

Text

Description automatically generated

**Multi-Threaded**

**Programming in Java**

CS330 Project deliverable

April 26th, 2023

Prepared By:

Sarah AlJurbua (220410528)

Raneem Alsoby (220410583)

Section 251

**Table of Contents**

Table of Contents.................................................................................................. 2

1. Introduction ..................................................................................................... 3
2. Design................................................................................................................ 4
3. Implementation .............................................................................................. 4
4. Results .............................................................................................................. 5
5. Conclusion ....................................................................................................... 8

5.1. Future Improvements................................................................8

1. References .......................................................................................................8
2. **Introduction**

The statement of this project is “using a multi-threading mechanism to solve the issue of summing/ adding elements of an integer array”. Multi-threading is a programming technique that aids multiple threads to run in parallel with a process, where every thread of execution is an independent order of instructions. This leads to faster and more efficient processing of tasks, but this introduces new challenges such as synchronization, deadlock, and race condition. Multi-threading is commonly used in web servers, database systems, and video games.

In the context of summing elements of an integer array, this can be used to speed up the process by dividing the array into multiple smaller sub-arrays and summing each sub-array in parallel using different threads.The decision to tackle this particular problem is driven by its potential to yield various advantages, such as enhanced efficiency, scalability, optimal utilization of system resources, and the ability to reuse code. Java is a multi-threaded programming language, and its threads are referred to as lightweight threads, meaning they run on the same memory space; therefore, making communications more accessible and easier.

The output received from the original given code after running it on Java NetBeans represents the time taken, on the right column of the output, by the called method ( sum(a), sum2(a), or sum3(a,#) ) in the ArraySum class to calculate the sum of the elements in the array of integers, as well as show the execution time for the sum of array elements ranging from 1000 to 65536000 elements.

The general approach taken to modify the original code for this problem is to divide the array into smaller sub-arrays based on the number of threads available. Then, create a worker thread for each sub-array and let them concurrently/simultaneously calculate the sum of squares of their respective sub-arrays. Once all the threads have completed their calculations, the results can be combined using the join() method to get the final result. It is important to ensure proper synchronization between threads to avoid race conditions and ensure correct results. The expected results should be shorter run times for the larger end of arrays when using multi-threading; since arrays are divided up and conquered concurrently. Although, that might not be the case for the smaller end of arrays, as it creates more work to divide it up into sub-arrays rather than doing it all at one together simultaneously. Therefore, the expected results would have longer run times for the smaller end of arrays when increasing the thread count.

1. **Design**

Instead of adding the individual elements in an array, the changed and modified code is intended to add their squares. Sum(a), sum2(a), and sum3(a, #) are three methods that were developed and 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.

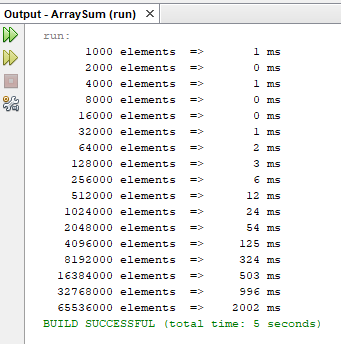
The sum2(a) method is a parallel variation that computes the sum using two threads. It generates two separate "summers" class, each of which affects a different area of the array (sub-divided). The threads are initiated, and after they are both finished, the results are combined. This method is run on 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 joined/summed together. When used on a computer with many processors, this approach is quicker than the single-threaded version since it can utilize every processor.

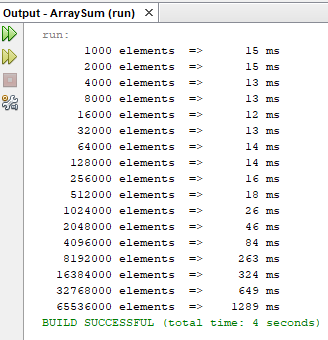
The Runnable interface is implemented by the assistance class known as Summer. It calculates the sum of the squares of the elements between the minimum and maximum indexes using an array, a minimum index, and a maximum index as input. The getSum() method can be used to retrieve a private variable where 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.

1. **Implementation**

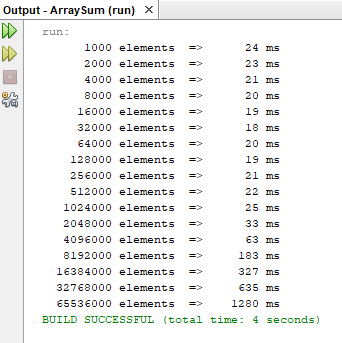
The design section of this report goes into deeper details about the code as a whole, explaining the modifications and changes made to the original given code to us as well as portraying to us how each algorith of the used methods work. But in this sections we’ll be explaining the more important parts of it, where the heart of the whole issue lies. 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 first technique is the sum(a) method, which computes the sum of squares of each element in the array sequentially/concurrently. The second technique is the sum2(a) approach, a parallel variation that adds 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 function/ 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.

1. **Results** 

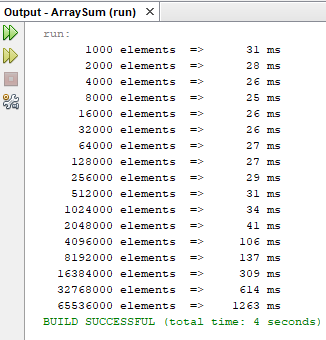
**Fig. 1:** This is the output result for the problem of “summing the square of the elements in an array” using only 1 singular thread. In this screenshot, we can see the length of int elements of the array (left side) opposite to its runtime in milliseconds (right side) (refers to the duration of time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



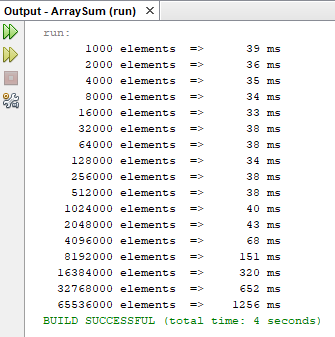
**Fig. 2:** This is the output screenshot when using a calling for the second method (sum2(a) ) that uses two threads to solve the issue presented. We can see that for shorter lengthed arrays, the runtime is higher in comparison to when using a single thread thats due to the fact that when an array is shorter, it's easier to let a singular thread carry out the work rather than splitting the array and dividing the work only to sum it back together again. But that's not the case for lengthier arrays, as 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. 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 that's due to the fact that when an array is shorter, it's easier to let a singular thread carry out the work rather than splitting the array and dividing the work only to sum it back together again. But that's not the case for lengthier arrays, as 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 solve the issue/ runs the code. And what is displayed in this screenshot has a similar explanation to the previous two screenshots ( fig. 2 and 3), but what we can see is that even tho the total runtime in fig. 2, 3, and 4 are the same (displayed in green, 4 seconds), we can see that the runtime for the larger arrays (towards the bottom of arrays list) decreases tremendously the more we use extra threads. The opposite is true for shorter arrays; the runtime increases the more we use extra threads.



**Fig. 5:** In this screenshoted output, it displays the outcome when calling on method ( sum3(a,8) ) which as we can clearly tell from the parameters used in the calling of the method, uses 8 threads to execute the code. What was stated for figures 2, 3, and 4 applies to this screenshot figure as well. This method helps produce the shortest runtime for the larger arrays.

1. **Conclusion**

The conclusion we can come to from this report is that when it comes to summing and using multi-threading in java, the use of 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 as a multi-threaded program would. So in our case, multi-threading was used efficiently in the larger arrays as the runtimes for them separately were decreased visibly the more threads we used. Although, this isnt the case for the smaller arrays, seeing as the run times for them separately were growing bigger and bigger as we used more threads. This helps prove our conclusion that multi-threading helps decrease the total runtime to a certain degree (from a total runtime of 5 seconds to 4 seconds as clearly stated in the figures above). Nevertheless, when looked into it deeper, we can see that multi-threading, aside from its previously stated constraints in the introduction, such as race condition and deadlocks, has a con that it does the complete opposite of its intended purpose (decreasing runtime) when used for smaller datas/ arrays.

**5.1. Future Implementations**

Looking at the data we 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 will help 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.

1. **References** (APA format)

* *CHAPTER 6 --Threads and Multithreading in Java. (n.d.).* [*https://cse.iitkgp.ac.in/~dsamanta/java/ch6.htm*](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*](https://www.youtube.com/watch?v=r_MbozD32eo)
* *Improvement Percentage Calculator - Calculator Academy. (2022, October 4). Calculator Academy.* [*https://calculator.academy/improvement-percentage-calculator/*](https://calculator.academy/improvement-percentage-calculator/)