizes do not necessarily give a different pattern as given in the examples here solely because of their size. It could be that the reason for the different patterns is due to the fact that the 1 million cell raster was just too small for a pattern to emerge.

[4]Latency and execution time are used interchangeably in this paper

5.2 Parallel algorithm run-time characteristics

A parallel cost-distance algorithm with a min-heap structure using distributed memory was successfully developed and tested as the methodology and results in chapters 3 and 4 display. When observing the results from table 6 and figure 7 the average elapsed time per number of processes is displayed. This shows that the run-time has significantly been improved by the parallel algorithm in comparison to the sequential algorithm (using cluster 2). The sequential run-time across all partition sizes is 1754.76 seconds on average. The best run-time with the parallel algorithm across all partition sizes with 5 processes is 674.74 seconds on average. This indicates a speed-up of 2.6 as the equation 4.1 displays. Evidently, equation 4.2 shows when taking the average latency of all number of processes the speed-up remains the same which indicates a overall significant improvement in speed-up. Essentially, the overall run-time was improved by 2.6 times by the parallel algorithm in comparison to the serial algorithm.

However, looking back at figure 7 noticeably after process number 7 the run-time begins to increase significantly again, indicating an issue and potentially a presence of a bottleneck. This should not be the case because as more processes are added the better the performance should be. To get a better understanding of the number of processes pattern, and how well the parallel algorithm actually performs, scalability measurements need to be conducted.

5.3 Scalability of the parallel algorithm

5.3.1 Strong scalability

The strong scalability measurement highlights quite well how well the processes are working together. Meaning, how well they coordinate and communicate as more of them are added to work on the same number of cells. The ideal scenario of the performance of the parallel algorithm regarding strong scalability is to contradict the saying 'less is more'. Ideally, the more processes that are added the more efficient the program should become.

When observing the number of processes and their corresponding latency in table 5, it seems that the parallel algorithm has significantly increased in comparison to the sequential one [5]. However, when observing the relative strong scalability efficiency (RSE) as the number of processes increases, the RSE decreases. This display an efficiency bottleneck.

To get a better understanding the RSE rate can be examined as displayed in table 5 and figure 8 where a clear pattern can be detected. Initially, the graph starts at a 100% on the y-axis indicating that with one worker the program is utilizing the complete worker. Then the second cluster of workers is added (4) and immediately there is a drop-off of approximately 25% in efficiency. At every cluster of workers that is added there is a significant drop-off. Even though the margins are becoming slimmer, the drop-off in efficiency is still significant. The performance of this parallel algorithm regarding strong scalability is not ideal. Looking at the dotted line in figure 8 that would be the ideal scenario, where the efficiency regardless of the number of processes added remains the same at 100%.

To identify why this problem occurs the 'C_profile'[6] logs need be checked, to find a potential explanation. The first potential noticeable indicator is that there seems to be a numerous amount of sequential fractions that take up a lot of time processing. Functions such as cost-distance or getting the accumulated cost calculations consume quite some time. The accumulated cost function seems to be the most intense sequential function. Strong scalability measurement has a hard-time dealing with serial fractions given that when more processes are added to the same problem-size the serial fractions remain the main time consuming component of the algorithm regardless of the amount of processors added (LUE, 2023). This refers back to Amdahl's law where some software inherently cannot be fully parallelized and in this case cannot achieve great strong scalability performances. In the figures 10 and 11 the logs for getting accumulated cost for process 4 and 8 are displayed. Noticeably, its the same pattern regardless of the number of processes added. Additionally, as the number of calls increases the longer the total time needed for the function. So the number of cells definitely increases the execution time however that is expected.

[5] cluster size 2, 2 processes but 1 worker
[6] Statistics of the parallel algorithm

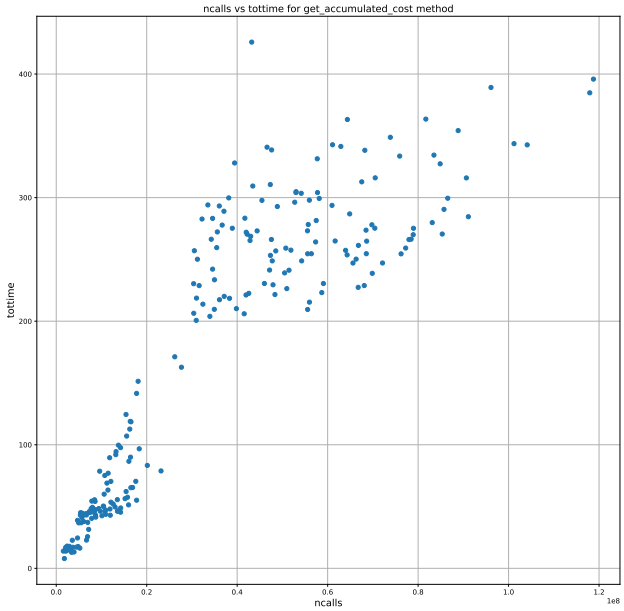


Figure 10: C_profile logs 'get_accumulated_cost' function vs number of calls & total time using 4 processes

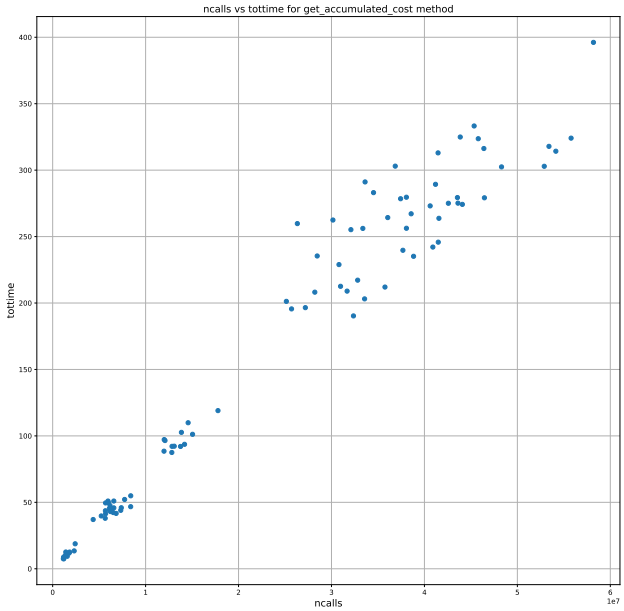


Figure 11: C_profile logs 'get_accumulated_cost' function vs number of calls & total time using 8 processes

The strong scalability efficiency of the parallel algorithm could be strongly affected by the sequential fractions of the initial algorithm. However, it is not a clear indication that this is certainly the reason of the poor efficiency performance. More tests need to be conducted to observe which cores are doing which tasks and maybe observe the distribution of the workload across processes more accurately. Great strong scalability efficiency will be difficult to achieve regardless whether the cost-distance algorithm being built as efficient as possible or if the workload distribution is perfect.

5.3.2 Weak scalability

When observing the weak scalability in table 6 or figure 9 initially there is an increase in efficiency when comparing the sequential algorithm (cluster 2) to the parallel ones that follow. However, once additional workers are added more and more the efficiency drops. Eventually at cluster 8 the marginal difference starts to decrease. The variables that are changing in RSE weak scalability is number of cells and the number of workers. Ideally, there should be no drop-off in efficiency, as the red dotted line in figure 9 suggests, it should be at 100% efficiency as more cluster of workers are added. Consequently, this leads to observe how the number of cells and the concurrent number of workers affect efficiency of the parallel algorithm. Beforehand, the peak increase in RSE at cluster 4 can be considered as unfair since the number of additional workers change goes from 1 to 3 whereas the following number of workers added remain with an interval of 2 as seen in table 6, so the RSE calculation is deceived.

The potential reason for the poor weak scalability performance of the parallel algorithm could be due to the fact that as more clusters of processes are added and the increase of the problem size there is more communication required between processes and resulting in communication overhead. When examining the 'C_profile' logs the time is mostly spent on the receiving end of the MPI communication structure. The receiving method is at the handler where the handler is expecting requests from the different workers. The issue lies that the handler is just waiting to receive the messages sent from the workers as the workers are processing their partitions. This potential issue could occur which was identified during the literature review. The figures 12 and 13 display that the higher the number of calls (is equivalent to the number of partitions) of the receiving method (comm.recv) the lower the the total time of the receiving time. The reason being when workers are working on smaller partition sizes, these partitions can be sent quick back to the handler and so on. However, with larger partition sizes the workers will need more time to process, whilst the handler is just waiting for the worker to send the partitions back.

When examining the C_profile logs the partition size and its corresponding number of partitions matter. The higher the number of partitions (which can be observed by the number of calls for the comm.recv method) the lower the receiving time and vice versa. For example there is a test with 4 processes that called the receiving method 14 times which indicates 14 partitions. Out of all the functions and methods, that was by far the highest time consuming function of the parallel algorithm. Hence, during the communication between handler and worker whilst the worker is undergoing cost-distance the handler is waiting for the 14 partitions. Given that there are only 14 partitions this indicates that the partition size is too large. So when the workers are doing there calculations on the larger partition size the handler is waiting to receive. This displays again how significant the partition size parameters affects the performance of the parallel algorithm. There is a pattern across the y-axis where with low number of calls the total time is high. This is due to the size of the raster. The higher the number of cells and with a high partition size (low number of partitions), this equates to low number of calls but a lot of time is spent on these small number of partitions. The two figures are shown to display that regardless of the number of processes used, the pattern remains the same. Which again reflects the overhead communication issue with the scalability of the parallel algorithm. The reason why the lower the partition size (to a certain extent) is better for the communication structure is because the workers will be finished quicker with it and send it back to the handler, so the handler wont be doing nothing for longer segments of time. With maintaining the workload per worker across the cluster sizes the inefficiency remains as its an communication bottleneck between the handler and the workers. The figure also indicates that at higher cluster the marginal difference decreases. In the logs this is also reflected which is due to the number of processes available send messages back to the handler. Meaning, when more workers are available there could be more messages that can be sent back so the handlers receiving time is reduced. However, going across the logs of the different number of processes the receive method remains one

of the top time consumers. Indicating that its the main bottleneck in relative terms for the poor weak RSE performance.

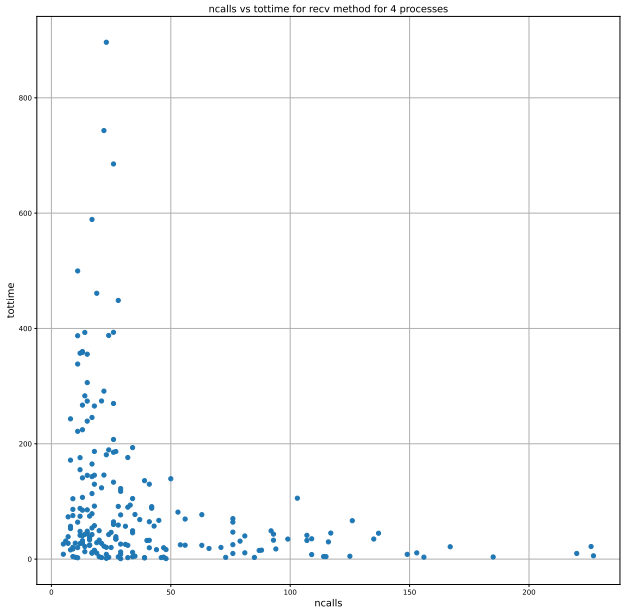


Figure 12: C_profile logs 'mpi.recv' method vs number of calls & total time using 4 processes

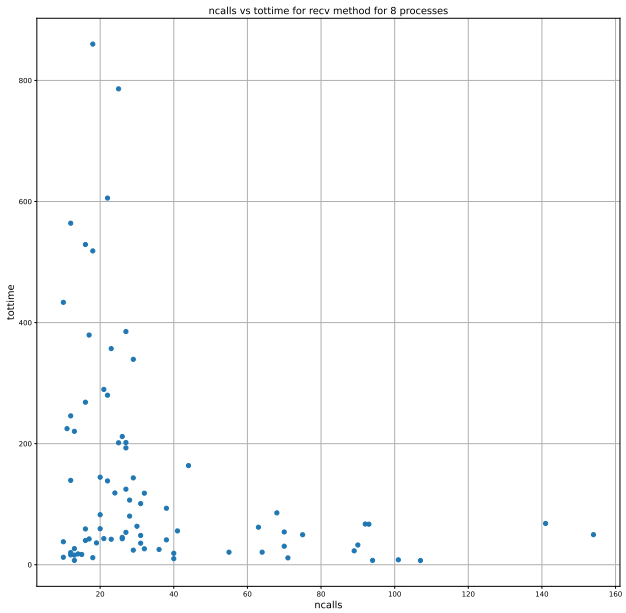


Figure 13: C_profile logs 'mpi.recv' method vs number of calls & total time using 8 processes

Additionally, the number of calls for cluster 4 indicates that the partitions are being received more often however that is due to the assertion that the number of partitions needs to be fully utilized. The weak scalability efficiency of the parallel program is affected by the communication structure since as the problem size increase concurrent with the number of workers, the handler will remain consuming time waiting to receive regardless of number of processes workload added.

Another potential indicator of why the weak scalability efficiency performed poorly is the the amount of time that is spent on checking boundaries of the partitions. The time spent on it is not as significant as with the receiving method however its still in the top 10% where the run-time is being spent on. The 'is_on_partition_boundary' function check if cells are on the partition boundary to eventually update and communicate it with neighbouring partitions. As the size of the raster increases so does the number of partitions, therefore the amount of boundary cells increases. This will make the number of processes added redundant since as the size increases concurrent to the number of processes the number of boundary cells becomes larger making 'is_on_boundary' function being called more often which again increases the run time, as the figures 14 and 15 display. Again the two figures for process 4 and 8 were added to show that the pattern is consistent across processes.

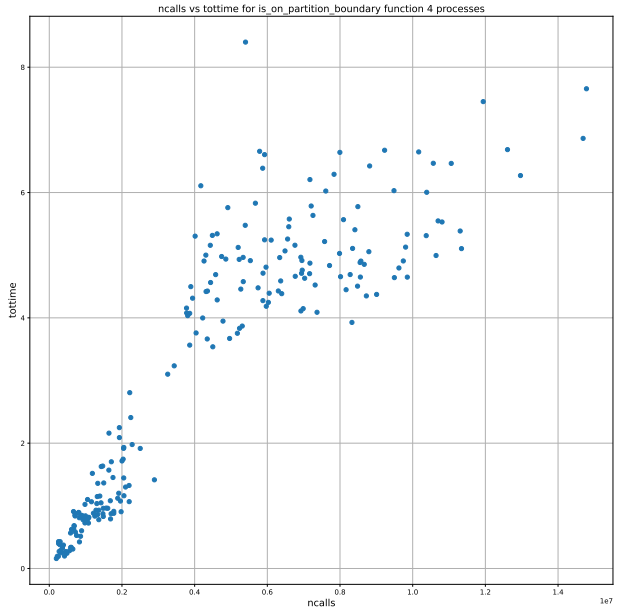


Figure 14: C_profile logs 'is_on_partition_boundary' function vs number of calls & total time using 4 processes

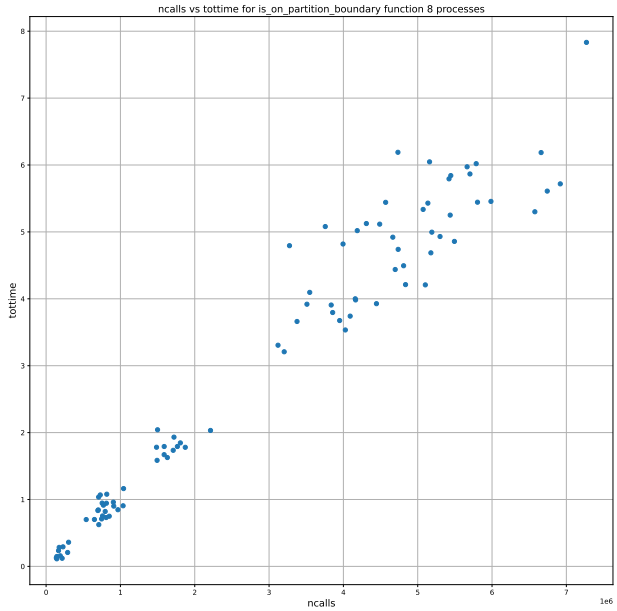


Figure 15: C_profile logs 'is_on_partition_boundary' function vs number of calls & total time using 8 processes

These two possible explanations for the poor RSE weak scalability performance are only potential explanations and do not verify the weak efficiency performance completely. To assure why these scalability performances are poor additional tests need to be conducted. Additional observations need to be measured regarding how these processes distribute the workload across the number of processes and how to the different variables affect that. So that during testing this is measured and can be examined furthermore.

5.4 Limitations

The first limitation of the parallel algorithm refers back to Amdahl's law that certain software contain sequential fractions and are embedded into the function of the program. For the cost-distance algorithm it is not any different. The cost-distance fraction consumes the majority of the run-time making the parallel algorithm still limited to improve run-time.

The second limitation refers to the MPI communication structure currently in place. Currently, the one Handler process is just waiting during receiving especially when the partition size increases. This limits the flexibility of the program when scaling and leads to major bottlenecks as presented by both scalability measures. Additionally when this is scaled to an exponential amount of workers the waiting time will increase exponentially as well or potentially it might be reversed where the handler is too much workload to process as the number of workers increase the more messages are being sent back. This was well reflected in the weak scalability part where the addition of processes and concurrent number of cells did not utilize the workers as it should therefore decreasing in efficiency.

Another major limitation that can be time consuming is finding the ideal parameters for the baseline of the cost-distance parallel algorithm. The way it is currently set-up the run-time is majorly affected by the partition size as described earlier. The lower the number of partitions the higher the time is spent on checking the partition boundaries and the longer the receiving segment. However, having more number of partitions, the more the number of partitions need to be processed by the workers which again takes time given the sequential fractions. Finding the right balance between the two aspects is therefore required but thus can result in significant time consuming efforts.

One other limitation is the way the input raster is divided. In the current form it can only be divided in square fractions and therefore the input raster has to have a square shape resulting in the algorithm not to be able to deal with all types of raster shapes.

5.5 Suggestions

To improve the parallel cost-distance algorithm it is suggested to focus on the MPI communication structure since the sequential fractions will always be sequential and therefore time consuming. Looking at the issues and bottlenecks that were identified there is one major concern which is the waiting time of the handler whilst the workers are processing the partitions. One suggestion could be is that while the handler is waiting and the workers are processing their partitions the handler can work on partitions that 1) are part of the remainder if the number of partitions cannot be directly distributed evenly, 2) if they are evenly distributed then work on the partitions if all the workers are currently not available (which the handler knows of due the available workers queue). These should follow a asynchronous communication where the handler is not halted by the workers to improve run-time. Hence, this suggestion could potentially improve the run-time and the performances of the scalability measures. Regarding the partition boundary this could be a preprocessing step instead of ingrained into the cost-distance algorithm and checked every time. So before starting the cost-distance the partition boundary can be already defined and the coordinates and indices can be stored in the list, similarly what was done to the source cells and x and y offsets. Furthermore, to get a better understanding of the run-time characteristics and scalability performances it is suggested that further observations are needed regarding the calculations and processing of the cells. One possible outlook can be the affect of the distribution of the source cells in combination with different partition sizes. These results can provide better understanding of how often each cell is processed and why they are processed so often. Thus could help in optimising the algorithm to improve run-time furthermore.

6 Conclusion

In this research paper a new parallel cost-distance with min-heap algorithm was introduced that was able improve the run-time characteristics by 2.6 times using a message passing interface (MPI) for distributed memory. The parallel program was developed to enable the algorithm to divide the input raster into different partitions so that each process can work on the cost-distance calculations simultaneously to accelerate the process. Once the algorithm was developed different parameters such as partition size, number of processes and number of cells were used to observe the affect on the parallel algorithm and to find the ideal parameter configurations to properly test the algorithms scalability. It turns out the partition size played a major role in finding the ideal execution time for the parallel algorithm. In retrospect this can be expected since the parallel strategy was to divide the input raster into partitions. On the other hand the raster size did not matter as much in terms of change in pattern when the size was varied. For the number of processes, although the overall pattern remained similar across the number of processes with some run-time improvements, it also showed that when adding more processes it did not always improve run-time. Thus was very apparent when testing the algorithms scalability. Although a faster run-time and a positive speed-up ratio was achieved eventually the parallel algorithm ran into overhead communication bottlenecks as the weak and strong scalability measurements clearly displayed. When running diagnostics the main possible explanation for these poor scalability performances is the receiving method at the handler process since it waits too long to receive messages from the workers especially when the partition size is large. Regardless of the number of processes added the waiting receiving time remains high. Other issues were also found that could also potentially explain the poor scalability performance such as the high run-time required for sequential fractions and the checking of cells on the partition boundaries. To get a better understanding of why the scalability performances are poor additional measurements are required such as checking how often each cell is being recalculated by each process whilst using different distributions of source cells. The proposed solutions for the current issues that were identified is to modify the handler so that while its waiting it can undergo other tasks that could potentially reduce run-time. Regarding the partition boundary this could potentially be solved in initial steps where the indices and coordinates of the partition boundary cells can be identified before the cost-distance instead of checking it within the cost-distance function. Overall, the research has provided a potential solution of how a parallel cost-distance algorithm can be constructed. Additionally, some insights were found on how the algorithm works and what parameters affect its performance regarding run-time and scalability. Future works could focus on optimising the communication structure to avoid or reduce overhead communication bottlenecks and to further examine exactly why the scalability performances are poor.

7 Appendix

7.1 Appendix A

7.1.1 The cost-distance algorithm with min-heap pseudo-code

```
function: do_cost_distance(input raster, raster layout, cell size = 1)
 cost_surface_pad = cost_raster.data
 accumulated_cost = cumulative_cost_raster.data
 padding = cost_raster.PADDING
 partition_size = cost_raster.PARTITION_SIZE
 # Create a mathematical set to store the change of the updated values
 change = set()
 neighbour_range = range(-1, 2)
 active_cells = create_active_cell_dict(accumulated_cost)
 assert that the cost_surface_pad shape == accumulated_cost shape /
 == (partition_size + 2padding, partition_size + 2padding)
 while active_cells is not empty:
 # use heap as a priority queue for active cells
 local_value, local_tuple = heapq.heappop(active_cells)
 # Iterate over neighbouring values to update cost values and change the queue if needed
 for i in neighbour_range:
    for j in neighbour_range:
        if not (i == 0 and j == 0):
            calc_y, calc_x = local_tuple[0] + i, local_tuple[1] + j
            calc_tuple = (calc_y, calc_x)
            if 0 < calc_y <= partition_size and 0 < calc_tuple[1] <= partition_size:
                distance = get_distance(i, j, cell_size)
                new_value = get_accumulated_cost(cost_surface_pad, local_tuple,/
                calc_tuple, distance, local_value)
                old_value = accumulated_cost[calc_tuple]
                if new_value < old_value and abs(new_value - old_value) > .5:
                    accumulated_cost[calc_tuple] = min(accumulated_cost[calc_tuple], new_value)
                    heapq.heappush(active_cells, (new_value, calc_tuple))
                    if is_on_partition_boundary(calc_y, calc_x, padding, partition_size):
                        neighbours = get_rel_neighbour_pos(calc_tuple, padding, partition_size)
                        change = change.union(neighbours)

 return the accumulated_cost and the change to be used to update the values.
```

7.1.2 Appendix A: The cumulative cost function pseudo-code

```
function: get_accumulated_cost(cost_surface_pad, local_tuple, distance, local_cost, calc_tuple)
  returns the cost-distance accumulated cost calculation = (cost_surface_pad[local_tuple] +
  cost_surface_pad[calc_tuple])/2) x (distance + local_cost)
```

7.2 Appendix B

7.2.1 Handler tasks pseudo-code

```
# Check if the handler is root node
if rank == 0
 initialize the output raster using GDAL
 write an array with infinite values for initialisation
 # check for no data values in the raster if so copy them to the output raster
 if no data value != None:
  raster.setnodatavalue(nodatavalues)
 # Create a set to check the numbers of partitions that need to be processed
 # Iterate over the number of partitions that need to be processed
 # The handler will receive message to see which partitions are still available
 partitions_to_check = set()
  for i in range(size - 1):
   message: [tuple] = comm.recv()
    for partition in message:
     partitions_to_check.add(partition)
  # Create a queue of avalaibale workers
  enter a while loop
   if worker_avail_queue size is (size - 1) and partitions_to_check is empty
      iterate over range from 1 to size - 1
          send a termination message to worker i
      break the loop if the termination message has been called
  while the queue is not empty:
   calculate valid_partitions_to_work_on as the difference /
   between partitions_to_check and partitions_working_on
  if the queue is empty:
   get a worker_to_send_to from worker_avail_queue
   get a partition to send from valid partitions to_work on
   remove partition to send from partitions to check
   add partition to send to partitions working on
   create cumulative raster as a new Raster object with the specified parameters
   send partition to send and cumulativeRaster to worker worker to send to

  receive worker_rank, worker_partition, new_partitions, and array from worker

  put worker rank back into workers available queue
  remove worker partition from partitions currently working on
      iterate over new_partitions
          add each partition to partitions to check

  overwrite data in CumulativeCostRasterBand with array
```

7.2.2 Worker tasks pseudo-code

```
# check if workers
if rank != 0
# Check the partition sizes, dimensions and offsets of the assigned partitions
set avalailable workers size - 1 (since handler = 0)
```

```
# identfiy cells to handle for equal distribution across workers
calculate number of cells to handle as the ceiling of the division of number of grid cells //
available workers
# calculation starting and ending positions of the partitions
starting position = (ranks - 1) x number of cells to handle
ending position = minimum value between the length list of cells - 1 /
rank x number of cells to handle
create empty list partitions to send
iterate over partitions
initiaize source raster
if in the source raster value = 1
 add x,y coordiantes of the source to send
send paritions to send to rank 0
enter a loop
 receive partition from rank 0
 if partition is equal to error message
  break the loop no more partitions
 create cost raster with global boundary as the result of adding global boundary and padding to costRa
 create cumulative cost raster with global boundary /
 as the result of adding global boundary and padding to cumulative cost raster
 create an empty to list niehgbours to update
 call do cost distance to calculate cost distance and change
 if change is true
  call the get neighbouring partitions function with partition, change and coordinates as arguments
  create new outpute raster without boundary as the result /
  of removing the global boundary from initial output easter
 send rank, partition, neighbours_to_update, new output raster to rank 0
```

7.3 Appendix C

7.3.1 Additional preliminary results

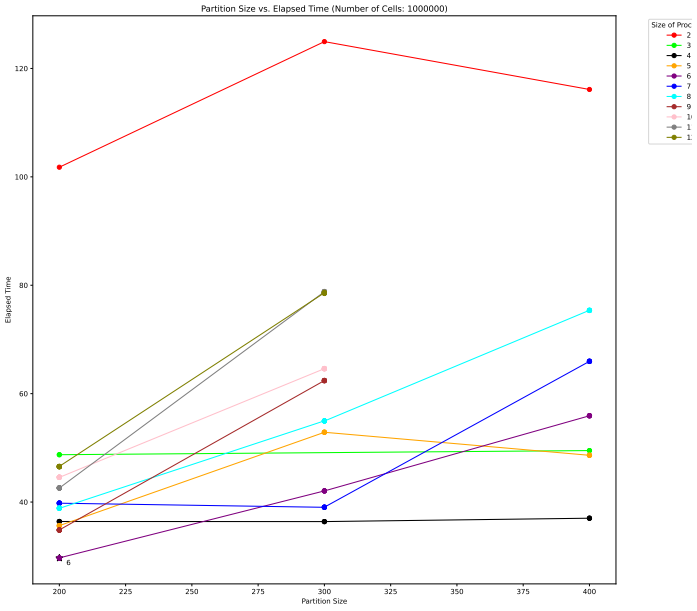


Figure 16: Partition size & number of processes vs execution time with 1 million cells

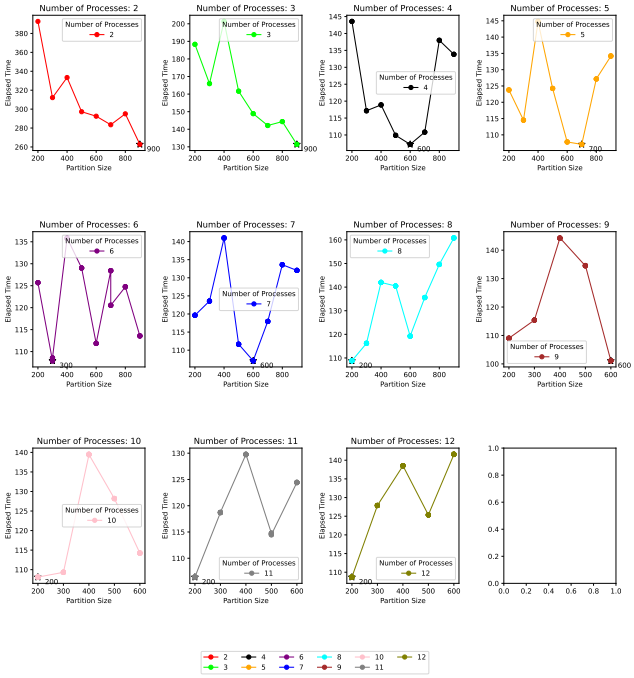


Figure 17: Sub-plots of across the number of processes & partition size with 4 million cells

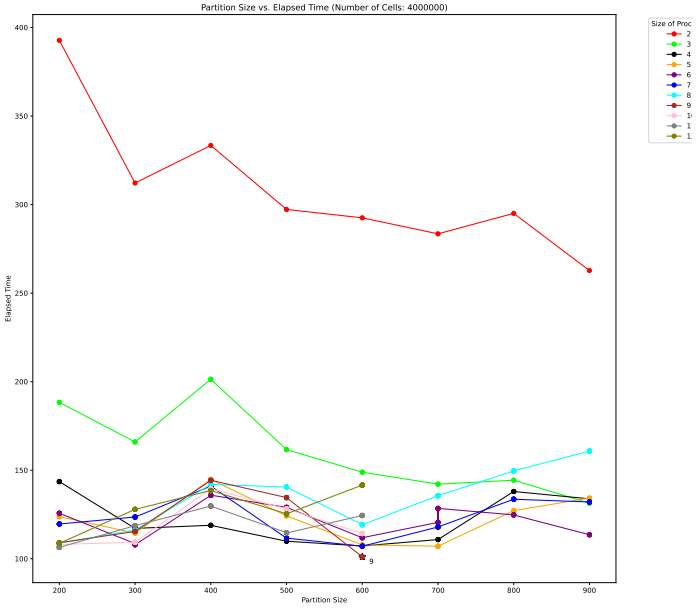


Figure 18: Sub-plots of across the number of processes & partition size with 4 million cells

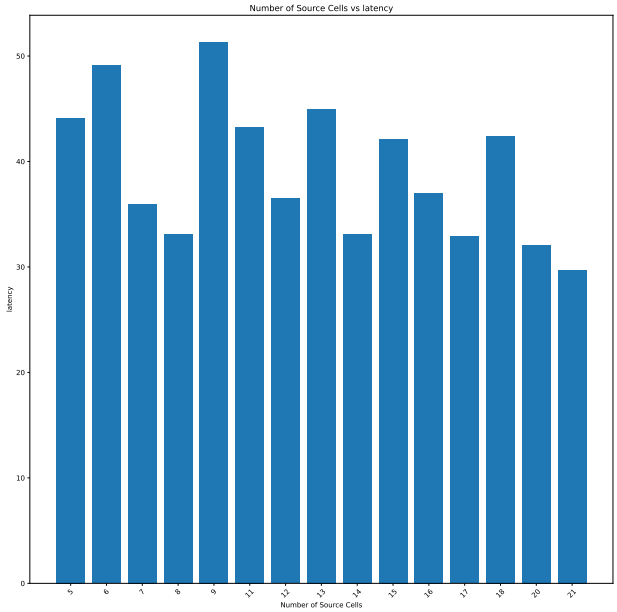


Figure 19: Bar chart of number of source cells vs latency 1M cells raster

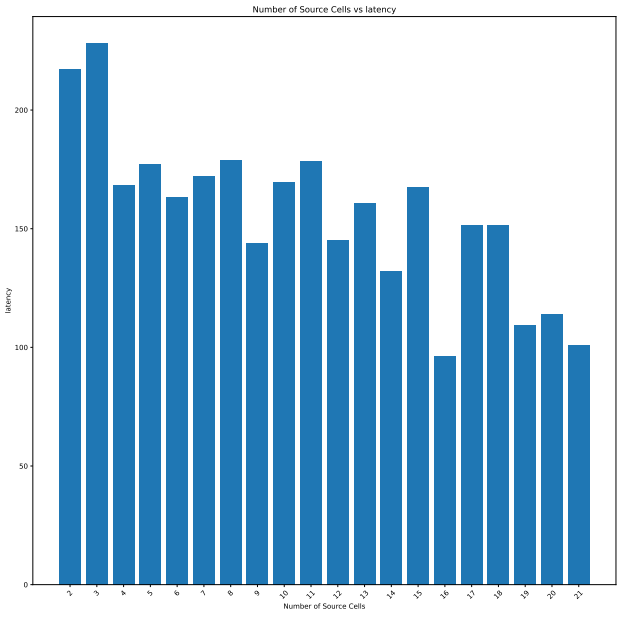
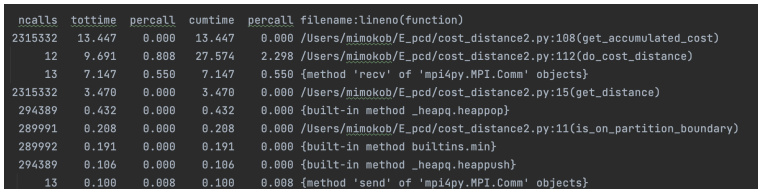


Figure 20: Bar chart of number of source cells vs latency 4M cells raster

7.4 Appendix D

7.4.1 Example of c-profile logs



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2315332	13.447	0.000	13.447	0.000	/Users/mimokob/E_pcd/cost_distance2.py:108(get_accumulated_cost)
12	9.691	0.808	27.574	2.298	/Users/mimokob/E_pcd/cost_distance2.py:112(do_cost_distance)
13	7.147	0.550	7.147	0.550	{method 'recv' of 'mpi4py.MPI.Comm' objects}
2315332	3.470	0.000	3.470	0.000	/Users/mimokob/E_pcd/cost_distance2.py:15(get_distance)
294389	0.432	0.000	0.432	0.000	{built-in method _heapq.heappop}
289991	0.208	0.000	0.208	0.000	/Users/mimokob/E_pcd/cost_distance2.py:11(is_on_partition_boundary)
289992	0.191	0.000	0.191	0.000	{built-in method builtins.min}
294389	0.106	0.000	0.106	0.000	{built-in method _heapq.heappush}
13	0.100	0.008	0.100	0.008	{method 'send' of 'mpi4py.MPI.Comm' objects}

Figure 21: Screenshot of C_profile data example

References

Balaji, P. et al. (2011). "Non-data-communication Overheads in MPI: Analysis on Blue Gene/P". In: *Mathematics and Computer Science Division, Argonne National Laboratory*.

Chen, GuoLiang et al. (2009). "Integrated research of parallel computing: Status and future". In: *Journal of Parallel and Distributed Computing* XX.X, pp. XX–XX.

Cornell University (2023). *Parallel Machines*. Cornell Virtual Workshop. URL: https://cvw.cac.cornell.edu/parallel/machines.

Fink, Zane et al. (2021). "Performance Evaluation of Python Parallel Programming Models: Charm4Py and mpi4py". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM. USA.

Graham, R. L. and G. Shipman (2008). "MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by A. Lastovetsky, T. Kechadi, and J. Dongarra. Vol. 5205. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. DOI: 10.1007/978-3-540-87475-1_21.

Greenberg, Jonathan A. et al. (2011). "Least cost distance analysis for spatial interpolation". In: *Computers Geosciences* 37.2, pp. 272–276. ISSN: 0098-3004. DOI: https://doi.org/10.1016/j.cageo.2010.05.012. URL: https://www.sciencedirect.com/science/article/pii/S0098300410002578.

Hoefler, Torsten, Florian Lorenzen, and Andrew Lumsdaine (2008). "Sparse Non-blocking Collectives in Quantum Mechanical Calculations". In: *Open Systems Lab, Indiana University* 1.

Kotov, V. and I. Sergienko (2009). "Measuring the Performance of Parallel Computers with Distributed Memory". In: *Cybernetics and Systems Analysis* 45.6, pp. 897–905. DOI: 10.1007/s10559-009-9192-3.

Laccetti, Giuliano, Marco Lapegna, and Valeria Mele (2016). "A Loosely Coordinated Model for Heap-Based Priority Queues in Multicore Environments". In: *International Journal of Parallel Programming*.

Lu, Hao et al. (2011). "An effective cost distance calculation based on raster data model improved algorithm". In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. Vol. 4, pp. 2214–2218. DOI: 10.1109/ICCSNT.2011.6182416.

LUE (2023). *Scalability*. Computational Geography. URL: https://lue.computationalgeography.org/doc/manual/framework/scalability.html (visited on 2023).

MPI4py: Python bindings for MPI (n.d.). https://mpi4py.readthedocs.io/.

Murekatete, Rachel Mundeli and Takeshi Shirabe (2018). "A spatial and statistical analysis of the impact of transformation of raster cost surfaces on the variation of least-cost paths". In: *Journal of Spatial Science*.

Open MPI: Open Source High Performance Computing (n.d.). https://www.open-mpi.org/.

Princeton research computing (2023). *Parallel Computing with MPI and OpenMP*. https://researchcomputing.princeton.edu/support/knowledge-base/parallel-code.

Schürhoff, Daniel (2021). *Performance Metrics & Measurements*. Presentation at RWTH Aachen University.

Shi, Yuan (1996). "Reevaluating Amdahl's Law and Gustafson's Law". In: *Computer and Information Sciences Department*.

Shoeb, Abu et al. (2012). "Performance Analysis of MPI (mpi4py) on Diskless Cluster Environment in Ubuntu". In: *International Journal of Computer Applications*. CITATIONS READS 2 154. DOI: 10.5120/9764-3701.

Skinner, David and Katie Antypas (2010). *Parallel Application Scaling, Performance, and Efficiency*. Presentation. Presented at the US Department of Energy. URL: https://www.nersc.gov/assets/NUG-Meetings/IntroMPIApplicationScaling.pdf.

Understanding Cost Distance Analysis (2023). https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/understanding-cost-distance-analysis.htm.

Wang, Y. et al. (n.d.). *Parallel Algorithm for Calculating Cost Distance of Raster Data*. School of Computer Science, China University of Geosciences.

What is Raster Data? (2023). https://desktop.arcgis.com/en/arcmap/latest/manage-data/raster-and-images/what-is-raster-data.htm.