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

Asir Alam Esteban Brugal Victor Huang Xavier Huetter Naim Shaqqou

***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. The performance factor for both concurrent algorithms was 3.5 - 4 times faster than the sequential algorithms.

The implementation of the concurrent algorithm using threadpools was slightly more efficient than the semaphore implementation, though the difference in performance is insignificant.

Our study demonstrates the potential for parallelism to improve the efficiency of the Karatsuba algorithm for large input sizes. These parallel implementations could have practical applications, especially in cryptography and other areas that involve the manipulation of large numbers.

I. Introduction

Today our computers perform millions of calculations to allow us to do our work and other tasks in our daily lives. Some of these tasks could be playing video games, rendering a video, compiling code, etc. To perform these tasks our computers will use many different algorithms to help with the efficiency of the tasks. The algorithm we are focusing on in this paper is the Karatsuba Algorithm.

C++ is a very popular programming language that is widely used for building large software infrastructure and applications that run on limited resources. It's used for operating systems, game development, web browsers, etc. It's excellent for building fast software because it can directly manipulate the hardware that it runs on which allows programmers to fine-tune their code to run efficiently in any environment. Due to this and C++'s multithreading support we decided to use it as our programming language of choice.

A. Problem Statement

The main goal for this project is to implement a parallelized Karatsuba algorithm in C++ and test it against the sequential version. As far as we know there doesn't seem to be a widely used parallel Karatsuba algorithm in C++. Once we have implemented the parallel version we will test it against the sequential implementation. We will mainly look at execution time during testing.

II. Background

A. The Karatsuba Algorithm

The Karatsuba algorithm is a fast multiplication algorithm that was discovered by Anatoly Karatsuba in 1960. It is a divide and conquer algorithm that reduces the number of multiplications needed to multiply two very big numbers. The traditional multiplication algorithm is $O(n^2)$, while the Karatsuba algorithm is $O(n^{\log_2 3})$.

Suppose we have two n-digit numbers x and y that we want to multiply in some base B. For any positive integer m less than n, the Karatsuba algorithm works as follows:

1) Both numbers can be rewritten as:
   - $x = x1 * B^m + x2$
   - $y = y1 * B^m + y2$
2) The product of xy becomes:
   - $xy = (x1 * B^m + x2) * (y1 * B^m + y2) = x1 * y1 * B^{2m} + x1 * y2 * B^m + x2 * y1 * B^m + x2 * y2$
3) These are 4 sub-problems that can be reduced to 3 sub-problems:
   - $a = x1 * y1$
   - $b = x1 * y2 + x2 * y1$
   - $c = x2 * y2$
4) And Karatsuba figured out that you can calculate b with the following formula:
   - $b = (x1 + x2)(y1 + y2) - a - c$
5) Finally xy becomes:
   - $xy = a * B^{2m} + b * B^m + c$

B. Parallel Computing

Parallel computing is a type of computation that performs many tasks simultaneously or in parallel. Large problems can be divided into smaller tasks which can then be done simultaneously. Parallel computing has gained a lot of interest due to it being able to reduce run-time, perform large calculations, and reduce energy consumption.

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 [2] 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 [3] 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. There is a large number of articles in the literature that aim improve the performance of common cryptography algorithms, such as Diffe-Hellman key exchange, RSA, ECC, etc., by implementing more efficient large digit multiplication algorithms.

For example, in [4], the authors propose a parallel version of the Karatsuba algorithm that uses a shared memory architecture. They implemented this algorithm with the goal of improving the performance of cryptography algorithms. The results of their experiments, using OpenMP and MPI, was that they were able to achieve a significant runtime improvements, and the algorithms seemed to run faster as they increased the core count.

Another example is [5], where the authors attempted to implement parallel homomorphic hashing on GPU's with a large number of cores. Part of the problem they were tackling was the efficiency of large number multiplication. As part of their experiment, they have attempted to use the Karatsuba-Montgomery algorithm, but they were able to find another algorithm that was better suited for their goals and hardware.

A multi-threaded version of the Karatsuba algorithm will prove useful for researchers in many fields by helping improve the efficiency of large number multiplication needed for their algorithms.

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++. 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:

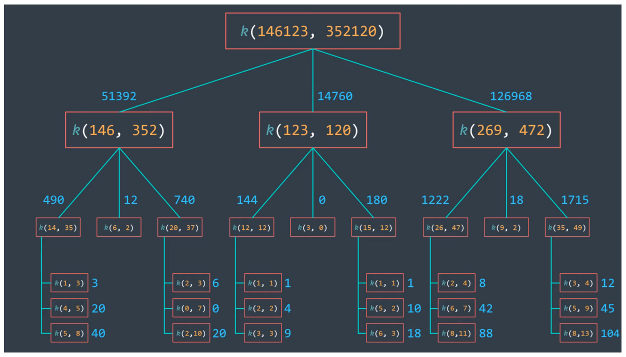

Fig. 1. Recursive tree for a Karatsuba algorithm.

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*

1) Anticipated Challenges: We have anticipated, and accounted for, many challenges and things that could go wrong during our experiment. Although the below challenges seem like considerable hurdles to our progress, we do not anticipate that they will detract from the quality of our solutions and results.

One of those challenges would be the fact that the performance of our algorithms could be affected by large constant factors due to string copy operations. Since our algorithms are meant for use with integers that have a large number of digits, that could have a significant effect on performance. We anticipate that handling those numbers, among other factors, efficiently will be a challenge that we need to overcome.

Second, three out of five people on our team have limited C++ programming experience. This challenge has the potential to delay our development and testing processes due to the anticipated learning curve associated with multithreaded programming using the C++ programming language.

Additionally, most of the team members do not have extensive experience dealing with parallelization techniques aside from concepts that were covered in class. We anticipate spending an extensive amount of time researching the appropriate techniques that could help us achieve our goal of efficiently parallelizing the Karatsuba algorithm.

Finally, we anticipate that we would need to spend additional time learning and analyzing the Karatsuba algorithm to ensure that we properly understand. Understanding the details of the Karatsuba algorithm is essential for us to be able to pinpoint some areas of the algorithm that can be improved with multithreading.

2) Encountered Challenges: Although we have anticipated many challenges throughout our experiment, we have not actually encountered any challenges that significantly impacted our progress. The challenges we encountered are not unique to our project. These challenges arise often in any software related project. With some diligent work, we were able to overcome those challenges.

The first challenge we encountered was that our sequential Karatsuba algorithm had some bugs which caused it to run slower than expected. These bugs were discovered because they caused us to have faulty tests. We were unable to proceed to parallelizing the algorithm until we were able to debug the sequential version of it.

Another challenge we encountered was with our thread pool approach. We were facing issues with finding a good thread pool library for the C++ language since C++ does not have built in thread pooling. We were not able to proceed with the implementing and testing the thread pool approach until we were able to find a good thread pool library for C++.

The biggest challenge we encountered was while setting up our code to test our algorithms. We were running the Karatsuba algorithm on the multiplication of numbers from -1000 to 1000 and for everyone's machine the program would freeze on a different specific number. This would happen specifically on the Parallel Semaphore Karatsuba algorithm. We couldn't find a clear reason for the issue but suspected that the semaphores were somehow causing a deadlock. After a lot of trial and error we were able to fix the bug by releasing the semaphore right before we returned the answer.

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.

Listing 1. Recursive calls to implement concurrently.
```
BigInt a = karatsuba(xh, yh);
BigInt d = karatsuba(xl, yl);
BigInt e = karatsuba(xh + xl, yh + yl) - a - d;

BigInt res = a.left_shift(len)
            + e.left_shift(lower) + d;
```

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.

Listing 2. Parallel portion of Karatsuba using a semaphore.
```
for (int i = 0; i < 3; i++)
  futures[i] = promises[i].get_future();

if (sem.try_acquire())
{
  spawned[0] = true;

  t[0] = thread([](promise<BigInt> &&p,
                   BigInt a, BigInt b) {
    p.set_value(ParallelKaratsuba(a, b));
  }, move(promises[0]), xh, yh);
}
else
{
  a = Karatsuba::karatsuba(xh, yh);
}

//...

if (spawned[0])
{
  t[0].join();
  a = futures[0].get();
}
```

D. Threadpool

For our second parallel implementation of Karatsuba, we used a threadpool to manage the parallelism. Instead of using a semaphore to control the number of threads running at a time, we use a fixed-size threadpool to allocate the threads instead. Unlike the semaphore implementation, the threadpool creates the number of threads to be used when instantiated and maintains them over the course of the program. This means that using the threadpool can avoid the overhead of creating and destroying threads when space is available. Rather, an existing, available thread is acquired when needed instead.

Similarly to our parallel implementation utilizing a semaphore, using the threadpool improves the performance of the Karatsuba algorithm over a sequential version by allowing the recursive calls of Karatsuba to be executed concurrently. However, unlike our version of the algorithm using a semaphore, the threadpool implementation does not create a new thread when trying to make a recursive call with thread space available. Instead, the program will check if there is an available, existing thread from the thread pool and use that thread to perform its recursive call with. Potentially, this could lead to a better performance than the semaphore implementation, as the threadpool would allow the program to avoid overhead present in thread creation and destruction.

Listing 3. Parallel portion of Karatsuba using a threadpool.

```
    if (pool.get_tasks_total() <
        pool.get_thread_count())
    {
      spawned[0] = true;

      futures[0] = pool.submit([](BigInt a,
                                  BigInt b) {
        return ParallelKaratsuba(a, b);
      }, xh, yh);
    }
    else
    {
      a = Karatsuba::karatsuba(xh, yh);
    }

    //...

    if (spawned[0])
    {
      a = futures[0].get();
    }
```

V. Testing

A. Testing Environment

We performed experiments to compare the performances of our two parallel implementations of the Karatsuba algorithm, the sequential implementation, and the grade-school multiplication algorithm against each other. We used a desktop computer with a i5 12400 processor with 6 cores, 12 threads, and 32GB of 3200MHz RAM. The code is written in C++20 and compiled using the g++ compiler and the -O2 optimization level.

B. Test Data

A test data set consists of input and output data. The input data consists of a set of various integers. Each integer is multiplied against every other integer in the set. The output data is the list of products resulting from these multiplications. We used two sets of testing data: (1) data for testing correctness and (2) data for measuring run time. The results (products) of the input data were generated using the grade-school algorithm to be used as the ground truth.

Our parallel implementations were built over many careful iterations, each meant to introduce additional efficiency on top of the previous version, starting with the grade school multiplication algorithm. To ensure that the changes from each iteration preserved the correctness of multiplication, we tested our code using the first data set. Since the purpose of this data set is to test correctness and since it is meant to be used repeatedly between iterations, it should be concise but comprehensive. Therefore, it suffices for the input data to be the set of all integers between -100 and 100, inclusive. It consists of a set of integers of various numbers of digits, even and odd parity, positive and negative signs, and zero. We added a few large multiplication tests as well. These are 5 pairs of integers with 1000 digits each.

The second data set allows us to measure the efficiency of our implementation, which is the primary focus of this paper. For this part, it is only necessary to consider integers with increasing numbers of digits starting from 2^0 number of digits to 2^{15} number of digits. This test will use all 4 implementations of multiplication. The grade school and Karatsuba single core represent our baseline and Threadpool and Semphore are used to compare against our baseline with the goal of finding the rate of speed vs input size. We fix the sign to positive since negative signs do not impact our run time. We run this code testing those range of inputs and output them into a CSV so that we can easily plot and graph them using Python.

Our implementation of the run-time test code uses a random number generator and appends it to the end of a string that is then inserted into our BigInt class. All of the functions use that BigInt class to calculate the very large numbers.

Listing 4. How we set up our run-time testing:

```
    int main(void)
    {
      // file output
      ofstream csv;
      ofstream textfile;
      csv.open("output.csv");
      textfile.open("output.txt");
      csv << "n,ThreadPool,Semaphore,Sequential,Grade"
      << endl;

      for (int i = 0; i < 16; i++)
      {
        string s = "";
        string y = "";
        int n = pow(2, i);

        // makes n number of digits
        for (int x = 0; x < n; x++)
        {
          s += (rand() % 9 + 1) + '0';
          y += (rand() % 9 + 1) + '0';
        }

        BigInt a(s);
        BigInt b(y);

        int tp = ThreadPoolTime(a, b);
        int sem = SemaphoreTime(a, b);
        int seq = sequentialTime(a, b);
        int grade = gradeTime(a, b);
        csv << n << "," << tp << "," << sem
        << "," << seq << "," << grade << endl;

        textfile << "n = " << n << endl;
        textfile << "ThreadPool: " << tp << endl;
        textfile << "Semaphore: " << sem << endl;
        textfile << "Sequential: " << seq << endl;
        textfile << "Grade: " << grade << endl;
        textfile << endl;
      }
    }
```

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 how fast the performance of the multi-threaded versions

TABLE I
EMPIRICAL RUNTIME ANALYSIS OF THREE IMPLEMENTATIONS OF THE KARATSUBA ALGORITHM

N	Sequential Implementation		Semaphore Implementation		Threadpool Implementation	
	$T(n)$	c	$T(n)$	c	$T(n)$	c
1	0	0	155	155	47	47
2	1	0.3322	155	51.4866	131	43.5145
4	3	0.3310	1021	112.6550	211	23.2813
8	14	0.5131	190	6.9637	259	9.4926
16	43	0.5235	225	2.7393	586	7.1342
32	94	0.3801	200	0.8088	220	0.8897
64	309	0.4151	268	0.3600	404	0.5427
128	759	0.3388	566	0.2526	399	0.1780
256	2356	0.3492	1041	0.1543	986	0.1461
512	7262	0.3575	2800	0.1379	1945	0.0958
1024	23575	0.3855	8887	0.1453	4807	0.0786
2048	70961	0.3855	31169	0.1693	15654	0.0850
4096	209481	0.3780	73115	0.1319	39614	0.0715
8192	615375	0.3689	256347	0.1537	118545	0.0711
16384	1930806	0.3844	689652	0.1373	372583	0.0742
32768	5757055	0.3801	2217743	0.1467	1380316	0.0913

in comparison to grade school and sequential Karatsuba. The point at which the sequential versions are slower happen around input size of 2^{10}.

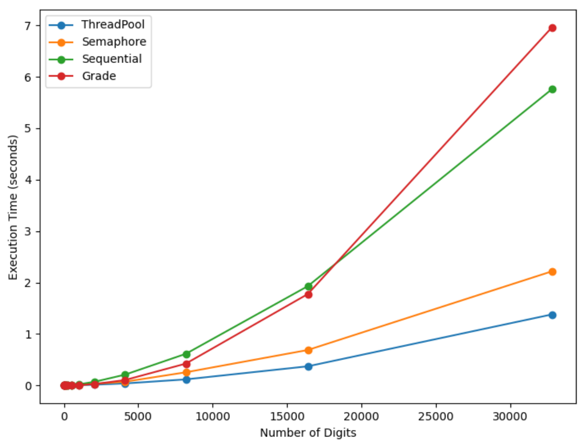

Fig. 2. This is the plot of run-time values from our test program.

B. *Empirical Runtime Analysis*

As a part of our testing process, we decided to incorporate empirical runtime analysis. This would prove that our implementations of the Karatsuba Algorithms are accurate to the official Big-O runtime of $O(n^{\log_2 3})$. The idea behind empirical runtime analysis is:

1) For an algorithm with runtime complexity of $O(n^{\log_2 3})$, the runtime on a given machine would be:
   - $T(n) = c \cdot n^{\log_2 3}$, where c is a constant.
2) Solving for c would give us:
   - $c = T(n) \div n^{\log_2 3}$.
3) The constant value is supposed to converge as n increases.
4) If the constant value converges, then we know that the runtime complexity of our implementations is the same as the official runtime complexity of the Karatsuba algorithm.

According to Table I, the results of our analysis show that the runtime complexity of our algorithms is in fact $O(n^{\log_2 3})$, since the constant values converge as n gets larger.

C. *Discussion*

The results from the testing of Semaphore and threadpool implementations show that they are both significantly faster than the single-core version of Karatsuba's fast multiplication. Though threadpools perform slightly better than semaphores. The run-time of grade school multiplication and single core Karatsuba's which we know have a $O(n^2)$ and $O(n^{1.59})$ are confirmed here. The Performance of the Parallel implementations are still $O(n^{1.59})$ because the number of threads cannot increase with the increase of the input sizes.

VII. Conclusion

In this paper, we presented the problem of making a parallel version of the Karatsuba algorithm for the purpose of speeding up multiplication operations involving high-digit numbers. The goal was to implement the sequential version of the Karatsuba algorithm and use that as a baseline comparison to the parallel version to gauge if our parallel implementations were more efficient than the sequential implementation.

We were successfully able to demonstrate 2 different implementations of the concurrent Karatsuba algorithm: one using semaphores, and the other using a threadpool. We also demonstrate an implementation of the single-core version as a baseline for comparison. We were able to ensure that the Parallel implementation of Karatsuba's algorithm would be faster than the single-core implementation for big integers.

Our study shows that the parallel implementations are indeed faster than the single-core version of this algorithm as depicted in Figure 2. This is due to a reduced constant factor, which divides the run-time proportionally. The time complexity is still $O(n^{1.59})$.

References

[1] Karatsuba, A. & Ofman, Y. Multiplication of many-digital numbers by automatic computers. *Dokl. Akad. Nauk SSSR*. **145**, 293-294 (1962)

[2] Bodrato, M. Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. *WAIFI'07 Proceedings*. **4547** pp. 116-133 (2007,6), http://bodrato.it/papers/#WAIFI2007

[3] Schönhage, A. & Strassen, V. Schnelle Multiplikation großer Zahlen. *Computing*. **7**, 281-292 (1971,9), https://doi.org/10.1007/BF02242355

[4] Tembhurne, J. & Sathe, S. Performance evaluation of long integer multiplication using OpenMP and MPI on shared memory architecture. *2014 Seventh International Conference On Contemporary Computing (IC3)*. pp. 283-288 (2014)

[5] Chu, X., Zhao, K. & Li, Z. Tsunami: massively parallel homomorphic hashing on many-core GPUs. *Concurrency And Computation: Practice And Experience*. **24**, 2028-2039 (2012), https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1826