“The Efficiency and Efficacy of Parallel Computing”

An Honors Thesis

by

Scott T. Sheppard

California, Pennsylvania

2018

California University of Pennsylvania

California, Pennsylvania

We hereby approve the Honors Thesis of

Scott T. Sheppard

Candidate for the degree of Bachelor of Science

Date

Faculty

4-24-18

Paul Sible, MS
Honors Thesis Advisor

4-24-18

William Dieterle, PhD
Second Reader

4-24-18

Gregg Gould, PhD
Honors Advisory Board

4/24/18

Craig Fox, PhD
Associate Director, Honors Program

24 April 2018

M. G. Aune, PhD
Director, Honors Program

The Efficiency and Efficacy of Parallel Computing

Scott Sheppard
University Honors Department
California University of Pennsylvania
California, Pennsylvaia, United States of America
She1472@calu.edu

Abstract— An n-body gravitational simulator was created and used to analyze the efficiency and efficacy of parallelizing a workload across a number of parallel processing cores in personal computing systems. While parallelizing does have definite performance advantages, there exist hurdles and limitations to implementing such.

Keywords—Parallel Processing, Amdahl's Law, Moore's Law, Multicore, Multithreaded

I. Introduction

Since the introduction of the first microprocessor chips over half a century ago, the processing power available for a single chip system has consistently grown exponentially throughout the decades. However, due to a variety of factors this trend has been beginning to stagnate recently. As a result, multicore processors have become the norm for the consumer market in recent years, with higher-end consumer chips being capable of processing up to sixteen simultaneous threads. However, in order to take full advantage of the multicore processors, programs will need to be rewritten with parallelization in mind. Not all workloads will see substantial benefit for this extra effort, but it may allow problems that were once unfeasible to become easily solvable in a short time. Here, this is demonstrated by analyzing the performance of an n-body gravitational simulator as an example of a parallelized workload.

II. Backgound Information related to Parallelization

A. Moore's Law

For most of the past sixty years or so, microprocessors have tended to have followed Moore's Law rather closely, roughly doubling a single chip's density of transistors every two years [1]. These transistors are the incredibly small electrical switch-like components that are the basis for most modern computer processors since the introduction of the Intel 4004 in 1971[2]. Since then, the transistor count has been consistently rising by a factor of two roughly every two years. This has been achieved by both increasing the total chip size and manufacturing ever smaller transistors into more densely packed areas. These smaller transistors have the advantage of needing less of an electrical charge to function, and as a result are faster, use less energy, and produce less waste heat.

For an example of how far microprocessor technology has progressed, in the previously mentioned Intel 4004, there was a total of 2,300 transistors on the 12 mm^2 size chip, with a spacing of about 10,000 nm between the transistors. More modern Intel processors, such as a Skylake based Core i7-7700k, have approximately 1.7 billion transistors on a 122 mm^2 chip, with only about 14nm between transistors.

However, in recent years manufacturers have been starting to encounter an ever-increasing difficulty in designing, manufacturing, and producing processors that still follow Moore's Law [3]. At present, transistors are so small they can easily measure 10 to 20 atomic diameters across their shortest axis. In fact, this is so small that the physics underlying many current technologies simply do not work if the transistors are made any smaller, either due to just the impossibly small scales that are being dealt with, or because of the increasing role that quantum effects start to play at these levels.

Physical constraints are not alone beginning to slow progress. As the transistors become even smaller, the cost of researching and trying to develop the technology to reliably produce them also continues to increase [4]. Intel has recently been struggling to create a fabrication process to reliably create transistors for their next generation products. The research and development cost of this struggle is estimated to be about one third of Intel's yearly profits, and as this price increases, the amount of time that it will take to produce newer, faster processors will also increase.

B. Amdahl's Law

As with just about anything, there are some tradeoffs when a program runs in a parallelized manner, rather than as a single threaded process. Gene Amdahl wrote a paper in 1967 titled *Validity of the single processor approach to achieving large scale computing capabilities* [5], in which he outlines many of these issues. Among these are Data Management Housekeeping, boundaries are likely to be irregular, and interiors may be inhomogeneous. Assuming most of this can be overcome, he writes about the diminishing returns in execution speedup due to parallelizing a portion of the program across n processors. While he did not write it as such in his paper, Amdahl's Law has been paraphrased as follows.

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}} \quad (1)$$

In this equation, r_s is the portion of the program that is serial, in other words, this is the part that is not, or cannot be, parallelized. Alternately, r_p is the portion that can be made to run in parallel on n number of processing cores, and the sum of r_p and r_s is always 1. *Speedup* is therefore the ratio of the speed of the parallelized version compared to that of the serial version, where a value of one indicates there is no difference in execution speed. Fig. 1 shows predicted speedups at different n values for programs with different portions being parallelized.

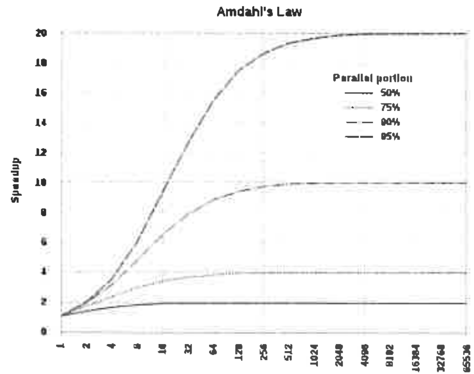

Fig. 1. Amhal's Law Graphed for Various Values [6]

As is visible in Fig. 1, for any given values of r_p and r_s, there is a maximum speedup that can occur that is approached asymptoticly, e.g. a program in which ninety percent runs in parallel (i.e. $r_p = 0.9$ and r_ss $= 0.1$) has a maximum speed up of ten times when run across a very large number of processors, compared to the entire program being run in a serial fashion.

As this demonstrates, for a program with any parallel portion, adding additional processing units should increase performance, but for each processor added the benefit of doing so diminishes, up until a point at which adding more processors will produce negligible, if any, noticeable performance increase [7]. This is all assuming the data set that is being processed is large enough to create enough work to be able to be split among all the processing units.

III. Testing Environment and Methodology

The testing method used here is intended to try to keep the environment sim ilar to what one might find on an average consumer computer, while eliminating as much variance from the system and minimizing outside influences on the results as is feasible. As such, these tests were conducted using a clean install of Microsoft Windows 10 with the latest updates. (Build number 16299.371 at time of testing) To minimize variation from run to run, any unrelated processes that are not being used by either the operating system or the program are manually terminated to avoid other programs from utilizing system resources. In addition, the computer has been disconnected from any network as an extra precautionary measure against any outside interference.

Full system specification can be found in Table I, but the core of the test is the processor being utilized, in this case an AMD Ryzen 5 1600x with a total of twelve logical processors. The processor's clock speed has been manually set to 3.89 GHz to avoid any variation from built in technologies that may try to temporarily boost performance.

TABLE I. Test System Part List

Computer System Part List	
Part Type	***Part Model***
Processor	AMD Ryzen 5 1600X
Motherbard	Asus Prime X-370 Pro
Memory	Crucial Ballistix Sport LT 16GB (2 x 8GB) DDR4-2400
Graphics	Sapphire - Radeon R9 Fury Nitro
Storage	Seagate 500GB 5400rpm SATA 2 HDD
Power	Rosewill 650W 80+ Bronze Certified ATX Power Supply
Operating System	Windows 10 Home - Build 16299.371

The program being used for the demonstration is one of the author's own creation. It is a simple n-body gravitational simulator. How this program works is to take a list of bodies, along with their mass, position, and velocity, and use this to iteratively calculate the movement of each body with respect to a central origin. This type of program is perfect for parallelization, since in this method the acceleration due to gravitational forces for each body can be calculated separately, and those forces are calculated from the summation of the gravitational force exerted on the current body by all bodies in the system.

Specifically, this program uses features built into C++11 standard, such as thread management and several other parallel processing specific features. While these are nice features to be added into the standard, it took a while to learn how to properly utilize these new tools. This difficulty was compounded by the difficulty already inherent in trying to write code to run in parallel, especially avoiding race conditions.

After creating the program, many scenarios were tested, varying the size of the problem (increasing the number of bodies in the system) and how many cores would be utilized to work on it (varying the number of threads that are being used to split the problem).

IV. Data and Analyses

Each scenario was tested multiple times, and the average time for completion was recorded. The speedup factor was then calculated for each point and graphed below in Fig. 2.

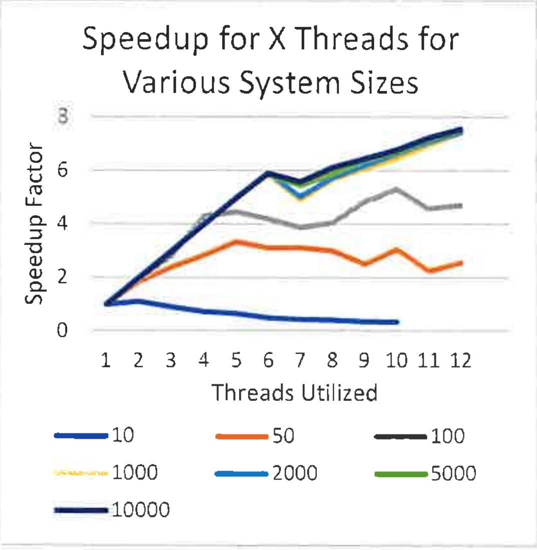

Fig. 2 Graph Displaying the Gathered Data

For the 10-body system, there is a very slight increase in speed when utilizing a second core, but each additional core causes a slowdown, increasing the time it takes to complete the same task. It is likely that this may be caused by the amount of overhead associated with parallelizing a task. This is a great example of how not everything will be able to take advantage of being split up into parallel tasks. While this program can be easily split to run in parallelized threads to work on the problem, with only ten bodies in the system there is not enough work for each thread to process to make up for the time required to create, manage, and stop each thread.

Both the 50-body and the 100-body systems start out having definite gains from being parallelized, but both seem to quickly level off. The 50-body system appears to peak at five threads giving a speedup factor of just above three times, but then slowly begins to fall, like the 10-body system, this is presumably due to the overhead of parallelizing overtaking the benefits of running in parallel in this program. Thee 100-body system is similar, but slightly more erratic. This is possibly an indicator that this system size is close to the point here the penalty for overhead is overtaken by the benefit for multithreading for programs running on less than a dozen threads.

These two body counts also share an odd spike in performance at the ten threads mark. While it is not clearly known the reasoning for this, it may be possible that it is due to being able to better split the workload evenly, therefore not requiring some threads to take on a higher workload than others. This reasoning steams from the fact that the program divides the work by bodies, meaning that if one hundred bodies are split among nine threads, eight of those will be working on eleven bodies each while the last one will have to work on twelve. The logic required to deal with this irregular boundary may be enough to have a noticeable performance hit for certain workloads.

The 1000, 2000, 5000, and 10000 body systems all follow a very similar path. Each of these systems has an almost identical and ideal speedup factor up until seven threads, at which they each take a sudden drop before continuing up, approaching similar speedup values. While it cannot be seen well here, since the larger systems have more to process within the parallel portion of the program, they have a slightly larger portion being parallelized, therefore will have a higher maximum theoretical speedup.

The slowdown present when increasing from six to seven threads was surprising at first. After a little bit of research and experimentation, it would appear that this is a byproduct of the Simultaneous Multi-Threading (SMT) technology built into the Ryzen processor. Simply put, this is a technology built into newer processors to minimize the processors' down time by having each processing core quickly alternate between processing two threads, reducing the amount of downtime the processor has. While this effectively doubles the core count of the processor (e.g. a six-core processor being able to process twelve simulations threads) the doubling is not perfect, and the overall speed is slightly diminished. This is demonstrated in Fig. 3.

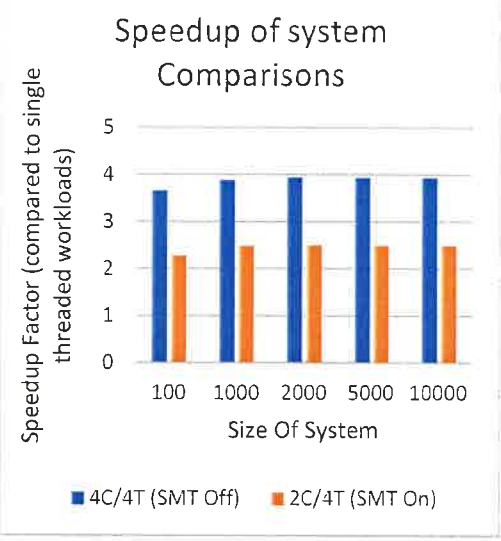

Fig. 3 Graph Depicting the Impact of SMT

For the data here, the settings of the test system's motherboard were changed, first to run as a quad-core system with SMT disabled, and then as a dual-core system with SMT enabled, effectively giving it four threads. For each of these setups, the program was run for various system sizes set to all run with four worker threads. As seen in Figure 3, the dual-core with SMT consistently preforms 35 – 38 percent worse than a quad core without SMT, even though both effectively have four processing cores.

Using the equation from Amdahl's Law, the portion of the algorithm that is running in parallel can be approximately extrapolated. The extrapolations are shown in Fig. 4 below.

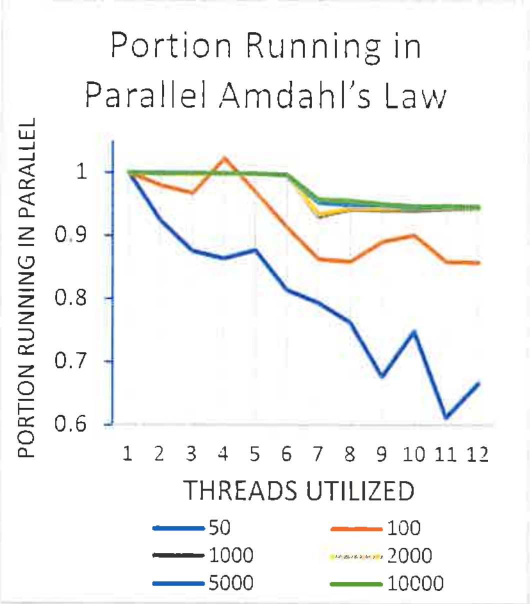

Fig. 4 Graph depicting the portion of code running in parallel according to Amdahl's Law

Looking at this graph, we can see that systems larger than 1000 bodies appear to be running about 99% of the code in parallel, up until it switches from 6 to 7 threads, where it appears to drop to about 95% of the code running in parallel. The systems with smaller body counts, namely the 50 and 100 count systems, initially have a large percentage that appears to be running in parallel, but as the thread count increases, the overhead becomes a larger part of the code overall, dramatically lowering the percentage of the code in parallel. The 10-body system is excluded from this graph since multithreaded run are slower than the single threaded run, it does not fit into the equation for Amdahl's Law.

V. Conclusions

For the scenario explored in this paper, systems such as the 10-body with very little data to prosses will not see benefit from parallelization, whereas systems that require exponentially more calculations to prosses such as the 1000-body and larger systems will see a very linear speedup, baring external factors such as SMT. Systems with sizes between these two categories, including the 50-body and 100-body systems start to see speedup factors similar to the larger systems until they reach a point at which there is not enough data to make adding more cores effective. The larger systems can be assumed to have points where this stagnation happens as well, but they are higher than can be tested with this apparatus.

While parallelizing programs on a system can have great speedup benefits, there are several drawbacks and limitations. First, the amount by which a program can be sped up is limited by Amdahl's Law, which states that any program with portions that cannot be parallelized there is a maximum amount it could be sped up by. Second, each processing core added will not be able to have as much of a performance increase as the previous core, except in specific cases. Third, from experience, the process of rewriting a program to run in a paralleled fashion may be difficult for many programmers and software engineers. Fourth, even when the problem appears to be easily be able to be parallelize each thread needs to be able to processes enough data to outweigh the overhead associated with being parallelized. Finally, there can be many factors that can have surprising impact on the performance of the code, including SMT and other quirks in the technology or data.

VI. Future Research

This is by no means the only way to parallelize workloads, there are many other forms of parallelization that can also be explored that may be able to be tested more extensively but are outside the scope of this study. These include utilizing the following:

- Processors with many more processing cores, such as the AMD EPYC or Intel Xeon Phi processors.
- General Purpose Graphical Processing Units (GPUs) that are designed for very highly parallelized workflows such as graphics rendering
- Distributed algorithm across a cluster of networked computers. This includes the recently popular blockchain technology that is the basis for cryptocurrencies.

While none of these are explored in detail in this paper, these would be great topics to research to learn more about various parallelization technologies. Both the use of GPUs and distribution across a network are planned as possible future expansions to the capabilities of this n-body simulator.

Acknowledgments

Thank you to the professors who helped me out with ideas along the way, including Prof. Sible, Prof. Sumey, Dr. Pyzdrowski, and Dr. Dieterle. I would also like to thank to all my friends who helped and supported me through this process, especially Rachael and Alaina.

References

[1] F. Peper, "The End of Moore's Law: Opportunities for Natural Computing," New Generation Computing, Osaka, 2017.

[2] The Economist Newspaper Limited, "Technology Quarterly - After Moore's law," The Economist Newspaper Limited, 2018. [Online]. Available: https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law. [Accessed 4 April 2018].

[3] M. Spencer, "The End of Moore's Law," U.S. Black Engineer & Information Technology, pp. 76-77, 2018.

[4] B. Crothers, "End of Moore's Law: It's not just about physics," CNET, 2013.

[5] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS spring joint computer conference, 1967.

[6] Daniels220, "English Wikipedia," 13 April 2008. [Online]. Available: https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg. [Accessed 15 4 2018].

[7] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," Computer, pp. 33-38, 2008.