Geospatial Processing in the Cloud

Hampus Paulsson
Abstract

Geospatial Processing in the Cloud

Hampus Paulsson

Cloud computing is an ubiquitous term that encompasses on-demand computing and related services over the Internet. This thesis work aims to develop a parallel program for processing geographical raster data and use it on the Amazon Web Services cloud platform for parallel, distributed processing. We will also investigate the program’s performance on the cloud platform and investigate different instances to see how they affect the program performance and network capabilities.

We found that Amazon Web Services could be used for parallel processing using Message Passing Interface and that the program scales with an increased number of workers used, but that the scaling is limited by the time it takes to transfer data between nodes in the cluster. We also saw how network performance as shown by bandwidth tests varies depending on what instance is used.

Ett sätt att få tillgång till datorer är via så kallade molnplattformar, en web-baserad platform som tillhandahåller datorkraft och relaterade tjänster, där man betalar för de resurser man använder. På så vis kan man spara pengar genom att man på egen hand inte behöver investera i dyra datorer.

I det här examensarbetet har vi undersökt om man kan använda en molnplattform som heter Amazon Web Services för att kunna göra krävande beräkningar på geografisk höjddata. Detta gjordes genom att implementera en algoritm för att beräkna så kallade isoliner på höjddatat, sedan parallellisera denna och slutligen använda den på molnplattformen för att undersöka om vi kan utföra våra beräkningar snabbare än en seriell implementation av algoritmen. Vi undersökte även kostnader för användandet av molnplattformen.

Vi såg att vår parallelliserade algoritm skalade med antalet processorer och antalet datorer som användes, men att skalningen begränsades av den tid det tar att skicka höjddata mellan de olika datorerna som användes på molnplattformen. Vi såg även att tiden det tar att utföra beräkningarna varierar med vilken typ av dator man använder på molnplattformen, men att skalningsegskaperna tycks vara lika för de olika datortyperna.
1 Introduction

1.1 Purpose

The purpose of this thesis work is to investigate a commercially available cloud platform (Amazon Web Services) for doing geospatial analysis. The focus will be to develop a parallel program that can be used to assess the scaling capabilities of the program in the cloud, as well as investigating the Amazon Web Services cloud platform from a more general standpoint to determine whether it could be used in future geographic information system (GIS) applications.

1.2 Background

There is a clear trend in most areas of computing towards the need for more powerful computers, be it scientific applications, gaming or commercial applications. Simulations and data analysis in physics and chemistry work with datasets with sizes in the order terabytes or even petabytes, requiring thousands of processors and highly sophisticated computer programs to be performed. The same trend is also observed in commercial and industrial applications such as in media and advertisement where companies analyze large datasets to find trends or correlations which could be used as a business advantage, this is generally known as Big Data. Another notable and criticized example of an application requiring immense computing is high-frequency trading, where financial institutions and corporations leverage computers coupled with sophisticated mathematical models and algorithms to automate trading [5].

The up-front costs for establishing an in-house data center are quite substantial. Although many companies do have their own data centers, it is getting more common for companies to use existing data centers when needed. This have given rise to what today is commonly called cloud computing. Some companies, most notably Amazon.com, Microsoft, Google and Salesforce.com saw a market opportunity in providing on-demand computing as a service, which essentially encapsulates the most fundamental part of cloud computing. Most cloud platforms comprises several important services to substitute an in-house data center, such as networking, databases, storage and also more high-level services such as analysis, artificial intelligence and software management.

The cloud platforms are not without their drawbacks; computers are virtualized on hardware that could be shared with several users, which could lead to serious performance degradation as well as security concerns. This loss of control is applicable to the other services, such as networking and storage and should be taken into account when deciding whether to migrate or use a cloud platform.

In this thesis work we will investigate some of properties of the Amazon Web Services cloud platform and assess whether it could be of use for a company (Carmenta AB) developing geographic information systems. We will investigate parameters such as performance, scaling, networking and host start-up times.
2 Problem Description

This work will be focused on two main tasks. The first task is implementing an algorithm to compute isolines on geographical raster data. The second task will be to use the above mentioned algorithm on the Amazon Web Services cloud platform as an assessment tool.

The first task will be to implement an algorithm called the Marching Squares algorithm. It will be implemented in the C++ programming language. The Marching Squares algorithm is used to calculate isolines on scalar data such as height or pressure. A large part of the implementation task will also be to parallelize the algorithm with multi-threading and distributed parallel computing to increase the performance and also enabling us to use several computers for calculations, which will be of importance for the second task.

The second task will focus on the Amazon Web Services cloud platform. Our interests will lie in investigating the cloud platform as a means of doing parallel computing. We will investigate how different cloud configurations will affect the performance of our application by measuring execution times, scaling capabilities, efficiency and network transfer rates.

3 Theory

This section will be dedicated to describing the marching squares algorithm that was used in this project to calculate the isolines, as well as giving a background on parallel computing and relevant performance metrics for our work.

3.1 The Marching Squares Algorithm

The Marching Squares algorithm is a special case of the Marching Cubes algorithm which was presented in 1987 [10]. It’s usage is diverse and ranges from image analysis to GIS, for which it will be used in this case. The MS algorithm is used to generate contour lines (isolines) from a two-dimensional scalar data set. Our scalar dataset will be a finite set of pairs \((W, H)\), where the set of coordinate vectors \(W = \{w_i \in \mathbb{R}^2 | i = 1, \ldots, n\}\) is spanning a domain \(\Omega \subset \mathbb{R}^2\), and with the accompanying set of scalar values \(H = \{h_i \in \mathbb{R} | i = 1, \ldots, n\}\) of the field \(f(x, y)\), sampled at \(W\) so that \(h_i = f(w_i)\). We also define a Cell grid as a mesh \(\Gamma\) that subdivides \(\Omega\) in rectangular cells with their vertices at the points of \(W\). For a given isovalue \(q \in \mathbb{R}\) we define an isoline as the set \(L(q) = \{l \in \Omega | f(l) = q\}\). In our case, \(L\) will consist of line segments that are defined on the cells \(\gamma_j \in \Gamma\). A cell is active at \(q\) if \(\min h_j \leq q \leq \max h_j\).

The marching squares algorithm is composed of the following steps:

1. Read the data that the contour lines is to be calculated for
2. Perform a binary filtration on the data with a chosen threshold value
3. Create a cell grid (matrix) of size one pixel less than the original data in each dimension
4. Center the grid on the filtered data set and let vertices of the cell grid, each placed in a separate filtered data cell, make up binary indices.
5. The indices will constitute a 4 bit binary value (0-15), each representing a certain contour line configuration in the cell.
6. Use the original data values to interpolate where the boundary crossings of the contour lines will be in each cell.

7. Iterate through the grid and build up lines from the line segments.

Figure 1 shows the relationship between the raster data and the overlayed cell grid. In a GIS application, the data matrix would typically contain some kind of raster data. The data matrix (black) will be of dimension $M \times N$, and the grid (red) will be of dimension $M - 1 \times N - 1$. After binary filtering the data matrix, according to a threshold value $q$, each data cell will contain a binary one or zero. When the grid is overlayed, each corner of a cell in the grid will contain a bit, and the four bits in total for each cell will constitute a binary index with integer values in the range 0-15.

Figure 1: A schematic picture of the relationship between the data and the grid used to calculate the contour line configurations. The value $G(i,j)$ is made up by the raster cell values at indices $(i,j), (i,j+1), (i+1,j)$ and $(i+1,j+1)$.

Figure 2 gives an overview of a single cell in the grid. The cell will contain information about the binary values of its vertices $v_{j1}, \ldots, v_{j4}$, as well as the local coordinates of the boundary crossing points $e_{j1}, \ldots, e_{j4}$. The most significant bit (MSB) of the binary points making up the binary number is in the upper left corner. Going in a clockwise direction toward the least significant bit in the lower left corner will build the binary number for that cell.
Using the vertices of the cell we will be able to represent a total of 16 ($2^4$) different cell configurations. Figure 3 displays all the different cell configurations. The black vertex points represents a binary one, meaning that the vertex point lies in the interior of the area enclosed by a contour line. Extra care has to be taken when considering case 5 and 10 because these configuration will emerge at saddle points, and one have to decide whether the contour line encloses the center of the cell or not. Figure 4 demonstrates this further.
3.1.1 Time Complexity of the Marching Squares Algorithm

Considering the fact that all the steps as described earlier in the algorithm description are independent of each other and are performed only once for each cell and isoline value, we can conclude that the time complexity of the Marching Squares algorithm is linearly proportional to the number of cells, \( n \), in the raster data. The number of line segments and hence the time complexity for constructing the isolines will be dependent on the number of active cells \( k \). From this we conclude that the algorithm has a time complexity of \( O(n + k) \).

3.2 High Performance and Parallel Computing

Since the early days of the microprocessor the processor, performance have been steadily increasing. Advancements in circuit design, faster clock frequencies and an increased transistor density are all factors contributing to this performance increase. However, a number of factors are making the quest for increased performance an ever increasingly difficult task. One factor contributing to this is the diminished gains in performance from increasing the clock-frequency of the processor. This manifests in the form of several problems [8];

- The power wall - The trend towards increasing the clock frequency as well as the transistor density leads to an increase in the power density and hence the heat generation in processors. Cooling has become increasingly difficult.

- The memory wall - The performance discrepancy between processors and memory have increased over the years. With the latency getting proportionally bigger for fetching data
from memory, this makes up a bottleneck in computer applications. CPU cache memories can reduce these effects some what.

Of course there are other factors contributing to the deceleration in performance increase of processors, such as physical limitations, design considerations etc., but the above mentioned factors are some of the main contributors.

One way to increase the performance of computer systems is the use of parallel computers. In a sense, most modern computers has some form of parallel processing capabilities. At the lowest level, the instructions executed in the processor are pipelined to increase the throughput, enabling execution of several instructions per clock cycle, such as fetching the instruction from memory, decoding, execution and accessing data from memory. Keeping the pipeline full is vital for optimal performance, but this is far from a trivial task and a failure to do so will result in a significant performance penalty. On a higher level, the use of multi-core processors is another way to enable faster execution. By increasing the number of cores, one has the ability to write programs which can distribute the workload between the processor cores, and either speed up the execution of existing problems, or solve larger problems in an equal amount of time.

3.2.1 Parallel Programming Paradigms

Parallel programming is usually divided into two paradigms, where the choice of whether to use one over the other comes down to what kind of problem need to be solved and the underlying system architecture. The two parallel programming paradigms are distributed-memory and shared-memory.

3.2.1.1 Shared Memory Systems

Shared memory systems can be described as a set of central processing units (CPU), operating independently but sharing the same memory resources. In networking and parallel computing, one often refer to a standalone computer as a node. Figure 5 gives a simple example of a shared memory system, where four CPUs are connected to a shared memory.

![Shared Memory System Diagram]

Figure 5: A schematic depiction of a shared memory system. Several CPUs share the same memory.

Shared memory systems has their advantages and disadvantages; letting the CPUs share memory gives the programmer a relatively easy programming perspective of the memory, making the development process easier in some cases. Since the CPUs have a small physical distance to the memory, data sharing between tasks and threads\(^1\) is fast and uniform.

\(^1\)A thread can be viewed upon as a “unit of execution” of a program; a program can often be split into multiple parts which can be run in parallel or concurrently. This is done by these parts in different threads, which are assigned by the programmer and often managed by the operating system.
One drawback with shared memory systems is that the scaling capabilities are diminished by the increased traffic between the memory and the CPUs, leading to slower access times and an increased risk of cache conflicts. The programmer is responsible for synchronization between threads to avoid this. One also have to be aware of other multi-threading conflicts such as race conditions and deadlocks, where frequent usage of shared variables can lead to unexpected results or that the program freezes.

3.2.1.2 Distributed Memory Systems

Distributed memory systems are comprised of several nodes, each with its set of CPUs and memory, connected through a network, commonly called interconnect for distributing data between them. One often used technique for distributed parallel programming is message passing, where a set of routines for point-to-point and collective communication is provided through an application programming interface (API). Figure 6 displays a simplified picture of several nodes in a network. The collective memory in the network can be seen as distributed between the nodes, hence the name distributed memory.

![Figure 6: A schematic depiction of a distributed system. Nodes are interconnected through a network for sharing data between them.](image)

3.3 Parallel Computing Performance and Performance Metrics

Making a parallel program is not a trivial task. Serious thought have to be put into the choice of algorithm or algorithm design, and the implementation might have to be customized depending on the underlying system architecture. Depending on the implementation, the performance gain for a parallel program can vary greatly, but there are a few relevant metrics that can be used either to measure the efficiency of the parallel implementation or to get an estimate of the expected performance.

3.3.1 Limiting Factors

There are many factors that limit the performance of parallel programs. One is the implementation - the program has to carefully be designed in order to fully exploit the underlying hardware of the system. Failing to do so can result in a parallel program with even worse performance than a serial implementation.
False sharing

False sharing is a sometimes subtle problem that happens when several CPUs are trying to write to the same variable, or to several variables lying close to each other in memory. The variables will be stored in each CPU’s cache and when one CPU writes to its cache-local variable, the cache line will be invalidated in the other CPUs, meaning they have to fetch the most recent version of the variables from memory again, leading to a lots of additional reads from the main memory.

Data dependency

Data dependency is another limiting factor - some problems are inherently hard to parallelize efficiently because the data used in one step is dependent on data which is processed in a previous step.

Load balancing

Load balancing is another crucial problem that have to be addressed in parallel programming. Not balancing the loads properly between CPUs or nodes will lead unused computing resources, which will limit the potential performance increases in a parallel application.

Network

Network performance is of great importance for distributed parallel programming applications as well as other application which rely on high bandwidth and low latencies, such as web applications and services. The fundamental idea of parallelization is to subdivide a problem into smaller ones, which then can be solved in parallel. For this to be done efficiently, one generally want to keep data dependencies at a minimum so that data synchronization and inter-process communication are kept at a minimum. But this will most of the time still be necessary. To keep the inter-process communication times at a minimum, parallel computer programs are generally deployed on high-speed networks, allowing for low latencies and high bandwidths. This is of course a trade-off when using distributed parallel computing; data has to be sent between nodes for processing.

The network, or interconnect that is used in a parallel application can be configured according to a certain topology to further increase performance. Simply letting all the nodes in the interconnect communicate with one another would not be optimal since the interconnect would become overburdened and processes would have to wait for communication. Instead one configures the interconnect according to a topology to limit the number of connections and increase performance [11] [7].

Amdahl’s Law

As stated before, many programs have inherently serial parts, which cannot be parallelized, these portions of the program will set an upper bound for the speedup of the program. Let $T_1$ be the time it takes to execute a program in serial, $n \in \mathbb{N}$ the number of CPUs and $C$ the serial portion of the program. The time $T_n$ to execute the program in parallel can be expressed as

$$T_n = T_1(C + \frac{1-C}{n})$$
and the speedup $S_n$ can be expressed as

$$S_n = \frac{T_1}{T_n} = \frac{T_1}{T_1(C + \frac{1}{n}(1-C))} = \frac{1}{C + \frac{1}{n}(1-C)}.$$ 

As $n \to \infty$ we see that the speedup function will always be bounded by the constant portion of the program and we will suffer diminishing returns in the performance gain of the program when running with more CPUs.

### 3.3.2 Performance Metrics

#### Speedup

This might be the most obvious performance metric when speaking of parallel computing. The speedup $S$ is defined as

$$S = \frac{T_{old}}{T_{new}} = \frac{T_{serial}}{T_{parallel}}$$

and it measures how much faster, or in worst case, slower, the parallel program is compared to the original version. The total runtime of a parallel program that is run with $p$ number of CPUs can be calculated as

$$T_{parallel} = \frac{T_{serial}}{p} + T_{ohd},$$

where $T_{ohd}$ is the parallel overhead, i.e. the time required coordinate the parallel tasks, such as synchronization, starting and terminating threads etc.

#### Efficiency

Efficiency measures how much of the potential resources are used in a parallel program. It is defined as

$$E = \frac{T_{serial}}{pT_{parallel}},$$

where $p$ is the number of processors used. Under ideal circumstances we would have an efficiency of $E = 1$, but this is rarely the case because of the same limiting factors as for the speedup.

#### Extending the Metrics to Include Combined Parallelization

We could extend this to include applications that are parallelized with a combination of shared-memory and distributed-memory parallelization. In an application where we use $p$ processes and let each process use $n$ threads, we could write

$$T_{parallel} = \frac{T_{serial}}{pn} + T_{ohd}.$$ 

Following the same logic we would write the combined efficiency as

$$E = \frac{T_{serial}}{pnT_{parallel}}.$$
4 Cloud Computing and Amazon Web Services

This section will serve as a foundation for understanding what Cloud Computing encompasses, as well as giving the reader familiarity with Amazon Web Services (AWS). AWS is Amazon.com’s cloud platform, which will be used for this thesis work. This section will not encompass all of the services on AWS, since that would be much too extensive to fit in this report. Rather we will focus on the services used in this thesis work and some closely related services. For full documentation we refer to the AWS official documentation [1].

4.1 Cloud Computing

The term Cloud Computing is a frequently used IT term today. While there’s no standardized definition of what Cloud computing really is, there’s a general consensus that the term encompasses the following:

Cloud computing fosters the provision and use of IT infrastructure, platforms, and applications of any kind in the form of services that are electronically available on the Web. The term cloud hints at the fact that these services are provisioned by a provider on the Internet (or on the intranet of a larger organization). Users of cloud services, on the other hand, can propose their own offerings as services on the Internet or on an intranet. [6].

There are many actors on the market providing cloud services of different kinds, the largest of them being Amazon.com with Amazon Web Services, Google with Google Cloud Platform and Microsoft with Microsoft Azure. Each vendor has their advantages and disadvantages as well as specific areas of specialization. When speaking of cloud computing, one often talk about three different layers. These layers describe the level of abstraction that the user will experience as well as the type of services provided.

**IaaS** or *infrastructure as a service*, is when the provider offers pure computing resources, often in the form of virtualized computers. This is not targeted towards end users, but rather toward developers that need a substitute for in-house hardware resources. Most often, vendors also provide accompanying systems, like networking and operating systems.

**PaaS** is an acronym for *platform as a service*. Services on this layer are primarily targeted toward developers, rather than the end users. The vendor provides an environment for developers to run, create and manage their applications on while the cloud service provider manages other tasks, such as virtualization, servers, networking and storage.

**SaaS** or *software as a service* is the cloud layer providing the highest level of abstraction. The vendor provides access to applications or databases without the user having any knowledge of the underlying systems. This could include storage applications, E-mail, gaming etc.

---

2Virtualization is the process of creating a virtual machine, separated from the underlying hardware resources, thus enabling one to have several virtual machines sharing the same hardware resources.
4.2 Amazon Web Services

Amazon Web Services (AWS) is a cloud computing platform that is provided by Amazon.com. It consists of a set of services, all managed and maintained by AWS. The services provided vary greatly but can be divided into the following categories:

- Compute
- Storage & Content Delivery
- Database
- Networking
- Developer Tools
- Management Tools
- Security & Identity
- Internet of Things
- Administration & Security
- Analytics
- Application Services
- Deployment & Management
- Mobile Services
- Enterprise Applications

The most fundamental service provided is the compute service Elastic Compute Cloud (EC2), which provides users with the ability to rapidly provision computing power in the form of virtual server in the AWS cloud. The other services are centered around managing or providing more functionality for the compute units, such as storage, networking and management of applications and code.

4.2.1 Infrastructure

Cloud Distribution and Networking

Amazon web services is hosted in data centers around the world, these are called regions. Each region in turn consists of multiple isolated locations which are known as Availability Zones (AZ). An EC2 instance is launched in a specified region and availability zone and can be launched in a virtual network called Virtual Private Cloud (VPC) which is logically separate from other VPC’s in the AWS cloud. A VPC is limited to a region, but can span several availability zones. To further sub-divide the VPC, one can specify subnets, which are network address ranges that are available to the instances placed in the subnet. Figure 7 explains this further.
Some AWS instance types support *clustering*, which allows instances to be placed in what is called a *placement group* - a logical\(^3\) grouping of instances within a single availability zone. Instances in a placement group benefit from a high bandwidth and low latency interconnect and can, depending on the instance, use the enhanced networking feature, which enable bandwidths of up to 10 Gbit/s. The instance networking performance are officially listed according to a few categories: low to moderate, moderate, high and 10 Gigabit.

### 4.2.1.1 EC2 Instances

The AWS landscape is constantly changing, with new instance types frequently made available. Currently AWS supports 9 instance classes, each with their scope of use. Each class have multiple instances, providing additional features such as more processors, enhanced networking or dedicated hardware. The list below gives a brief explanation of the different instance types.

**T2** - General purpose instances with up to 2 virtual CPUs. Low cost.

**M3, M4** - General purpose with up to 40 virtual CPUs and 160 GiB of memory. Also supports enhanced networking. M4 is the latest generation of instances in the m-family.

**C3, C4** - Compute optimized instances with high up to 36 high frequency virtual CPUs, 60 GiB memory and with the ability of enhanced networking. C4 is the latest generation of C-class instances.

**R3** - Memory optimized instances with up to 32 virtuals CPUs and 244 GiB of memory. Supports enhanced networking.

**G2** - Instances with NVIDIA GPUs for applications requiring graphics processing.

\(^3\)Logical refers to the functional description of the instances being placed in a group, as seen by the user. The physical view of this system could, however be a whole different story.
I2 - Optimized for applications requiring high SSD I/O.

D2 - Dense storage instances with up to 48 TB of HDD-based storage.

Since the instances are virtualized on shared hardware, one can’t expect to be the single user of that hardware. This could in principle lead to performance losses, therefore AWS support dedicated instances which run on hardware that is dedicated to a single user.

EC2 instances can use many different operating systems, such as Microsoft Windows Server, Ubuntu, Red Had Linux and Amazon’s own Linux distribution Amazon Linux. These are pre-configured into what is called Amazon Machine Images (AMI). These contain the necessary software to quickly set up virtual machines. AMIs can be saved after they have been modified by the user, so that they can be re-used later on.

Storage

AWS supports several storage options, each with its advantages and drawbacks.

The Elastic Block Store (EBS) storage option provides block level storage that is hosted on magnetic volumes, solid state drives (SSD) or hard disk drives (HDD). Block level storage refers to that the drive which the storage is allocated on is formatted into blocks, where each block in turn can be controlled as an individual drive. Each EBS volume can be attached to a running instance and detached and re-used when an instance is terminated to conserve data.

Instance Store is similar to EBS, with the main difference being that the instance storage is physically attached to the host computer and that it’s stored data is lost when the instance is terminated, stopped or have failed. It’s ephemeral nature makes it suitable for storing AMI startup data, buffers, caches and other temporary data.

S3 is a storage service that provides a repository primarily for web applications. It is also used to backup copies of data volumes.

4.2.2 Pricing

The fundamental principle of AWS is that one pays for the resources that are used. Instances are priced according to an hourly fee, storage is billed according to how much data is stored and network costs are determined by the amount of data transferred. Costs vary slightly between regions.

Instance Pricing

As said previously, instances are mainly billed according to an hourly fee, and the user pays for each hour that is initiated, i.e. if an instance is launched and used for ten minutes, the user will be billed for the whole hour. There are however a few different type of instances, that differ in the way they are deployed on the AWS infrastructure. These are:

On-Demand Instances are deployed on shared hardware and are payed hourly according to a fixed fee. The fee varies with the instance class and are generally more expensive for high performance instances of newer generations.

Spot Instances are available in a market of sorts. Spot instances are currently unutilized instances within an availability zone that users can place a bid price on. The price is determined by the market and a spot request instance run the risk of being terminated if the market
price exceeds the user’s bid price. These are generally much cheaper than the ordinary instances. If the instance is terminated by a price increase, the user will not be billed for the partial hour.

**Reserved Instances** are instances that are reserved for a set time period in the range of 1 to 3 years. The benefit is that reserved instances come with a discounted price, with up to a 75% discount.

**Dedicated Hosts** are physical servers with instance capacity that is fully dedicated to a specific user.

In general, one can say that spot instances are the cheapest, but also quite unreliable. On-demand instances vary in price, depending on the instance type used and are more reliable compared to spot instances. Reserved instances are cheap, but require a long-term commitment, which is not ideal in many cases. Dedicated hosts are good for some use cases, but keep in mind that the number of instances running on the reserved host does not influence the cost.

**Storage and Data Transfer Pricing**

Storage pricing is determined by the amount of data stored, the time the data is stored and the storage option used. EBS storage for example has a fixed fee per GB-month (average storage over the course of a month) and an additional fee for EBS-storage with extra support for high IOPS application. The price per GB-month is lower for magnetic volumes than SSD volumes.

The S3 storage service price is determined by the access frequency and the data amounts stored. The price is lower for the infrequent access options. The price per GB-month is also reduced when the user stores more data in total.

The data transfer rates are generally very cheap and depend on the amount of data transferred and to where the data is transferred. The most expensive transfer rate is when transferring data from EC2 to internet, but this is cost is only a small fraction of a dollar and is reduced when the total transfer rate increases.

5 Implementation

This section will describe the implementation of the Marching Squares algorithm.

5.1 Implementation of the Marching Squares Algorithm

The algorithm and the testing program was developed in C++, mainly due to performance considerations, but also because of the possibility of object oriented programming and for the large amount of libraries and tools available for development, profiling, debugging etc.

The starting point of the algorithm is in `MainFunctions.cpp`, where a pointer to the raster data is received together with other parameters, such as the number of threads to be used, the dimensions of the raster and parameters for the elevation levels to calculate the isolines at. The next step is to set up the environment for multi-threading.

Up to this point, everything have been executed in a serial fashion, and now starts the parallel region. After the threads are set up, the number of contour levels and data storage for the isolines are allocated locally for each thread. For this we use a nested vector structure from the C++
standard library (commonly written std). The vector containing all the isoline information holds a set of vectors containing a point objects representing a two-dimensional point with x and y coordinates.

At this point, the iterative work with creating the isolines for each level starts. The basic workflow is as follows: for each contour level - perform steps 2 to 7 in the marching squares algorithm description. The first step is to perform a binary filtration according to the current threshold value, this is done in the function `binaryFilterPar` by iterating through each element and setting a value of zero or one, depending on the raster data value. The result is stored in a thread-local array called `filteredData`. The original raster data is read-only.

The next step is to calculate the permutation values for each cell in the cell grid. This is done by reading the values in the four neighbouring cells in the `filteredData` array, and for each neighbouring value, starting in the top left corner, do a logical right-shift and perform a bitwise OR operation, eventually building up a four-bit binary index that will constitute the line configuration in that cell.

When the permutation values are calculated, each cell contain information about the line segments, but not exactly where the line segments will cross the cell boundary. To make the lines more representative of the underlying data, we use linear interpolation on the raster data to calculate where the boundary crossings will be. This is all done in the function `interpolateContourLineCrossingsPar`.

The last iterative step is to build the sublines. This will be the most time consuming part of the program, partly because of memory access patterns that are far from optimal, i.e. accessing elements that are located far from each other in memory, and because of branches that will be hard for the CPU to predict, which will result in a performance penalty. To build the lines, we start by scanning all the four edges of the cell grid for line segments. These segments will be a part of an open line, i.e. a line whose endpoints will lie at the cell grid boundary. Upon detecting a line segment at the edge of the cell grid, we traverse the line one segment at a time, adding the points in each cell to a vector. To make sure we don’t have any stray line segments that will interfere with further building of lines we delete the segments that have already been added. Figure 8 show a comparison of building only the open lines, compared to including the closed lines.

![Figure 8](image)

**Figure 8**: A comparison of including only the open lines as compared to include both closed and open lines.

Special care have to be taken when building the isolines; to avoid duplicate data in the isoline vectors, we only add one point for each cell in the cell grid - since the endpoint of each line segment will be the starting point of the next line segment. To know which point to add we have to know
where we entered the cell. This is done by having the algorithm keep track of which side of the
previous cell it exited while traversing the isoline, and then using this information combined with
the current cell value to get the right point in the next cell. Figure 9 displays a scenario where the
current cell is highlighted in the grid. To add the correct point, in this case the bottom point, the
algorithm knows that the previous cell side was left, it then uses this information in combination
with the current cell value to get the correct point, simultaneously setting the previous side to the
bottom side, so that we will get the correct result in the next cell.

Figure 9: The figure displays cell grid containing a set of line segments that belong to the same
isoline. To build the isoline, we add the second point of each cell. The second point, in this case
the bottom point, is determined by using information about the current cell value, as well as which
side we entered the cell from.

When all the open isolines have been accounted for, we simply scan the rest of the raster,
starting in the top left corner, for new line segments and in a similar fashion we add all the points
making up a line to a vector. For calculating isolines at another level, we simply repeat the above
steps for some other threshold value.

5.1.1 Multi-threading the Marching Squares Algorithm

The Marching Squares algorithm lends itself nicely to multi-threading since we have no data
dependencies between adjacent cells, allowing us to process subsections in parallel with multiple
threads. Multi-threading is implemented with the OpenMP API, which supports shared-memory
parallel programming in C/C++ and Fortran [3]. OpenMP is relatively easy to use compared to
other APIs such as POSIX Threads (Pthreads). Each thread is assigned to work on a subsection
of the raster, called sub-raster. The layout of these sub-rasters will be so that they as much as
possible are in a square grid layout. This is of course not possible for numbers other than perfect
square integers. When handling the other cases, the layout is biased towards keeping the number
of sub-rasters per row higher than the number of sub-rasters per column. Prime numbers except
the number two are omitted.
Because the extents of the cell grid is one element smaller in each direction than the raster data, we have to be careful when doing computations on the sub-rasters; using smaller cell grids would result in a gap in the sub-raster boundaries, where no isolines would be calculated. To address this problem, the cells grids are created so that they will overlap each other, ensuring that there will be no gaps between thread boundaries. To ensure load balancing between threads, an algorithm distributes the number of data points as equally as possible, making sure the difference will be at most one row or column of raster data among the threads. When each thread has their sub-raster to process, steps 2-7 in the algorithm description will be performed in parallel by each thread on its sub-raster, hopefully making the algorithm run faster.

5.1.2 Implementing Distributed Memory Parallelization

Since the cloud platform allows for easy set-up and configuration of new virtual computers, we would like to use this to scale our application, so that we can process larger datasets in a shorter amount of time, thus reducing latency in an application. To do this we use Message Passing Interface (MPI). MPI is widely used in academia and industry for parallel computing. MPI allows for parallel computing in distributed memory systems, e.g. in a network with several interconnected computers, where each computer can run a separate process as a part of a distributed program.

Our application will use MPI to distribute parts of a large raster to several nodes in the same network, effectively distributing the work among several computers. The node that delegates the work is called the Master node and receiving nodes are called Worker nodes. Combined with the multi-threading we will achieve parallelization on two levels: at the top level we distribute the original data on several nodes, in the second layer we perform calculations in parallel using multi-threading with OpenMP.
Figure 11: The left grid represents the raster data on the master node. The master node will scatter the raster data in sub-rasters to the worker nodes in the network. Each worker node, as well as the master node, will perform multi-threaded calculations in parallel on their respective sub-raster.

The work-flow is as follows: The master node gets information about which part of the raster to read, on what levels the lines are to be calculated, as well as how many processes and threads are to be used. The application then distributes the raster data to the different processes as equally as possible, which is calculated by an algorithm in order to avoid load imbalance between nodes. The act of distributing the sub-rasters to the different worker nodes is called scattering.

When the raster is scattered to the workers, the workers perform the isoline calculations on their respective part of the raster. The isolines are temporarily stored locally on each node and sent back to the master node.

This application was implemented using two MPI implementations; one standard implementation, i.e. Open MPI or MPICH, as well as an MPI implementation from the BOOST C++ libraries. This is because they both complement each other nicely in functionality. The standard MPI implementations supports the creation of custom MPI data types, more about that below, while the BOOST MPI implementation supports automatic serialization of data.

A Note About Serialization

Serialization is the process of converting an object into a byte stream in order to store the data or send it over a network, with the purpose of recreating it in the same state when needed. Serialization can be performed in several ways. The most method common in applications requiring high performance is binary serialization, where objects are serialized into a pure byte stream. Other serializations methods, such as XML (Extensible Markup Language) or JSON (JavaScript Object Notation) convert the objects into a data structure represented by either of those text formats.

MPI requires information about the size and type of data to be sent, and can in its original implementation not easily send user-defined objects and data structures. Since the number of isolines and the number of points in each line will vary on each node, this would be a hindrance if we could not send arbitrarily sized collections of point objects. The BOOST MPI implementation lets us implement automatic serialization of objects and data structures containing these, so that we easily can send these between the worker nodes and the master node.
A Note About Memory Layout

The raster data in previous pictures are conceptually displayed as a two dimensional grid of cells, each containing a numeric value. This is because it is how it later is displayed in the screen. This would conveniently be represented as a two-dimensional array, i.e. a matrix. When doing lower level programming, one often have do discern between the representation of data and the actual layout of data in the computer memory; a matrix in C++ is actually stored as a one-dimensional row of contiguous memory elements, where an element can be accessed by pointing to the memory address of the first element in the matrix and specifying an offset into the current memory block. Figure 12 shows an example of this.

\[
\begin{array}{|c|c|c|c|}
\hline
0 & 0 & 1 & 2 \\
\hline
\end{array}
\]

\[
9 = \text{#columns x row + column} = 4 \times 2 + 1
\]

Figure 12: The figure show a mapping from 2D index to the actual 1D index as used in the computer memory. The first row and column is indexed with zero, respectively.

For mapping any 2D index \((i, j)\) of a matrix with \(M\) elements in each row, to a 1D index \(k\), we perform the following mapping:

\[(i, j) \mapsto iM + j = k, \text{where } i, j \in \mathbb{N}^0.\]

The 1D memory layout creates a problem when we want to scatter a part of the raster that in a 2D memory representation would be a contiguous block; this does not represent the physical, 1D, layout of the data in memory. Figure 13 illustrates this further.
Figure 13: The figure show the discrepancy between the 2D and 1D memory layouts for a matrix which is divided in four sub-matrices illustrated by the different colors.

To counter this problem the MPI interface comes with functionality for specifying custom data types. The function `MPI_Type_vector` lets us create a contiguous data structure representing a sub-matrix by specifying a block length, i.e. the number of contiguous elements, the number of these blocks and the number of elements between the starting element of each block. For representing the red sub-matrix in figure 13 we would have a block length of two, the number of blocks would be two as well, and the stride between the start of each block would be four elements. With this we will reduce the number of scattering operations, as well as making it easier to receive and process the data for the worker nodes.

6 Parallel Program Assessment

In this section we will analyze the performance of our program to see how it scales on the Amazon EC2 cloud with different numbers if worker nodes and threads on each node.

6.1 Cloud Setup and Test Environment

All the tests performed in this study were done on Ubuntu Server 14.04, a Linux-based, open-source operating system. Some additional software packages were installed to support the compilation of the program, these packages include C++ BOOST for complementary C++ libraries and GDAL for reading raster data files [9].

6.1.1 MIT StarCluster

To automate much of the tedious parts with setting up a cluster of virtual computers in the AWS cloud, we used the open source toolkit StarCluster [4]. StarCluster comprises a large suite of functionality for managing EC2 clusters; by configuring a cluster template on a local computer an EC2 cluster can be completely managed with simple commands from a terminal. The cluster template can be configured to use existing or user-defined AMIs, to be in a specific region, subnet and security group. It can also be set up to use a Network File System (NFS) along with existing EBS storage.
6.2 Multi-threaded Program Performance

We start by investigating the program’s performance when run on a single computer using different number of threads. All subsequent test were performed on a m3.2xlarge AWS EC2 instance, using an Intel Xeon E5-2670 v2 Ivy Bridge processor and with the following listed performance specifications:

Table 1: m3.2xlarge instance performance specifications from the AWS official website

<table>
<thead>
<tr>
<th>Model</th>
<th>vCPU</th>
<th>Mem (GiB)</th>
<th>SSD Storage (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.2xlarge</td>
<td>8</td>
<td>30</td>
<td>2 x 80</td>
</tr>
</tbody>
</table>

As test data we use a 6000 x 6000 pixel portion of a raster file representing elevation data. The isolines were calculated from 0 to 380 meters with steps of 10 meters.

We start by investigating how run time proportions of the different program parts differ when using different number of threads; if one part of the program would scale worse than other parts, this could likely lead to a performance bottle neck and would thus have to be addressed. All tests were done three times for each thread-count and the results are the average values of these run times.

![Figure 14](image)

Figure 14: The proportional run times of the main parts of the program. Each bar display the run time proportions of the different parts of the program when run with different thread counts.

Figure 14 show how the run time proportions of the program parts varies when using different
thread counts. We see that the run time proportions of each part of the program are more or less fixed, varying at most with a few percentage points.

Next we want to see how the program scales with multiple threads. The result is displayed in figure 15.

![Figure 15: The graph shows how the program scales with multiple threads by displaying measured speedup as a function of the number of threads used.](image)

The figure shows a linear relation for all program parts up to four threads, with fairly high efficiency. With more than four threads, the efficiency is reduced. The m3.2xlarge features 8 virtual CPUs, as indicated by table 1. Looking further on the specification of the Intel Xeon E5-2670 v2 processor, we see that there are 10 cores and 20 threads [2]. This is enabled with the Intel Hyper-Threading Technology which does the following according to Intel:

*Intel® Hyper-Threading Technology (Intel® HT Technology) delivers two processing threads per physical core. Highly threaded applications can get more work done in parallel, completing tasks sooner.*

While we can’t be sure, we can suspect that two virtual CPUs as defined by Amazon for the m3.2xlarge instance would equate to one physical CPU with hyper-threading. Testing the program on a laptop with an Intel Core i5-3317U processor with 2 cores and 4 threads, the program would scale very inefficiently with more than two threads. This indicates that the program scales with the number of physical cores on the computer. With the results from figure 15 we can then further presume that two virtual CPUs, at least for the m3.2xlarge instance, equates to one physical CPU with hyper-threading.

To see the efficiency behavior we plot this using the same data as in figure 14 and 15:
Figure 16: The efficiency as a function of the number of threads used. We see a clear decrease in parallel efficiency as the thread count increase above the available number of physical CPUs.

As figure 15 indicated, figure 16 show more clearly how the parallel efficiency decreases as the thread count exceeds the available number of physical CPUs. We also see that the total efficiency often is lower than the efficiency of its constituents. This is because the total time includes tasks that are inherently serial, such as thread management, reading input parameters and raster data, synchronizing data between threads etc. This is essentially the basis for Amdahl's law; however much one would be able to parallelize parts of the program, this serial part would ultimately limit the performance.

6.3 Distributed Program Performance

Here we will look at the performance when using several nodes for parallel computing. While the multi-threaded performance study focused on the four main parts of the program: binary filtering, calculate contours, interpolate contours and build lines, we will in this section just measure their total time, which we will call the processing step. Since the processing will run on each node in parallel, each node will process the received data in different amounts of time. The program cannot finish until the slowest node have finished processing, hence we will only use the recorded time of the node that finished last.

Using distributed parallel programming introduces new steps in our program which we would like to measure: distributing data to the worker nodes and collecting data from the worker nodes to the master node, also known as scattering and gathering data, respectively.

We start by investigating the speedup of the program as we use various number of worker nodes.
Figure 17: The speedup as a function of the number of nodes using different thread counts.

Figure 17 shows the speedup of our program as a function of the number of nodes used in parallel. The different lines in the graph shows the speedup for a fixed number of threads per node. As indicated by the legend we used 1, 2, 4 and 8 threads per node. As expected, we see that the speedup increases with both the number of nodes and the number of threads used per node. We also plot the parallel efficiency with the same data to see how well our computing resources are used:
Figure 18: The parallel efficiency as a function of the number of nodes using different thread counts.

Figure 18 show the parallel efficiency for the distributed program, we see that the efficiency is reduced with the number of threads used and the number of nodes used. This could partly be explained by the following bar diagram which show the proportions of the distributed program parts; scattering, computing and gathering:

Figure 19: The distributed program run time proportions as a function of the number of nodes used. Eight threads per node were used.
Figure 19 shows how the run time proportions of the program parts vary when using different number of nodes. We see that as the number of nodes increase, more time is spent sending and receiving data, while the computing time is decreased, which would explain our parallel efficiency decrease in figure 18.

We compare the above results from the m3.2xlarge instance type with the c3.2xlarge instance, which is optimized for compute intensive tasks. Figure 20 and 21 show the results of the speedup and parallel efficiency tests, respectively.

When comparing figure 20 with figure 17 we see that the scaling performance is very similar, with the c3.2xlarge instances performing marginally better. Since the parallel efficiency is calculated from the speedup results, we would expect the efficiency graphs to be similar, which is confirmed by figure 21.
Since the scaling performed very similarly for the two instance types, we also compare the raw numbers from our tests to see if one instance performed better than the other in terms of absolute run time. Table 2 show the result of this.

Table 2: Execution times and comparison between the m3.2xlarge and c3.2xlarge instances

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Number of Nodes</th>
<th>m3.2xlarge [s]</th>
<th>c3.2xlarge [s]</th>
<th>m3/c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>22.013</td>
<td>20.933</td>
<td>1.052</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12.766</td>
<td>11.626</td>
<td>1.098</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7.015</td>
<td>6.412</td>
<td>1.094</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4.232</td>
<td>3.876</td>
<td>1.092</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11.850</td>
<td>11.302</td>
<td>1.049</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7.228</td>
<td>6.713</td>
<td>1.077</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4.253</td>
<td>3.793</td>
<td>1.121</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3.000</td>
<td>2.760</td>
<td>1.087</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>6.928</td>
<td>6.758</td>
<td>1.025</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4.824</td>
<td>4.668</td>
<td>1.033</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.676</td>
<td>3.528</td>
<td>1.042</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.678</td>
<td>2.417</td>
<td>1.108</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>6.532</td>
<td>6.444</td>
<td>1.014</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4.505</td>
<td>4.284</td>
<td>1.052</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.043</td>
<td>2.889</td>
<td>1.054</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.478</td>
<td>2.287</td>
<td>1.084</td>
</tr>
</tbody>
</table>

In the table we see that the c3.2xlarge instance consistently outperforms the m3.2xlarge instance, with up to a 12% faster execution time when running with four nodes and two threads per
6.4 AWS Network Performance

Network performance is very important for distributed computing applications. We will here investigate how a ping pong MPI program performs when two nodes send and receive fixed sized data packages to each other when placed in the same availability zone. And to make a more general network performance study on the AWS cloud we will use the network testing tool Iperf to measure the bandwidth between two nodes placed in two different configurations: both nodes in the same availability zone, each node placed in separate availability zones within the same region.

6.4.1 MPI Ping Pong Network Performance

Network performance for a Ping Pong program implemented with MPI will be measured here. The program sends a data package consisting of a contiguous array of double precision values from one node to another. The receiving node then sends another package of the same size in return. The process is repeated ten times. The time for sending the data between the nodes is recorded whereby an average bandwidth can be calculated. This process is repeated several times over a specified time interval so that possible variations in networking performance can be observed.

![EC2 Instance Network Bandwidth](image)

Figure 22: Measured bandwidth with the Ping Pong program over a two hour period for a t2.micro and a m3.2xlarge instance.

Figure 22 show the result of the Ping Pong program when run on two instance types over a two hour period. Note that two tests were not run at the same time. We see that the measured bandwidth of the t2.micro instance fluctuates quite heavily while the m3.2xlarge instance is hit by a loss in bandwidth once during the time of the test.

6.4.2 Network Performance Tests with Iperf

Here we used the Iperf network testing tool to measure the bandwidth between two nodes. Iperf is set up on two nodes where one acts as a server, listening for incoming calls from the client.
node. By placing the node and server in separate availability zones we can see if that influences the bandwidth.

![Network Bandwidth For Selected Instances Within the Same VPC](image)

Looking at figure 23 we that the bandwidth is quite similar for the intra- and the inter availability zone communication. Worth noting here is the instances official networking performance; the t2.micro, t2.medium are classified as having low to moderate networking performance, the m3.medium and m3.large as having moderate networking performance, while the m3.2xlarge, c3.2xlarge and the c3.4xlarge instances are classified as having high networking performance.

**6.5 Price Comparison**

A common incentive for migrating to the cloud is saving money. Although the scope of this work doesn’t cover drawing any general conclusions about what instance type will be most efficient on a cost-performance basis, we can use our test results to see whether one instance type if preferable over the other.

Our performance tests covered the usage of two mid-tier instances, c3.2xlarge and m3.2xlarge, with the former instance being adapted for compute intensive applications, and the latter instance being a general purpose instance with large amounts of memory. We saw in the scaling performance tests that both instances scaled similarly, with the c3.2xlarge instance being ever so slightly better. Table 2 also show that the c3.2xlarge performed better in terms of shorter execution times, which would indicate that the c3.2xlarge instance would be the better alternative for our purposes. As of writing this report the price for the c3.2xlarge and m3.2xlarge instance in the Frankfurt region is $0.516 and $0.632 per hour, respectively. With the m3.2xlarge being 22% more expensive per hour and not as performant as the c3.2xlarge, we can conclude that c3.2xlarge instance is the
preferred choice for our application.

7 Discussion and Conclusions

In this thesis work we have successfully made a C++ multi-threaded, distributed parallel implementation of the marching squares algorithm. This has been deployed on the AWS cloud where we’ve done performance tests to assess the cloud platform as well as the algorithm implementation. This was done by gathering data on several relevant performance parameters.

We managed to show how the Amazon Web Services cloud could be used to leverage parallel computing for geospatial processing. The algorithm implementation scales well with the amount of physical CPUs available on the AWS instances tested but scales more inefficiently when using Message Passing Interface for distributed parallel processing. The reason for the inefficiency is because the time it takes to send data between the master node and the workers is large compared to the processing time. This could certainly be addressed by using a more high-performing instance type that uses the 10 Gigabit interconnect or by minimizing the data passing between nodes.

We also performed network bandwidth tests for several instance types where we could see how the t2.micro instance were subject to varying network performance over a 2 hour period, whereas the more expensive m3.2xlarge instance was not, except for one performance dip during the hour test. The Iperf network tests also showed how the bandwidth varies among the different instance types. A 10 Gigabit enabled instance was not tested because it would require the installation of network drivers to fully utilize the 10 Gigabit network.

Although the scope of this work is limited, the tests performed in this work seem to show that the AWS cloud platform is quite suitable for scalable distributed GIS applications. The many services and room for configuration on the cloud platform makes it a diverse, practical platform for many kinds of computing applications.

7.1 Using Amazon Web Services

Amazon Web Services is a highly sophisticated and comprehensive cloud platform, in fact so comprehensive that we could reasonably not come close to using or presenting all of the services and functionality available. It is recommended that a beginner user of AWS has a quite strong computer background, not necessarily as a developer, but at least some experience with managing computers from a systems manager’s point of view, preferably with some Linux experience and with some basic knowledge of network technologies.

The manual configuration of instances from the AWS management console is notably tedious over time, hence a third party tool such as StarCluster for managing multiple instances is highly recommended. The tests performed in this study would not have been possible without it, since it required lots of starting and terminating clusters with different instance types, the transfer of data to and from the clusters and the setup of VPCs, subnets etc., all of which StarCluster had functionality for alleviating. With the official AWS software development kits available for multiple programming languages and frameworks including .NET, Java and Python, users can develop their own tools for managing and automating AWS.
7.2 Pricing

As mentioned before in this report, the general pricing model of AWS is that you pay for what you use. This applies for both data storage, data transfer and the time that the resources where utilized. The user is billed for each hour of resource usage, even though the resource wasn’t utilized for the whole hour. There are AWS services for optimizing one’s application, not only from a price perspective, but also a resource utilization perspective. This include services such as auto scaling groups, load balancers, and other means of automating employment and deployment of AWS resources.

As for what resources, such as instances, to use for a specific application, it’s the user’s choice. The AWS online documentation is quite extensive and can certainly guide a user towards making an informed decision about what to use.

7.2.1 Project Specific Notes About Pricing and Costs

We saw that that for our specific application, the best instance type would be the c3.2xlarge, both in terms of the hourly fee and performance. But we also have to keep in mind that the efficiency of the parallelization was quite low when running on several nodes, meaning we utilize the potential computing capacity poorly, and thus paying for unused capacity. But again, this is limited to our specific application.

7.3 Further Work

For mission critical applications, further work should be put in to investigate the reliability of the AWS cloud, especially as it relates to how instance performance vary over time and compare this to dedicated instance performance. Security is also of great concern - although there are many security features on AWS, it’s the users responsibility to work in a manner so that these are used. A performance comparison between dedicated instances and on-demand instance would also be of interest, especially for large scale applications.

References


