Peri Hassanzadeh
ECE 1570 High Performance Computing
Dr. Bigrigg
November 5, 2021

# Solution Scaling and Applications

## Scaling

Solution scaling deals with how the solution time varies with the number of processors. I will now analyze the scalability of each solution and how it could potentially increase or decrease the performance based on which implementation was chosen.

Before I get into analyzing each solution and its corresponding execution time and relate that to its performance, I wanted to mention that each time is smaller when using the ssh bigrigg server compared to my local machine. Although the server is a shared machine and the times may differ depending if it is being used by other processes, each time I have tested the times of each implementation, the times were faster than that of my local machine.

Now I will start analyzing the times of each implementation, mentioning that of the server and my local machine, why I think certain implementations are faster than others, and what I could have done to possibly speed up each implementation.

The serial implementation ran at a speed of 269,864 microseconds on my local machine which is approximately 0.269864 seconds. On the server, it ran at a speed of 123,376 microseconds which is approximately 0.123376 seconds. The serial version is more than two times as fast on the server as on my local machine. In order to speed up the execution, parallel programming is introduced with data parallel first and then task parallel implementations.

The data parallel version ran at a speed of 179,443 microseconds or 0.179443 seconds on my local machine and 72,279 microseconds or .072279 seconds on the server. Meaning the data parallel version is almost 2.5 times as fast when run on the server compared to the local machine. In order to improve this implementation, I think I could've created more threads to split up the files to be worked on rather than just splitting the data in half and distributing it to two threads. I feel like the more threads present, the less time it would take for each thread to complete its tasks and therefore since they can all run at the same time, the overall execution time would be sped up. When comparing the data parallel implementation to the serial implementation, there is a much greater speed up of nearly 3.7 times on the local machine and 1.7 times on the server. This would lead me to believe that the program has strong scalability since the ability of the parallel algorithm to achieve performance gains proportional to the number of processors being used increases. When programming in parallel, the number of processors being utilized increases. This is the same for task parallel as well, but the results may differ in terms of scalability.

The task parallel version ran at a speed of  888,178 microseconds or 0.888178 seconds on my local machine and 649,240 microseconds or 0.649240 seconds on the server. When run on the server, the task parallel implementation is only 1.3 times faster than when it is run on my local machine. In terms of comparing task parallelism to data and serial parallelism, this implementation is much slower. Compared to the serial version when run on the server, the task parallel version is 5.2 times slower and when compared to the data parallel version when run on the server, the task parallel version is nearly 9 times slower. The scalability for this implementation is not very strong as more processors are being used compared to the serial and data parallel implementations, yet the execution time is much slower. I believe this version is slower compared to the other two because of the way I implemented mutexes and have busy-waits at the beginning of some of the threads causing them to sit idle for a certain time period. I should have done more research on the proper implementation and way to design a program utilizing task parallelism.

---

## Applications

Serial, data parallel, and task parallel implementations can be used for many applications and aid in speeding up the execution of many use cases. By adding additional processors in concurrent use, scalability, performance enhancement, and efficiency are all likely to increase or strengthen. When doing research for this project, I saw many implementations of thread and mutexes related to pointers and linked lists. It is a common application to apply task parallelism to this problem. Any problem that could implement a pipeline would also be able to implement task parallelism. For example, in class we talked about how Amazon confirms your order online once you submit your information for purchasing. Task parallelism is used to check if an item is in inventory and then in order to update the inventory number for a certain item, the “thread” in this case would need to wait to access the data when no one else was trying to access or change it.

Data parallelism can be used any time there is a large amount of data that needs to have the same operations on it. For example parsing resumes for job applicants and filtering out which applicants are potential candidates for an interview without having to look through every resume by hand. A program could split the resumes into equal groups and have the same operations performed on them and then at the end have the main thread create a list of all the potential candidates that would be likely to have the qualifications to receive an interview.

Serial implementations are very common in many programming examples such as programming a webpage or demonstrating a depth-first search algorithm; these types of applications run each line by line and nothing runs at the same time as each other. In other words, there is no threading, busy-waits, or mutexes used in a serial version of a program. Note that some of these applications could be changed to data parallel or task parallel in certain scenarios where scalability, efficiency, and performance metrics are important to the developer or critical to the application.