Graph 500 on the public cloud

Master System and Network Engineering - RP2

Author:
Harm Dermois
harm.dermois@os3.nl

Supervisor:
Ana Lucia Varbanescu
a.l.varbanescu@uva.nl

July 10, 2015

Acknowledgments

I would like to thank my supervisor Ana Varbanescu, for all the help and feedback she has given me. I also want to thank Ana Opresecu with the advise and help she provided with working on the cloud. Kees Verstoep for helping me with the DAS-4 and OpenNebula. Yuri Demchenko for providing a Amazon account for my experiments. Amazon Web Services help desk for their quick response to my request to increase the instance limit.

Abstract

In this project the viability of the Graph 500 benchmark on the cloud has been tested. The performance one Graph 500 benchmark application has been evaluated on the DAS-4 and Amazon EC2. We found that using the cloud is competitive with super computers. We also created a model for predicting the performance of the benchmark given the resources.

Contents

1. Introduction **6**

2. Research Question **6**

3. Background and Related Work **7**
3.1. Graph 500 Benchmark 1 ("Search") 7
3.1.1. Generating the edge list 8
3.1.2. Graph Construction 8
3.1.3. Sampling 64 search keys 8
3.1.4. Breadth First Search 8
3.1.5. Validation 8
3.1.6. Timing and Performance metrics 9
3.2. Related Work 11

4. Methodology **12**
4.1. Tools 12
4.1.1. Das-4 12
4.1.2. OpenNebula 12
4.1.3. Amazon EC2 instance types 13
4.1.4. Reference Implementations 14
4.1.5. Message Passing Interface and OpenMP 15
4.1.6. Intel MPI Benchmark 16
4.2. Experiments 16
4.2.1. Communication 16
4.2.2. Measurements 17
4.2.3. OpenMP 17
4.2.4. MPI 18

5. Results **20**
5.1. Communication 20
5.1.1. IMB benchmark 20
5.1.2. Message count 20
5.2. OpenMP 20
5.3. DAS-4 21
5.3.1. Turning validation off 22
5.3.2. Nodes and Scale 22
5.3.3. No InfiniBand 23
5.4. OpenNebula 24
5.5. Amazon 24

6. Model **26**

7. Discussion **28**
7.1. Validation vs No Validation . . . 28
7.2. DAS . . . 28
7.3. OpenNebula . . . 28
7.4. Amazon . . . 28
7.5. Reference implementation . . . 29
7.6. Model . . . 29

8. Conclusion **30**

9. Future Work **30**

A. OpenNebula Templates **34**

B. Intel MPI Benchmark compilation **36**

C. Intel MPI becnhmark “PingPong” **36**

D. DAS-4 environment script **37**

E. Amazon preparation scripts **37**

1. Introduction

The Graph 500 [1] is a list of top 500 graph processing machines. As of late the need for network analysis is skyrocketing. Network analysis includes social networks, road direction and even text analysis.

The Graph 500 is inspired by the TOP500 [2]. The TOP500 uses the LINPACK benchmark which solves linear equations and linear least-squares problems. The metrics used in the LINKPACK are useful for CPU intensive problems, but do not quantify the ability to process graphs. For this reason, a new benchmark is made with metrics which better suit data intensive problems. The benchmark was made with the following ideas in mind: the kernel should generic and apply to many applications, the results should map to real world problems and the data set should be comparable to real-world problems. The benchmark is meant to push the industry to invest into building specific hardware to more efficiently tackle data intensive of problems.

To get on the list, a benchmark should be run. This benchmark needs to comply to the specifications which can be found on the Graph 500 website[3], but beyond these specifications anything goes. This means that the benchmark can be optimized for the hardware it runs on.

The benchmark consists of two kernels: the graph construction and the Breadth-First Search(BFS). Both these kernels will be timed. The kernels are preceded by the creation of an edge-list using a data Generator. The results of the second kernel is also validated and the performance information is output.
The Graph 500 list, at the moment, consists mostly of super computers. The aim of this project is to get an entry on the Graph 500 list using the public cloud. The research focuses on defining a model to predict the performance given a certain amount of resources and vice versa.

2. Research Question

The research question for this project is as follows:
Is it possible to make a model that can predict the performance of Graph500 on the cloud, depending on the amount of resources?
To answer our main question, the following sub-questions have been formulated:

- What size of graph is reasonable to benchmark while working on the cloud?
- How well does the reference implementation[4] run on the public cloud?
- What model fits the results acquired from running the benchmark on the cloud?

We approach this research question empirically, i.e., we use the Graph 500 reference implementation and perform experiments to determine the benchmarks behavior. The reference implementations are created by the authors of the Graph 500 and there is no doubt that the implementations adhere to the specifications. The Graph 500 times two kernels: the graph generation and the BFS. From the two kernels timed in Graph

500, we focus on the BFS. The BFS performance is the most important metric for the benchmark and it is the one shown on the list.

3. Background and Related Work

This section contains information needed to understand the project and the related work.

3.1. Graph 500 Benchmark 1 ("Search")

Benchmark 1 ("Search")[3] consists of two kernels accessing a single data structure representing an undirected graph. These kernels are: the construction of the graph from an edge list and a search through the constructed graph.

The benchmark has defined a few different problem classes. These are shown in table 1. These problem classes give a sense of the size of the graph and the amount of storage needed to run the benchmark. The scale is defined as a combination of the amount of vertices and the edges connected to each of these vertices. The number of vertices is given by the scale. For example, the Toy problem is a graph of 2^{26} vertices. The other parameter is the edgefactor. The edgefactor is ratio of edges to vertices. For each of the problem classes the edgefactor has been set to 16.

Problem class	Scale	Edgefactor	Approx. Storage size in TB
Toy (level 10)	26	16	0.0172
Mini (level 11)	29	16	0.1374
Small (level 12)	32	16	1.0995
Medium (level 13)	36	16	17.5922
Large (level 14)	39	16	140.7375
Huge (level 15)	42	16	1125.8999

Table 1: The lists of problem classes as defined by the Graph 500, assuming the storage of the edge list is done n a 64-bit integer.

The benchmark consists of the following steps:

1. Generate the edge list.
2. Construct a graph from the edge list.
3. Randomly sample 64 unique search keys.
4. For each search key:
   a) Do the Breadth First Search and store all all visited vertices in an array.
   b) Validate that the array is a correct BFS search tree for the given search tree.
5. Compute the performance

In the following subsections all steps of the benchmark are explained in more detail.

3.1.1. Generating the edge list

The first step of the benchmark is generating the edge list. For this a data generator is used. The data generator constructs a list of edge tuples containing the start and end vertex, but also contains a vertex identifiers, thereby creating a list of undirected edges.

The intent of the graph generation is to convert a the edge list, which has no locality, into a data structure which can more easily be used. The list of generated tuples could have some locality because of the way the edges are generated. To lose this locality the edge list is randomized before inserted into the graph generation kernel.

The data generator is a Kronecker generator, which has many properties which are seen in graphs in the real world[5].

3.1.2. Graph Construction

The graph construction is the first timed kernel. The kernel takes the previously created edge list and transforms it into a data structure of choice. Examples of data structures to store graphs are Compressed Row Storage(CRS) and Compressed Sparse Column(CSC)[6, 7]. The kernel takes only two parameters, the edge list and the size of the edge list. The number of vertices and other information that might be inferred from the edge list must computed by the kernel.

One thing to note is that the data structure created in this kernel cannot be altered by subsequent kernels.

3.1.3. Sampling 64 search keys

After the graph is constructed a 64 searches need to be done. By performing these searches the speed of traversal can be measured. The search keys need to have at least one other vertex connected to them, to prevent trivial searches. The search is done in a Breadth First Search way.

3.1.4. Breadth First Search

Breadth First Search[8] (BFS) is the graph traversal which has been chosen by the Graph 500. BFS visits all vertices each on the level before moving on to the next level. It keeps traversing the Graph 500 until all levels in the graph have been visited. The level of a vertex is defined as the minimum number of edges that need to traversed to get to the root. An example of the traversal can be seen in figure 1. Note that the Graph 500 benchmark only specifies the end results is a BFS, but does not specify how the program gets to this result.

3.1.5. Validation

After the each of the 64 searches a validation is done. The validation does soft checking of the results. Because there randomness in the process, validation against a reference is not possible. The validation checks for the following things:

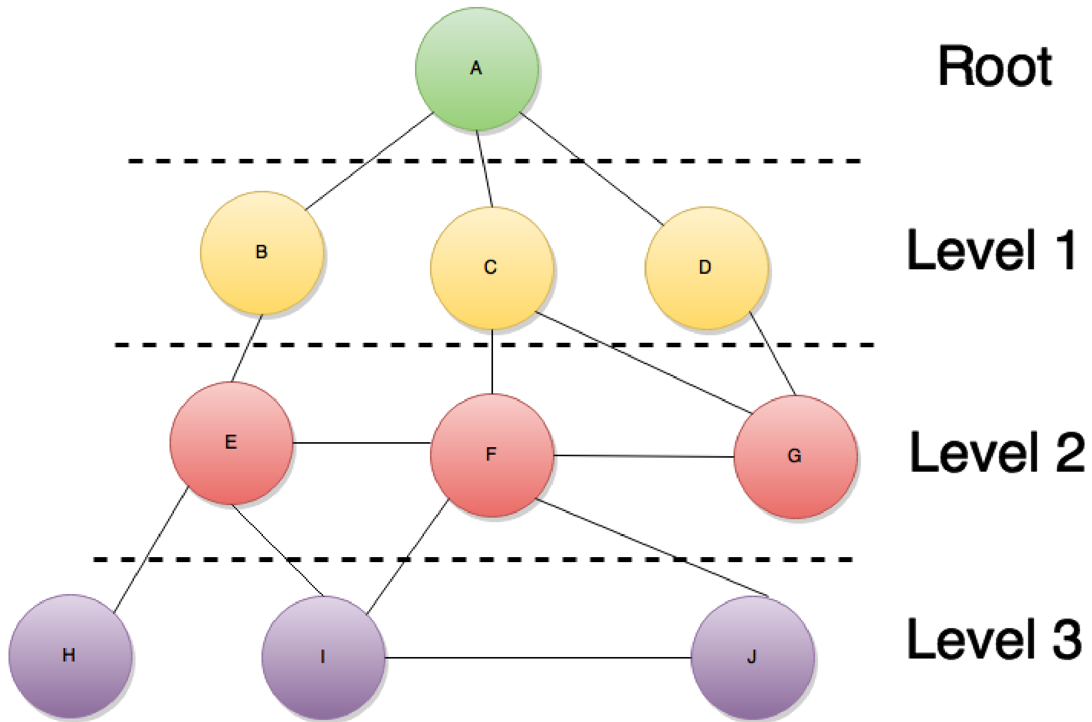


Figure 1: An example of BFS traversal is shown here. The nodes are traversed in alphabetical order.

1. The BFS tree is a tree and does not contain cycles.
2. Each tree edge connects vertices whose BFS levels differ by exactly one.
3. Every edge in the input list has vertices with levels that differ by at most one or that both are not in the BFS tree.
4. The BFS tree spans an entire connected component's vertices.
5. A node and its parent are joined by an edge of the original graph.

3.1.6. Timing and Performance metrics

In code block 1 the output of the `graph500_mpi_simple`.

Listing 1: Output of the benchmark.

```
SCALE:                        24
edgefactor:                   16
NBFS:                         64
graph_generation:             16.2365
num_mpi_processes:            8
construction_time:            15.4957
min_time:                     13.9349
```

```
8  firstquartile_time:           15.6177
9  median_time:                  15.7516
10 thirdquartile_time:           15.8136
11 max_time:                     16.1873
12 mean_time:                    15.6812
13 stddev_time:                  0.318657
14 min_nedge:                    268435456
15 firstquartile_nedge:          268435456
16 median_nedge:                 268435456
17 thirdquartile_nedge:          268435456
18 max_nedge:                    268435456
19 mean_nedge:                   268435456
20 stddev_nedge:                 0
21 min_TEPS:                     1.65831e+07
22 firstquartile_TEPS:           1.6975e+07
23 median_TEPS:                  1.70418e+07
24 thirdquartile_TEPS:           1.71879e+07
25 max_TEPS:                     1.92635e+07
26 harmonic_mean_TEPS:           1.71183e+07
27 harmonic_stddev_TEPS:         43826.4
```

The most important values from this output are:

SCALE Is the size of the graph used in the benchmark.

edgefactor The edgefactor used in the graph.

NBFS Number of BFS searches run.

graph_generation Time for the edge list generation.

construction_time Time for first kernel.

num_mpi_processes The number of processes used for this benchmark. It does not specify how many nodes were used.

harmonic mean TEPS Mean of the kernel 2 TEPS. Note: Because TEPS is a rate, the rates are compared using harmonic means[9].

The timing of the BFS starts right before visiting the root and ends when search has written the last value to the memory. The metric used to define the performance is called traversed edges per second (TEPS). The TEPS are measured through the benchmarking of kernel 2. The equations is.

$$TEPS(n) = \text{total number of edges } / \text{time of kernel 2} \quad (1)$$

3.2. Related Work

There is a lot of related work on the Graph 500 benchmark. Most papers focus on the implementation of better BFS kernels and are not applicable to our study.

In a paper, by Chaktranont et al.[10], the difference is shown between `graph500_mpi_simple` on a virtual private cluster and on a physical cluster. The paper shows that the virtualization overhead head is about 5% on the HPC Cloud they created. This is a cloud solution specifically created for super computers. Comparing their results with other cloud solutions might give some insights.

Toyotaro Suzumura et al. [11] investigated the Graph 500 reference implementations. The paper gives a detailed explanation of three of the four implementations and provides a performance evaluation of these implementations. Suzumaru et al. managed to reach eight GTEPS for a scale 34 problem. The paper also provides insights into optimizing the reference implementations. The implementation used in this project is described in their paper. Their insights about the implementations and communication model have been used for our research.

A technical paper by Angel et al.[12] runs the Graph 500 simple implementation on UMBC High Performance Computing Facility. In this paper the simple implementation runs on their cluster up to scale 32 and 64 nodes. They share a way of removing the validation from the program. The experiments are done by running multiple instances of the program on the same node for up to 64 nodes and explain the implication of running the program in such a way. The hardware used by Angel et al. is similar to the DAS-4. They also propose a method of turning off validation, which is used in our project.

To summarize, there has been a lot of work done on performing and optimizing the Graph 500 benchmark, but to our knowledge no one has attempted to run the Graph 500 benchmark on a public cloud yet.

4. Methodology

4.1. Tools

The goal of the project is to design a model to predict the performance of Graph 500 on the cloud. To do so, we use an empirical approach: we benchmark the Graph 500 kernels on the cloud and build a numerical model from the observed performance. To create a baseline, experiments were also done on the DAS-4 and OpenNebula.

4.1.1. Das-4

The Distributed ASCI Supercomputer 4 (DAS-4) is a six-cluster wide-area distributed system designed by the Advanced School for Computing and Imagining (ASCI)[13]. The six-clusters are the following:

- Vrije Univeristeit (VU)
- Leiden Univeristeit (LU)
- Univeristeit van Amsterdam (UvA)
- Technische Universiteit Delft (TUD)
- Univeristeit van Amsterdam - The MultimediaN (UVA-MN)
- Astronomy institute Netherlands Institute for Radio Astronomy (ASTRON)

All computations have been done on the clusters from the VU and LU. The LU cluster was not actively used at that time. The VU cluster has the most nodes available of the all the clusters and has an OpenNebula cluster installed. The specifications of the clusters can be found in table 2.

Each of the nodes in the clusters has a dual quad-core processor with a speed of 2.4 GHz. All the nodes are also connected with InfiniBand[14] to improve the inter-node communication. The nodes on the cluster have CentOS release 6.6 (Final) installed.

Cluster	Number of nodes	Memory(GB)
VU	74 (41) for all purposed	24
LU	16	48

Table 2: The specifications of the clusters used in this project.

4.1.2. OpenNebula

To be able to compare the results from the public cloud to another cloud solution, OpenNebula was used. “OpenNebula provides the most simple but feature-rich and flexible solution for the comprehensive management of virtualized data centers to enable private,

public and hybrid IaaS clouds. OpenNebula interoperability makes cloud an evolution by leveraging existing IT assets, protecting your investments, and avoiding vendor lock-in." [15]. The OpenNebula software installed on the VU cluster is version 3.8. The OpenNebula cluster on the DAS-4 consists of eight nodes. Table 3 shows the specifications of the OpenNebula nodes. OpenNebula uses templates to define resources to give virtual machines. The template needed to know how to instantiate virtual machines using a given image and makes the resource provisioning flexible. The created virtual machines used the following two templates seen in Appendix A.

CPU	Number of cores	RAM
Intel(R) Xeon(R) CPU E5-2620 0 2.00 GHz	24	66 GB

Table 3: The specifications of the OpenNebula host nodes.

4.1.3. Amazon EC2 instance types

The Amazon Elastic Compute Cloud(Amazon EC2) has many different types of instances, each optimized for specific purposes[16]. The following types are available:

T - Burstable Performance Instances provide burst of CPU with a low baseline. This means that it cannot maintain a constant high CPU load.

M - Balanced Well balanced in terms of memory, CPU and network resources.

C -Computation Optimized Used for computationally intensive tasks.

R - Memory Optimized Used for memory intensive tasks.

G - GPU Has a GPU.

I - High I/O Has large SSDs for fast I/O.

D - Dense Storage Has large HDD for storing a lot data.

From these types, two types are interesting for this project, namely: C and R. They were selected because we want to determine whether using a compute- or memory-optimized instance will make a difference for the performance of the benchmark. The R and C types both have five different sizes of instances and different generations of hardware. Generation three has been chosen for both the R and C type. Size "large" is the smallest size which can be used for this type. We chose this size and generation because these are the cheapest instances that fit the project[1]. The specifications c3.large and r3.large can be seen in table 4.

[1] All experiments have been run on the Ireland region

Type	Number of vCPUs	Memory (GB)	Processor
c3.large	2	4	Intel Xeon E5-2680 v2 (Ivy Bridge) Processors 2.80 GHz
r3.large	2	16	Intel Xeon E5-2670 v2 (Ivy Bridge) Processors 2.50 GHz

Table 4: The specifications of the Amazon EC2 instances chosen for this project.

4.1.4. Reference Implementations

The Graph 500 reference code[4] used is version 2.1.4. This version has four different MPI implementations which all perform the BFS in the same way. The differences between the implementations are the data structures used to store the graph and the way the graph is distributed. The names of the applications are: `graph500_mpi_simple`, `graph500_mpi_one_sided`, `graph500_mpi_replicated` and `graph500_mpi_replicated_csc`. The `graph500_mpi_one_sided` is not be considered, because the one sided implementation expects high performance remote memory access to work properly. This is a technique which can not be relied on in the public cloud, because you have no control over the environment. The `graph500_mpi_replicated` and `graph500_mpi_replicated_csc` store the complete graph in each of the nodes. This will take a lot of RAM per node. We would like to use as little hardware as possible on the public cloud, so we avoided these applications.

Thus, we are using `graph500_mpi_simple`. Other reasons to choose this application are: it is the most simple to understand, it has been thoroughly studied and it requires the least amount of RAM on each of the nodes.

A detailed description of the `graph500_mpi_simple` application can be found in [11].

The implementation uses 2 queues for the BFS. The first queue is used to store all nodes that should still be visited in this iteration. The second queue is used to store all the nodes that should be visited in the next iteration. When the first queue is empty the roles will be swapped of the queues and the next iteration will start. This is done until there are no more nodes that should be visited. On each level the vertices are evenly distributed between all participating processes. The graph is stored by using Compressed Row Storage[7] to minimize the amount of data that needs to be stored in the RAM.

Listing 2: Pseudo code taken from paper [11]

```
1  for all vertex v do
2     |  pred[v]     -1;
3     |  visited[v]     0;
4     CQ      Empty;
5     NQ     Empty;
6     CQ[root]     1;
7
8  fork;
9     this     GetMyRank();
10    loop
```

```
|   while CQ != Empty do
|   |   for each received vertex v and its predecessor u do
|   |   |   if visited[v] = 0 then
|   |   |   |   visited[v]     1;
|   |   |   |   pred[v]     u;
|   |   |   |   Enqueue(NQ, v);
|   |   u     Dequeue(CQ);
|   |   for each vertex v adjacent to u do
|   |   |   r     GetOwner(v);
|   |   |   if r = this then
|   |   |   |   if visited[v] = 0 then
|   |   |   |   |   visited[v]     1;
|   |   |   |   |   pred[v]     u;
|   |   |   |   |   Enqueue(NQ, v);
|   |   |   else
|   |   |   |   send (v, u) to r;
|   if new queue of all the processes is empty then
|   |   break;
|
swap(CQ, NQ);
  join;
```

In the pseudo code CQ is the current queue and NQ is the queue for the next level.

Initial modeling

Graph traversal is a traditional memory-intensive application. When running on a distributed system, the inter-node communication can also become a bottleneck, depending on the speed of the nodes and the communication links. The paper by Suzumura [11] proposes an estimate of the amount of communication in the `graph500_mpi_simple` . The formula is:

$$C(n, M) = A * B * C * D \text{ (bytes)}. \tag{2}$$

Where $A = M * 2$, $B = (n-1)/n$, $C = 2$, $D = 8$, M = total number of edges, and n = the number of MPI processes. The outcome of the equation is the communication size in bytes.

Knowing the amount of data is useful to calculate the amount of messages that need be sent and could also be used to calculate the network load.

4.1.5. Message Passing Interface and OpenMP

The Graph 500 reference code is created in C and uses MPI and OpenMP to parallelize the program.

"Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers."[17]

"OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors."[18].

4.1.6. Intel MPI Benchmark

Because Graph 500 is a communication intensive benchmark, we need to understand the inter-node communication performance. For this we use the Intel MPI benchmark(IMB). The IMB is a free benchmark used to determine how well MPI performs on a certain platform. The benchmark consists of a few different tests, each measuring a different aspect of MPI. Of these tests "PingPong" is the one important to us. "PingPong"[19] is used to measure the start up and throughput of a single message sent between two processes.

For our purposes, IMB's "PingPong" is compiled using the OpenMPI compiler; details of the compilation can be found in Appendix B.

4.2. Experiments

This section explains which experiments have been done and which parameters are used.

4.2.1. Communication

The MPI Graph 500 benchmark is a communication heavy application. The time it takes to send messages might have an impact on the performance. To evaluate how much time is spent on sending messages, the total amount of messages send needs to be approximated. The model shown in equation 2 approximates inter-node communication size. This can be used to calculate the total number of messages that needs to be sent. To confirm that the model fits application, we have logged each time a MPI message is send by a node. By doing this, the total number messages sent can be observed.

There are 3 types of messages sent in this program. The first type is sent when the message buffer is full. The buffer has a fixed size (2 kB) in each of the experiments but it can be changed to get better performance.

The second type of message is used to flush the buffer. If all nodes have been visited by the program and the buffer has some vertices for other nodes visit, the node will flush the buffer.

The last message is used to report that the program is done. The node sends an empty message to other nodes meaning that sender is done visiting vertices.

Because the communication is asynchronous, all nodes can send messages at the same time. The time it takes for one node to send all it's messages is the communication time of the whole program. We assume that on average all nodes need to send the same amount of messages. This assumption is based on the observation that the queues generate a reasonable load balancing for the application. With this assumption in mind the total number of messages needs to be divided by the number of nodes to get the number of messages per node. This can then be multiplied by the time it takes to send a message to obtain the theoretical communication time.

The IMB benchmark will be performed for each of the environments to measure the time it takes to send messages (see section 4.1.6).

4.2.2. Measurements

The MPI program takes two parameters: scale and edgefactor. The measurements are run for the scale 9,12,15,21,24,27 and 30 on 2,4,6,16 and 32 nodes. Each node of the MPI program will only contain one process. The benchmark will be run one time for each of the experiments. It is run only one time because the benchmark already does 64 different searches to get statics of the performance. The platforms on which the measurements will be done are: DAS-4, OpenNebula and Amazon Webservices. The specifications of these platforms can be found in section 4.1.

The memory consumption is important determine which scales can be run on which machines and how many machines are needed. Equation 3 determines the memory footprint of each scale, and it is used to calculate the memory footprint of different scales, seen in table 5 which predicts the memory consumption:

$$M(Scale) = (2^{Scale} * (2 * edgefactor + 1)) * 8 \quad (3)$$

Scale	Predicted total memory consumption (GB)
9	0.000135168
12	0.001081344
15	0.008650752
18	0.069206016
21	0.553648128
24	4.429185024
27	35.433480192
30	283.467841536

Table 5: The predicted total amount of memory used by the program. To find out how much memory is used per node divide the total by the number of nodes that will be used.

4.2.3. OpenMP

The `simple_mpi_simple` uses OpenMP for some of the loops in the program. To find out what the performance effect of OpenMP is, we have tested the benchmark on a single node, by measuring the performance when fixing different numbers of CPUs to use. By default, OpenMP will decide how many nodes will be used for the problem at hand by checking the amount of available cores. By forcing OpenMP to use a specific amount of CPUs, the influence of the number of CPUs on the performance can be measured. For the OpenMP experiments a small change needed to be made in the code to turn off the dynamic assignment, such that the number of CPUs to be used can be specified. The experiments were done on one node of the LU cluster. The experiments were only

done for a limited number of scales, because only one node is used. Validation was still turned on for these experiments. We expect that using more CPUs for the program will improve the performance.

4.2.4. MPI

The MPI measurements have been conducted to find out the effect of the number of nodes and scale on the performance[2]. The experiments are done on the DAS-4, OpenNebula and the Amazon EC2. The DAS-4 and OpenNebula experiments serve as a baseline for the Amazon EC2 results. DAS-4 will serve as the comparison of the cloud results to a super computer. The OpenNebula will serve as comparison for another cloud platform. The following experiments sets were done:

1. With validation
2. Without validation
3. Without Infiniband
4. On the OpenNebula
5. On Amazon EC2 c3.large and Amazon EC2 r3.large

The first two sets of experiments are, the `graph500_mpi_simple` with and without validation. These experiments are done to decide if turning of the validation has an impact on the results and how much large this impact is. We expect the results to be similar in behavior, but the experiments without validation will give a higher performance, because the execution time is divided by the total number of edges instead of the number of visited edges.

The third set of experiments, running the bechmark without InfiniBand, will provide a better comparison between the DAS-4 and Amazon EC2. The cloud does not have InfiniBand, turning will bring the results closer together with this the locality of the environment can be measured. An Amazon data center is larger than the DAS-4 and the machines will most likely be further apart, this means that the results will most likely be comparable, but the Amazon EC2 would perform worse.

The OpenNebula experiments are done in the same environment as the DAS-4 experiments, but use different hardware and a virtualization layer on top. The expected results from the OpenNebula experiments is that they will be comparable to the results of the Amazon EC2 and the DAS-4.

The expectations of the Amazon EC2 set of experiments, is that the results is will be comparable to the DAS-4 without Infiniband, but have lower performance.

[2] Technically experiments could be done with 64 nodes, but some of the nodes were out of service and even if all nodes were available there is always someone else using one of other nodes which made it impossible to run an experiment with 64 nodes.

Removing validation

To run the experiments for scales larger than 15, validation needed to be removed. For these scales the program gets stuck while validating. During the validation also the performance of the search is calculated. To be able to run the program without validation[3] and still measure the performance, the following has been done. Instead of dividing BFS search time by the number of traversed edges during the search it will be divided by the total number of edges as proposed in [12]. Calculating the performance in this way can have a impact and needs to be investigated. Therefore the first set experiments were run. The code with the changes to remove validation can be found here [20]

DAS-4

To run programs on the DAS-4, `prun` needs to be used. This program reserves nodes and places all the required files on the node for the `mpirun` command. For `prun` to run the specified program, it needs to run a script to find out all environment variables and what flags should be used for `mpirun`. One adjustment needed to be made to the default script to get `prun` working for `graph500_mpi_simple`. The file can be found in Appendix D.

OpenNebula

Running experiments on OpenNebula was done by using `mpirun`. The experiments done with 2, 4, 8 and 16 nodes had 24 GB of memory available. This amount has been chosen because this is the same amount of memory as the nodes on the VU cluster. For the experiment with 32 VMs, the nodes have only 10 GB. 10GB was chosen because the eight physical nodes could not handle 32 nodes with 24 GB of RAM. With this setup, the experiments could still be run. The VMs used for the OpenNebula experiments have CentOS release 5.11 (Final) installed. The version of MPI was 1.4. On the VMs no InfiniBand has been installed.

Amazon EC2

For the experiments on Amazon EC2 a new image was created similar to the image used on OpenNebula. To create this image a publicly available image was used with 5.x version of CentOS. There is one difference between the experiments done on OpenNebula and the Amazon EC2. The `mpirun` command is started from one of the participating nodes instead of running it from a node which is not participating. Running MPI in such a manner might have an influence on the performance.

[3]The graph does not need validation, because this is the reference implementation

5. Results

In this section the results of this project are shown.

5.1. Communication

As mentioned in section 4.2.1 the Graph 500benchmark is communication heavy and this might be the determining factor for its performance. With the following results we would like to get feel for the amount of time spent sending messages.

5.1.1. IMB benchmark

On each of the platforms the IMB benchmark has been performed to determine time it takes to send messages between two machines. The communications which are relevant are 0, 1028 and 2048 bytes. The 0-bytes result shows the time it takes to send the final message of the MPI program, 1028 bytes gives an indication of how much time it will take to send a message a not completely filled buffer, and lastly the 2048 shows the time it takes to send a full buffer. The complete table is shown in Appendix C.

Bytes	DAS-4 (μsec)	DAS-4 no InfiniBand(μsec)	OpenNebula (μsec)	AWS EC2 (μsec)
0	3.81	46.55	112.75	81.82
1024	4.93	56.97	130.76	91.40
2048	5.96 4	68.36	269.74	102.96
4096	7.36	79.08	344.64	125.58

Table 6: This figure shows the IMB benchmarks on the different platforms used. All times are an average of a 1000 messages sent. Only the relevant sizes have been shown.

5.1.2. Message count

The message count is a metric to calculate the communication time. The amount of data which is sent per message can be estimated. By using this estimation the number of messages can be calculated. As mentioned before in [11], an estimation of the amount of bytes is computed using Equation 2. Figure 2 shows that the observed number of messages is lower than the calculated number. We found that computed messages sent is two times larger than the observed amount of messages. The model is a good approximation of the number of message, but it is a factor two off.

5.2. OpenMP

In figure 3 we present the results from the OpenMP experiments. The figure shows that the performance of the modified `graph500_mpi_simple` does not depend on the number of CPUs used. This does not change as the scale of the problem increases (see figure 3b). Essentially, these results show that the performance does not depend on the number of

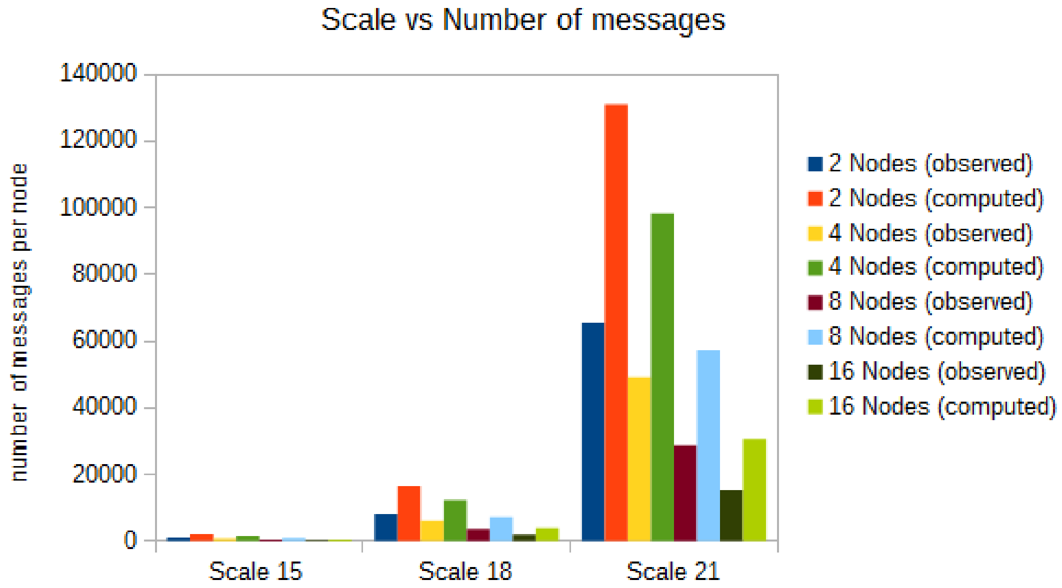
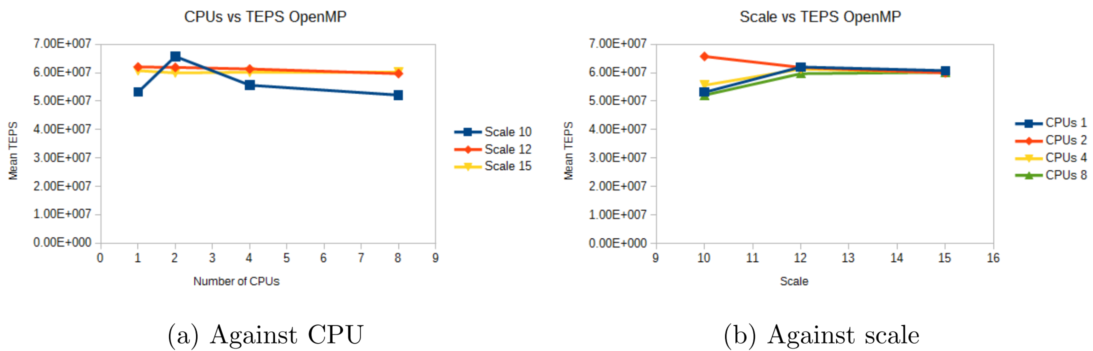


Figure 2: The number of messages sent per node for 4 different scales and 2 to 16 nodes. The observed number is the number of messages sent per node when the buffer is full plus the messages send with left overs(see section 4.2.1)

CPUs (i.e. , cores or threads) used per node. This is an unexpected result, which needs investigation. However, as this analysis is beyond the scope of our project, we simply use no OpenMP for our further cluster and cloud analysis.



(a) Against CPU (b) Against scale

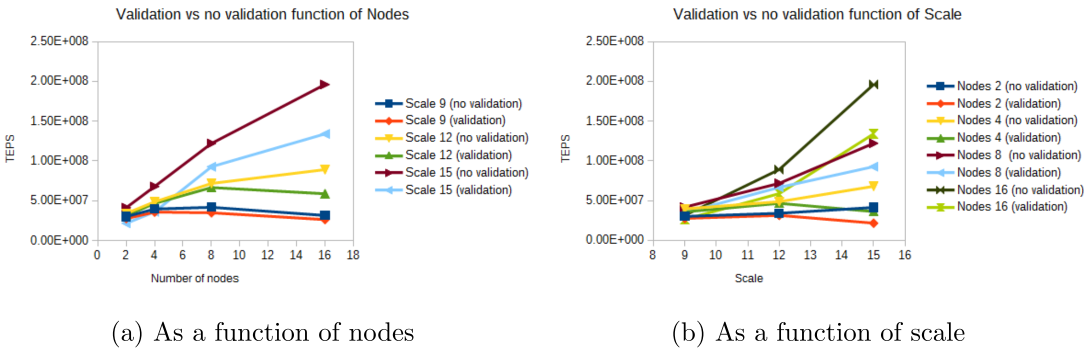
Figure 3: The effect of different scales and the numbers of CPUs on the (TEPS)

5.3. DAS-4

In this section we discuss the results of the experiments on DAS-4.

5.3.1. Turning validation off

Figure 4 shows the performance, in TEPS, as a function of the scale (figure 4b) and the number of nodes (figure 4a). The results show that there is a difference in the number of TEPS with and without the validation. The difference in performance can be up to 50% in the case of scale 15 with 16 nodes. What can be noticed in figure 4a is that the same trends are followed as the number of nodes increases. Figure 4b shows that for larger scales the difference between the validation and non validation becomes larger. As mentioned in section 4.2.4 the way the performance is measured between validation and no validation is different. Dividing the search time by the total amount of edges instead of the visited edges does make a difference for the performance. Although the performance differs the behavior seen in the results is similar. Having a higher performance without validation, but still having the same behavior in the results is the expected result.



(a) As a function of nodes

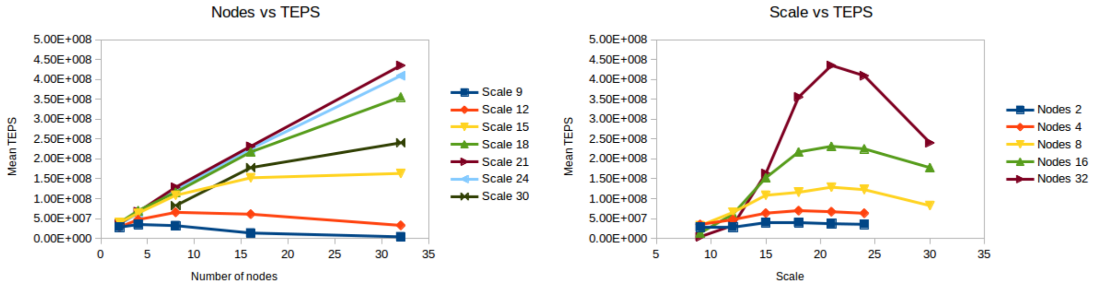
(b) As a function of scale

Figure 4: The effect of different scales and the numbers of nodes on the TEPS between the program with and without validation.

5.3.2. Nodes and Scale

Figure 5 shows that the performance increases as the number of nodes increases. The more nodes participate, the more edges can be traversed, as seen in figure 5b. This increase in performance can be seen up till a certain point. After this tipping point the amount of TEPS decreases again. The tipping point should be found for any number of nodes and it happens at scale 15,18, and 21 for 2, 4, and 8+ nodes.

Figure 5a shows the same data, but then as a function of the number of nodes. What can be seen in this figure is that the amount TEPS increases as the nodes increase for each scale. There is an almost linear correlation between the number of nodes and the TEPS for higher scales except for scale 30. One interesting behavior which can be observed in figure 5b the decline after the tipping point. The performance decline is significant, and seems to be related with the behavior of the network and the inter-node communication. In section 5.3.3, we see that when InfiniBand is disabled, this behavior disappears, confirming our hypothesis that the network becomes a bottleneck when the number of nodes and the amount of messages grow larger.



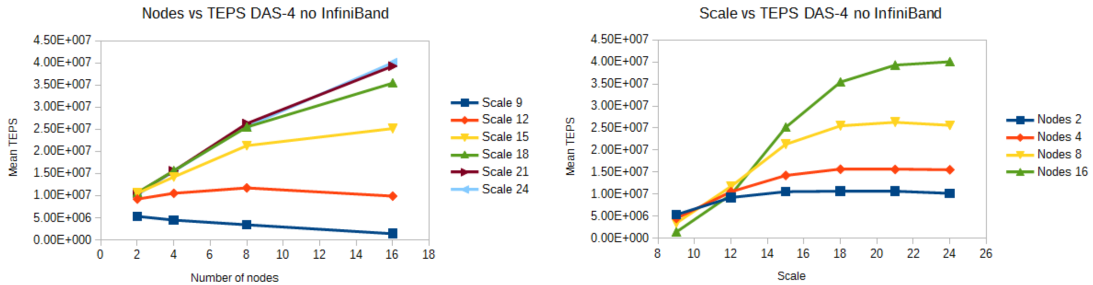
(a) TEPS as a function of nodes for different scales.

(b) TEPS as a function of scale for different amount of nodes.

Figure 5: Performance (in TEPS) as a function of the number of nodes (a) and scale (b) for DAS-4 with Infiniband.

5.3.3. No InfiniBand

The experiments on the DAS-4 have also been run without InfiniBand. The results show similarities with the DAS-4 results with InfiniBand. Figure 6b shows an increase in performance as the scale increases till a certain tipping point, but unlike what has been seen in figure 5b the tipping point has not been yet been reached at scale 24. Figure 6a shows similar results to figure 5a. The performance difference can be as large as 6 times when InfiniBand is used, which further proves that this application is heavily dependent on the inter-node communication. Figure 6b also shows the same trends as figure 5b. However, we note that the bottleneck has disappeared, and the performance does not deteriorate after the tipping point.



(a) TEPS as a function of nodes for different scales.

(b) TEPS as a function of scale for different amount of nodes.

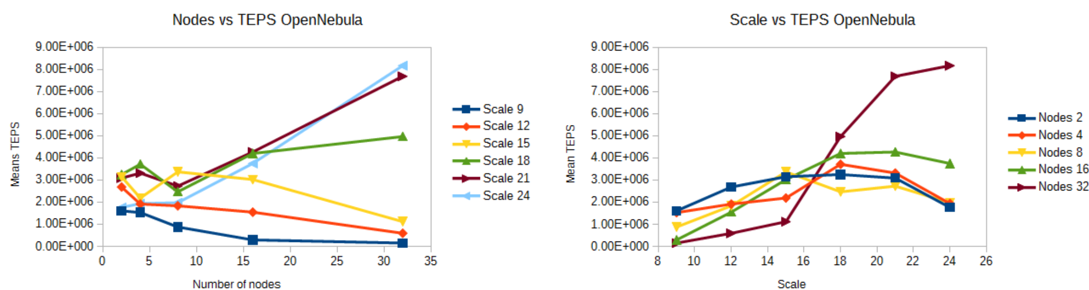
Figure 6: Performance (in TEPS) as a function of the number of nodes (a) and scale (b) for DAS-4 without Infiniband.

5.4. OpenNebula

The OpenNebula results can best be compared with results on the DAS-4 without InfiniBand, because the VMs on the OpenNebula do not use InfiniBand. Looking at figure 7, the performance seems to vary significantly. The performance that can be achieved on OpenNebula is much lower than that achieved on the DAS-4 even without InfiniBand.

The difference in performance between using four and eight nodes is much less clear on the OpenNebula, and higher performance can sometimes be achieved with four nodes compared to eight nodes as seen in figure 7a. This was not the case on the DAS-4. For the larger scales 21 and 24, a linear correlation can be seen between the number of nodes and the performance.

The performance is an order of magnitude lower than what can be seen on DAS-4 without InifiBand. One of the possible reasons for this is the hardware it is running on. By looking at table 3 we see that the clock speed is lower than the other machines. Also the communication time in the OpenNebula cluster 6 is more than two times longer than the communication time within Amazon EC2 and about four times longer than the DAS-4 without InfiniBand. Furthermore, the fact that there are only eight physical machines hosting all the nodes plays a part in the performance. By looking at 7b the tipping points are harder to distinguish but for four and eight nodes it can be found at scale 18. For the experiments on the DAS-4 can be found a bit later around the 18-21 point.



(a) TEPS as a function of nodes for different scales.

(b) TEPS as a function of scale for different amount of nodes.

Figure 7: Performance (in TEPS) as a function of the number of nodes (a) and scale (b) for OpenNebula on DAS-4.

5.5. Amazon

In figures 8 and 9 we show the results of the experiments done on Amazon. The figures show almost identical in terms of performance and behavior, but the r3.large can run experiments of a higher scale. The experiments done with the c3.large show that the c3.large reaches linear behavior faster than the r3.large which can be seen in figure 8a and 9a. For the c3.large scale 18 already shows linear behavior where as r3.large this

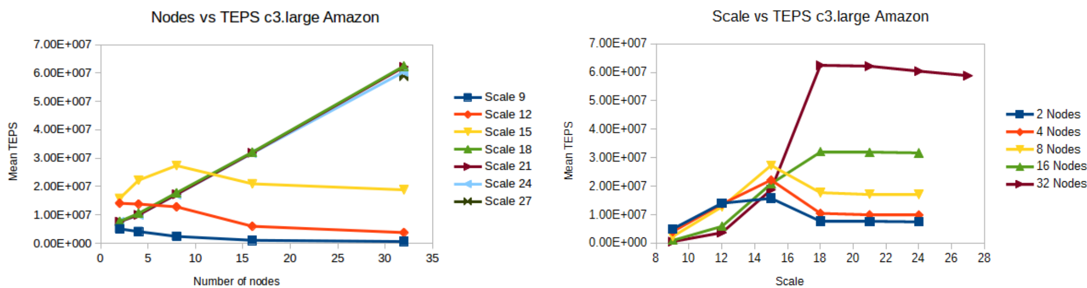
starts at scale 21. All lines above scale 18 tightly packed on each other, as seen in figure 8a. For the results of the r3 machines figure 9a shows a more discrepancies between the scales 18 and higher. The thing to notice is that by doubling the amount of nodes used the performance also doubles.

Both figures show that for scales lower than 15 the performance is almost constant for each number of nodes that have been tested. For these scales, we have empirically found the tipping point (i.e. the point at which adding more nodes does not improve performance). For larger scales, the tipping point has not been found.

Comparing these results with the results of the OpenNebula and the DAS-4 without InfiniBand, the Amazon experiment can traverse about ten times as much edges comparing to OpenNebula. The Amazon has about 50% less TEPS than DAS-4 without Infiniband.

Both figures show a slow decline in performance after the tipping point in figure 8b and 9b. The parallelism becomes saturated at an earlier point for the Amazon EC2 instances than for the DAS-4. With the Amazon instances it happens at about scale 18 and for the DAS-4 18 and higher scales. Furthermore, when looking at the figures 8a and 9a the tipping point has not been reached for scales above 15.

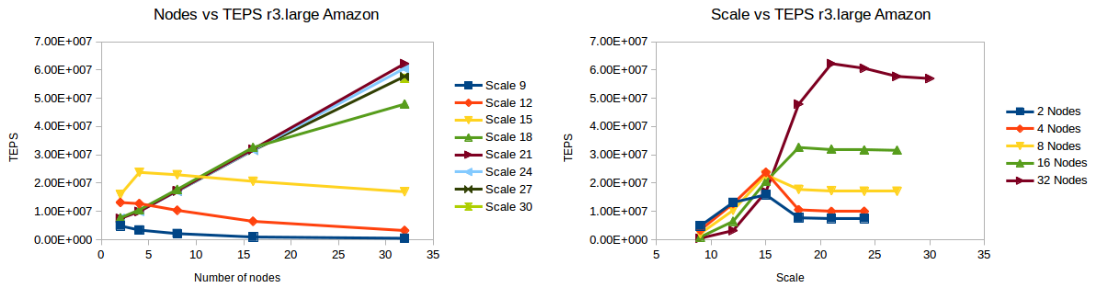
The behavior for the scales up to scale 15 is most likely due to the communication time. As shown in table 6 the time taken to send messages on Amazon EC2 is twice as long compared to the DAS-4 no InfiniBand. While looking at the OpenNebula results where the communication is even longer than for EC2 the constant performance for each number of nodes is still seen up to scale 18. This makes us believe that this constant behavior for lower scales is the communication time.



(a) TEPS as a function of nodes for different scales.

(b) TEPS as a function of scale for different amount of nodes.

Figure 8: This figure shows the TEPS vs the scale and nodes for the experiments done on AWS c3.large machines



(a) TEPS as a function of nodes for different scales.

(b) TEPS as a function of scale for different amount of nodes.

Figure 9: Performance (in TEPS) as a function of the number of nodes (a) and scale (b) for Amazon c3.large.

6. Model

The results in section 5.5 show that there is a linear correlation between performance and number of nodes for each scale above 15.

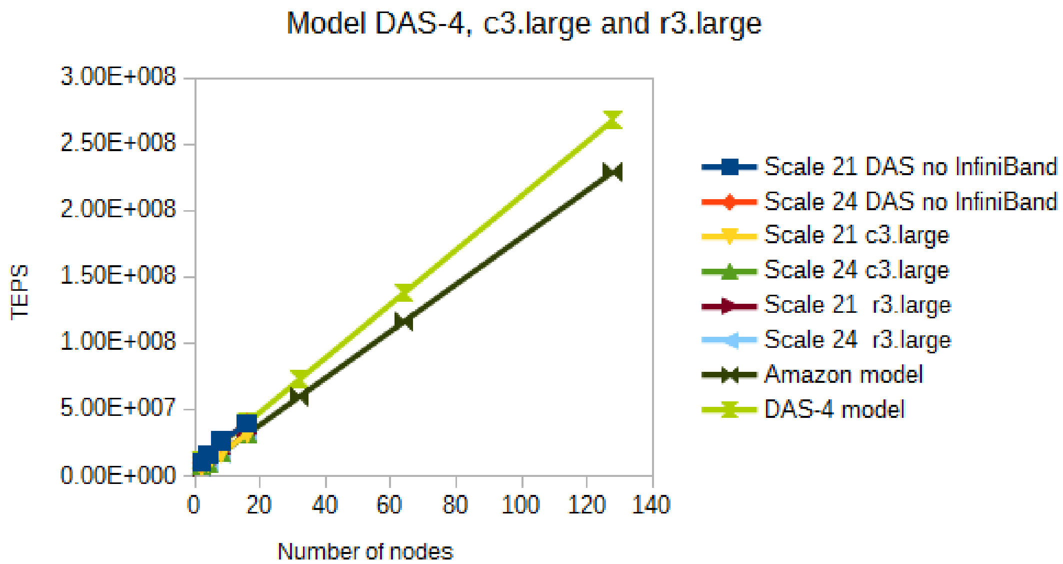


Figure 10: The linear behavior for the DAS-4,c3.large and r3.large.

We general model will be:

$$TEPS(scale) = \begin{cases} a * \#nodes + b \leq T & \text{if scale} > 15 \\ \text{slow decrease} > T & \text{if scale} > 15 \end{cases} \tag{4}$$

Where T is the tipping point and is defined as: $T = f(scale, architecture)$. Equation 4 calculates the performance for given scale for a specific number of nodes.

Experiment	a (MTEPS)	b (MTEPS)
DAS-4 scale 21	2.03	7.70
DAS-4 scale 24	2.11	7.00
Amazon EC2 c3.large scale 21	1.76	3.45
Amazon EC2 c3.large scale 24	1.76	3.41
Amazon EC2 r3.large scale 21	1.78	3.36
Amazon EC2 r3.large scale 24	1.75	3.42

Table 7: The values for a and b from equation 4 for different scale and architectures.

To calculate the a and b for a different architectures and scale, a regression line has been made for the experiments seen in figure 10. Table 7 shows the results of creating these regression line. The table show there is not much difference between the a and b on the same architecture. This means a more general formula can be created for each of the architectures.

First the equation for the DAS-4:

$$TEPS(scale) = 2.03 * x + 7.70 \tag{5}$$

Second the equation for Amazon EC2:

$$TEPS(scale) = 1.76 * x + 3.45 \tag{6}$$

Where x is the number of nodes used for the application. Both equations return a value in 10^6 TEPS (MTEPS).

As can be seen from these two equations the slope and the starting value depend on the architecture on which the Graph 500 benchmark runs. To calibrate the experiments at least 3 experiments need to be done with the hardware of choice. When a and b have been found the model is complete and a prediction can be made for different experiments.

The model made can only predict the performance of the as a function of the number of nodes for large scales. The point where maximum parallelism can be achieved and the point of the diminishing return cannot be calculated with equation 5 or 6.

7. Discussion

7.1. Validation vs No Validation

Figure 4 shows that approximation to divide the search time by the total number of edges instead of the actually visited edges by Angel et al.[12] is a rough approximation. The scales used for the validation experiments were small and different behavior might be observed for higher scales. To further investigate approximation, experiments need to be done on a platform with a newer version of MPI installed, because then validation can be left on.

7.2. DAS

There are two kinds of tipping points in these graphs. The first can be seen in figure 5a, in which the TEPS as a function of nodes is shown. For example, a tipping point can be seen at 8 nodes with scale 12. After this tipping point, adding more nodes leads to a sharp decrease in performance. Thus this tipping point, shows the optimal number of nodes for a certain scale. In the case of the DAS-4 results in figure 5, this is for 8 nodes for scale 12. Figure 5b shows the other tipping point, which can be seen at scale 21 for 32 nodes. This tipping point shows the point where paralellization has reached its peak. Adding more data(i.e., using a graph of larger scale) to the application will not improve the performance anymore.

7.3. OpenNebula

There were some problems with getting the program to work on OpenNebula. The cluster on which the latest version of OpenNebula was installed was shutdown in preparation of the new DAS-5. For this reason the older version 3.8 needed to be used. The OpenNebula marketplace does not store any images for version 3.8 anymore, which made the setup time longer. Also public images available on the OpenNebula VU cluster did not work out of the box. One thing to note is that when you are running MPI it is best not to have a firewall between the hosts. The MPI nodes need to be able to open ports to each other for the messages.

7.4. Amazon

Experiments on the Amazon EC2 were only done once and on one region. The results did not show much variance, the standard deviation is about one percent of the mean value.

The experiments were run on Friday and over the weekend. We expect that, during the day or peak hours the performance might can change, due to higher demands on Amazon EC2 (assuming these instances, r3.large and c3.large, see demand variability over time). More experiments should have been done to confirm that the cloud is this stable. The performance might also change per region.

Using Amazon EC2 does have some limitations. With a new account you can only make a limited amount of machines. To raise this limit you need to present a business case. This might not be a problem in most cases, but for huge experiments might not be possible. Getting the machines ready is also a huge task when working with more instances. Starting up 32 machines on amazon took longer 15 minutes and the VMs also need to the latest code after creation. When doing experiments with a lot amount of nodes it is better to create a special image where everything is ready. After all the machines are ready, all the machines need to be added to the `known_hosts` file(see Appendix E for the Amazon setup scripts.). If this is not done the MPI program will get stuck, because it needs to accept a key. Adding all machines to the `known_hosts` might also take a long time, depending on how this is done.

7.5. Reference implementation

There were some problems with running the `graph500_mpi_simple` on the DAS. The first problem was that the MPI version on the DAS-4. The version used is 1.4. This MPI 1.4 is not compatible with the reference code, which uses data types defined in later versions. Because of this the texttworkaround.h needed to be changed. Changing the file made it possible to compile this program on the DAS-4. In the commands of this file, it is stated that this would help out with the validation and will prevent the program from hanging. This was the case for a scale up to 15. At a scale higher than 15 program would get stuck while validating the first BFS. To avoid this problem, experiments have been run with validation turned off.

7.6. Model

The initial intuition for the model is that there are two factors which contribute to calculating the TEPS, namely: the computational time and the communication time. The computation time is the taken to traverse all the edges in the graph. The communication time is the time it takes to send all the messages. Trying to make a model with these two parameter resulted in straight lines. The computation is a linear function of the number of nodes and the communication is a function of the scale and the number of nodes, because everything is parallel and the number of and the messages which are sent are non-blocking the calculations need to be done for one node. The paper by Angel proposes that there is another factor which is contention on the network. What the paper suggest is that at some point the network is getting saturated. This is due to the shear amount of nodes that need to be transported over the network. This is also what can be seen in the results. As the nodes stay constant and the scale goes up there is a tipping point as mentioned in the results. After this tipping point the amount of TEPS keeps reducing. This can be explained, by looking at contention. If the scale goes up but the number of nodes stay the same the chance of contention keeps increasing. As the contention increases the network will reach is minimum value which is the maximum one link can handle. This means that it will return to a state between two nodes instead of having a parallel system of n nodes. This problem would be less prominent in the

Amazon EC2 environment. In this environment congestion will occur much less likely because the network is much larger and there are no direct connections between the machines which might get clogged. With this new insight in mind there are multiple ways to test if this is true. One easy way to increase the buffer size. By increasing the buffer size the amount of messages that will not decrease but the timings of these messages will differ and there will not be a continues stream coming from one node only bursts.However, this test is left as future work.

8. Conclusion

The project focused on finding a model to predict the performance of the a Graph 500 benchmark depending on the amount of resources.

A reasonable size of graph to process on Amazon EC2 is a scale of 30. Processing a scale 30 graph can be done with 32 r3.large instance and takes for five hours to complete. Getting 32 instances of the r3.large on Amazon EC2 does not take a lot of effort.

The performance of the Graph 500 benchmark on the DAS-4 has been compared to the performance of the Amazon EC2. From the results we can conclude that the public cloud can be competitive with a super computer. Although there is no InfiniBand in the cloud, the latency and compute power of the machines is competitive with the DAS-4 in terms of performance.

A model has been created. This model is dependent on the tipping performance point. Before the point, the model is linear. After the tipping point, we approximate the model with a constant performance, which is an optimistic prediction. The parameters of this linear function are dependent on the architecture on which the benchmark is running.

9. Future Work

Our results are promising so far, but more research needs to be done in validating and refining the model.

To validate the proposed model, more experiments still need to be run. Larger amount of nodes and higher scales should be tested. Behaviors can differ for larger scales. It has only been shown that the behavior is different for smaller, because the tipping point has been identified empirically. The same behavior could also occur for larger scales and more nodes.

The `graph500_mpi_simple` could run multiple processes on one node. The program does not make use of multiple processes during the BFS. This means that if there is enough memory on the computer it would be possible to run multiple processes per node. This has already been shown in the paper by Angel et al.[12], but it is something that we have not looked at yet.

Other implementations could also be used to run on the cloud. The `graph500_mpi_simple` is a not an optimized implementation. The performance can be improved by having a more optimized implementation on each node. The implementation should be hardware

agnostic, if it is to run on cloud. The implementation by [21] which do a two dimensional version of the BFS algorithm might be a good first step.

Furthermore, other types of cloud instances and public cloud services should be tested. Only Amazon EC2 has been benchmarked in this project and only two of its instances. Although these instances seemed the most relevant for this project, larger instances should be tested to find out if the model still holds. Also the, using the m3.large might provide a good comparison to the other results. The Google cloud might also be interesting option to run the application on.

DAS-4 with InfiniBand showed a different behavior than all the other experiments we have performed. This behavior is unrelated to the Graph 500 on the cloud, but it is interesting to investigate further.

References

[1] Richard C Murphy et al. "Introducing the graph 500". In: *Cray Users Group (CUG)* (2010).

[2] *Home — TOP500 Supercomputer Sites.* URL: http://www.top500.org (visited on 06/07/2014).

[3] *Graph 500 Benchmark 1 ("Search") — Graph 500.* URL: http://www.graph500.org/specifications (visited on 06/02/2014).

[4] *Graph 500 Reference Implementations — Graph 500.* URL: http://www.graph500.org/referencecode (visited on 06/02/2014).

[5] Jure Leskovec et al. "Kronecker graphs: An approach to modeling networks". In: *The Journal of Machine Learning Research* 11 (2010), pp. 985–1042.

[6] *Compressed Column Storage (CCS).* URL: http://netlib.org/linalg/html_templates/node92.html (visited on 06/07/2014).

[7] *Compressed Row Storage (CRS).* URL: http://netlib.org/linalg/html_templates/node91.html (visited on 06/07/2014).

[8] *Breadth-first search - Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/Breadth-first_search.

[9] *Harmonic mean - Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/Harmonic_mean (visited on 06/07/2014).

[10] Nuttapong Chakthranont et al. "Exploring the Performance Impact of Virtualization on an HPC Cloud". In: *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on.* IEEE. 2014, pp. 426–432.

[11] Toyotaro Suzumura et al. "Performance characteristics of Graph500 on large-scale distributed environment". In: *Workload Characterization (IISWC), 2011 IEEE International Symposium on.* IEEE. 2011, pp. 149–158.

[12] Jordan B Angel et al. *Graph 500 performance on a distributed-memory cluster.* Tech. rep. Citeseer, 2012.

[13] *DAS-4: Distributed ASCI Supercomputer 4.* URL: http://www.cs.vu.nl/das4/.

[14] *InfiniBand - Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/InfiniBand.

[15] *OpenNebula — About the Technology.* URL: http://opennebula.org/about/technology/.

[16] *AWS Amazon EC2 Instance Types.* URL: http://aws.amazon.com/ec2/instance-types/ (visited on 03/07/2014).

[17] *Message Passing Interface - Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/Message_Passing_Interface.

[18] *OpenMP - Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/OpenMP.

[19] *Intel MPI Benchmarks - User Guide and Methodology Description*. URL: http://www.hpc.ut.ee/dokumendid/ics_2013/imb/doc/IMB_Users_Guide.pdf (visited on 03/07/2014).

[20] *RP2*. URL: https://github.com/cheeseit/RP2 (visited on 09/07/2014).

[21] Koji Ueno and Toyotaro Suzumura. "Highly scalable graph search for the graph500 benchmark". In: *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM. 2012, pp. 149–160.

[22] *How to build a MPI PC cluster using CentOS 5.5 and Open MPI*. URL: http://na-inet.jp/na/pccluster/centos_x86_64-en.html (visited on 09/07/2014).

A. OpenNebula Templates

The templates used for OpenNebula.

Listing 3: Template used for 16 nodes on OpenNebula

```
NAME   = Centos5-disk
CPU    = 1.0
MEMORY = 24000
VCPU = 8

REQUIREMENTS = "CLUSTER_ID = 101"

OS     = [
arch = x86_64
]

DISK   = [
IMAGE_ID=908
]

NIC    = [
# NETWORK = "Small network"
# OpenNebula API change: now refer to NETWORK ID:
NETWORK_ID=1
]

GRAPHICS = [
TYPE    = "vnc",
LISTEN  = "0.0.0.0"
]

FEATURES = [
# Needed for graceful shutdown with KVM:
acpi="yes"
]

RAW = [
type = "kvm",
data = " <serial type='pty'> <source path='/dev/pts/3'/> <target port
    ='1'/> </serial>"
]

CONTEXT = [
hostname   = "$NAME",
# OpenNebula API change: now refer to NETWORK ID:
dns        = "$NETWORK[DNS,     NETWORK_ID=1]",
gateway    = "$NETWORK[GATEWAY, NETWORK_ID=1]",
netmask    = "$NETWORK[NETMASK, NETWORK_ID=1]",
files      = "/cm/shared/package/OpenNebula/current/srv/cloud/configs/
    centos-5/init.sh /var/scratch/hdermois/OpenNebula/id_dsa.pub",
target     = "hdc",
root_pubkey = "id_dsa.pub",
username    = "opennebula",
```

```
47 user_pubkey = "id_dsa.pub"
48 ]
```

Listing 4: Template used for 32 nodes on OpenNebula

```
1 NAME   = Centos5-disk
2 CPU    = 1.0
3 MEMORY = 10000
4 VCPU = 8
5
6 REQUIREMENTS = "CLUSTER_ID = 101"
7
8 OS     = [
9 arch = x86_64
10 ]
11
12 DISK   = [
13 IMAGE_ID=906
14 ]
15
16 NIC    = [
17 # NETWORK = "Small network"
18 # OpenNebula API change: now refer to NETWORK ID:
19 NETWORK_ID=1
20 ]
21
22 GRAPHICS = [
23 TYPE    = "vnc",
24 LISTEN  = "0.0.0.0"
25 ]
26
27 FEATURES = [
28 # Needed for graceful shutdown with KVM:
29 acpi="yes"
30 ]
31
32 RAW = [
33 type = "kvm",
34 data = " <serial type='pty'> <source path='/dev/pts/3'/> <target port
     ='1'/> </serial>"
35 ]
36
37 CONTEXT = [
38 hostname   = "$NAME",
39 # OpenNebula API change: now refer to NETWORK ID:
40 dns        = "$NETWORK[DNS,     NETWORK_ID=1]",
41 gateway    = "$NETWORK[GATEWAY, NETWORK_ID=1]",
42 netmask    = "$NETWORK[NETMASK, NETWORK_ID=1]",
43 files      = "/cm/shared/package/OpenNebula/current/srv/cloud/configs/
     centos-5/init.sh /var/scratch/hdermois/OpenNebula/id_dsa.pub",
44 target     = "hdc",
45 root_pubkey = "id_dsa.pub",
46 username    = "opennebula",
47 user_pubkey = "id_dsa.pub"
```

```
48 ]
```

B. Intel MPI Benchmark compilation

To compile the Intel MPI Benchmark the following has been done on all platforms[22].

Listing 5: Compilation of IMB

```
1 # For example  the MPI_HOME needs to be the OpenMPI compiler.
2 make -f make_mpich MPI_HOME=/usr/lib64/openmpi/1.4-gcc
```

C. Intel MPI becnhmark "PingPong"

Bytes	DAS-4 (μsec)	DAS-4 no InfiniBand(μsec)	OpenNebula (μsec)	AWS EC2 (μsec)
0	3.81	46.55	112.75	81.82
1	2.25	47.68	114.01	81.32
2	4.19	45.65	114.98	80.25
4	2.62	41.81	114.93	80.95
8	2.18	44.69	117.94	80.7
16	5.11	42.08	115.68	80.71
32	3.26	47.92	114.91	81.02
64	5.85	48.22	100.11	82.87
128	5.47	48.43	100.87	81.76
256	3.69	49.51	105.15	83.15
512	4.04	50.95	108.65	87.67
1024	4.93	56.97	130.76	91.4
2048	5.96	68.36	269.74	102.96
4096	7.36	79.08	344.64	125.58
8192	9.93	103.12	362.49	143.53
16384	19.66	209.36	519.48	216.97
32768	25.84	262.14	3715.68	258.57
65536	29.42	581.46	1755.55	479.69
131072	53.2	1031.22	2577.59	568.67
262144	98.92	1581.03	4411.18	1010.47
524288	260.8	2796.03	8466.25	1660.28
1048576	561.83	5116.38	18058.36	8829.55
2097152	738.45	9648.28	37539.18	17482.83

Table 8: The timing results for benchmark of the IMB.

D. DAS-4 environment script

Listing 6: Script used to run MPI on the DAS

```
#!/bin/sh

# Sanity checks to make sure we are running under prun/SGE:
if [ "X$JOB_ID" = X ]; then
echo "No JOB_ID in environment; not running under SGE?" >&2
exit 1
fi
if [ "X$PRUN_PE_HOSTS" = X ]; then
echo "No PRUN_PE_HOSTS in environment; not running under prun?" >&2
exit 1
fi

# Construct host file for OpenMPI's mpirun:
NODEFILE=/tmp/hosts

# Configure specified number of CPUs per node:
( for i in $PRUN_PE_HOSTS; do
echo $i slots=$PRUN_CPUS_PER_NODE
done
) > $NODEFILE

# Need to disable SGE's PE_HOSTFILE, or OpenMPI will use it instead or
    the
# constructed nodefile based on prun's info:
unset PE_HOSTFILE

. /etc/bashrc
module load openmpi/gcc

$MPI_RUN $OMPI_OPTS --hostfile $NODEFILE $PRUN_PROG $PRUN_PROGARGS
```

E. Amazon preparation scripts

Listing 7: Script to start the Amazon instances and create the hosts file

```
#!/usr/bin/python

import boto.ec2
import boto.ec2.address
import sys

def create_instance(connection):
instance_t = "r3.large"
image = "ami-5ab4f02d"



connection.run_instances(image, key_name='key',security_group_ids=['sg-
    ee16dc8b',
```

```
14 'sg-01c99764'], instance_type=instance_t)
15
16 nr_instances = 0
17 if len(sys.argv) > 1:
18 nr_instances= int(sys.argv[1])
19
20 connection = boto.ec2.connect_to_region("eu-west-1")
21
22 print connection.get_all_instance_status()
23 print connection.get_all_instances()
24
25 for i in range(nr_instances):
26 create_instance(connection)
27
28 instances = connection.get_only_instances()
29 #
30 f = open('hosts', 'w+')
31 for i in instances:
32 if i.ip_address and i.key_name == "key":
33 f.writelines("root@%s\n"%i.ip_address)
```

Listing 8: The script used to prepare all Amazon EC2 instances.

```
1 #! /bin/bash
2 #hosts is the file with all the participating instances.
3 HOSTS=$(cat hosts)
4 FIRST=$(head -1 hosts)
5 for i in $HOSTS
6 do
7 tmp=$(echo $i | cut -c 6-)
8 ssh -oStrictHostKeyChecking=no $i "cd project; git pull" &> /dev/null
9 ssh -oStrictHostKeyChecking=no $i "cd project/code/graph500/mpi;make
     clean; make" &> /dev/null
10 # ssh $i "cd project; git pull"
11 # ssh $i "cd project/code/graph500/mpi;make clean; make"
12 ssh -oStrictHostKeyChecking=no $FIRST "ssh-keyscan -t rsa,dsa $tmp 2>&1
     >> ~/.ssh/known_hosts;"
13 done
14 scp hosts $FIRST:/root
```