Summer 8-7-2020

Using High-Performance Computing Profilers to Understand the Performance of Graph Algorithms

Costain Nachuma
cnachuma@uno.edu

Using High-Performance Computing Profilers to Understand the Performance of Graph Algorithms

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

By

Costain Nachuma

B.S. University of Zambia, 2011

August, 2020

ACKNOWLEDGEMENT

I wish to acknwlodge the support and guidance that I received from several sources. Among them, the faculty in the department of Computer Science and fellow students.

Firstly and foremost, I specifically acknowledge my supervisor, Dr. Shaikh Arifuzzaman, asssitant professor of Computer Science at the University of New Orleans. His support for me was unwavering and inspiring.

Secondly, I am grateful for Dr. Minhaz Zibran and Dr. Tamjidul Hoque for their dedication to help me a better researcher. I also would like to thank Joseph Imseis and Md Abdul Motaleb Faysal who aided me a lot in the different phases of the research. Furthermore, I am grateful to Feng Chen, a LONI support staff, who helped me immensely with the hardware used in this research.

Thirdly, I would like to acknowledge the many friends at home and abroad who helped me cope with living in away from my home country, Zambia. Specifically, I want to mention all my friends from Chi-Alpha student organization.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

An algorithm designer working with parallel computing systems should know how the characteristics of their implemented algorithm affects various performance aspects of their parallel program. It would be beneficial to these designers if each algorithm came with a specific set of standards that identified which algorithms worked better for a specified system. Therefore, the goal of this paper is to take implementations of four graphing algorithms, extract their features such as memory consumption and scalability using profilers (Vtunes /Tau) to determine which algorithms work to their fullest potential in one of the three systems: GPU, shared memory system, or distributed memory system. The features extracted in this study were scalability, speedup, parallel efficiency and memory consumption. We find that when looking at various parallel algorithms: Community Detection, Communities through Directed Affiliations (Coda), Cluster Affiliation Model for Big Networks (BigClam), and Breadth First Search all achieved noticeable speedup with increasing processors.

Keywords – Parallel Computing, Community Detection, Graph Mining, Performance tuning, Big Data.

1 INTRODUCTION

Fluid dynamics, aerospace engineering, genomics, and astronomy, are just a few topics that use parallel or high-performance computing (HPC) to answer a variety of academic research questions. Traditionally, before the days of high-performance computing and parallel programming, an individual would sit in his/her office and think about a particular problem and then propose an answer for it. However, now, with the power of HPC and parallel computing, the scope and possibilities of the academic research questions asked are much larger. This is because we have much more data available for us to process than we did in the past. The amount of data that a standard application needs to keep track of is far beyond what one person can handle. Hence, the current trend, in almost all areas of research, is to have some sort of HPC facility/parallel computing resource that will help ask/answer as many questions as fast and efficient as possible [1,2,3].

The problem is that most algorithms that you can find online today are implemented in serial, meaning that one task will begin executing once another has finished. Serial programs typically can be optimized via parallel computing that ideally provide concurrency which will save designers/developers time and money. The general, idea behind parallel computing is that one can take a computational task and break it down into several similar sub-tasks that can be processed independently and whose results are combined afterwards, upon completion [4].

Although parallelism is ideal, it can be very difficult for some programs to be parallelized. Why exactly? Well, understanding the performance characteristics of applications in HPC environments can be overwhelming because of the increase in the complexity of architecture and programming paradigms that have been developed over the past couple of years. Even so, algorithm developers and researchers alike, aim to understand how parallel algorithms run in HPC environments in order to extract as much performance as possible. Taking this into consideration we aim to answer the following three research questions:

Q1). What are the performance characteristics of running the serial versions of graph algorithms?

Q2). What is the "optimal" running environment for parallel implementations of graph algorithm or set of parallel algorithms?

Q3). How much parallelism can we achieve before losing a significant increase in performance?

2 DESIGN AND APPROACH

This section includes the design of the study which is broken down into two phases. Figure 2 gives an overview of our approach. First, we gather implementations of multiple C/C++ serial and parallel graphing algorithms. Secondly, we run these algorithms on an HPC system along with Vtunes and other performance tools. In doing so we are also able to extract features such as memory consumption, communication overhead, scalability, speedup, etc.

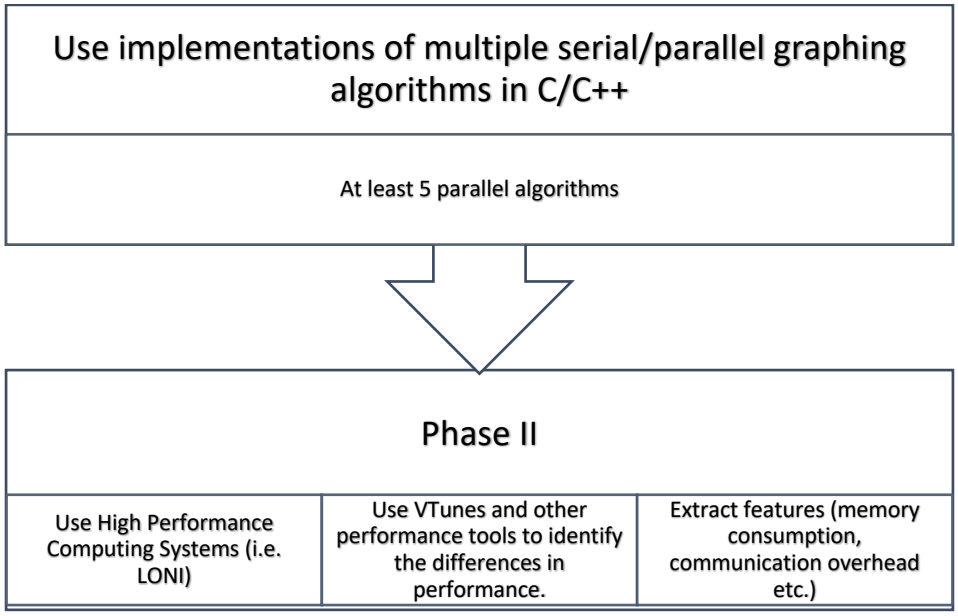

Figure 2: Study Design

A total of four algorithms were examined. The BigClam algorithm formulates community detection problems into a non-negative matrix factorization and discovers community membership factors of nodes. BigClam is an overlapping community detection method that scales to large networks which can consists of millions of nodes and edges. Our approach is based on the paper presented by Jaewon Yang and Jure Leskovec [7]. The second algorithm implements a large-scale overlapping community detection method known as Communities through Directed Affiliations (Coda) [8]. Coda handles both directed, as well as undirected networks, and is able to find 2-mode communities where the member nodes form a bipartite connectivity structure [14].

Another community detection algorithm was obtained thanks to University of New Orleans student Md Abdul Motaleb Faysal and is available online as of December 12, 2019 [19]. The last parallel algorithm examined was the Breadth First Search algorithm.

2.2 Experimental Setup

Two setups were used to carry out experiment I and experiment II.

For experiment I, the network used to connect to the HPC system was the Louisiana Optical Network Infrastructure (LONI) network. Using the LONI network we were able to access Queen Bee 2 (QB2), LSU's HPC cluster, which we then used execute our parallel algorithms and perform our analysis. QB2 is a 1.5 Petaflop peak performance cluster which contains 504 compute nodes, 960 NVIDIA Tesla K20x GPU's and over 10,000 Intel Xeon processing cores. The operating system used on these nodes is RedHat Enterprise Linux 6 OS. [11].

For experiment II, LONI infrastructure was used but a different cluster. Using the LONI network we were able to access Queen Bee 3 (QB3), LSU's HPC cluster, which we then used execute our parallel algorithms and perform our analysis. QB3 has 192 compute nodes each having two 24-core Intel Cascade Lake (Intel® Xeon® Platinum 8260 Processor) CPUs. Each node also has 600 GB HDD and 192GB memory. Operating system used on these nodes is RedHat Enterprise Linux 7 OS. [23]

To obtain our features, two profiling tools were used: VTunes Amplifier, TAU and hardcoded analysis. VTune Amplifier is a performance profiler application that allows for software performance analysis of 32 and 64-bit x86 based machines [12]. It can be run via the command line or via its graphical user interface (GUI) which is easy to use on a system given the right permissions at set on the server side. A few types of general analyses that can be done via Vtunes are hotspot analysis, memory consumption, HPC performance characterization and threading analysis. Vtunes is free for students and hence the preferred choice. Tuning and Analysis Utilities, also known as TAU, is a portable profiling and tracing toolkit that is used for performance analysis of parallel programs which are written in Fortran, C, C++, UPC, or Java. TAU is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling [13]. TAU also has a profile visualization tool known as ParaProf, which provides graphical displays of all the performance analysis results in aggregate (max, mean, std. dev) and in single node/context/thread forms. Examples of this visualization will be shown and explained in more detail. In addition to ParaProf, TAU also has PerfExplorer, a framework for parallel performance datamining. PerfExplorer allows for comparative feature analysis via graphical charts. Some of the charts that can be generated are time-steps per second, relative efficiency, and speedup of the entire application.

2.3 DATASETS

The datasets used range from a network 0.036 million (M) to 0.1M nodes and edges from about 0.2M to 3M. For evaluating BigClam and Coda algorithms, we used the *Enron email* network which includes approximately half-a-million emails. This data was originally made public, and posted to the web, by the Federal Energy Regulatory Commission during its investigation. [19]

The dataset used, by the community detection algorithm, was the *YouTube* network dataset. The final dataset used pertained to Facebook's social network. [20] Table 2.3 gives the summary of the datasets showing nodes, edges and network descriptions. Note that the YouTube data set is has more nodes and edges when compared to the Enron and Facebook data sets)

Using the execution time from the log files as our feature we were able to derive three metrics that were used for our analysis: Scalability, Speedup, and Parallel Efficiency.

NETWORK	NODES	EDGES	DESCRIPTION
Enron-Email	36692	183831	Half a Million Enron emails
YouTube	1134890	2987624	YouTube social Network
Facebook	4039	88,234	Facebook social Network

Table 2: Dataset for Experiments

3 EXPERIMENTAL ANALYSIS I

3.1 Implementation

After logging into the LONI network, we are given access to LSU's QB2 cluster. By default, the user will be placed in his/her home directory, which works off of the head node. The head node itself is limited in terms of storage space and number of available cores that can be used to run a particular program. So, in order to run our parallel programs and measure our features (scalability, speed up, and parallel efficiency) we must put in an allocation request via the qsub command. The standard qsub command that is used by the LONI network is shown here:

qsub -X -I -l walltime=hh: mm: ss, nodes=n: ppn=20 -A allocation name

-I is a flag which allows us to specify which resources we require. -l flag allows us to set the walltime, which can be thought of as the amount of time that we have access to our allocation. Note that we can also specify the number of computing nodes (nodes = n) and processors per node (ppn =20). In the case of QB2, every computing node has 20 processors available to us

3.2 Vtunes Analysis

Using VTune Amplifier, we can connect to the LONI network, via SSH connectivity, to execute the program and to extract the needed performance features. To do so, Vtunes requires a small script which contains a run command and the needed parameters that are specific to that particular parallel program (i.e. # of processors used, input graph used etc.). Figure 3.1 and Figure 3.2.1 shows the sample results obtained. Here is an example of a run command that could be found in a

small script: *mpirun -n 8 ./BFS.out*

The analysis shows important performance characterization metrics like, clocks per instruction (CPI) and execution time. Execution time is the main feature of this research, in this case: 8.292 seconds. Vtunes also provides an analysis of how effectively the available CPUs were utilized. In this case the effectiveness was only about 36 %. This indicates that more parallelism can be attained and hence better CPU utilization. Additional experiments using the same algorithm, with an increasing number of utilized CPUs are discussed in chapter 4.

Using the BigClam algorithm shown in figure 3.2.2, we see an execution time of 45.4 seconds and effective CPU utilization of about 50 %. Note, that this second experiment focuses only on threading efficiency.

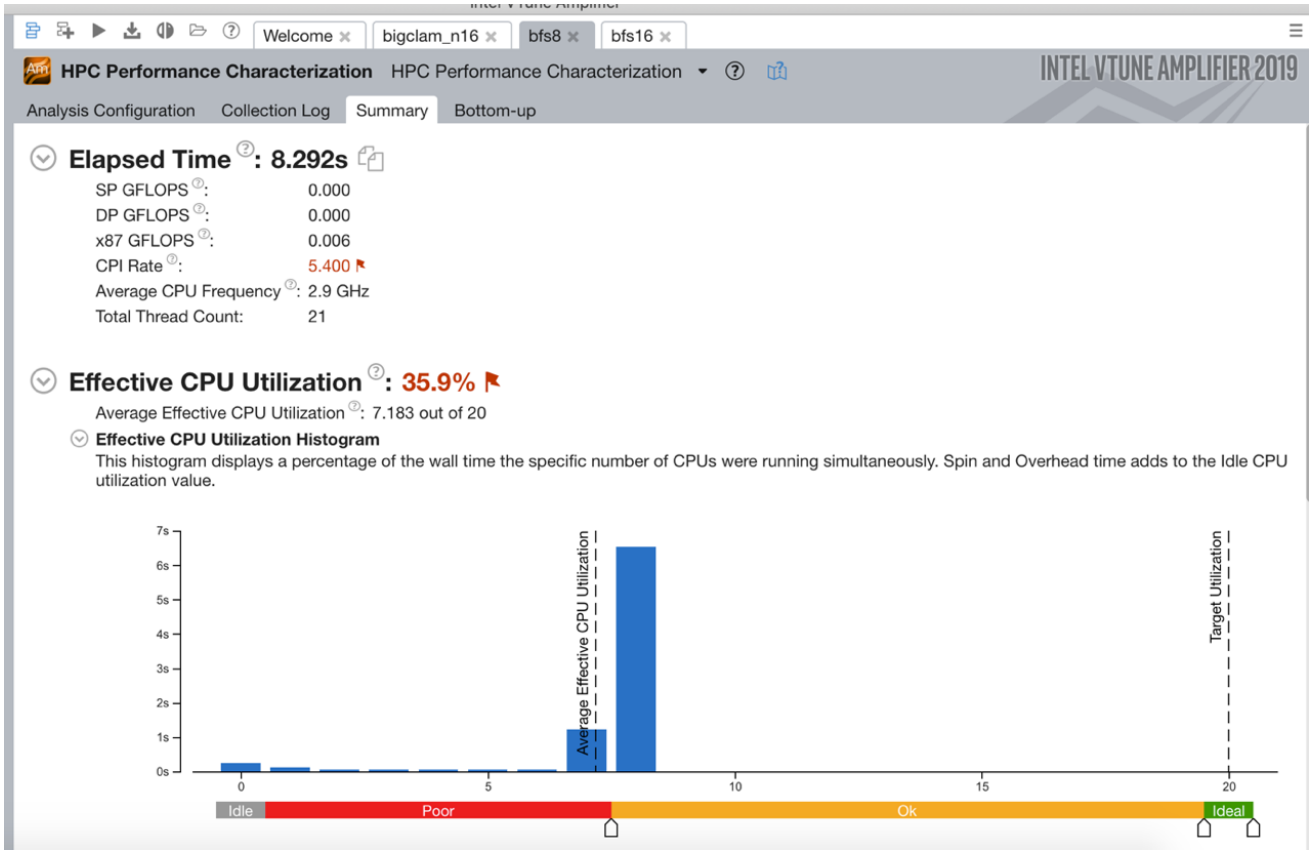

Figure 3.2.1: Vtunes Analysis using 8 Core

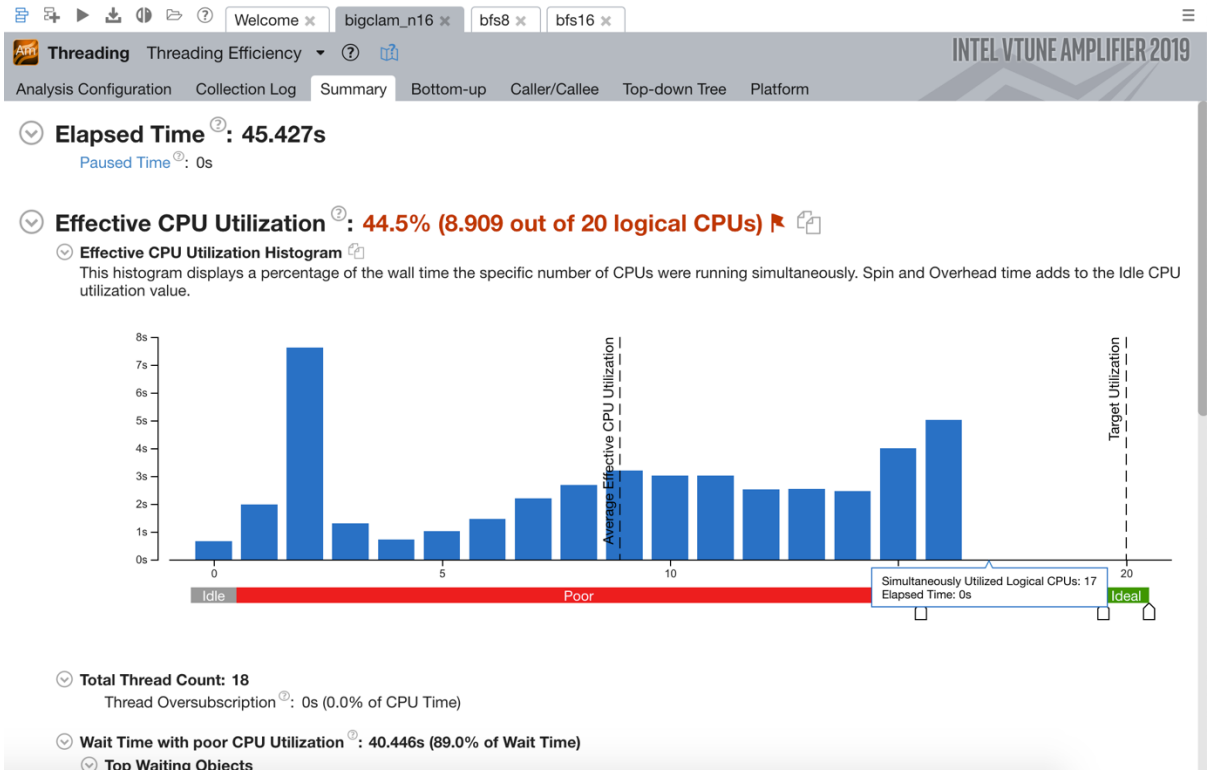

Figure 3.2.2: Vtunes example for BigClam algorithm with 16 cores in use.

3.3 TAU Analysis

Profiling via TAU is slightly different when compared to profiling via VTune Amplifier, in that, it must be run through the command line interface of LONI. Once we connect to the LONI network we must load the TAU module in order to use its profiling tools. We do this by inputting the command as such: module load tau. Once the module finishes loading, we add tau_exec to the run command. This command ensures that tau profiles the parallel program. An example run command using TAU:

mpirun -np 64 tau_exec./ompRelaxmap

We are then able to use TAU's visualization tool, ParaProf, by typing paraprof into the command line. Figure 3.3.1 and figure 3.3.2 is an example visualization using execution time as the metric measured. Note that for this example we ran the program with 64 processors.

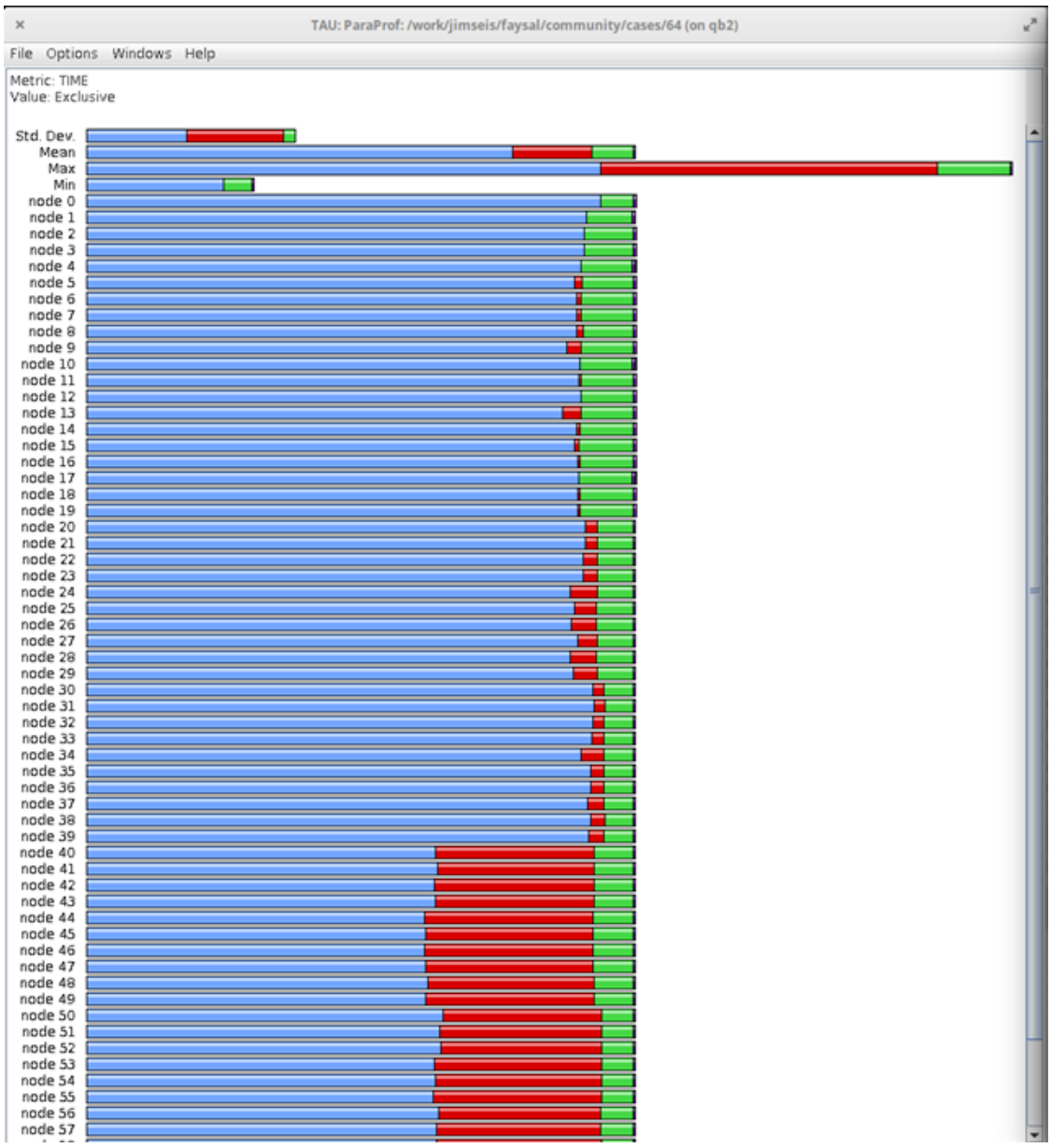

Figure 3.3.1: TAU Analysis of Execution time

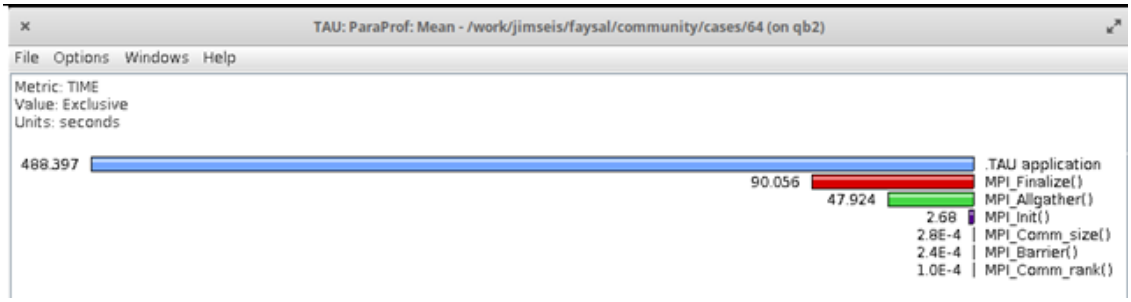

Figure 3.3.2: An example of TAUs ParaProf visualization tool.

3.4 Logging Analysis

Another effective and reliable way we found to analyze the algorithms was through logging the time at code level. Vtunes and Tau results were compared to the results the manual logging.

4 RESULTS AND EVALUATION I

4.1 Scalability

The scalability of a parallel algorithm on parallel architecture is a measure of its capacity to effectively utilize an increasing number of processors [14]. In this study we use strong scaling which is defined as how the solution time varies with the number of processors for a fixed total problem size.[21]

Table 4.1 describes the scalability of the four algorithms in terms of execution time and processor count.

# Processors	Coda	BFS	BigClam	Infomap
1	41.00	16.65	397.00	397.94
2	25.00	8.48	61.00	345.92
4	15.00	4.64	35.00	282.18
8	10.00	3.77	22.00	274.37
16	8.00	3.43	15.00	215.34
32	6.00	1.4	14.00	178.89
64	6.00	0.69	13.00	178.73

Table 4.1: Processor Count and Execution time for each algorithm

When we look at the results, all four algorithms analyzed show considerably strong scalability. For example, Coda runs serially at 41 seconds. Note that as we add more processors, the execution time improves significantly with the introduction of just one additional core (2 in total). Two cores alone, reduces the execution time by nearly 40 % from 41 to 25 seconds. Further addition of cores reduces the execution time until a point when the addition of more hardware no longer yields tangible improvement in performance (You could kind of just say that you have a plateau point here rather than this lengthy sentence). This happens at 32 processors for Coda, with an execution time of 6 seconds. Hence the optimum number of cores is 32. Figure 4.1.1 Shows the plot.

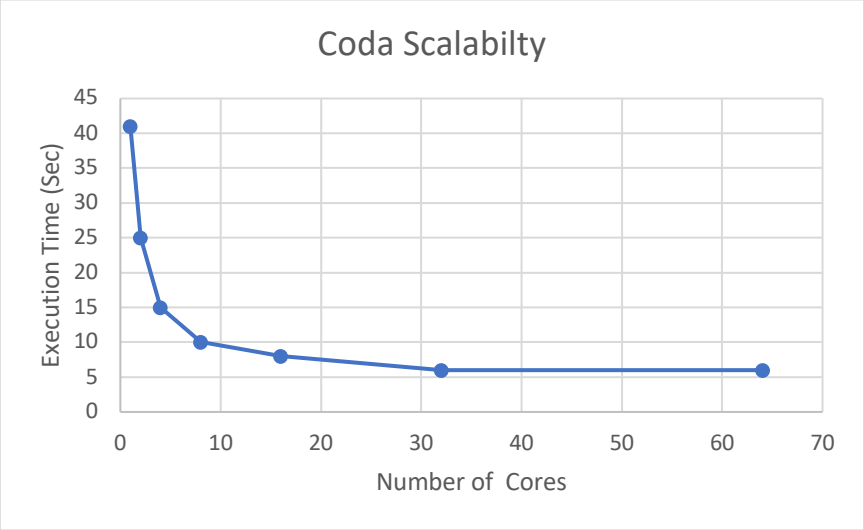

Figure 4.1.1: Coda Scalability Plot

The BFS algorithm shows a significant reduction in execution time with the addition of a second core (~50% from about 17 seconds to 9 seconds). Note, that no noticeable gains, in terms of execution time, were seen after 8 cores (execution time for 8 cores and 16 cores is nearly the same time avg.~3.5 seconds). Therefore, the optimum number of cores is 8. Figure 4.1.2 shows the plot.

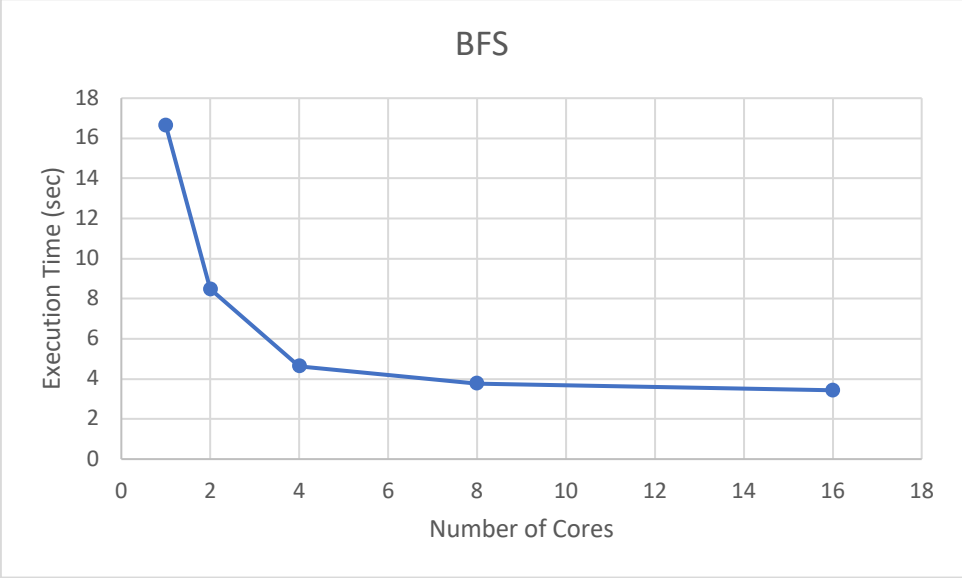

Figure 4.1.2: BFS Scalability Plot.

As for the BigClam algorithm, we see the greatest improvement of all the algorithms studied after the addition of a second core. The improvement in execution time is about 600%, from 397 seconds to 61 seconds. The improvement diminishes slowly with the addition of more cores as seen in the previous plots above. The optimum number of processors is seen at about 16 cores with the execution time reduced to 15 seconds (26x better than the serial execution time). Figure 4.1.3 shows the plot.

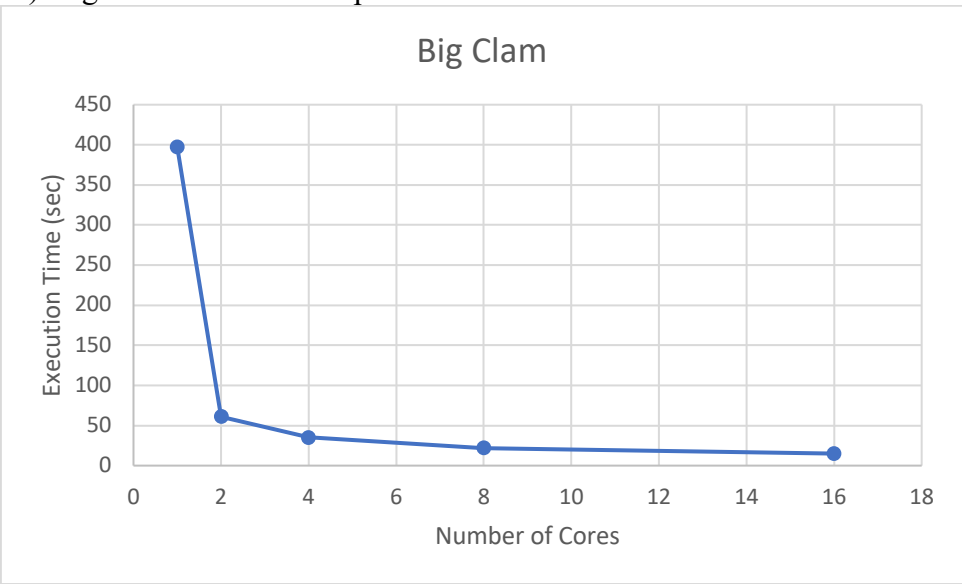

Figure 4.1.3: Big Clam Scalability Plot

Finally, the Infomap algorithm begins at 397 seconds when run serially and improves to an optimum of 178 seconds when 32 cores are used. Beyond 32 cores, we are merely wasting hardware, as no significant improvements are seen when 64 cores are utilized. Figure 4.1.4 depicts the plot.

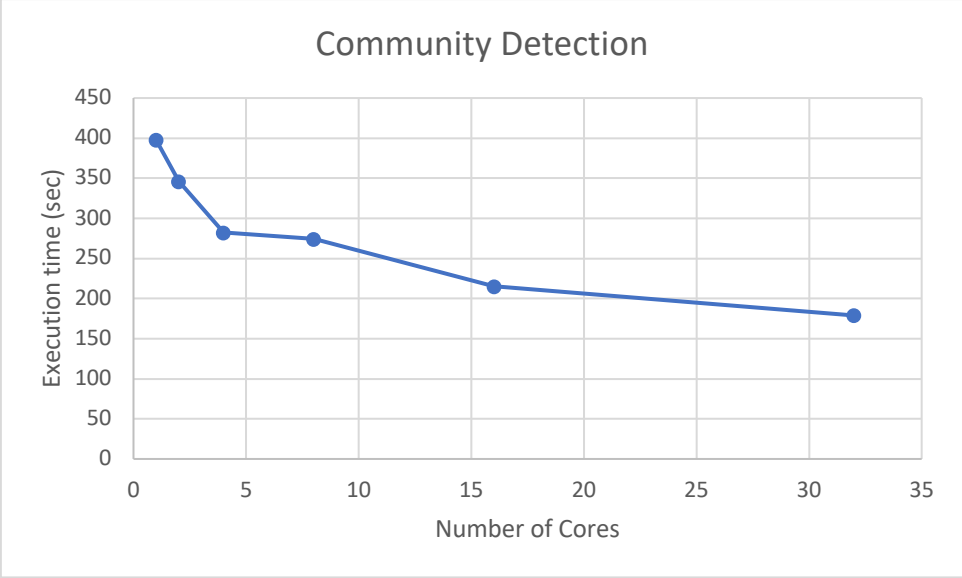

Figure 4.1.4: Infomap Community Detection plot.

4.2 Speedup

Speedup is simply defined as the execution time using one processor divided by the execution time using multiple p processors. The notion of speedup was established by Amdahl's law, which was particularly focused on parallel processing [22]. Using results from table 4.1 we evaluate the speedup for all the algorithms depicted in table 4.2.

# Processors	Coda	BFS	BigClam	Infomap
1	1.00	1.00	1.00	1.00
2	1.64	1.96	6.51	1.15
4	2.73	3.59	11.34	1.41
8	4.10	4.42	18.05	1.45
16	5.13	4.85	26.47	1.85
32	6.83		28.36	2.22
64	6.83			2.23

Table 4.2: Speedup for each algorithm

Looking at table 4.2, we see the optimum speed up for Coda is 6x at 32 processors. The BFS algorithms has ~5x speedup with 16 processors. On the other hand, BigClam has a considerable 26x speedup at 16 processors. The Infomap algorithm shows a speedup of 2x at 32 processors. In the case of a machine learning model, Algorithms similar to Coda or the Infomap would have 32 cores for optimum speedup. Algorithms similar to BFS implementation and BigClam would have 16 cores for the predicted optimum speedup.

Further testing would be required to solidify these findings.

4.3 Parallel Efficiency

The last metric is parallel efficiency, which is defined as the speedup obtained divided by the number of processors used. Using table 4.2 we derive table 4.3. Parallel efficiency helps us measure the quality analysis of our parallel implementation, in terms of workload balancing, as the processing nodes increase. We observe that all algorithms exhibit diminishing efficiency as nodes are added. At some point, which we deem as the optimum, there is little efficiency gained as more nodes are added. It is at this point for each algorithm that we stop adding more nodes.

# Processors	Coda	BFS	BigClam	Infomap
1	1.00	1.00	1.00	1.00
2	0.82	0.98	3.25	0.58
4	0.68	0.90	2.84	0.35
8	0.51	0.55	2.26	0.18
16	0.32	0.30	1.65	0.12
32	0.21		0.89	0.07

Table 4.3: Parallel Efficiency for all Algorithms

5 EXPERIMENTAL ANALYSIS II

To obtain our features, code level logging analysis was utilized Similar runs were done as in experiment 1 on different hardware. The QB3 cluster was used to run the tests. Based on the results of experiment 1, we note that only one compute node having 48 cores will be enough for experiment II. This is because the optimum number of cores for experiment did not exceed 32 cores for any the algorithms.

5.1 Implementation

To run an algorithm on the QB3 cluster, a different approach is detailed below.

First, we request a compute node:

srun -p workq --pty /bin/bash

Next, we run the algorithm once the node is allocated.

srun -n16 ./bfs ./enron.csr

This command was used to run the BFS algorithm. Similar commands were used for the other algorithms with different parameters being passed depending on the specific algorithm.

6 RESULTS AND EVALUATION II

6.1. Scalability

The Results obtained were as follows: Dataset Enron-emails on QB3 cluster.

# Processors	BFS	Coda	BigClam
1	12.15	35.41	365
2	6.17	24	62
4	3.17	15	41
8	1.56	11	35
16	0.99	6	21
32	0.75	5	17
48	0.71	5	15

Table 6.1.1 QB3 Execution times (in seconds) for the algorithms profiled

The second experiment demonstrates that the results and parallel efficiencies of the algorithms will hold even for different hardware. With this knowledge we are able to confirm that we can predict the optimum running parameters for similar algorithms. However, we need to establish the definition of similarity when referring to various differing algorithms. More research is required for the specifics each algorithm and its numerous implementations in order for us to obtain features which could be used for a machine learning model.

6.2 Speedup

From table 6.1.1 we obtain the speed up and compare the values with the results of experiment I. For discussion purposes, we use two of the algorithms to demonstrate the correlation of the results.

Coda	Speed up Values	
# Processors	**Experiment I QB2**	**Experiment II QB3**
1	1.00	1.00
2	1.64	1.48
4	2.73	2.36
8	4.10	3.22
16	5.13	5.90
32	6.83	7.08

Table 6.2.1: Speed up for Coda on QB2 cluster vs QB3 cluster

Figure 6.2.2 visualizes the similarity in performance for different clusters

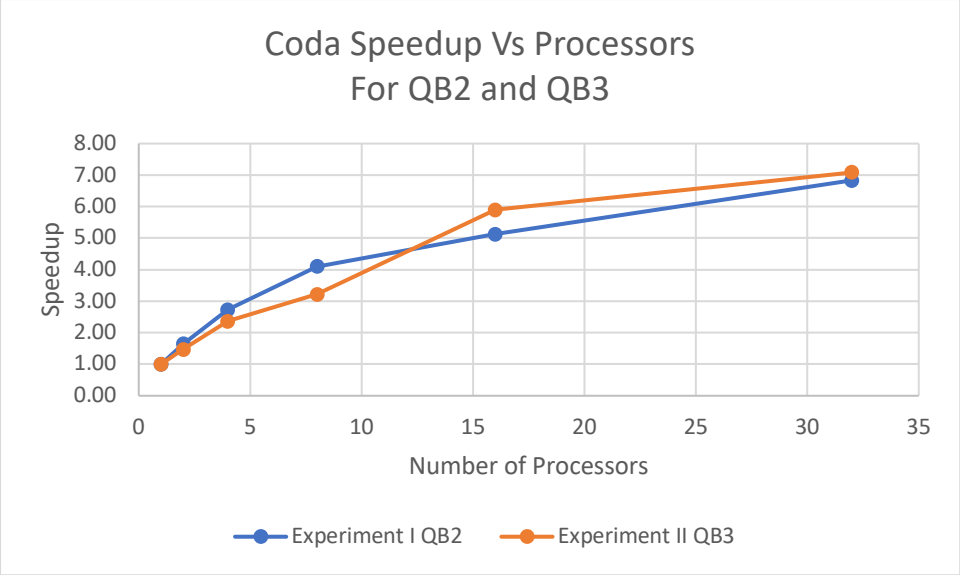

Figure 6.2.2: Coda Speedup Vs Processors for QB2 and QB3

Looking at figure 6.2.2, we see that almost identical speedup is achieved for both experiment I and experiment II. This validates the quality of our results. This gives much promise to the automation of optimum system specifications for similar graph algorithms.

We also obtain the results for BigClam as follows:
Note: to avoid repetition of similar results, we are only presenting the speedup graph here.

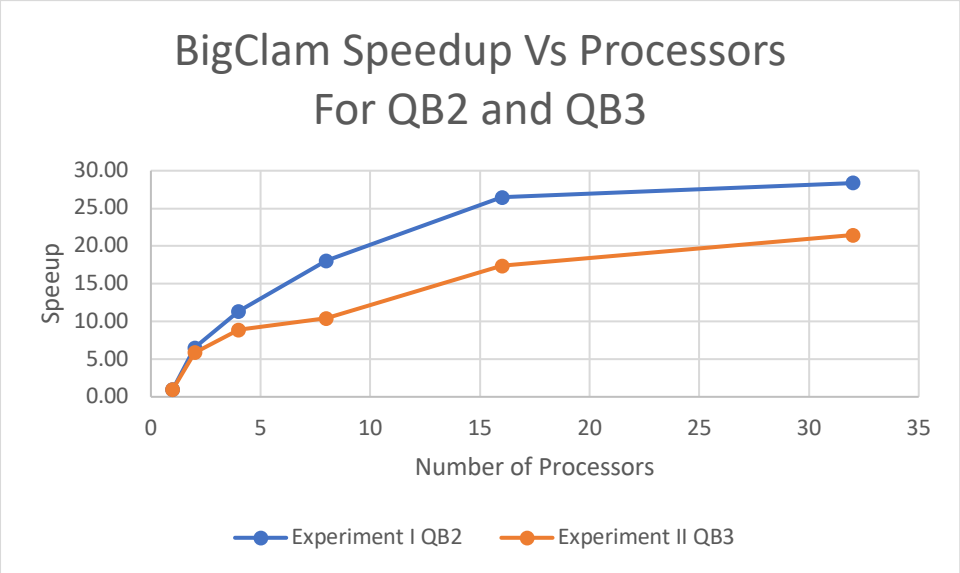

Figure 6.2.3 BigClam Speedup Vs Processors for QB2 and QB3

Once more, we see that the performance of the algorithms on different clusters is similar. Although not replicated in this paper, the results show an analogous pattern for both profiled BFS and Infomap algorithms. Examining BigClam, we notice the speed starts to sharply decline around 16 to 32 processors where it nearly flattens out. This shows that more

computing resources yield little gain, hence we stop right there and set the points where the graphs flatten out as optimum number of cores for each algorithm respectively (Sentence runs on to long). We notice the speedups are slightly impacted by the different computational capabilities of the different clusters.

6.3 Experimentation with other Networks

Further experimentation was done for the two algorithms using different datasets to help validate results of the optimum number of cores. The results obtained for the Facebook dataset are discussed as follows in figure 4.5.4. for the Coda algorithm.

Coda Execution time

# Processors	Facebook	YouTube
1	68	173
2	25	34
4	23	19
8	11	15
16	8	10
32	7	9
48	6	8

Table 6.3.1: QB3 Scalability on Facebook Dataset for the Coda Algorithm

We see from the results that both algorithms, for this dataset, also exhibit scalability as more processors are added.

We obtain the speed-ups and compare them with results obtained from Enron emails dataset. The results are shown in table 4.6 and figure 4.7.

Coda

# Processors	Enron	Facebook	YouTube
1	1.00	1.00	1.00
2	1.48	2.72	2.35
4	2.36	2.96	3.33
8	3.22	6.18	4.18
16	5.90	8.50	5.58
32	7.08	9.71	6.76

Table 6.3.2: Coda algorithm Speedup on QB3 for different datasets.

In order to better understand the results, we use table 6.3.2 to plot figure 6.3.3

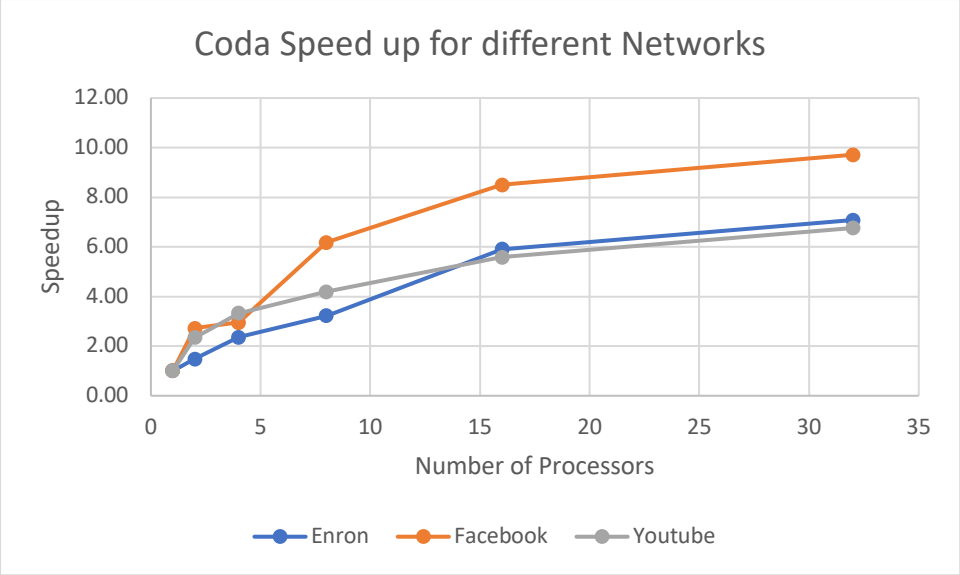

Figure 6.3.3: Speedup over different networks for Coda

Speedup for every network in the dataset is measured against the runtime of detecting communities in a single processing unit.

We noticed that different networks show a similar pattern in speedup, but the values are not the same. This is due to the fact that the number of nodes and edges in a dataset plays an important role in computation and communication. With more nodes and edges, it means more computation and communication and perhaps more memory. All networks exhibit similar behavior as the speedup declines around 16 core and almost flattens out around 32 cores. This consistency is the basis we can use to model predictions for similar algorithms.

We also run the network test on the Big Clam algorithm and the results are shown in table 6.3.4 and illustrated in figure 6.3.5

BigClam

# Processors	Enron	Facebook	YouTube
1	1.00	1.00	1.00
2	5.89	2.72	7.72
4	8.90	2.96	11.71
8	10.43	6.18	15.88
16	17.38	8.50	22.02
32	21.47	9.71	24.83

Table 6.3.4: Speedup over different networks for BigClam

The resulting plot for the above results is shown in figure 6.3.5

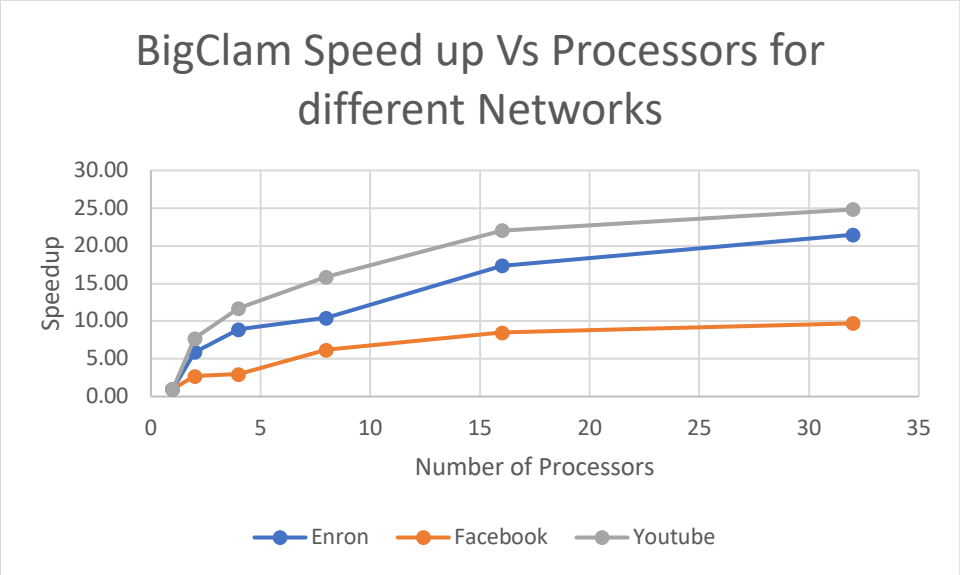

Figure 6.3.5 Speedup over different networks for BigClam

The experiments for the two algorithms show clear similarity in Algorithm behavior over different networks. The results should hold for other algorithms as well. The concept has been thoroughly tested with regard to scalability, speedup and also different hardware. When we profile more algorithms in the future, a proper model needs to be developed and tested to finalize the desired outcome of our preliminary research. The YouTube network was the largest and also benefited most from parallelism. We also noted that the smallest network profiled also depicted the smallest speedup. However, no conclusions can be made at this point, as the previous results differ. In figure 6.3.3 we note that Facebook network exhibited the greatest speedup.

7 EXPERIMENTAL ANALYSIS III: MEMORY CONSUMPTION

The third and final experiment done was on memory usage with respect to scalability of the graph algorithms. Two algorithms were tested; Coda and BigClam. The experiment was done on QB3 cluster using command line-based analysis using Intel® Parallel Studio XE 2020.

Memory Consumption analysis explorse memory consumption (RAM) over time and identify memory objects allocated and released during the analysis run. During Memory Consumption analysis, the VTune Profiler data collector intercepts memory allocation and deallocation events and captures a call sequence (stack) for each allocation event (for deallocation, only a function that released the memory is captured). VTune Profiler stores the calling instruction pointer (IP) along with a call sequence in data collection files, and then analyzes and displays this data in a result tab. [24]

To get start: A dedicated node had to be requested using the following command:

srun -p workq -w nodeName --pty /bin/bash

I needed a specific node because for the analysis to take place, we need the perf_event_paranoid value to be set to 1 or less. This allows performing of analysis on the computing node.

Next, export the path variable to point to where the analysis software is installed.

Export PATH=/usr/local/compilers/Intel/parallel_studio_xe_2019.5/vtune_amplifier_2019/bin64/:$PATH

Then we run the analysis
amplxe-cl -report summary -report-knob show-issues=false -r ./myresult ***-collect memory-consumption*** *srun -n16 ./x ./enron.csr*

After the results were obtained, I securely transferred them to an ordinary computer with Vtunes GUI installed. This was done to easy the analysis. Figure 7.0.1 shows the results of the BigClam for 16 cores being used.

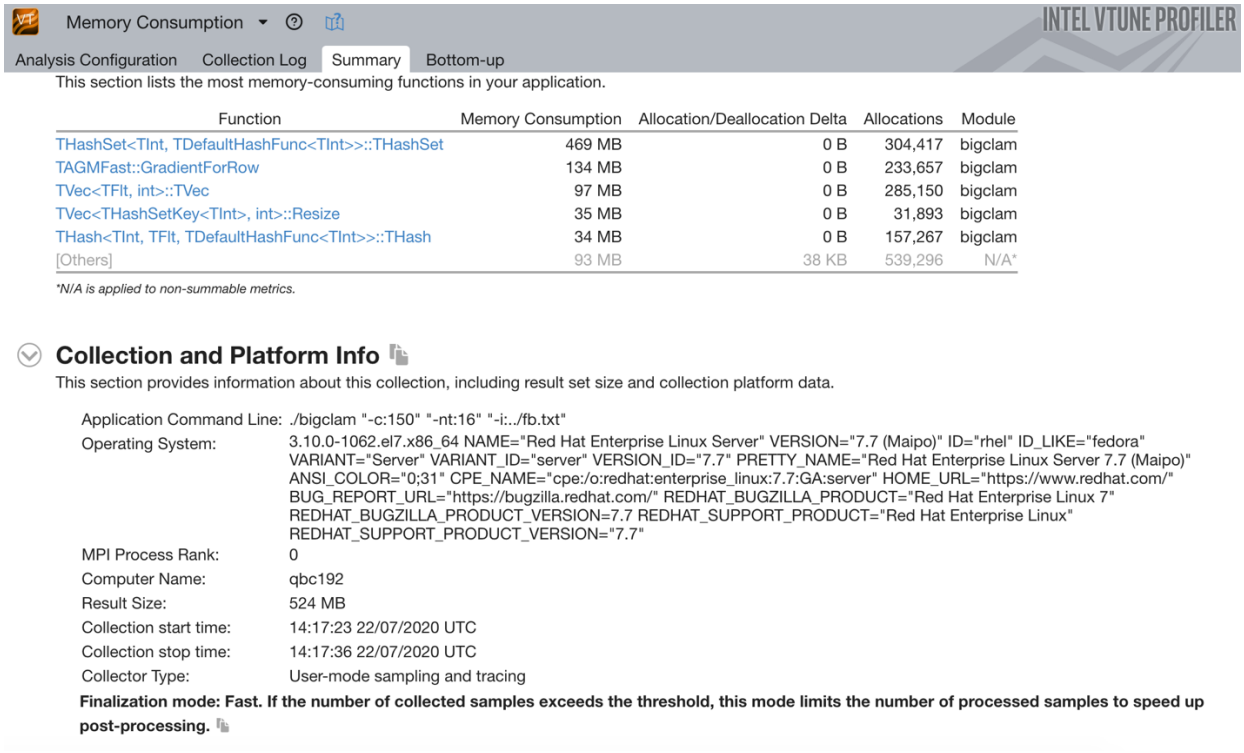

Figure 7.0.1: Vtunes Memory Analysis for BigClam with 16 processors run.

8 RESULTS AND ANALYSIS III

We run memory analysis on the Coda and BigClam algorithms. The both algorithms had a top memory-consuming function. Table 8.0.1 shows the memory consumption functions for both algorithms in relations to parallelism. This test was performed on the Facebook Network dataset.

Facebook		MB
# Processors	**Coda**	**BigClam**
1	8192	1024
2	627	471
4	624	469
8	624	469
16	624	465
32	599	465
48	599	465

Figure 8.0.1: Top memory-consuming functions memory usage.

From table 8.0.1, we see that the serial versions of the algorithms use up a lot of memory as compared to the parallel versions. For the Coda algorithm, it was noticeable that additional cores had a significant reduction in memory usage of ~1200 %. However, the memory consumption remained more less the same even though more cores were used from the 2nd to the 48th core. It may have to do with how the program was written.

On the other hand, for BigClam, running the algorithm with one core verses two cores shows a reduction in memory consumption of about ~55% for the parallel version. Like the Coda algorithm, additional cores did not significant alter the memory consumption.

Other tests may need to be performed on memory usage to have a better idea of how the memory is being utilized.

9 LIMITATIONS

When using VTune Amplifier we were unable to come up with the correct configuration to profile the parallel algorithms using more than 20 processors. Recall that the QB2 cluster requires the user to submit an allocation request in order to use more than 20 processors. Also recall that VTune Amplifier also requires a script which allows for the execution and profiling of a parallel program. We believed that in order to run a parallel program with more than 20 nodes we would also have to insert the allocation command into our script which contained our run command. However, when running this script, VTune Amplifier obtains the allocation but only profiles the running script rather than profiling our actual parallel program. To overcome this, A breakthrough came in later in the research with the introduction of QB3 cluster which has 48 computing cores on one node. However due to permission limitations, an allocated cluster does not allow ssh connections which is required by an external Vtunes application. I had to resort to command line-based analysis using Vtunes which is not as visual or user friendly. To further overcome this problem, I had to export the command line-based results to another ordinary computer which helped me view the visualizations of the results. The process ate away valuable time as I spent countless days of debugging and learning how to use the system with limitations. To avoid all the challenges, all that was needed was to set perf_paranoid value on the server to 0. However, LONI, possibly for security reasons, does not allow this. They were gracious enough to set it to 1 for me on one single node after much discussion. Setting the per_paranoid value to 1 allows for command-line based analysis.

Failing to find a timely solution to the VTune Amplifier problems, we then moved onto TAU as an alternative profiler. With TAU we were able to obtain the execution time of our community detection algorithm. However, when we compared the results obtained from TAU with the results of our program logs, it was discovered that the execution time measure via TAU was significantly larger. We are still unsure as to why this was the case, but we believe it might have something to do with TAU inefficiently utilizing the allocated processors (load balancing issues or not using all the nodes as specified). We are learning more about TAU by reading its documentation so that we can determine exactly why we have this large difference in execution time.

It must also be noted that finding and testing these parallel algorithms was an extremely challenging task. The reason being, is that there are very few publicly available parallel algorithms online and of the few that are available most of them are poorly documented and/or require various amounts of refactoring before they are able run.

10 RELATED WORK

In this section we will discuss some related works in terms of feature classification and parallel algorithms. Our report relies heavily on feature analysis. Islam et al presents a customizable framework for analyzing performance measurements and visualizing through a web-based interactive dashboard for interactively exploring a large volume of hierarchal information. The interactive framework they developed known as DASHING allows for interactive visualizations that provide both coarse and fine-grained information about the causes of a target performance metric (e.g. efficiency loss) [15].

Arifuzzaman et al describes that big graphs can consists of millions or even billions of node and edges and that these graphs demand the development of parallel computing algorithms. The triangle counting algorithm they developed uses dynamic load balancing such that the algorithm can compute the exact number of triangle ins a network with 1 billion edges in only 2 minutes using only 100 processors (i.e. good speed-up and scaling) [16].

Rastogi et al describes the history, significance, and need for parallel algorithms. They also discuss load distribution, synchronization, fault tolerance and communication overhead along with the PRAM model for parallel computation [17].

11 CONCLUSION AND FUTURE WORK

In this thesis, we analyzed the four parallel algorithms and were able to extract execution time as our defining feature. Using execution time, we were able to determine three other metrics: scalability, speed-up, and parallel efficiency. For all algorithms profiled, we noticed that scalability was plateaus, between 8 and 64 processors at some point for each graph, the parallelization overhead produced resulted in little to no speed-up. Our work done up to this point is far from complete. In the future, we need to obtain more parallel implementations. Further, we need to fully understand how TAU works in order for us to extract even more features (hardware counters, memory analysis etc.…). We would also need to run our algorithms on the various environments that that the QB2/3 cluster offers (i.e. test our algorithms on their GPU's instead of their CPUs). After this stage, we would apply machine learning techniques to classify algorithms according to similarity in optimum performance features and come up with a model to automate this process.

LIST OF REFERENCES

[1] Blaise Barney, “Introduction of parallel computing” Lawrence Livermore National laboratory Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, "Introduction to Parallel Computing" http://www-users.cs.umn.edu/~karypis/parbook/

[2] "Book Reviews," IEEE Concurrency, vol. 2, no. 2, pp. 81-84, Summer, 1994.

[3] Thomas W. Crockett, An introduction to parallel rendering, Parallel Computing, Volume 23, Issue 7, July 1997.

[4] “Parallel Computing.” Wikipedia, Wikimedia Foundation, 4 Dec. 2019, en.wikipedia.org/wiki/Parallel_computing.

[5] Narayanan Sundaram Linkedin Profile, 5 Dec. 2019, www.linkedin.com/in/narayanan-sundaram-94431626/.

[6] Google Scholar Results for Narayanan Sundaram, 5 Dec. 2019, scholar.google.com/citations?hl=en&user=9HiV_AEAAAAJ&view_op=list_works&sortby=pubdate.

[7] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: A nonnegative matrix factorization approach. In Proceedings of the Sixth ACM International Conference on Web Search and Data Mining, WSDM ’13, pages 587–596, New York, NY, USA, 2013. ACM.

[8] Snap Algorithm Descriptions https://snap.stanford.edu/snap/description.html

[9] Owwwlab.com. Shaikh M. Arifuzzaman: Publications, 5 Dec. 2019, www.cs.uno.edu/~arif/publication.html#.

[10] SNAP datasets https://snap.stanford.edu/data/index.html

[11] “High Performance Computing.” HPC@LSU | Documentation | User Guides | QB2, www.hpc.lsu.edu/docs/guides.php?system=QB2#access.

[12] “VTune.” Wikipedia, Wikimedia Foundation, 29 Aug. 2019, en.wikipedia.org/wiki/VTune.

[13] “Tuning and Analysis Utilities.” TAU - Tuning and Analysis Utilities -, www.cs.uoregon.edu/research/tau/home.php.

[14] Kumar, V.P., and A. Gupta. “Analyzing Scalability of Parallel Algorithms and Architectures.” Journal of Parallel and Distributed Computing, Academic Press, 25 May 2002, www.sciencedirect.com/science/article/abs/pii/S0743731584710999.

[15] Islam, Tanzima Z., et al. “Towards A Programmable Analysis and Visualization Framework for Interactive Performance Analytics.” Tanzimaislam, www.tanzimaislam.com/.

[16] Arifuzzaman et al. “A Fast-Parallel Algorithm for Counting Triangles in Graphs Using Dynamic Load Balancing” - IEEE Conference Publication. ieeexplore.ieee.org/document/7363957.

[17] Rastogi, Shubhangi, and Hira Zaheer. “Significance of Parallel Computation over Serial Computation Using OpenMP, MPI, and CUDA.” SpringerLink, Springer, Singapore, 1 Jan. 1970, link.springer.com/chapter/10.1007%2F978-981-10-5577-5_29.

[18] M. A. M. Faysal and S. Arifuzzaman, "Distributed Community Detection in Large Networks using An Information-Theoretic Approach," 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 4773-4782, doi: 10.1109/BigData47090.2019.9005562.

[19] http://snap.stanford.edu/data/email-Enron.html.

[20] https://snap.stanford.edu/data/com-Youtube.html

[21] https://en.wikipedia.org/wiki/Scalability#Performance_tuning_versus_hardware_scalability

[22] https://en.wikipedia.org/wiki/Speedup

[23] http://www.hpc.lsu.edu/docs/guides.php?system=QB3

[24] https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/hotspots-analysis-group/memory-consumption-analysis.html

VITA

The author was born in Lusaka, Zambia. He obtained his bachelor's degree in Computer Science from the University of Zambia in 2011. He joined the University of New Orleans Computer Science Graduate program in 2018. He became a Graduate Assistant under Dr. Christopher Summa in Spring and fall of 2019. In Spring and Summer 2020, he worked as a Graduate Assistant in the Office of Enrolment Services under Ann Lockridge.