# Introducing Parallelism Using A Practical Example

In Woo Park

December 2021

ICS 632

High Performance Computing

University of Hawaii at Manoa

# Introducing Parallelism Using A Practical Example

**In Woo Park**

Department of Information and Computer Sciences, University of Hawaii at Manoa

**First complete draft:** 11 December 2021

## Abstract

The benefits of parallel computing is unmatched when working with large data sets. To provide insight on reducing sequential programming runtime, we use a simulated Brute Force Dictionary Attack and observe runtime behavior between Sequential, Parallel in OpenMP, and Parallel in MPI. The goal is to introduce students to parallelism by combining two different Computer Science topics (Cyber Security, Parallelism) and using a simple but practical example to represent when a program running in parallel is beneficial.

*Keywords:* Computer Science, Cyber Security, Parallelism, Brute Force Attack

# 1 Introduction

As the world continues to advance in technology, an increasing number of established threats attaches itself to the forefront of discussion. Alarmingly, since the start of the COVID-19 pandemic, cyber attacks such as theft, embezzlement, data hacking, destruction, etc, have increased by 600% [1], and unfortunately, mainstream companies were not spared from this statistic. In May of 2021, the computer manufacturer Acer faced a high profile ransomware attack by REvil where sensitive and confidential financial documents were leaked. In June of 2021, the popular video game development firm CDProjekt Red had their most recent video game source code stolen and placed under ransomware [2]. Even industry leading live streaming platforms such as Twitch.tv experienced a data breach that included the entire source code for the company's streaming service, unreleased confidential projects with partner details, and full transparency of content creator payouts [3].

It's important to note that cyber attacks at this caliber are also considered zero-day attacks*. This is not a valid excuse for cyber security negligence. All leading industries have the obligation to secure and detect against cyber threats in the current era as they are associated with technology. As technology evolves, not only does cyber security have to match the stature of cyber threats, but also prepare for unknown cyber threats.

However, to prepare the next generation of cyber security professionals, they need to understand the basics of cyber attacks using practical scenarios that exemplify real cyber threats. This paper targets introductory level cyber security professionals and introduces them to two heavy handed concepts that compliment each other naturally.

# 2 The Vision

The first subject in discussion is the Brute Force Algorithm. The Brute Force Algorithm is designed to solve a problem using sheer computational power and attempting every possibility [5]. The classic example used to represent this concept is the traveling salesman. Suppose a salesman needs to travel to 10 different cities and wants to determine the shortest route between each city to make the most optimal trip [6]. Using the brute force technique, the salesman would calculate every single combination of cities to determine the shortest route. The time complexity of this algorithm is $O(m \cdot n)$ where n is the number of cities and m is the number of city comparisons. Although the algorithm will successfully solve the problem, it is terribly inefficient and impractical when n grows substantially.

Suppose a salesman needs to travel to 1 million different cities and wants to determine the shortest possible route between each city. With a time complexity of $O(m \cdot n)$, this algorithm is unsuitable for today's standards of efficiency. Not only does the algorithm depend on the size of n, but also depends on the number of calculations you can process per computation. How can we improve the efficiency of this algorithm without changing the core idea? This brings us to the second subject matter of discussion, parallelization.

Parallelization is the idea of having calculations or processes running simultaneously. This is possible as computers are equipped with multi-core and multi-threading capabilities that allow the computer to simply do more than one task at a time. It is unfortunate that this subject is not

*Zero-Day-Attack: Unknown exploits or vulnerabilities in software or hardware [4]

usually taught to undergraduates as a core component of computer science (i.e. Data Structures, Algorithms, Operating Systems) but rather its own special higher level subject.

Although parallelization does deserve its own category, we should consider introducing this topic in combination with other core aspects of computer science such as cyber security. Therefore, if we can pair both a brute force attack with parallelization, we can properly visualize the benefits of parallelization and most importantly, prepare the next generation of cyber security professionals by revealing how easily attackers can brute force passwords if proper precautions are not in place.

## 3 The Problem Set

To begin with, we create an interactive scenario to introduce both topics of a brute force attack and parallelization. Suppose a company has hired you as a forensic analyst to attempt a penetration test[†] on their network. You discover that the company's intrusion prevention system is outdated and you can observe decrypted network traffic. You quickly scan through the network and find out how the company generates a master passphrase to access their software.

The master passphrase is generated using a cryptographic key[‡] and a cryptographic salt[§]. The key is created by hashing a word from a dictionary. Similarly, the salt is created by hashing a word from the same dictionary. The words are chosen at random from a dictionary of 466,550 words [13]. The sum of the key and salt creates a new passphrase that will allow access to their software. A new master passphrase is generated every hour.

Given the hashing algorithms used to generate a master passphrase, design an algorithm that will successfully find the key, salt, and master passphrase generated within one hour.

## 4 The Solution

The solution to this problem set is to use a Brute Force Dictionary Attack. Provided the dictionary, we compare every word in the dictionary with every single combination of words in order to make master passphrases until we find the right set of keys and salts. The solution loads the dictionary in dynamically allocated memory using a 2D array, then we use a nested for loop to compare array indexes or actual word strings. After the words are converted with their respective hash functions, we compare their values in order to find a matching pair. A GitHub repository is provided to observe the source code [12].

## 5 Runtime Analysis

We will break up the runtime analysis into 3 separate parts. First we will discuss the sequential runtime, then explore the time complexities between OpenMP and MPI. The values are graphed

[†]Penetration Test: An ethical hacker attempts to bypass a company's cyber security to find exploits or vulnerabilities[8].

[‡]Cryptographic Key: String of characters used within an encryption algorithm so it appears random [9].

[§]Cryptographic Salt: Additional set of random characters added to a key to increase complexity [10].

by the number of inputs denoted by n, versus the time it took to complete the program in minutes. To reduce the number of variables that could arbitrarily affect our program, we will run our tests using the University of Hawaii at Manoa, Mana Cluster [11]. The source code is tested on a local machine running a Ubuntu Virtual Seed but all runtimes are based on the Mana Cluster.

In addition, our program will be submitted as a $SBATCH[¶] job. The conditions we requested include 4 nodes as we strictly want to observe sequential and parallel improvement and MPI implementations aren't as useful with 1 node. We also decided to use 8 cores as most next-generation CPUs have 8 cores and it'll be interesting to see how our program behaves on the (relative) higher end of thread count. Our last condition is that our program runs with 4GB of memory. In realistic scenarios, most modern machines default to 8GBs of memory or higher. In our case, our dictionary file is only 6MBs in size. We should have more than enough memory to load the dictionary into memory.

### 5.1 Non-Parallel Brute Force Dictionary Attack (Sequential)

The sequential version of our Brute Force Dictionary Attack depends on a nested loop.

**Algorithm 1**

```
1: function SEQUENTIAL
2:     ...
3:     for i = 0 → n                                  ▷ 466,552 total words
4:         for j = 0 → n
5:             ...
6:             master_passphrase = key + salt
7:             UNLOCK(master_passphrase)
```

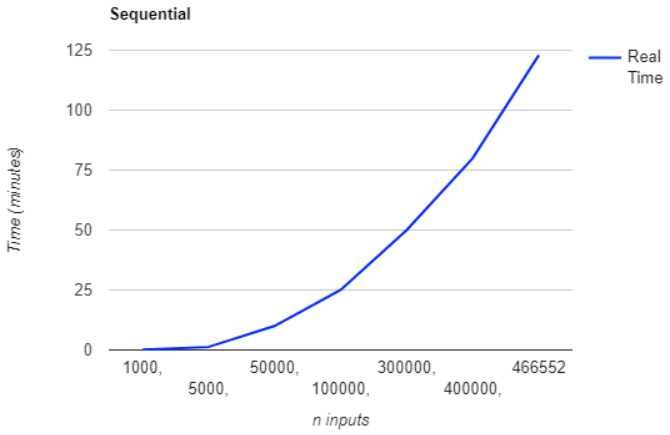


Figure 1: Sequential Runtime

Both loops will traverse $n$ iterations in the worst case where $n$ is 466,552, the last index of our 2D array. Therefore, we should expect a runtime of $O(n^2)$.

[¶]SBATCH: Submit a batch script to Slurm.

Our assumptions are validated by observing Figure 1. The line represents the sequential runtime of our algorithm in real time. The biggest take away from the sequential program is that when $n = 300,000$, the program has already taken 50 minutes. Therefore, we are motivated to find a parallel solution to reduce the time complexity.

### 5.2 Parallel Brute Force Dictionary Attack (OpenMP)

We begin by using the OpenMP library to run our nested loop in parallel. We look at two different OpenMP implementations where one focuses on parallelizing the outer loop, and one that focuses on parallelizing the inner loop. The reason for using both implementations is to observe the behaviors in runtimes.

For the outer loop, we use a simple OpenMP[‖] call that will parallelize the amount of work it takes to compare a range of words against the other 466,552 words:

**Algorithm 2**

```
1: function OpenMP_Outer
2:     ...
3:     #pragma omp parallel for
4:     for i = 0 → n                                  ▷ 466,552 total words
5:         for j = 0 → n
6:             ...
7:             master_passphrase = key + salt
8:             UNLOCK(master_passphrase)
```

The behavior of our program is seen in Figure 2 where:

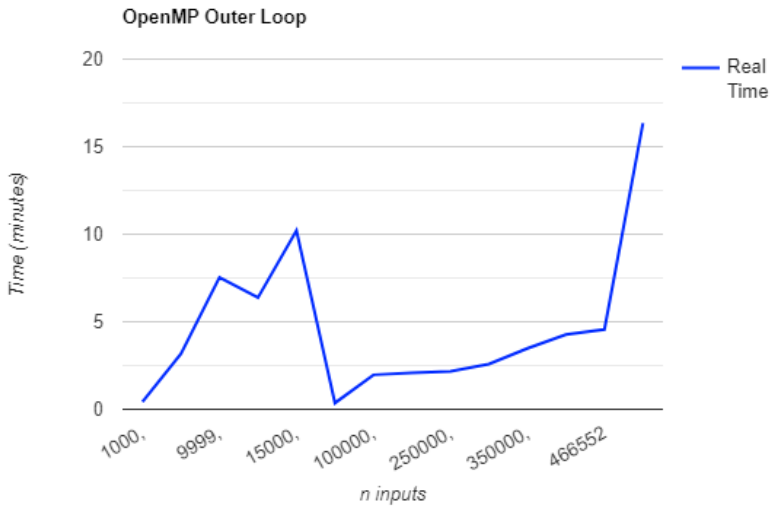


Figure 2: Runtime of Parallel Outer Loop

[‖]OpenMP: An API that supports multi-platform shared-memory multiprocessing programming in C and C++.

The runtime is a bit odd. We see several outliers that indicate weird behaviors in our program. Namely, in between 1,000 and 15,000 inputs, we see an expected increase of $O(n^2)$ into a sudden decline in runtime. When we run the program again to find a word between the 100,000 and 400,000 range, we see a steady runtime that almost looks linear. Finally, at the worst case where the program needs to find the last word in the dictionary in combination with itself, we see a sudden spike in runtime.

The threads should be dividing up the work by splicing where they access the dictionary array. Areas in the runtime where it seems to be "faster" (i.e. $n = 1,000, n = 100,000, n = 400,000$), indicate areas where a new thread has taken in a new range of the array to do work on.

After our initial runtime experiment with these values, we checked values that roughly surround the outlier areas and we reproduced the same runtime.

Although we have these outliers, we've still met the requirement of finding a matching pair within the allotted time of 1 hour by looking at the runtime of the worst case in Figure 2.

For the inner loop, we also use an OpenMP call that will parallelize the amount of work it takes to compare one word against the other 466,552 words:

**Algorithm 3**

```
1: function OpenMP_Inner
2:     ...
3:     for i = 0 → n                                  ▷ 466,552 total words
4:         #pragma omp parallel for
5:         for j = 0 → n
6:             ...
7:             master_passphrase = key + salt
8:             UNLOCK(master_passphrase)
```

Having the inner loop parallelized returns expected behavior shown in Figure 4.

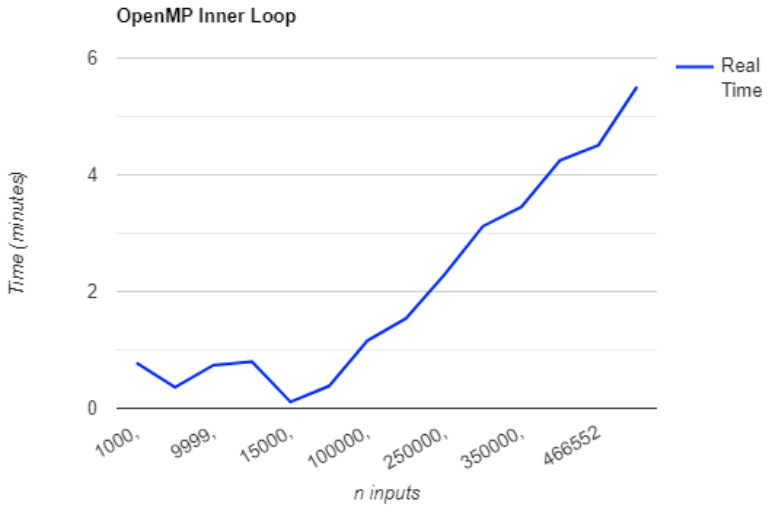


Figure 3: Runtime of Parallel Inner Loop

Figure 3 follows a runtime of $O(n^2)$ that almost seems linear. This is because when we parallelize the inner loop, each thread takes a range of words that need to be calculated for the current index of the outer loop. Instead of having each thread start on a new index range, all threads are sharing the same index but not sharing the range of words that need to be calculated.

This explains why the runtime for the worst case takes the longest time as all threads are working on the current index at the same time.

However, if you compare both Figure 2 and Figure 3, Figure 3 has an objectively lower **maximum** runtime of less than 6 minutes whilst Figure 2 has a **maximum** runtime of less than 17 minutes.

For our OpenMpP implementation on our Brute Force Dictionary Attack program, it is safe to assume that having the inner loop parallelized is faster than having the outer loop parallelized. Either implementation is valid as we have found the matching pair of master passphrases, keys, and salts in under 1 hour.

### 5.3 Parallel Brute Force Dictionary Attack (MPI)

The MPI implementation is similar to the OpenMP implementation, in that, we want processes to work on a certain range of the dictionary. For instance, if we specified 2 processes, the MPI implementation would split the dictionary in half so the first process could attempt to find the master passphrase in the first half of the dictionary and the second process would attempt to find the master passphrase in the second half of the dictionary.

**Algorithm 4**

```
1:  function MPI
2:      ...
3:      MPI_Bcast(&randomSeed, 1, MPI_LONG, ROOT, MPI_COMM_WORLD)
4:      ...
5:      MPI_Barrier(MPI_COMM_WORLD);
6:      ...
7:      for i = iBegin → iEnd                                   ▷ 466,552 / #ofprocesses
8:          for j = jBegin → jEnd
9:              ...
10:             master_passphrase = key + salt
11:             UNLOCK(master_passphrase)
```

The MPI implementation will search through a certain section of the entire range of the dictionary.

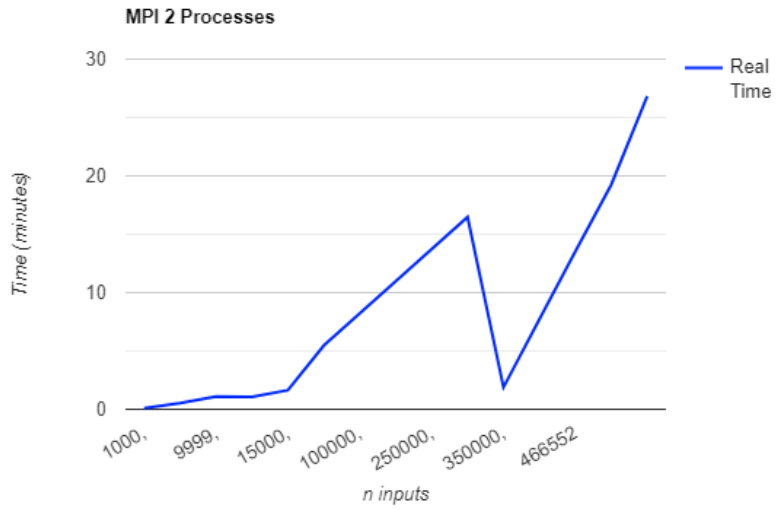


Figure 4: Parallel MPI with 2 Processes

Figure 4 shows the runtime of the MPI implementation when we specify the number of processes equal to 2 processes ($np = 2$). You can roughly see where MPI decides to split the dictionary based on the number of processes by the sharp decline in runtime around 350,000 inputs.

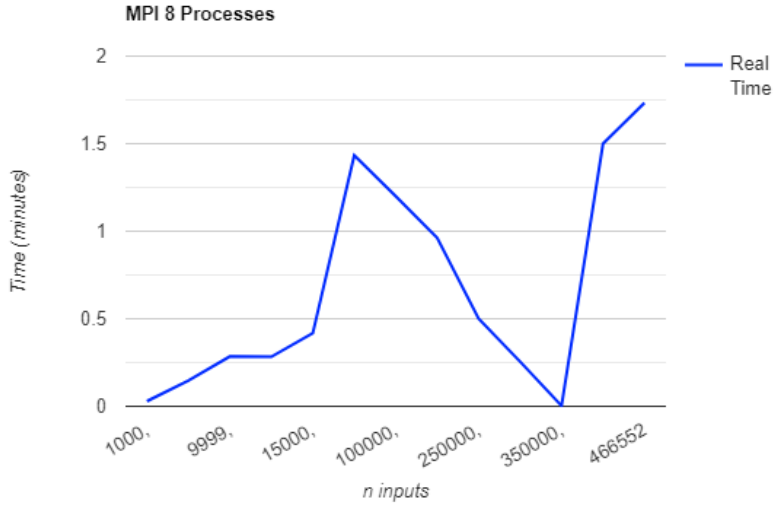


Figure 5: Parallel MPI with 8 Processes

Figure 5 shows the runtime of the MPI implementation when we specify the number of processes equal to 8 processes ($np = 8$). Interestingly, the **maximum** runtime of the program at the worst case is less than 2 minutes. This is the expected behavior as we are splitting the dictionary into 8 equal parts and all processes would do the same amount of work.

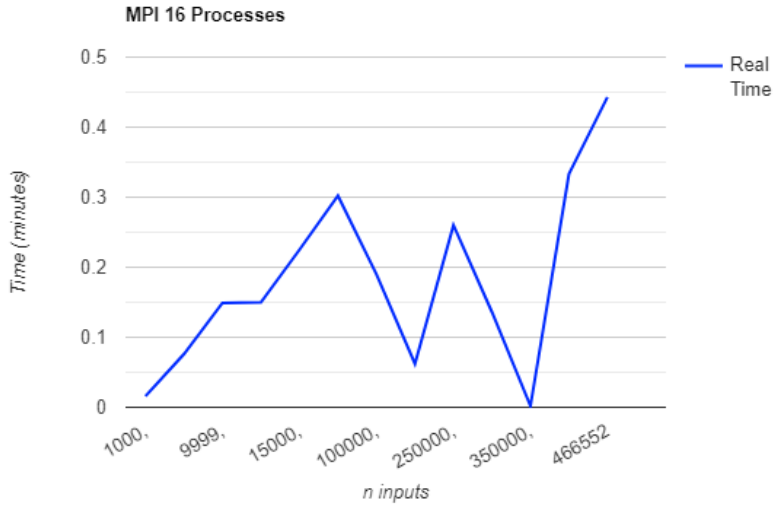


Figure 6: Parallel MPI with 16 Processes

Figure 6 shows the runtime of the MPI implementation when we specify the number of processes equal to 16 processes ($np = 16$). Interestingly, the **maximum** runtime of the program at the worst case is less than 0.5 minutes (30 seconds). This is also the expected behavior as we are splitting the dictionary into 16 equal parts and all processes would do the same amount of work.

The MPI implementation is successful in that if we used the **minimum** amount of processes to create a parallel program ($np = 2$), we still manage to complete the dictionary attack within 1 hour.

### 5.4 Outliers and Challenges

Although the main goal of running a Brute Force Dictionary Attack is completed, it is still interesting to view the runtimes of our parallel implementations.

An odd behavior is seen in the MPI 2, 8, and 16 processes implementation where if $n = 350,000$, to find the master passphrase is almost instant. Perhaps with how MPI divides the work between each processes, a process got, "lucky" and started near the 350,000 index of the dictionary. We've attempted specific runtime tests around this value and reproduced the same runtimes. If we had more time, we would attempt runtime calculations again but be more strict with our ranges.

Similarly, the OpenMP Outer Loop implementation also shares an outlier where at 100,000 inputs, the runtime is almost instant. It could mean a thread was, "lucky" on their dictionary range index.

In addition, although the dictionary has a total of 466,550 words, when we load the dictionary file into our data structure, the maximum size of our data structure is 466,552. We've attempted to find the reason for this size increase when the data structure uses dynamic memory allocation based on the size of the dictionary, but we could not find a solution to this problem. Perhaps somewhere along the dictionary there could have been an empty *newline* character, however we

double checked dictionary inputs and confirmed that all inputs were valid. Luckily, having the dictionary size increased by 2 did not affect our implementations.

Whatever the case, our implementations have an adequate Parallel Time as our primary goal is to visualize the benefits of parallelization using a Brute Force Dictionary Attack as a practical example.

# 6 Conclusion

Ultimately, our research is primarily focused on merging to two core computer science topics: cyber security and parallelization. Instead of keeping parallelization in its own course, we should consider adopting a method of implementation where parallelization can be introduced to students by merging course topics. Our research used a dictionary brute force attack that could potentially be used as an assignment students can solve by themselves. The OpenMP implementation requires one line of code to turn the sequential version into a parallel one. Having the sequential program embarrassingly parallel allows students to understand quickly that $O(n^2)$ isn't ideal when it takes 50 minutes at 300,000 inputs. Motivating students to find a parallel solution on their own.

Students who successfully complete this assignment will now have an additional tool to help their programs reach optimal runtimes without having to take a high performance computing course. On the other hand, students who do end up taking a high performace computing course are more prepared as they are not novices in the topic. We hope this is a motivator to find other computer science topics that can introduce parallelization by merging topics of their own.

## References


[1] "2021 Must-Know Cyber Attack Statistics and Trends." Embroker, 10 Dec. 2021, https://www.embroker.com/blog/cyber-attack-statistics/.

[2] "Brute Force Algorithm: A Quick Glance of Brute Force Algorithm." EDUCBA, 8 Mar. 2021, https://www.educba.com/brute-force-algorithm/.

[3] Dan AriasR&D Content EngineerHowdy!"Adding Salt to Hashing: A Better Way to Store Passwords." Auth0, 25 Feb. 2021, https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/.

[4] freeCodeCamp.org. "Brute Force Algorithms Explained." FreeCodeCamp.org, FreeCodeCamp.org, 28 Apr. 2021, https://www.freecodecamp.org/news/brute-force-algorithms-explained/.

[5] "High Performance Computing Services (HPC) " Hawai'i Data Science Institute." Hawai'i Data Science Institute, 19 Nov. 2021, https://datascience.hawaii.edu/hpc/.

[6] Security, Contrast. "What Is Penetration Testing: Definition Methods." What Is Penetration Testing — Definition Methods, https://tinyurl.com/4ca26wet.

[7] Touro College. "The 10 Biggest Ransomware Attacks of 2021." Touro College Illinois, Touro College, 12 Nov. 2021, https://illinois.touro.edu/news/the-10-biggest-ransomware-attacks-of-2021.php.

[8] Warren, Tom. "Twitch Confirms It Was Hacked after Its Source Code and Secrets Leak Out." The Verge, The Verge, 6 Oct. 2021, https://www.theverge.com/2021/10/6/22712365/twitch-data-leak-breach-security-confirmation-comments.

[9] What Is a Cryptographic Key? — Keys and SSL ... - Cloudflare. https://www.cloudflare.com/learning/ssl /what-is-a-cryptographic-key/.

[10] "What Is a Zero-Day Exploit?" FireEye, https://www.fireeye.com/current-threats/what-is-a-zero-day-exploit.html.

[11] "What Is Parallelization?" What Is Parallelization?, 26 Apr. 2017, https://tinyurl.com/2p9dkdrh.

[12] Park, In Woo. "InwooCS/ICS632_HPC_INTRO." GitHub, https://github.com/inwooCS/ICS632_HPC_INTRO.

[13] Dwyl. "Dwyl/English-Words: A Text File Containing 479k English Words for All Your Dictionary/Word-Based Projects E.g: Auto-Completion / Autosuggestion." GitHub, https://github.com/dwyl/english-words.