Brute Force threading with the CPU

Data Structures & Algorithms Presentation 2

Michael J Kleinman

Student ID : 1601618

Hypothesis

- Threading a brute force algorithm over the CPU will be more effective than running it in serial

Background

Brute force – Quick run down

- User enters a password (between 1 – 5 chars).
- That password is hashed and saved.
- An attempt is created from the char array.
- The attempt is then hashed & compared with the password hash.
- Program stops when the password is found.

Brute force – Character array

```
const char alpha[64]
{
    'A', 'B', 'C', 'D', 'E', 'F', 'G','H', 'I', 'J', 'K', 'L', 'M', 'N','O', 'P', 'Q', 'R', 'S', 'T', 'U','V', 'W', 'X', 'Y', 'Z',
    'a', 'b', 'c', 'd', 'e', 'f', 'g','h', 'i', 'j', 'k', 'l', 'm', 'n','o', 'p', 'q', 'r', 's', 't', 'u','v', 'w', 'x', 'y', 'z',
    '0','1','2','3','4','5','6','7','8','9','.',' '
};
```

- Attempt is created from the character array above
- Total of 64 chars includes ' ' & '.'.
- Creates each attempt hash with the hash of the password.

AAAAA
AAAAB
AAAAC
AAAAD
……….
AAABA
AAABB
AAABC
……….

9999A
9999B
9999C
9999D

Hashing Algorithm

- sha256 (http://create.stephan-brumme.com/hash-library/)
- Chosen over MD5 as its more secure
  (sha256 has not been cracked yet).
- Used to hash the password & the attempt created by the password cracker.
- Increases over all security of the program.

CPU Threading

Threading

- Program will use as many CPU cores as is available to it.
- Designed to utilise hardware concurrency to ensure efficient computation.
- Always runs at least 2 threads :

    - The first thread always runs the listener function.

    - The other threads are there to handle the tasks in the farm.

Primary work pattern: Farm

- Main adds the limits for each part of the character array to the que.
- Once a thread completes a task, it takes the next available task in the thread
- Dynamic threading means the program will allocate task accordingly & all tasks will be executed.

Task 1	Task 2	Task 3	Task 4

Task 1 : 0-16

Task 2: 16-32

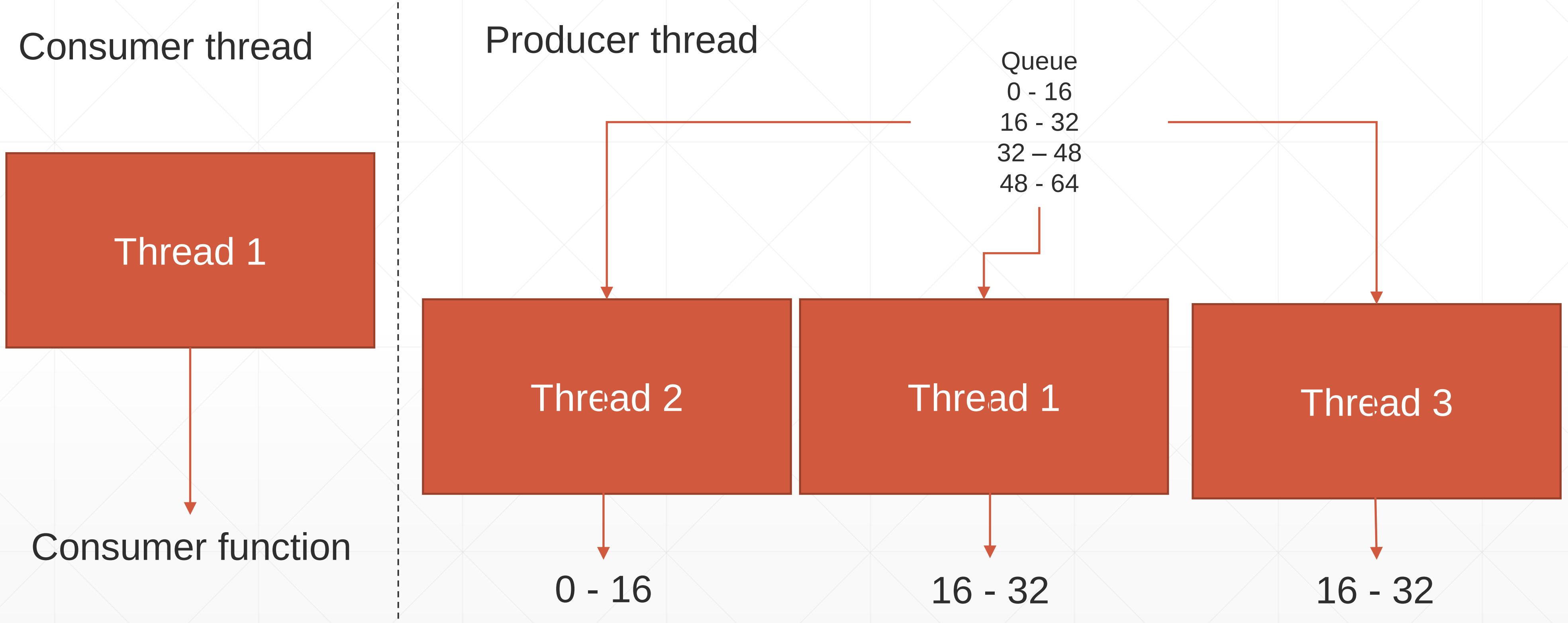
Task 3: 32-48

Task 4: 48-64

```
while (a <= 64)
{
    f.add_task(new attack(pword, attempt, a, b, &mainContainer));

    a = a + 16;
    b = a + 16;                    //splits the work into sections
}
```

Threading Diagram example



Secondary work pattern: Producer -> consumer

- The producer is the check() function & the consumer is the endall() function:
  - **endAll()** acts as a consumer as it waits for the check function to produce a true bool variable. which point the consumer thread will set the global variable "finished “.
  - **Check()** acts as a producer by supplying the bool variable once a hash has been matched with the password hash.
- In terms of task decomposition, The listener thread is dependent on the producer thread.
- The listener thread acts as a consumer function, when a true bool variable is passed to it.

Resource management

Sharing resources with the mutex

- Protects the variable "tries"
- Tires calculates how many attempts to find the password.
- Drastically slows down the program
- Necessary for the tries variable to be accurate.
- Otherwise race conditions occur.

```
if (found == true)
{
    return;
}
else
{
    attempt = string() + alpha[Ch1] + alpha[Ch2] + alpha[Ch3] + alpha[Ch4] + alpha[Ch5];
    //cout << attempt << endl;
    if (attempt == pword)
    {
        found = true;
        cout << "\n===============PASSWORD FOUND===============\n" << attempt << endl;
        return;
    }
    unique_lock<mutex> lock(tries_mutex);
    tries++;
```

Sharing resources with the mutex

- Mutex is also used on the farm
- Protects the queue
- If 2 threads try and access the same task at the same time it creates a race condition.
- The mutex prevents this.
- Slows down the program as it creates a bottle neck

```
while (finished == false || !myq.empty())
{
    Task* t = nullptr;

    qc.lock();

    if (!myq.empty())
    {
        t = myq.front();
        myq.pop();
    }
    qc.unlock();

    if (t != nullptr)
    {
        t->run();
        delete t;
    }
}
```

Performance evaluation

Consumer thread – endAll()

- **Listener function** is set up in an independent **thread,** this is started before the farm is run
- Waits for the password to be found by using a **condition variable** that locks the function & waits
- Waits for the **producer thread**
- The listener function waits until the password has been found.

```
void listener::endAll()     //ends all the threads running if the password is found
{
    std::unique_lock<std::mutex> lck(signalContainer->endThreads_mutex);
    while (!signalContainer->endThreads) signalContainer->endThreads_cv.wait(lck);
    //while loop in place to ensure consumer waits for producer

    finished = true;
}
```

Producer thread – Check() function

- **Producer thread** compares the hashes of the attempt & the password.
- Sends a bool variable to signal the password has been found.
- **Consumer thread** reads the bool. if true, it ends all the other threads running in the program.

```cpp
void attack::check(std::string attempt)
{
    attempt = attemptHash(attempt);

    if (attempt == pword)       //compares the hashes (pword Vs attempt)
    {
        std::cout <<"Tries : " << tries;

        //sends data to the listener
        {
            std::unique_lock<std::mutex> lck(signalContainer->endThreads_mutex);      //unlock mutex
            signalContainer->endThreads = true;                                      //sents end threads to true
            signalContainer->endThreads_cv.notify_one();                             //notifies endAll
        }

    }
    else
    {
        std::unique_lock<std::mutex> lock(tries_mutex);
        tries++;
    }
}
```

Signaling between threads

- Received by **listener function**
- Once it receives :
  "endThreads = true"
  it sets a global variable to true.
- Farm only ends once the que is empty OR **Global variable** is set to true.

```cpp
void listener::endAll()     //ends all the threads running if the password is found
{
    std::unique_lock<std::mutex> lck(signalContainer->endThreads_mutex);
    while (!signalContainer->endThreads) signalContainer->endThreads_cv.wait(lck);
    //while loop in place to ensure consumer waits for producer

    finished = true;
}
```

```cpp
void Farm::run()
{
    vector <thread> myt;
    int n = thread::hardware_concurrency();

    //  thread consumer(attackObjEnd.endAll);
    for (int i = 0; i < n; i++)
    {
        myt.emplace_back([&]
        {
            while (finished == false || !myq.empty())
            {
```

Testing

The Set-Up: Test environment

- To ensure the results are fair and valid, all testing done in the same test environment.
- The same computer (lab computer 4506), same amount of applications running (Visual studio).
- Different versions of the algorithm will be given the same test data.
- Limiting outside influence as much as possible means that the algorithms can utilize all the necessary resources.
- Prevents invalid results due to having less processing power to work with.

The Set-Up: Testing Method

- Password used for testing is “brute”.
- Will work as a control for testing the overall results
- Allows static version of brute force attack Vs the parallel version to be viewed fairly.
- Used **4 Threads** in the farm.
- **Measuring how long it takes to find the password**

Averages

- Median was used instead of a mean as the median doesn't rely on specific data distribution.
- Give a more accurate example of the algorithm's day to day run time.
- Balances out background discrepancies as they can often skew the results of calculating the mean (the outliers).

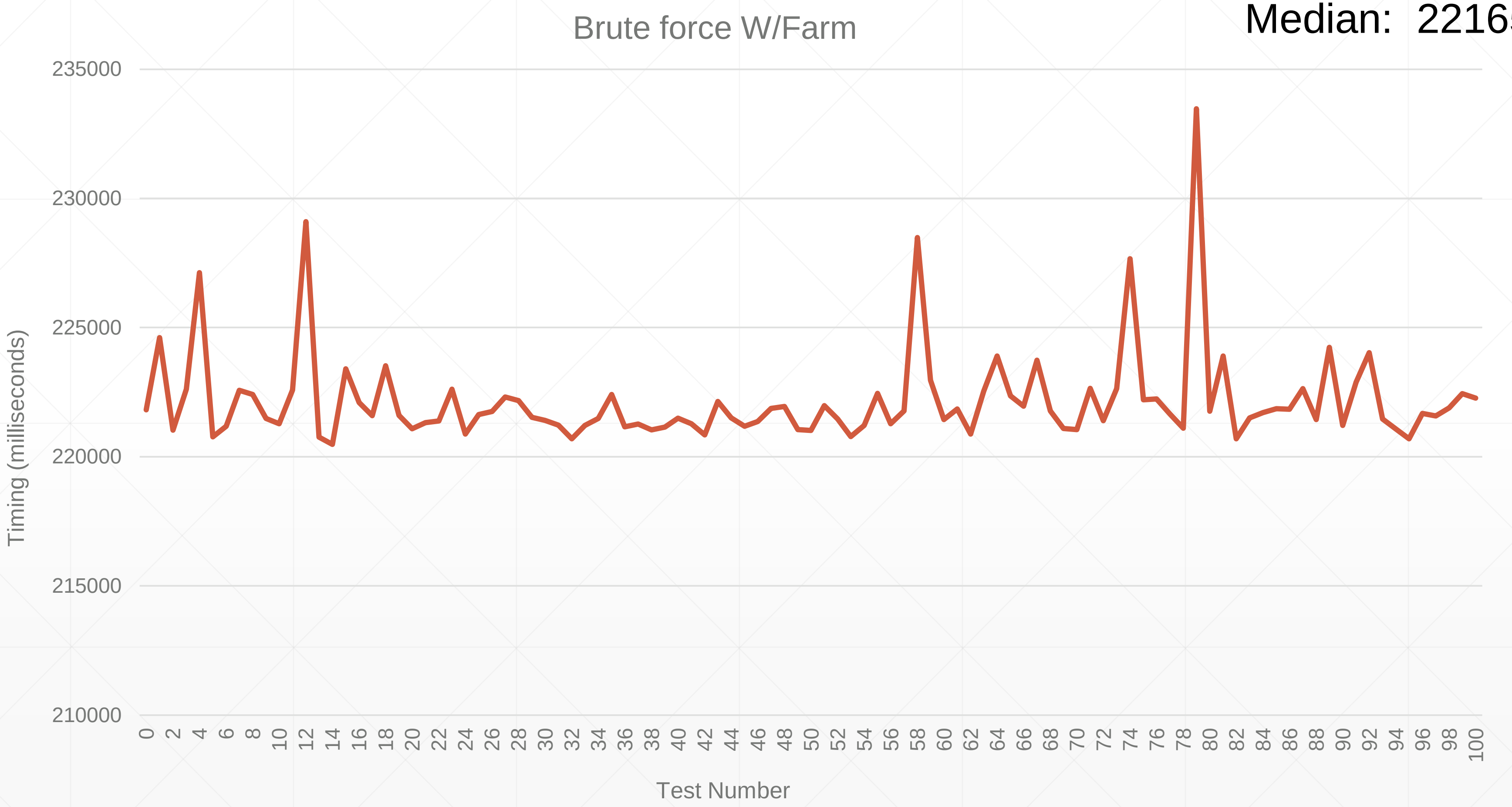
Median

~~Mean~~

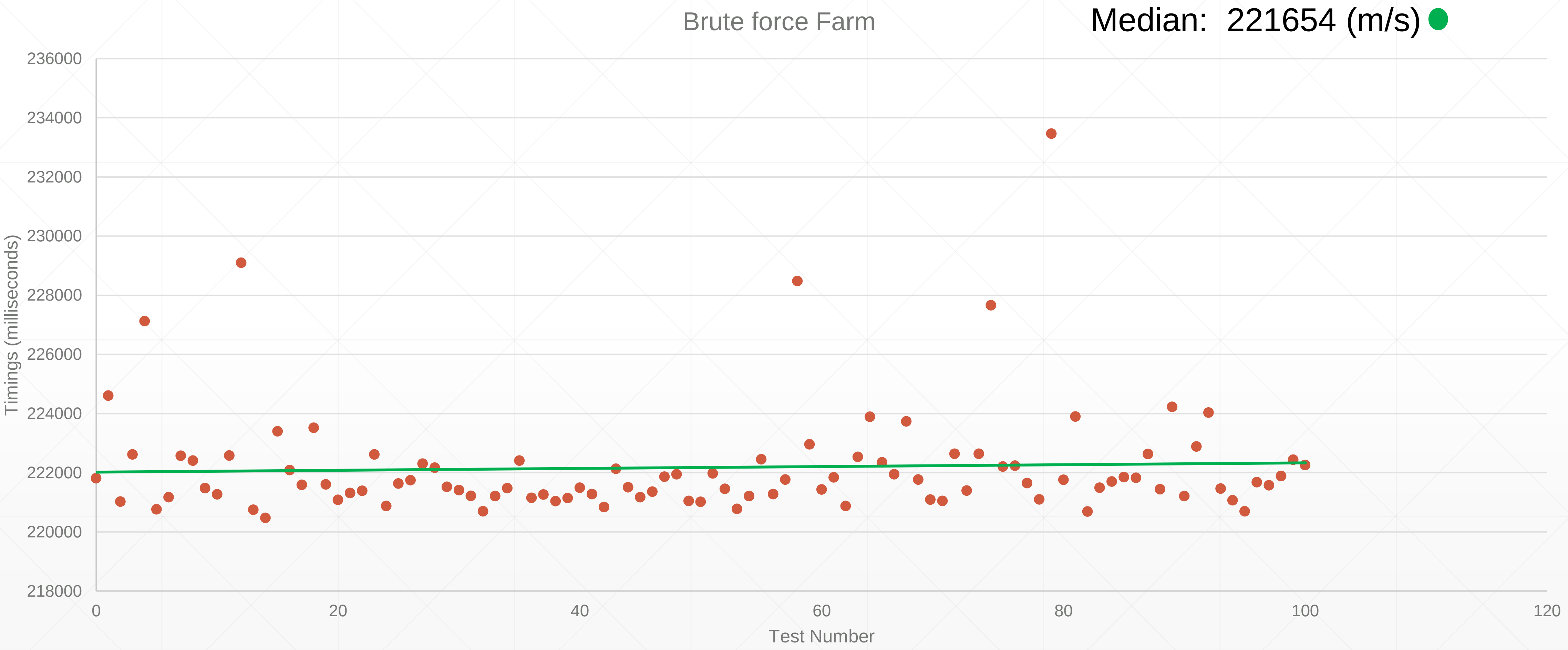
Farm Vs Serial

Farm - Results

Median: 221654 (m/s)

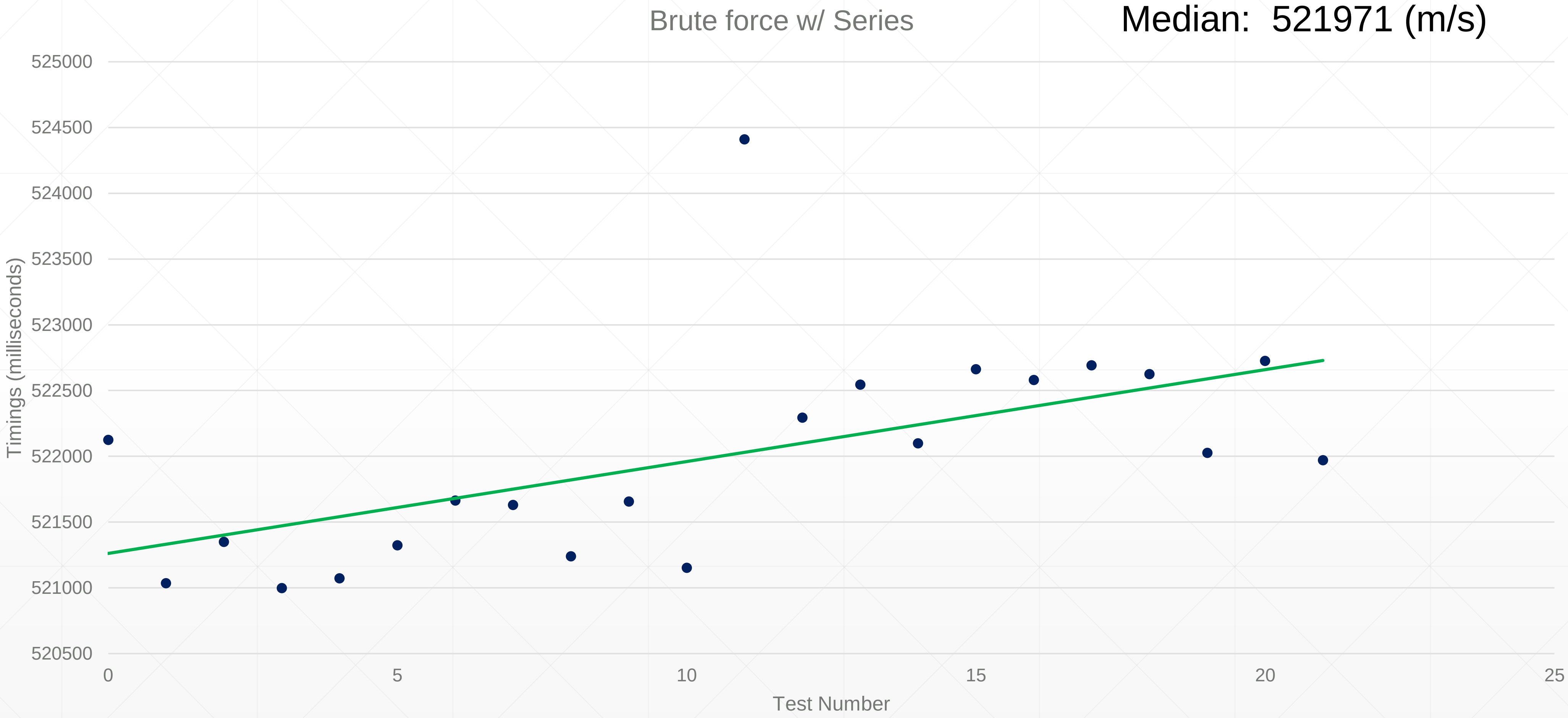


Farm - Results

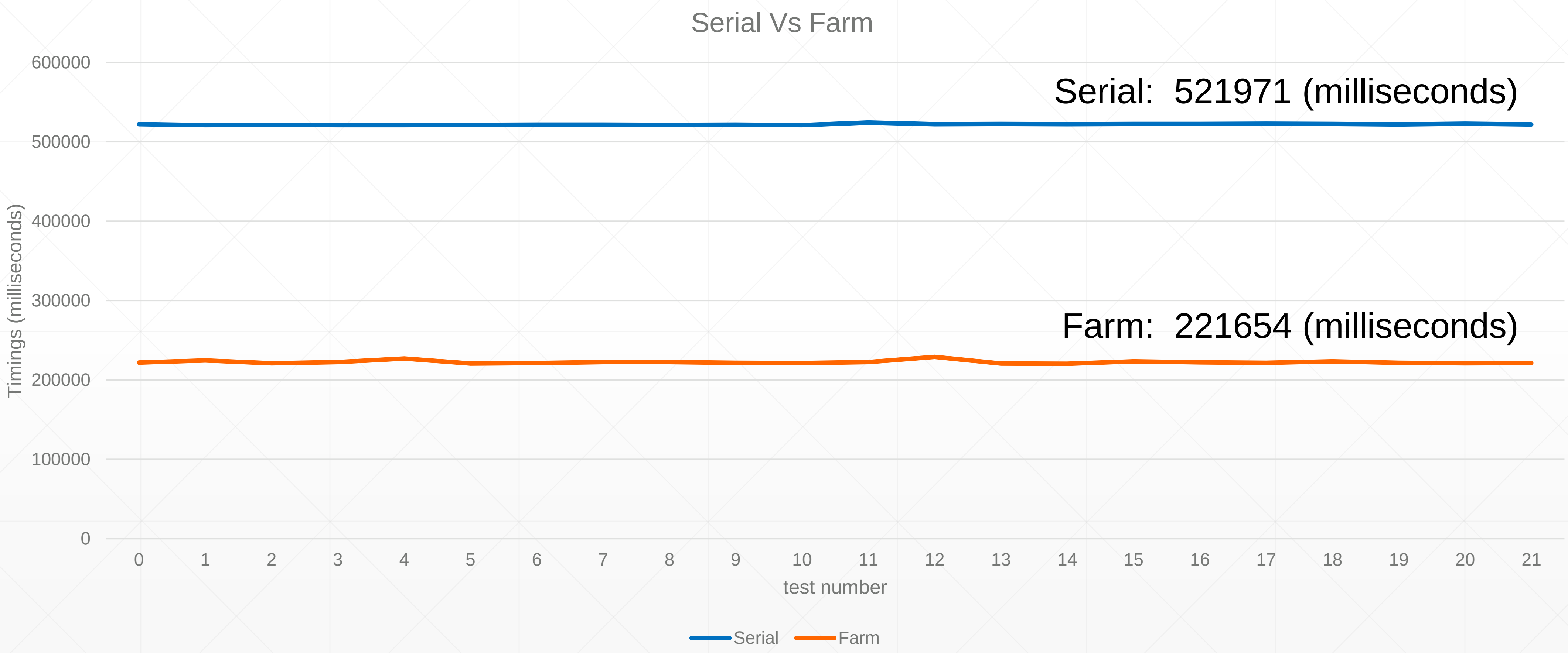


Serial - Results

Median: 521971 (m/s)

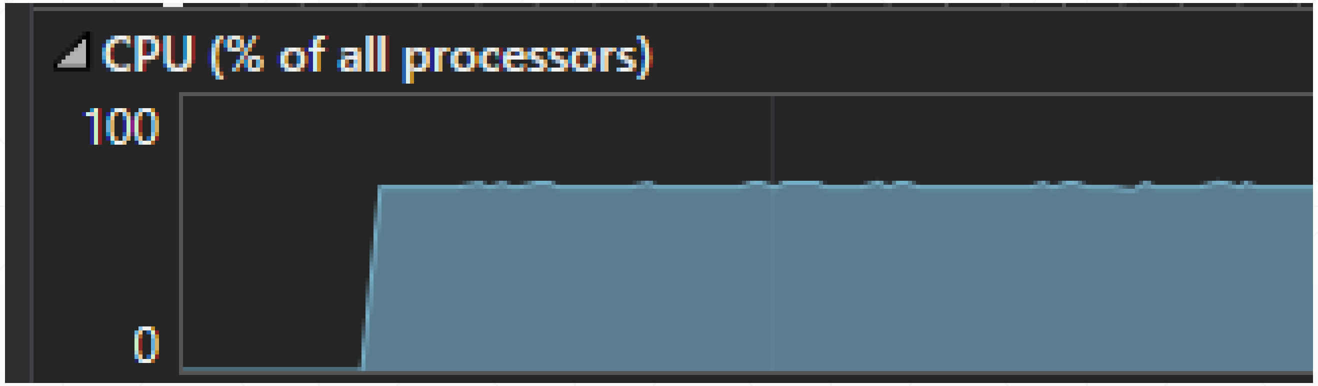


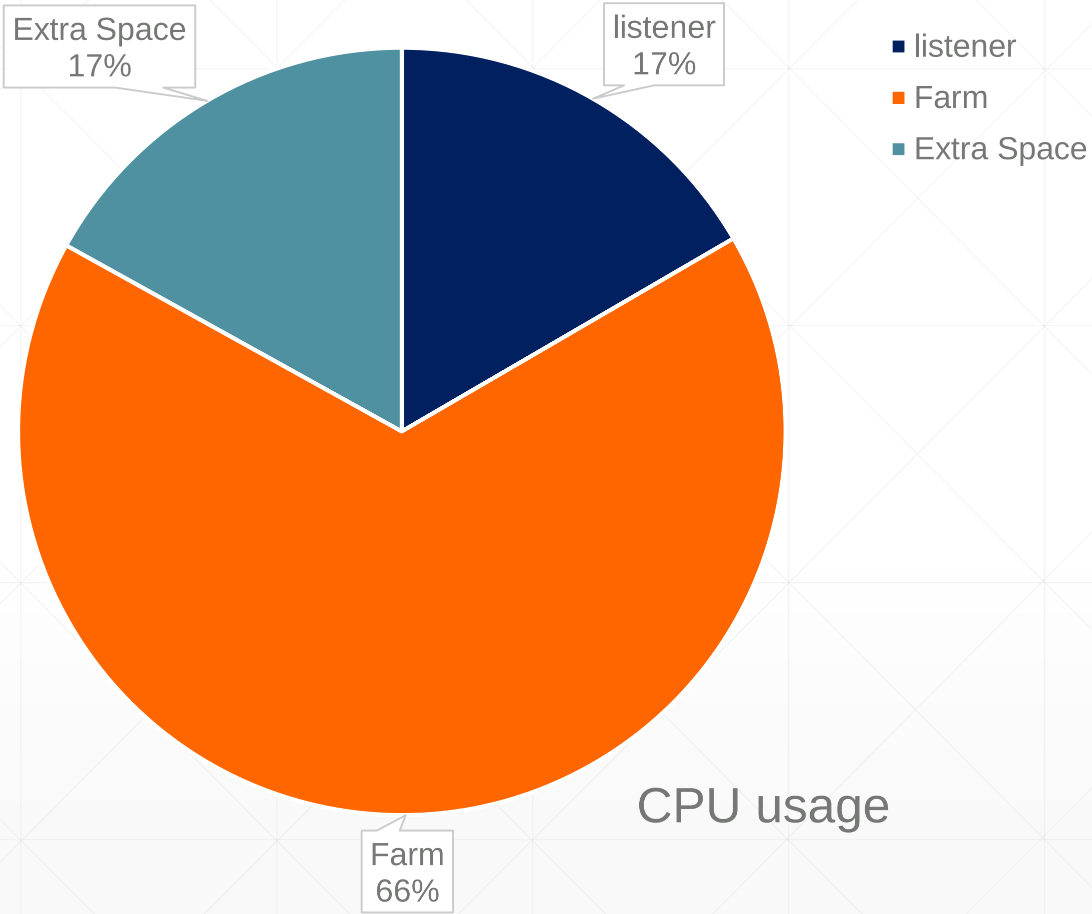
Farm Vs Serial - Results



Farm profiler

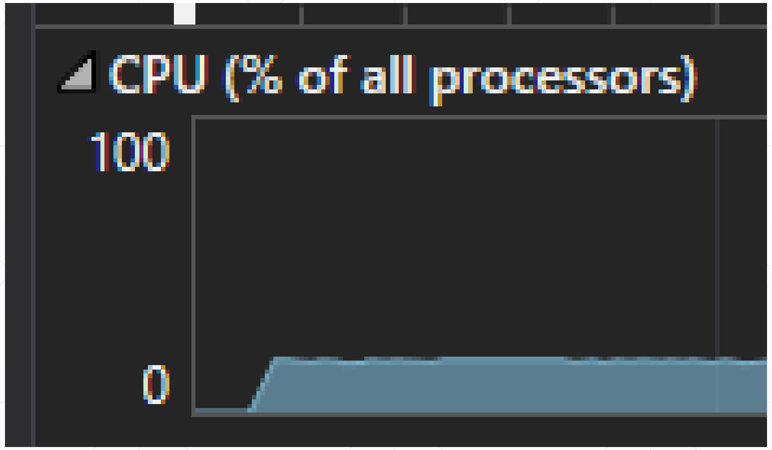
- Profiled on a different system (6 CPUs) as the lab profiler was inaccessible.

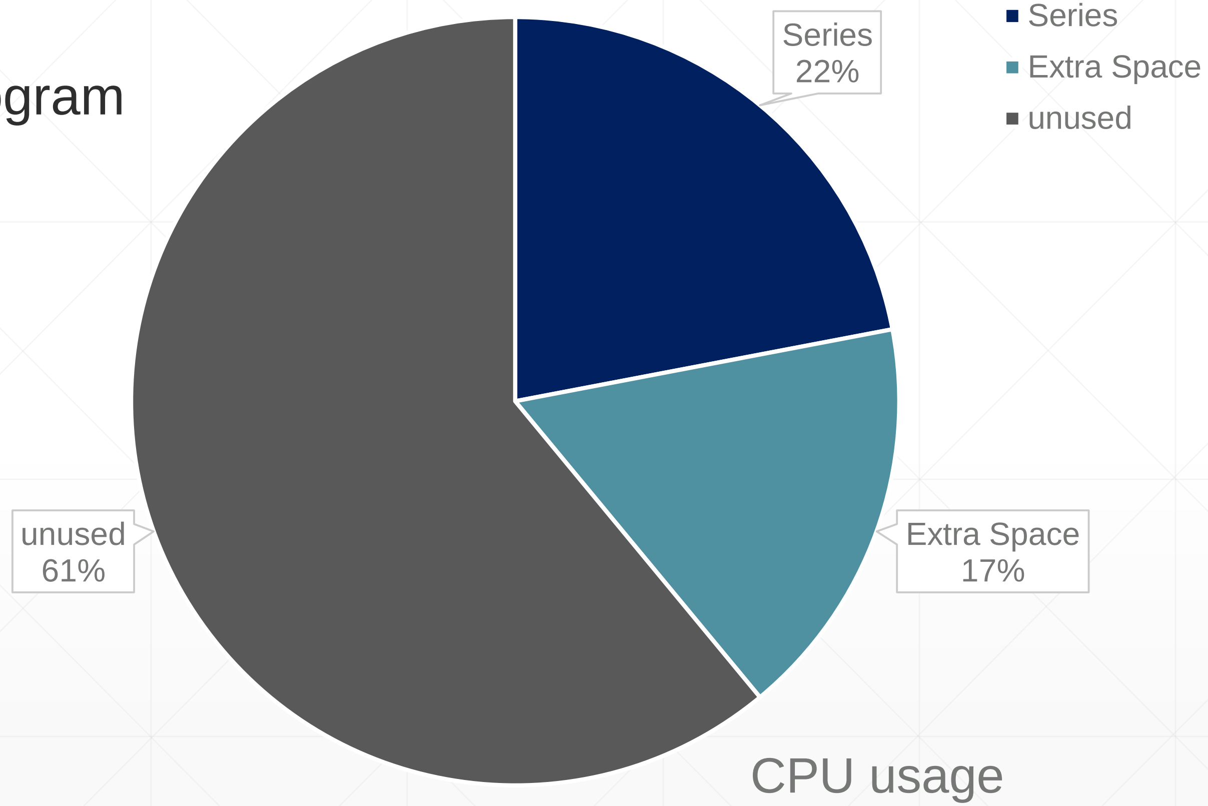




Serial profiler

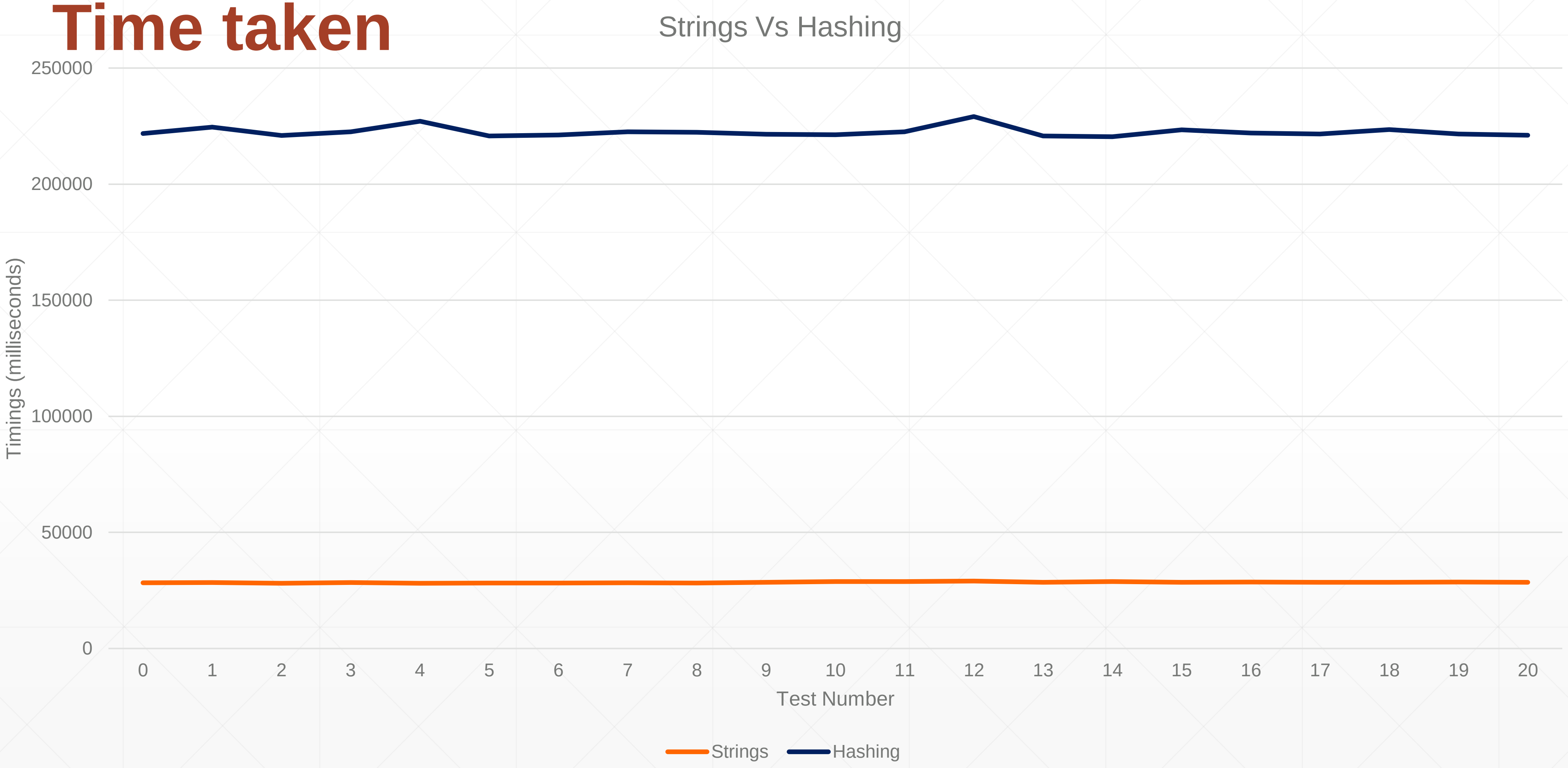
- Shows that the serial version of the program doesn’t make use of the CPU cores





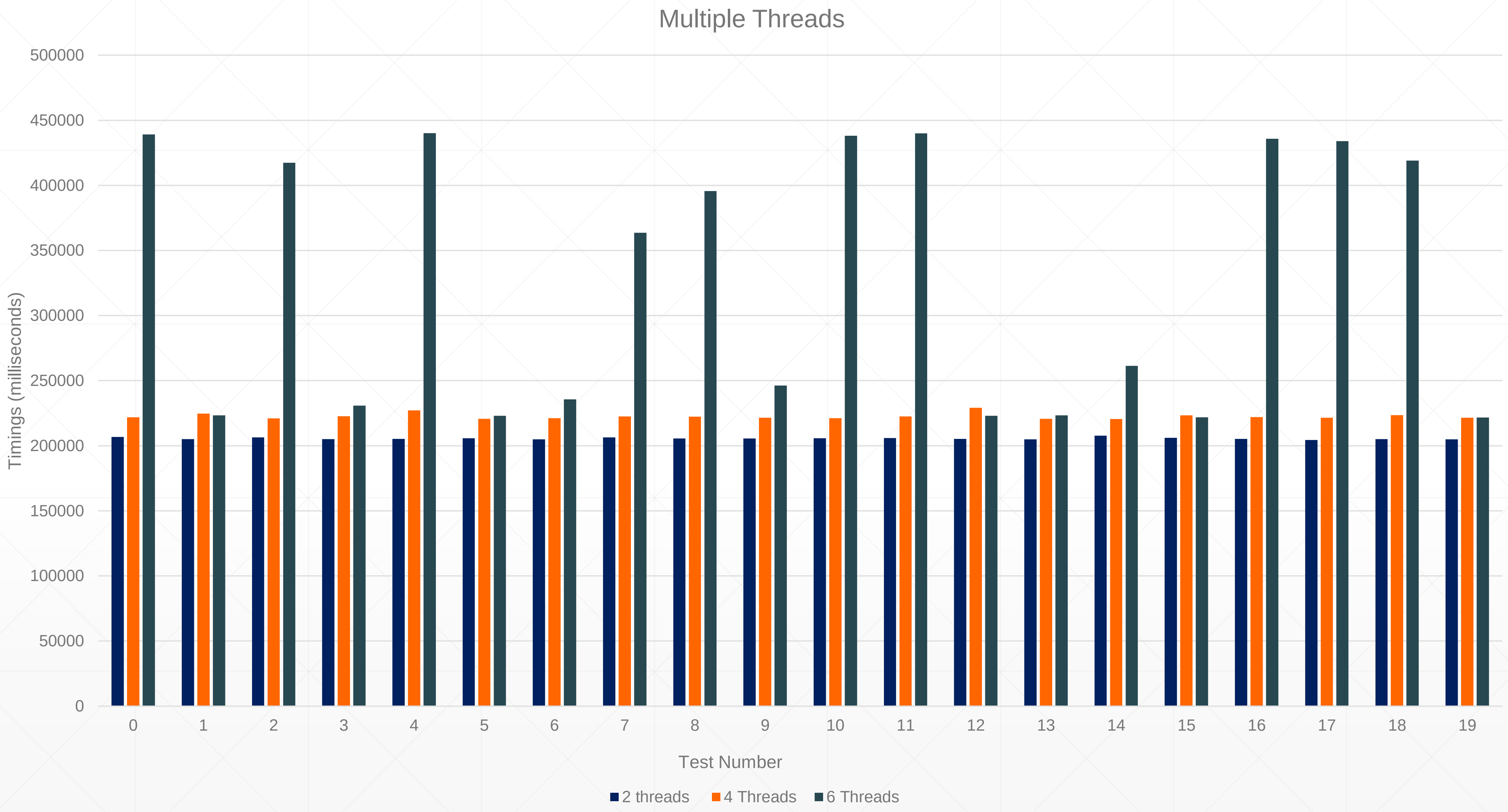
Hash Vs Strings

Time taken

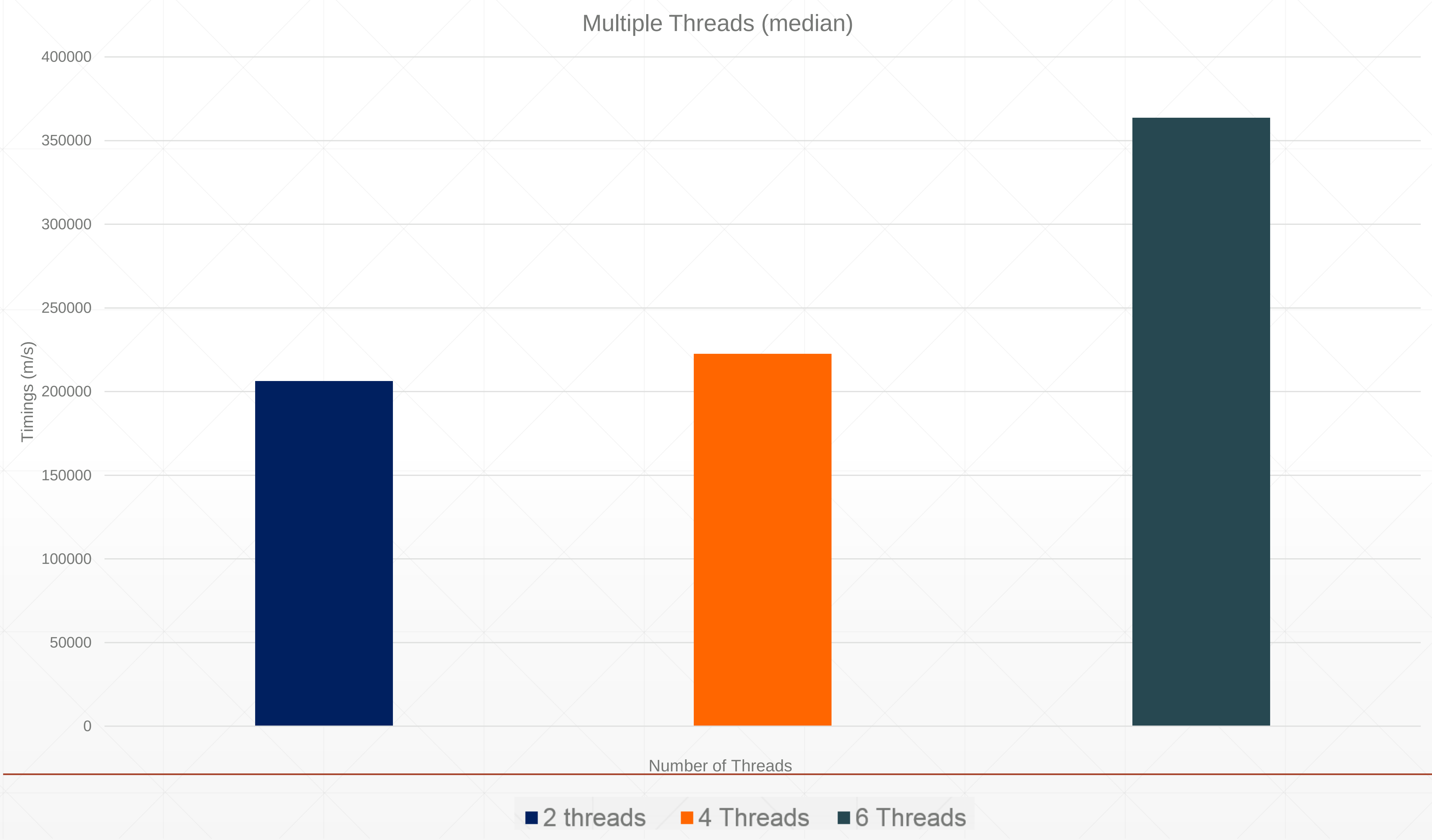


Strings Vs Hashing
250000
200000
150000
100000
50000
0
Timings (milliseconds)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Test Number
Strings
Hashing

Variable Threads



Multiple Threads
Timings (milliseconds)
500000
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Test Number
2 threads
4 Threads
6 Threads


Multiple Threads (median)
400000
350000
300000
250000
200000
150000
100000
50000
0
Timings (m/s)
Number of Threads
2 threads
4 Threads
6 Threads

Variable thread reasoning

- The lab computers only have 4 cores.
- Means the 6 thread version of the program is open and closing the other two threads over and over again.
- Cant accommodate the other 2 threads, hence its slower.

Performance Evaluation

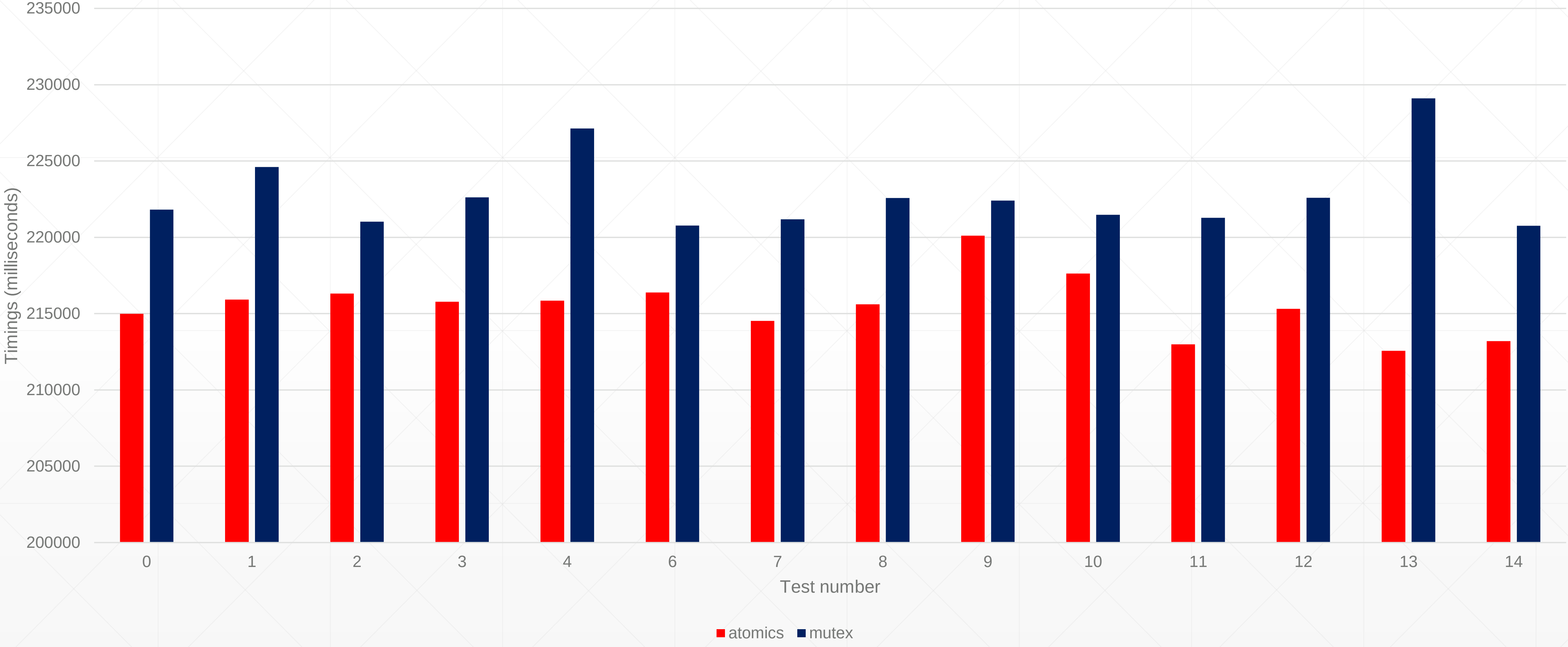
The critical section

- The tries variable is a critical section of the program.
- If 2 threads try and increment it at the same time the overall result will be in correct as one of the increments will not have registered.
- A synchronisation object, a Mutex, is used to avoid this problem and protect the tries variable
- However this creates a deliberate bottleneck in the program as the threads must wait to access the variable, less time for threads to do other work.
- No cases of deadlock as there is a global order of mutexes in place. (unlocks, uses tries, then locks)

Mutex Vs Atomics

- Comparing the two resource sharing options under the same test conditions
- Allows for an accurate representation of which one is more efficient when used in the brute force algorithm.
- Atomics are inherently atomic so there is no need for mutex locking.
- However it only works for one variable, blends perfectly for the tries counter.

Mutex Vs Atomics



In Conclusion

- Running the program in a parallel farm is more efficient and effective than running it in serial. (when using 4 cores its more efficient).
- The atomic variable should be used over the mutex when keeping track of the tries variable.
- Future work :
  - comparing hashing algorithms (SHA3 Vs SHA256).
  - Run further tests on machines with more cores.

The End ?