CPU Parallel Programming

Oregon State University

CS 271 Winter 2020

Connor Bentzley

Connor Baldes

Introduction to Parallel Programming

Parallel computing is a type of computing where multiple processes are being carried out simultaneously. This can be beneficial by increasing the performance of a computer through being able to do more work in the same amount of time, taking less time to do the same amount of work, and makes some tasks more convenient to implement [1]. There are three types of parallelism in computing: Instruction Level Parallelism(ILP), Data Level Parallelism(DLP), and Thread Level Parallelism(TLP) [1]:

1. ILP is either performed by a compiler in which is called Static ILP, or by the CPU in which is Dynamic ILP [1]. ILP is when a program has a continuous stream of assembly instructions but does not execute those instructions continuously. A “pause” is implemented halting the current process until whatever initiated that pause is ready to proceed [1].
2. DLP is performed by the software engineer and is when the same instructions are executed on different parts of data [1]. For example if a program has a large set of data that needs to be looped through the data can be split into sections and run by different loops.
3. TLP is when different instructions are performed simultaneously by the computer [1].

Threads vs. Processes

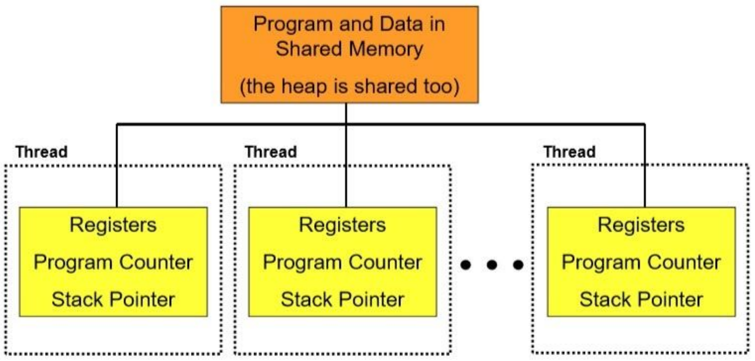


Figure 1: Visual representation of multiple threads in program [1].

Moving forward it is important to understand what the terms “process” and “thread” mean in terms of computing, and how they are related. A **process** executes a program in memory, and keeps a state(program counter, registers, and stack) [1]. **Threads** are separate independent processes, executing in a common program and sharing memory (Figure 1) [1]. Each thread has its own registers, program counter, and stack pointer [1]. But each thread also has access to the same global data in memory, because all threads are executing on some part of the same program [1]. Simply speaking a process is a program being executed, and processes divided into independent units are threads.

Multithreading vs. Multicore

As stated above a “thread” is an independent path through the program code [1]. **Multithreading** is the process of having multiple instruction streams or “threads” running simultaneously. The main benefits of Multithreading can be broken down into four key categories:

1. Responsiveness: Multithreading may allow a program to continue running even if parts of it are blocked or performing a lengthy operation which increases responsiveness to the user [3].
2. Resource Sharing: Processes can only pass resources through techniques such as message passing and shared memory, both of which must be explicitly organized by the programmer [3]. Threads however share memory with the processes they belong to which allows applications to have several threads within the same space [3].
3. Economy: Allocating memory and resources for processes is often very costly for the computer [3]. Since threads share memory it is generally much quicker to create switch threads [3].
4. Scalability: The benefits of multithreading increase as the size of the task or data increase. The more data there is the more that can be split and run concurrently making what would be a long process much faster.

These benefits, and many others that multithreading has, makes scenarios such as when: specific operations can be CPU-intensive, specific operations have higher or lower priority than other operations, performance can be gained by overlapping I/O, and accelerating a single program on multicore CPU chips ideal for the use of multithreading [1].

While multithreading is an extremely powerful tool in a programmers arsenal, it comes with its issues and must be used carefully. In order to use multithreading a programmer must make sure their code is “thread-safe” meaning that it does not keep an internal state between calls [1]. If your code does keep an internal state between calls it runs the risk that a second thread will come in and change the state and the first thread will not know it has been changed [1]. Two common issues that occur when code is not thread-safe include: **Deadlock Faults** when two threads are waiting for each other to do something [1], and **Race Condition Faults** where a condition is dependent on which thread gets to a section of code first [1].

Multicore is the term used to describe a computer processor with two or more separate processing units which are called cores [4]. Each one of these cores can read and execute program instructions same as if the computer had multiple processors [4].

Multicore and multithreading are two seperate ideas that apply to different areas of computing. Multicore refers to a computer or processor that has more than one logical CPU core, and that can physically execute multiple instructions at the same time [5]. Multithreading refers to a program that can take advantage of a multicore computer by running on more than one core at the same time [5].

Flynn's Taxonomy & Parallel Designs

Flynn's taxonomy is a type of computer architecture that consists of four classifications (Figure 2) based on the number of continuous instruction and data streams available in the architecture [6]. The four classifications include:

1. Single-instruction, single-data (SISD) systems: Single processor machine capable of executing one instruction at a time on a single data stream [7]. Instructions are processed in a sequential manner and all data to be processed has to be stored in primary memory [7]. The speed of an SISD system is limited by the rate a computer can transfer data internally [7].

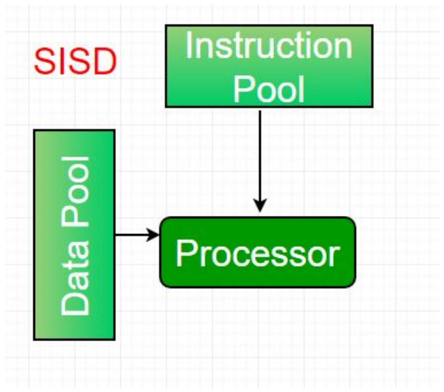


Figure 2: Visual representation of SISD system [7].

2. Single-instruction, multiple-data (SIMD) systems: Multi processor machine capable of executing the same instructions on all CPUs but operating on different data streams [7]. SIMD systems are often used for scientific computing since they involve lots of vector and matrix operations [7].

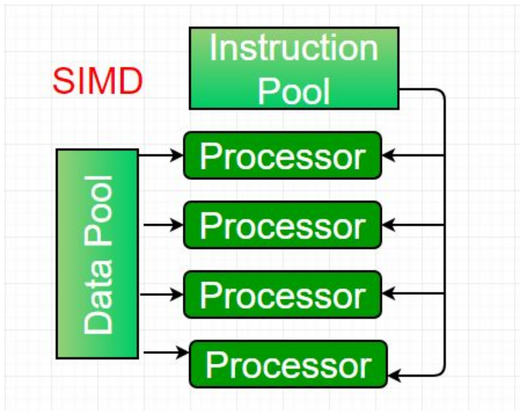


Figure 3: Visual representation of SIMD system [7].

3. Multiple-instruction, single-data (MISD) systems: Multi processor machine capable of executing different instructions on different processing elements but all of them operating on the same dataset [7].

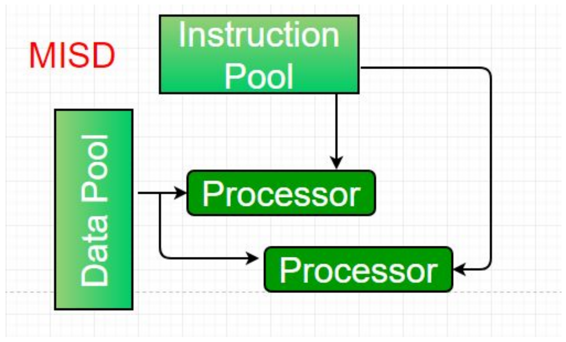


Figure 4: Visual representation of MISD system [7].

4. Multiple-instruction, multiple-data (MIMD) systems: Multi processor machine capable of executing multiple instructions on multiple data sets [7].

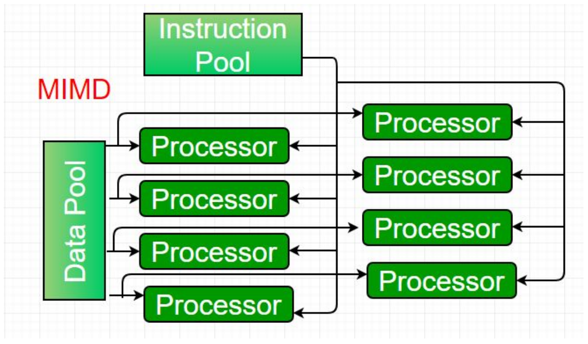


Figure 5: Visual representation of MIMD system [7].

Moore's Law

Moore's Law is a computing law that states that the number of transistors on microchips grows exponentially, while the price of microchips decreases simultaneously. The co-founder of Intel, Gordon E. Moore, inferred this in 1965 after watching manufacturing trends at Intel. He stated the number of transistors that can be packed in a given space will double every two years. While the actual time for transistors to double is closer to 18 months, Gordon E. Moore's inference has set a precedent for the computing industry since [2].

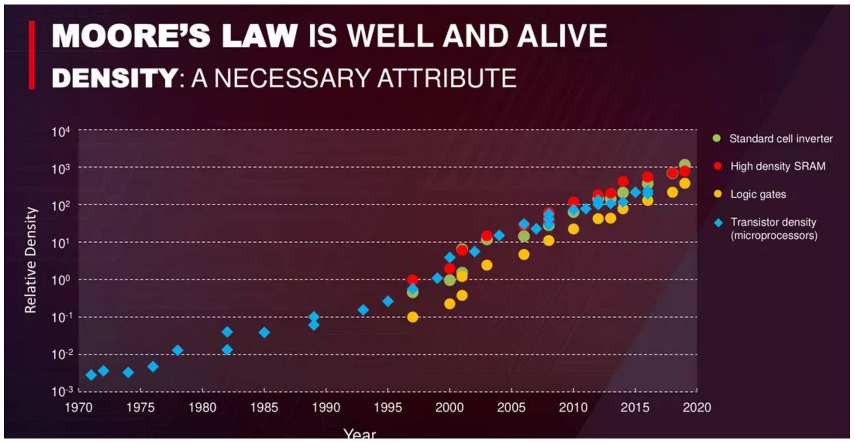


Figure 6: Transistor Density as a Function of Time [8]

Amdahl's Law & Gustafson's Observations

Amdahl's Law is a formula to quantify the speed-up of a task of fixed-size based on portions of code executable in parallel and number of parallel parts. Gene Amdahl, a computer architect, proposed this formula in 1967. The formula is as follows: $S = 1/((1-p) + p/s)$, where S is theoretical speed-up, p is the portion of code executable in parallel, and s is the number of processors [9].

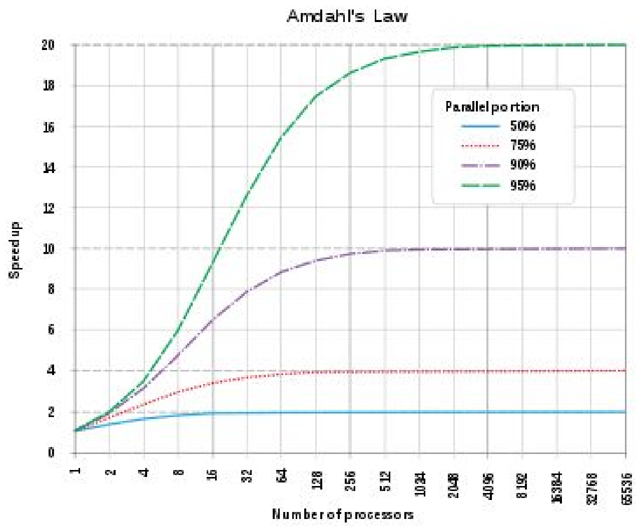


Figure 7: Amdahl's Law Graphed

One of the most common misconceptions about Amdahl's Law is that it is inferred that adding more processors equates to a larger speed-up. This is not always the case as Amdahl's Law states that speed-up is also based on proportion of code executable in parallel. As one can see in the graph, Amdahl's law follows the rule of diminishing return. This means that after a certain point, adding processors will equate to a very miniscule difference in computing time. As seen in the graph, running more code in parallel sets the point of diminishing return higher for the number of processors.

In 1988, American Computer Scientist, **John Gustafson** re-assessed Amdahl's Law in a famous paper. Reviewing Amdahl's Law, Gustafson realized that adding more processors not only increased speed-up, but also allowed computers to process larger tasks.

OpenMP

OpenMP stands for Open Multi-Processing, and is an application programming interface that allows for the support of parallel programming in shared memory. OpenMP is supported in C, C++, FORTRAN, and many other platforms. OpenMP includes a variety of options to manage which parts of the code run in parallel. It adds a level of obscurity between the user without the complexity of having to manage threads individually. In other words, OpenMP allows you to use a directive to run parts of code in parallel without the complexity of having to manage the threads. It also has an includable runtime library that can be used to set parallel programming. In this library, environment variables can also be used to define runtime parallel programming operation parameters [10].

General Format for Designating Portions of C/C++ Code to be run in parallel is as follows:

```
#pragma omp parallel private(var1, var2) shared(var3) {
        //Code here
}
```

Xeon-Phi

Xeon-Phi is an Intel product family of processors specifically optimized for running applications that are highly parallel. The Xeon-Phi family processors are designed to run in Intel's Many Integrated Core architecture, meaning they are designed to be running in conjunction with several other coprocessors. In the case of supercomputers, one supercomputer could have several hundred Xeon-Phi coprocessors. Each coprocessor has several smaller cores, with most models in the product family having over 50+ cores per unit. As the Xeon-Phi product line was designed with parallel tasks in mind, the number of cores was maximized, as well as thread count.

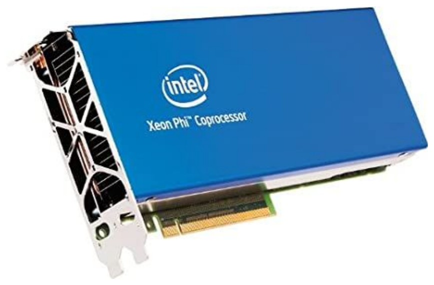
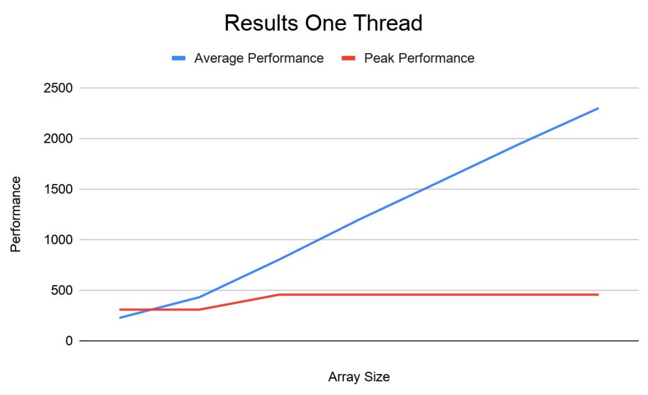


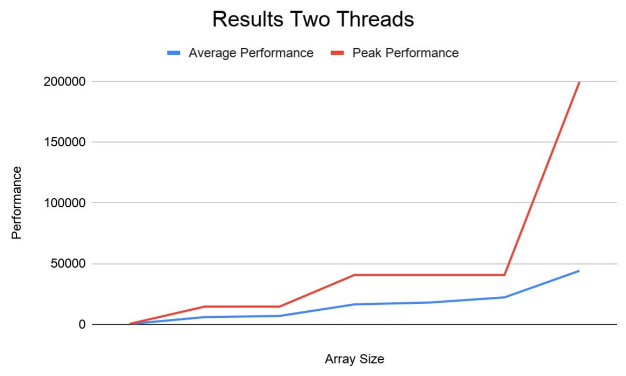
Figure 9: Xeon-Phi Coprocessor

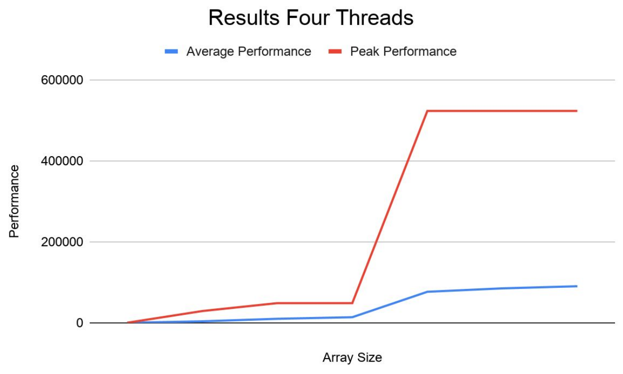
Async Programming vs. Parallel Programming

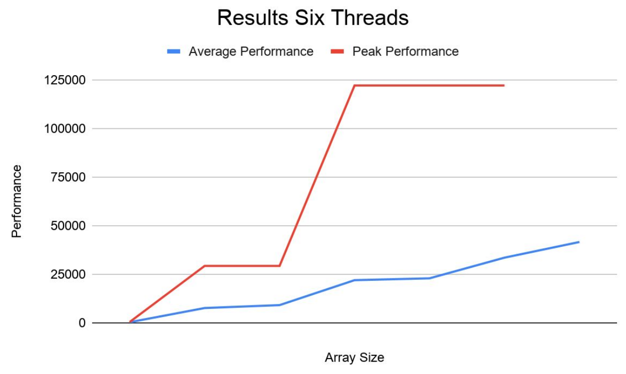
Asynchronous programming is a task management concept closely related to parallel programming. In a synchronous program each task is executed one after another. Only one task can be executed at a time. In an asynchronous program each task is executed without the main thread waiting for it to be completed. The main thread is then notified later when the task is completed. It is worth noting that the group of tasks are running asynchronously to the main thread, meaning the main thread is still free to do other tasks. However, the tasks in the assigned thread will run synchronously. Asynchronous programming is typically mentioned when threads are limited because multiple tasks are typically split off into another thread by the main thread, and the main thread is called back to the result. This is different in comparison to parallel programming, where tasks are typically designated to individual threads [11].

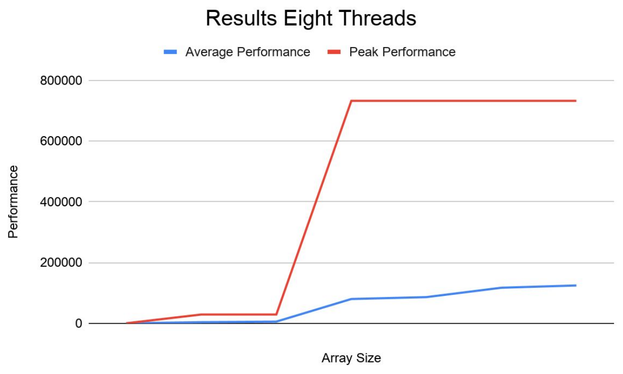
Exercise Results: Intro to Parallel Execution







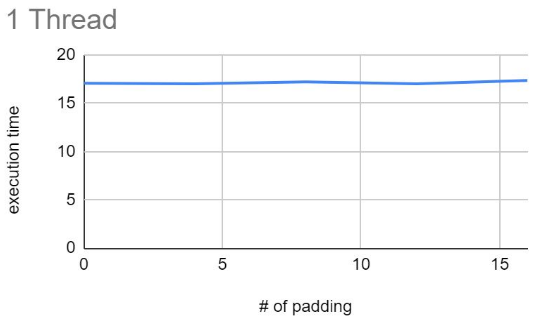


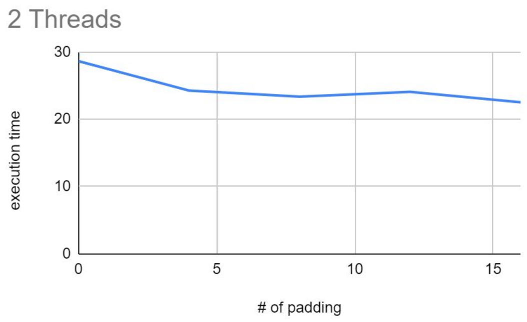


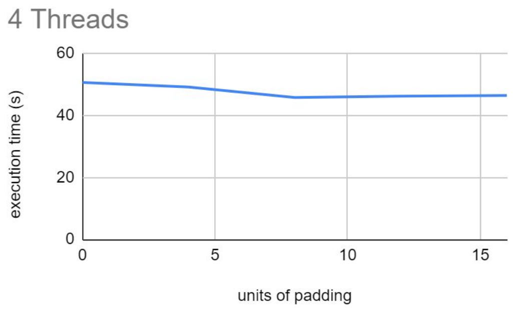
Findings

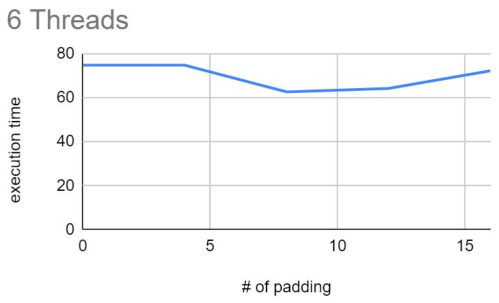
The results from the thread performance test were very interesting. The results of the test with a single thread I believe must have been an error(even though I ran the test multiple times) as while peak performance remained consistent for every data array size, average performance was greater. The remaining four tests all showed similar results: a steady incline in average performance as array size increased. Peak performance on the other hand seemed to increase at a fairly steady rate then have a drastic increase in the middle followed by a plateau. The results from threads 4-8 all showed this result with their results scaled. The sharp increase in performance I expected as that is one of the main upsides of multithreading, increased performance when performing the same operation on large sets of data. What I found slightly unexpecting was the sudden plateau as the array size reached a certain point. Average performance continued to have a steady climb but peak performance did not change. I suspect this is just because all of the processors being used reached 100% of their load capacity and remained there for the rest of the program. Another theory I have is that the sudden drop off is in relation to Amdahl's law but I'm fairly certain that because performance continued to increase as threads were added that, that is not true. Overall it was clear that as threads increased so did average and peak performance which is consistent with the ideas behind parallel programming.

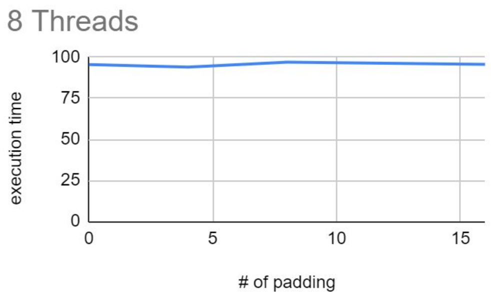
Exercise Results: Cache Memory Improvements











Based on my findings, I believe adding padding to cache lines narrows down the chance of **false cache line sharing** up to a certain point, but beyond that adding padding elongates the processing time. I found that my fastest execution times were occuring at 1 thread, however, changing the amount of cache padding had little to no noticeable effect. I believe this is because tasks had to be done sequentially, so there was little risk of multiple threads writing or reading to the same cache line. Beyond this point however, I did not necessarily see a definite pattern. There was a general decline in performance because of adding padding. I believe this is because adding padding to the cache means the retrieval times will be higher.

References

Lecture Slides:

[1] M. Bailey, *Parallel Programming: Background Information*, 08-2020.

Websites:

[2] C. Tardi, “Moore's Law Explained,” *Investopedia*, 05-Feb-2020. [Online]. Available: https://www.investopedia.com/terms/m/mooreslaw.asp. [Accessed: 09-Mar-2020].

[3] aastha98, “Benefits of Multithreading in Operating System,” *GeeksforGeeks*, 14-Aug-2019. [Online]. Available: https://www.geeksforgeeks.org/benefits-of-multithreading-in-operating-system/. [Accessed: 09-Mar-2020].

[4] Wikipedia contributors. (2020, February 18). Multi-core processor. In *Wikipedia, The Free Encyclopedia*. Retrieved 06:55, March 9, 2020, from https://en.wikipedia.org/w/index.php?title=Multi-core_processor&oldid=941465521

[5] “Multithreading and multicore differences,” *Stack Overflow*, 06-Aug-2012. [Online]. Available: https://stackoverflow.com/questions/11835046/multithreading-and-multicore-differences. [Accessed: 09-Mar-2020].

[6] Wikipedia contributors. "Flynn's taxonomy." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 18 Dec. 2019. Web. 9 Mar. 2020.

[7] Siddharth_Pandey, “Computer Architecture: Flynn's taxonomy,” *GeeksforGeeks*, 06-Jan-2020. [Online]. Available: https://www.geeksforgeeks.org/computer-architecture-flynns-taxonomy/. [Accessed: 09-Mar-2020].

[8] S. Shankland, “The company that builds Apple's iPhone chips has a rosy forecast for processor progress,” *CNET*, 21-Aug-2019. [Online]. Available: https://www.cnet.com/news/processor-progress-is-alive-and-well-tsmc-builder-of-apple-iphone-chips-says/. [Accessed: 09-Mar-2020].

[9] Astha_SinghCheck out this Author's contributed articles., Astha_Singh, and Check out this Author's contributed articles., “Computer Organization: Amdahl's law and its proof,” *GeeksforGeeks*, 01-Apr-2019. [Online]. Available: https://www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof/. [Accessed: 09-Mar-2020].

[10] B. Barney, “OpenMP Tutorial,” *OpenMP*. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/#API. [Accessed: 09-Mar-2020].

[11] *C# Parallel and Async Programming*. [Online]. Available: http://www.software-architects.com/devblog/2014/09/22/C-Parallel-and-Async-Programming. [Accessed: 09-Mar-2020].