Parallelization of the Karatsuba Algorithm Using Semaphores and Threadpool in C++

Asir Alam Esteban Brugal Victor Huang Xavier Huetter Naim Shaqqou

Contents

***Abstract*—In this paper, we present three parallel implementations of the Karatsuba algorithm for multiplying large numbers. The first implementation is designed to spawn an unlimited number of threads. The second implementation uses Semaphores, while the third implementation uses a threadpool. We compare the performance of the three parallel implementations with the sequential implementation of the Karatsuba algorithm using experimental results. We also use the School Grade algorithm as a baseline for our experiment.**

Our experimental results show that all three parallel implementations of the Karatsuba algorithm outperform the sequential Karatsuba algorithm for large input sizes.

Our study demonstrates the potential for parallelism to improve the efficiency of the Karatsuba algorithm for large input sizes.

I. Introduction

A. *Problem Statement*

II. Background

A. *The Karatsuba Algorithm*

B. *Parallel Computing*

III. Related Work

Since the algorithm was first introduced by Anatolii Alexeevitch Karatsuba in the 1960s [1], many improvements have been proposed in the literature to further improve its performance.

For instance, the Toom-Cook multiplication algorithm [] was developed as a modified version of the Karatsuba algorithm. The main difference between the Toom-Cook alogrithm and the Karatsuba algorithm is that the former is capable of breaking down the numbers into more than 2 parts unlike the Karatsuba algorithm. The idea behind this algorithm was to provide a faster generalization of Karatsuba's algorithm.

Another example is the Schönhage–Strassen algorithm [] which has a runtime complexity of $O(n \cdot \log n \cdot \log\log n)$. Similar to the Karatsuba algorithm, this algorithm also takes advantage of the divide-and-conquer concept in order to achieve efficient multiplication. This algorithm is known to be effective, especially with numbers with more than $2^{2^{15}}$ digits.

In recent years, there has been more interest in parallelizing the Karatsuba algorithm in order to achieve a higher level of efficiency.

IV. Implementations

The following is the breakdown of the tasks we completed throughout this project:

- Write our own BigInt class.
- Write our own sequential Karatsuba algorithm.
- Implement a parallelized Karatsuba algorithm using semaphores.
- Implement a parallelized Karatsuba algorithm using a threadpool.
- Create a testing environment to obtain performance results.
- Compare benchmarks between the sequential and parallel Karatsuba algorithms.

A. *Plan for Implementation*

Our goal is to implement a parallelized Karatsuba algorithm in the C++ programming language. Our implementation would be much faster than the sequential Karatsuba algorithm. Once the implementation is complete, we plan on testing it against sequential Karatsuba algorithms in C++ and other languages as well as testing it against parallelized versions of the Karatsuba algorithm. Karatsuba recursively calls itself to perform three operations, so we plan on using a thread for each of these recursive calls. This way each operation can run at the same time and would only have to wait for its child threads to finish.

Figure 1 shows an example of how the recursive tree looks:

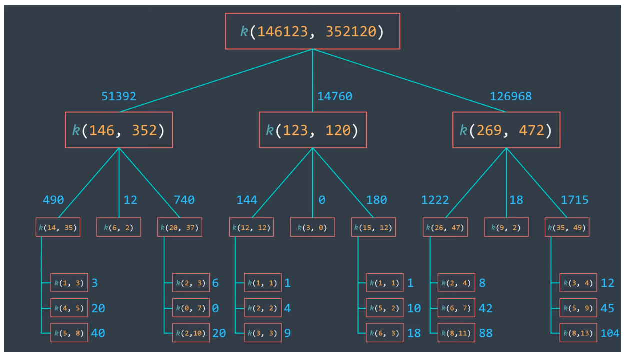

Fig. 1. Recursive tree for a Karatsuba algorithm.

So using this tree as an example, our implementation would first spawn a thread for each call on the first level of the tree. Then a thread would be spawned for each call in the second level of the tree and so on. Each thread can now run at the same time, though they will have to wait for their child threads to finish running since they need the result from the child.

To do proper testing we need to use very big numbers since the algorithm is only useful when the numbers have many digits. C++ doesn't have its own BigInt library so we will create our own so that we can properly test the algorithm.

B. *Anticipated and Encountered Challenges*

The performance could be affected by large constant factors due to string copy operations.

C. *Semaphores*

Our first parallel implementation of the Karatsuba algorithm uses a semaphore to manage the parallelism. In this implementation, we used a semaphore to keep track of and limit the creation of threads to run concurrently. For example, on a machine with a four-core processor, the semaphore would allow for a maximum of 4 threads to be running at the same time and keep track of when a thread has finished to allow for a new thread to begin running in its place. Only when one of the currently running threads has finished its task would the semaphore permit our algorithm to create and use another thread.

By using a semaphore within our parallel implementation of the Karatsuba algorithm, we are able to speed up the performance of the algorithm by allowing recursive calls to be made and performed concurrently. Where a recursive call is made in the sequential version of the algorithm, our implementation checks the semaphore to see if there is available space for a new thread to be created. If a thread is available, the parallel karatsuba will decrement the count of available threads in the semaphore and create a new thread to use to perform the call. If there are no threads available, the algorithm will perform as usual, making the recursive call it normally would. When a thread has finished, it will increment the count of available threads in the semaphore, allowing the next recursive call made to create a thread to perform its calculation.

D. Threadpool

In the next section, we present the experimental setup and results for the two parallel implementations and compare them with the sequential Karatsuba algorithm.

V. Testing

A. Baseline

B. Testing Environments

We performed experiments to compare the performance of the two parallel implementations of the Karatsuba algorithm against each other and against the sequential implementation. We used a desktop computer with a Ryzen 5 5600X processor with 6 cores, 12 threads, and 16GB of 3200MHz RAM. We implemented the algorithms in C++20 using the Visual Studio Code editor. We used the g++ compiler with the -O2 optimization level to compile the code.

We generated random numbers of various sizes and measured the time taken to compute their product using each of the algorithms. We measured the time taken for 10 runs and calculated the average time for each algorithm.

Our parallel implementations were built over several iterations, each meant to introduce some additional efficiency to the previous version, starting with the grade school multiplication algorithm. To ensure that each iteration preserved the correctness of multiplication we made sure to stress-test our code using concise, but comprehensive test data. The data consists of a set of integers of various numbers of digits, even and odd parity, positive and negative signs, and zero. Testing involves multiplying each pair of numbers from this set and validating the result against the grade school multiplication algorithm.

VI. Experimental Results

A. Results

The results of our experiments show that both parallel implementations of the Karatsuba algorithm outperform the sequential Karatsuba algorithm for large input sizes. Figure 2 shows the performance of the three algorithms for multiplying two numbers of size up to 10^8 bits.

B. Discussion

VII. Conclusion

References

[1] A. Karatsuba, Yu. Ofman Multiplication of many-digital numbers by automatic computers Dokl. Akad. Nauk SSSR 1962 145 2 293–294 http://mi.mathnet.ru/dan26729

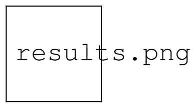

Fig. 2. Performance of the three algorithms for multiplying two numbers of size up to 10^8 bits