Multithreaded Programming (CS330 Project)

For: Dr. Heyfa Ammar

By: Tala Alhazmi
ID: 220410073

1. Introduction

Multithreaded programming refers to the execution of multiple threads within a program simultaneously, thus allowing for concurrent operations and potentially speeding up the execution time of a program. Multithreaded programming allows for parallel processing, where each thread of execution is independent of order of instructions.

During this experiment, I will be implementing multithreaded programming to sum integers in an array while measuring the runtime using the Java programming language on NetBeans.

One common use case for multithreaded programming is when performing computationally intensive tasks, such as, in the context of this experiment, summing integers in an array. By dividing the array into smaller sub-arrays and summing each sub-array in parallel, we can accurately observe the runtime of each array summation.

Optimal configuration depends on the hardware provided and the scope of the task.

2. Design

By modifying the original code, the program will add the squares of the arrays (rather than summing the individual elements). Sum(a), sum2(a), and sum3(a, #) are three methods that were developed/modified to address this issue.

The sum(a) method squares each element of the array before adding it to a running total using iterations within a for loop. This approach uses a single thread to carry out the calculation sequentially.

The sum2(a) method is a parallel variation that computes the sum using two threads. It generates two separate "summers" class, each of which affects a different area of the array(sub-divided). The threads are initiated, and after they are both finished, the results are combined. This method is run on a system with two or more processors, it is quicker than the single-threaded version.

A parallel version of the sum2(a) is the sum3(a, #) method, which could employ any quantity of threads (4, 6, or 8 threads) to compute the sum. The method produces a Summer

object for each section of the array after determining how long each segment of the array will be handled by each thread. The Summer items are begun and started as distinct and different threads. When they have all finished, the results are pooled and summed together. When used on a computer with many processors, this approach is quicker than the single-threaded version by utilizing every processor.

The Runnable interface is implemented by the assistance class known as Summer. It calculates the sum of the squares of the elements between the minimum and maximum indexes using an array, a minimum index, and a maximum index as input.

The getSum() method can be used to retrieve a private variable where the outcome is stored. The sum2(a) and sum3(a, #) methods use this class to calculate the sum of the squares of the array's members. The modified code offers three modified and very effective methods for computing the sum of squares of array elements, each of which offers a different degree of parallelization for use on multi-core or
multi-processor devices.

3. Implementation

The sum()method has been changed to total the components' squares rather than the individual elements. Three other ways to sum the square of the items are included in the modified code.

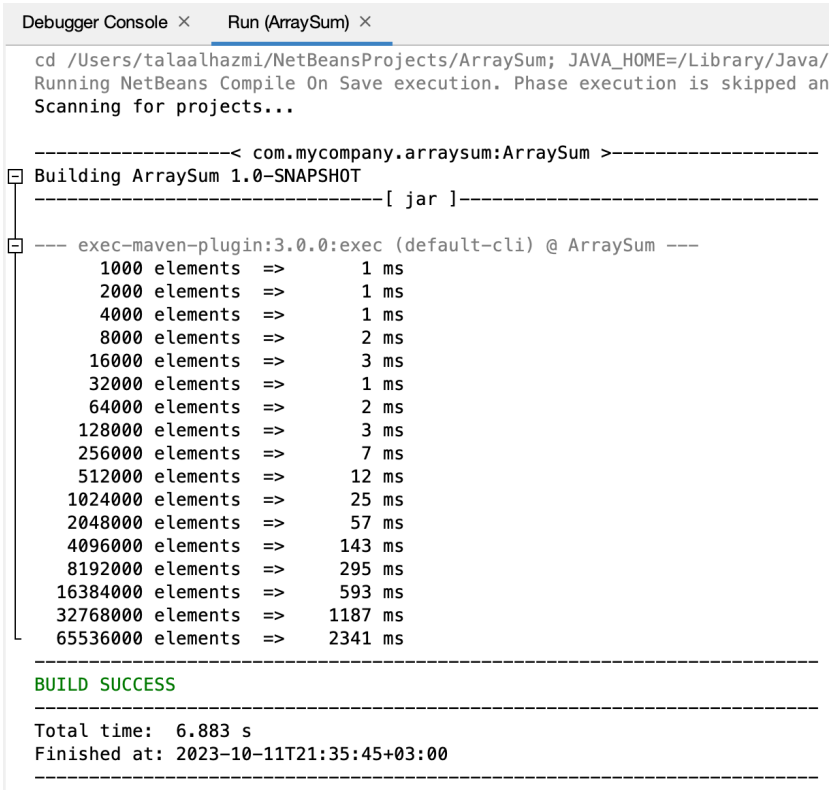
The firsttechnique is the sum(a) method, which computes the sum of squares of each element in the array concurrently.

The second technique is the sum2(a) approach, a parallel variation thatadds up the squares of elements using two threads. The results of the two threads are combined using this procedure, which creates two "Summers" to run as separate threads.

The sum3(a, #)method, which has a parallel variant as well but supports any number of threads (4, 6 or 8 threads in this case), is the third technique. This technique creates a "Summer" for each sub-array and divides the array into several sub-arrays. Each summer computes the sum of squares of the elements in its sub-array and operates on a separate thread. The result of each Summer is then added to determine the total of all sub-arrays.

The provided code also has a helper method called sumRange() that calculates the sum of squares of items in a particular range of the array. The sum2(a) and sum3(a, #) methods use the class Summer (which is included in the code) to generate summers that execute on distinct threads. The Summer class computes the sum of squares of items in a particular range of the array and implements the runnable interface.

4. Results

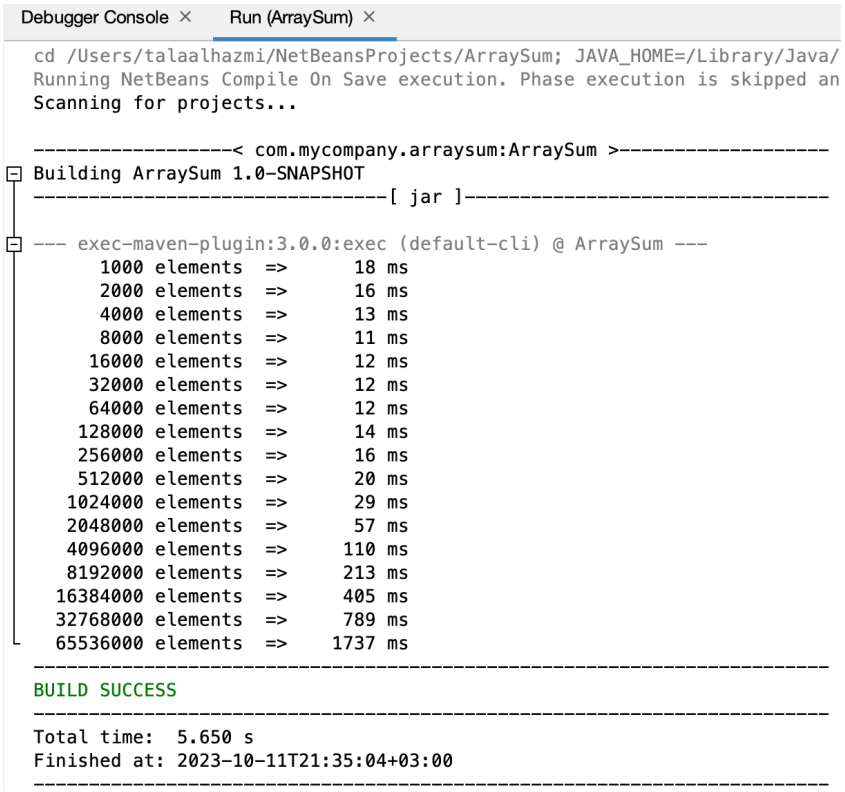


```
Debugger Console × Run (ArraySum) ×
cd /Users/talaalhazmi/NetBeansProjects/ArraySum; JAVA_HOME=/Library/Java/
Running NetBeans Compile On Save execution. Phase execution is skipped an
Scanning for projects...

------------------< com.mycompany.arraysum:ArraySum >-------------------
Building ArraySum 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ ArraySum ---
      1000 elements  =>       1 ms
      2000 elements  =>       1 ms
      4000 elements  =>       1 ms
      8000 elements  =>       2 ms
     16000 elements  =>       3 ms
     32000 elements  =>       1 ms
     64000 elements  =>       2 ms
    128000 elements  =>       3 ms
    256000 elements  =>       7 ms
    512000 elements  =>      12 ms
   1024000 elements  =>      25 ms
   2048000 elements  =>      57 ms
   4096000 elements  =>     143 ms
   8192000 elements  =>     295 ms
  16384000 elements  =>     593 ms
  32768000 elements  =>    1187 ms
  65536000 elements  =>    2341 ms
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  6.883 s
Finished at: 2023-10-11T21:35:45+03:00
------------------------------------------------------------------------
```

Fig. 1: This is the output result for the problem of "summing the square of the elements in an array" using only 1 singular thread. In this screenshot, we can see thelength of int elements of the array (left side) opposite to its runtime in milliseconds (right side), which refers to the duration of time taken forthe computer to run/ execute or complete the task, which in this case is summing the square of the elements in said length array. We see that the longer the arrays' length, the more time it takes to execute on a single thread.



```
Debugger Console × Run (ArraySum) ×
cd /Users/talaalhazmi/NetBeansProjects/ArraySum; JAVA_HOME=/Library/Java/
Running NetBeans Compile On Save execution. Phase execution is skipped an
Scanning for projects...

------------------< com.mycompany.arraysum:ArraySum >-------------------
Building ArraySum 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ ArraySum ---
      1000 elements  =>      18 ms
      2000 elements  =>      16 ms
      4000 elements  =>      13 ms
      8000 elements  =>      11 ms
     16000 elements  =>      12 ms
     32000 elements  =>      12 ms
     64000 elements  =>      12 ms
    128000 elements  =>      14 ms
    256000 elements  =>      16 ms
    512000 elements  =>      20 ms
   1024000 elements  =>      29 ms
   2048000 elements  =>      57 ms
   4096000 elements  =>     110 ms
   8192000 elements  =>     213 ms
  16384000 elements  =>     405 ms
  32768000 elements  =>     789 ms
  65536000 elements  =>    1737 ms
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  5.650 s
Finished at: 2023-10-11T21:35:04+03:00
------------------------------------------------------------------------
```

Fig. 2: This is the output result for the second method (sum2(a)) that uses two threads to solve the issue presented. We can see that for shorter lengthed arrays, the runtime is higher in comparison to when using a single thread. This is due to the fact that it's easier to let a singular thread carry out the work for shorter arrays rather than splitting the array and dividing the work only to sum it back together again. However, for lengthier arrays, the runtimes decrease in comparison to using a single thread. This is because that the work load is done faster split between two threads when the array is lengthier.

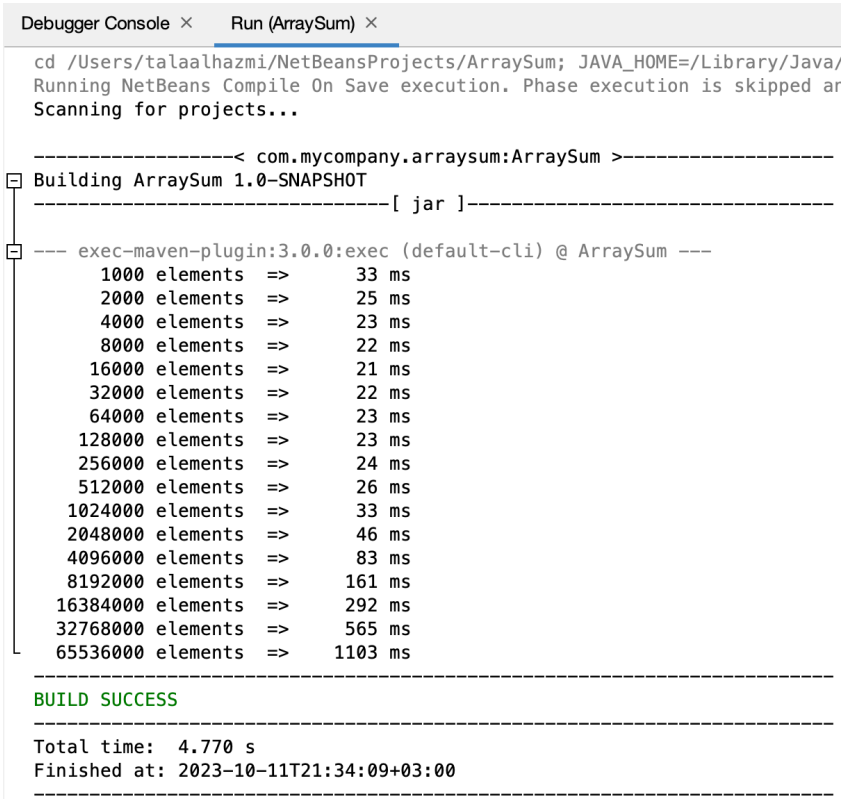


Fig. 3: This is a screenshot of the output received for the third method (sum3(a,4)), which calles on to use 4 threads to execute the code. And as stated in Fig.2, for shorter lengthed arrays, the runtime is higher in comparison to when using a single thread. Also similar to Fig. 2, for lengthier arrays, we can see that after a certain breaking point, the runtimes decrease in comparison to using a single thread. And that's due to the fact that the work load is done faster split between two threads when the array is lengthier.

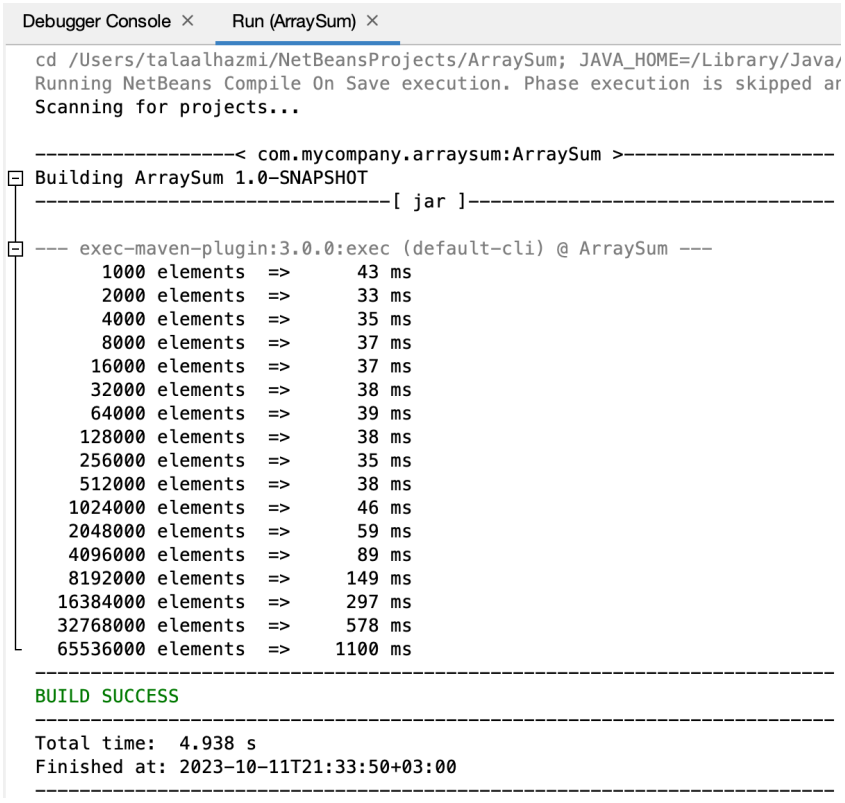


Fig. 4: This is the output received when calling method (sum3(a,6)), which uses 6 threads to runs the code.What is displayed in this screenshot hasa similar explanation to the previous two figures (Fig. 2 and 3), but what we can see is that even though the total runtime in Fig. 2, 3, and 4 are the same (displayed in green, ~4 to 5 seconds), we can see that the runtime for the larger arrays decreases tremendously when mire threads are used. The opposite is true for shorter arrays; the runtime increases the more we use extra threads.

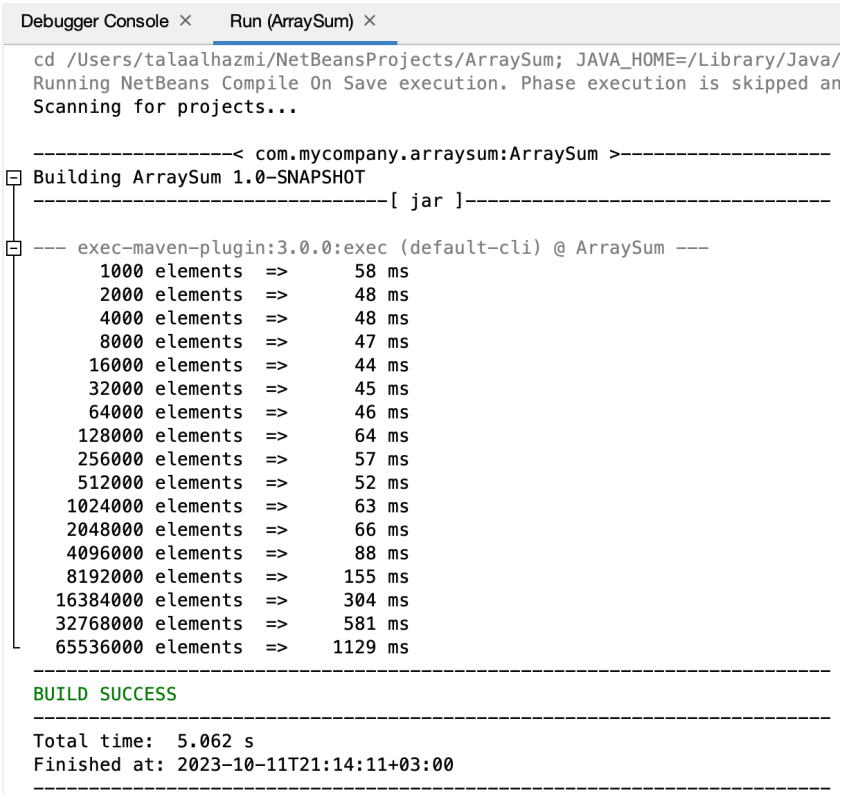


Fig. 5: This output screenshot displays the outcome when calling on method (sum3(a,8)), which uses 8 threads to execute the code. What was stated for Fig, 2, 3, and 4 applies to this figure as well. This method produces the shortest runtime for the larger arrays while producing the longest runtime for the shorter arrays.

5. Conclusion and Future Improvements

When it comes to summing and using multi-threading in java, multi-threading can be used to its full efficiency when the work load is considered too much for a singular thread to carry out at a similar timing in comparison to a multi-threaded program. So, in the case of this experiement, multi-threading was used efficiently in the larger arrays, as their resulting runtimes visibly decreased with the more threads used.

However, this isnt the case for the smaller arrays, seeing as the run times for them separately increased as more threads were used. This helps prove our conclusion that multi-threading aids in decreasing the total runtime to a certain degree (from a total runtime of almost 7 seconds to 5 seconds, as visible in the figures above).

Nevertheless, when looking deeper, we can see that multi-threading, aside from its previously stated constraints in the introduction, may do the complete opposite of its intended purpose when used for smaller data/arrays, resulting in an increase of runtime instead.

Looking at the data received from the outputs of the modified code, and at the data of percentage of improvement in the table submitted separately, we can see that multi-threading could have been used more efficiently utilized if it was only implemented for arrays larger than a specified number of elements (ex: arrays larger than 16000). This may decrease the total runtime even more since most of the resources for multi-threading will be allocated for the larger arrays that actually need to divide the work between working threads, unlike the case for smaller arrays.

6. References

- CHAPTER 6 --Threads and Multithreading in Java. (n.d.). https://cse.iitkgp.ac.in/~dsamanta/java/ch6.htm
    - Coding with John. (2021, June 28). Multithreading in Java Explained in 10 Minutes [Video]. YouTube. https://www.youtube.com/watch?v=r_MbozD32eo
Improvement Percentage Calculator - Calculator Academy. (2022, October 4).
    - Calculator Academy. https://calculator.academy/improvement-percentage-calculator/