Making Multiplication Faster: Parallelizing Karatsuba's Multiplication Algorithm

By Joseph Capper, Zachary Gilman, and Luke Jennings

Abstract

In this paper we attempt to produce a parallel multiplication algorithm on massively large numbers, millions and billions of digits long, using the GNU Multiple Precision Library, better known as GMP,and Posix threads, better known as pthreads, for the C programming language. This parallel algorithm parallelizes the calculations of the necessary partial products of Karatsuba's algorithm in order to speedup multiplication. Alongside this, there are many notable struggles that come alongside parallelizing multiplication, including but not limited to, trading off smaller multiplicands with more bit shifting, compromising the initialization and calculation of smaller multiplicands with additional overhead. Despite these flaws, it appears that parallelizing multiplication can reap noticeable benefit over using serial efficient multiplication algorithms such as Fast Fourier Transforms (FFT) or Toom-Cook when using task parallelism with Karatsuba's algorithm as its structure. Any further research on the subject must take account for these problems, likely in the form of parallelizing more complicated efficient multiplication algorithms.

Introduction

Long multiplication, the grade-school technique of multiplying numbers by calculating partial products from taking each digit of one multiplicand and multiplying it with the other, shifting each subsequent partial product one digit corresponding with the digit place of the singular digit, had long remained the best understood way of multiplying any two numbers. This however changed in 1962 when a disproof of the Kolmogorov n^2 conjecture, a conjecture that would implicate that the asymptotic lower bound for any multiplication algorithm as $\Omega(n^2)$, was discovered by Soviet mathematician Anatoly Karatsuba. A further explanation of the algorithm will be described later, however the basic premise of the algorithm is that by separating the bits of two multiplicands into two chunks, referred to as 'a' and 'b' in the first multiplicand, and 'c'

and ‘d’ in the second multiplicand, by use of partial sums and products, it reduces the asymptotic complexity of multiplying any two numbers to $n^{\log_2 3}$. Since then a variety of efficient multiplication algorithms have been developed such as the Toom-Cook algorithm developed by Andrei Toom in 1966, and algorithms involving Fourier transforms, were first discovered by German mathematicians Volker Strassen and Arnold Schrönage in 1971 and since have improved.[2] [3] In part to these algorithms, massive speedup on multiplication computations are possible, making multiplying numbers millions to billions of digits long trivial in comparison to any attempts possible by long multiplication. Such multiplication of massive numbers is particularly relevant to a few fields notably cryptography, where multiplication of prime numbers is particularly relevant for encoding messages.[4] Given that the largest known prime number as of December 2021, $2^{82589933} - 1$, is 24,862,048 digits long in decimal, discovered by the Great Internet Mersenne Prime Search project, and with multiple other prime numbers millions of digits long, an ability to quickly multiply two prime numbers is extremely important for the quick transfer of secure data.[5] Although efficient multiplication algorithms have been utilized in such calculations, attempts to parallelize such algorithms for additional speedup remain relatively unexplored, despite the application of parallelization in other mathematical problems such as matrix multiplication.

GMP Library

This project heavily utilizes version 6.2.1 of the GNU Multiple Precision Library, better known as GMP, is a publicly available math library for C/C++ that allows for operations on numbers of varying precisions, only gated by the memory of the computer performing such operations.[6] For the sake of this project we will be performing multiplication on only integers. As a result, we will be using GMP’s integer struct implementation mpz_t. Although all common mathematical operations such as addition, subtraction, division, and so on are accounted for, here are the functions particularly relevant to our parallel multiplication algorithms.

- `mpz_init(mpz_t x)` - Function initializes a mpz_t struct for use, be it for storing a value or as an output buffer. By default this sets the integer to 0. Although not frequently used, it is helpful in some edge cases later, or when we wish to parallelize the overhead of initialization.

- `mpz_init_set_str(mpz_t x, const char *string, int base)` - Function initializes an mpz_t struct for use starting with an integer of the value of a string given some input base. This is extremely useful for easily splitting the multiplicands into smaller parts and bit shifts easier by setting the base to hexadecimal, thus making a shift of one digit in hex equal to four bit shifts.
- `mpz_mul(mpz_t rop, mpz_t x, mpz_t y)` - This function takes two mpz_t structs and multiplies the integers contained inside them, outputting the product to the return mpz_t struct. The notable part about the GMP implementation of multiplication is that the algorithm used is dynamic based on the size of the integers ranging from long multiplication for very small numbers, to Karatsuba's, to variations of Toom, to FFT. As such in our project, we are attempting to parallelize multiplication using the fastest implementations one could reasonably use. Unfortunately, this function is not safe to run in parallel when two different functions are returning their result to the same mpz_t struct.
- `mpz_add(mpz_t rop, mpz_t x, mpz_t y)` - This function simply takes the integers inside two mpz_t struct and returns the sum into another rop struct. Similar to the other mpz_mul function, this function is not safe to run in parallel if two functions are returning to the same mpz_t struct. However, if a struct being used as an addend can also be used as a struct for the sum.
- `mpz_mul_2exp(mpz_t rop, mpz_t x, mp_bitcnt_t op2)` - This function serves as the bit shift that allows us to shift over a desired number of digits. This is especially important in efficiently shifting over partial products to be added together for our final product.

Serial Multiplication

For point of comparison, we will be using a program without pthreads to which all subsequent timings will be compared to. For this and all subsequent programs, given that we are multiplying two integers together, of which both or written in hexadecimal in order to compute digit shifts more easily, integers will start as strings scanned from text files into a buffer up to 1,073,741,824 characters long (or the equivalent of one Gigabyte). Each string will then be used

in an mpz_init_set_str function call to initialize two different mpz_t structs, then using the mpz_mul function, we multiply the integers together, and measure the time elapsed. Although the overhead of the scanning of the strings is important, in order to measure the speedup of the multiplication in particular, the timer for each program will begin after the two strings have been completely scanned. As such the overhead for the dividing of the substrings into partial multiplicands will be accounted for in the serial implementations.

Naïve Parallelization

As a proof of concept, our first goal was to create a single process, multiple execution, shared memory program using the pthread library that would parallelize the process of multiplying any two numbers without regard to efficiency or overhead.

The premise of our naïve parallelization algorithm is that by dividing an integer into multiple substrings, we can multiply each of those substrings with the other multiplicand, shift the number of digits such that each partial product then matches the digits places that such partial products would appear in a standard long multiplication. In this case AB does not represent the product of an integer A and an integer B, but instead represents a string, where the first arbitrary number of digits are represented by A, and the remaining digits are represented by B. Similar notation will be used for explaining Karatsuba's algorithm later.

```
          C
*A        B
--------------
         BC
ACz
```

In long multiplication, for all subsequent digits past the one's place we must shift left the remaining partial products over once for each digit that has come before it in the multiplicand. As such A doesn't merely represent a string of integers, but rather an integer with a number of zeros at the end as long as the number of digits in B. As such our naïve parallelization emulates this partial sum and digit shifting found in long multiplication.

Our naïve multiplication parallelization consisted of five major steps.

1. Finding the length of both strings of each hexadecimal integer. This step is necessary in order to calculate how many digits of the longer integer should be given to each thread. This process was not parallelized in our implementation, however could be very easily parallelized across two threads before the main calculations occur. The shorter of these two strings is set and initialized as a global mpz_t variable using the mpz_init_set_str function. This was done in order to minimize the discrepancy in length between the two multiplicands, which for this naïve parallelization is very high, as each thread is at best multiplying an n digit number by a n/p digit number, thus calculating approximately n+ n/p digit number, where p represents the number of threads used.
2. Based on the length of the longer string, we block partition the corresponding amount of digits on each thread, copying the desired substring. For example, if our longest string is four digits long, and two pthreads are being used, the first two digits are given to thread 0 and the latter two digits are given to thread 1. If instead five digits, then the first thread would receive three digits instead of two. This step is parallelized in our implementation, as these substrings are only relevant to these local threads. These substrings are used to initialize mpz_t variables similar to that in step 1, each local to their respective thread.
3. Using the mpz_mul function, we multiply each thread's local multiplicand derived from the substring of the larger of the two factors with the global, shorter integer, placing the partial product into a new mpz_t struct. This part is also done in parallel, thus resulting in a partial product on each thread without the correct bit shifts over to match
4. For all threads not including the last, each partial product must be digit shifted left over by the total length of the longer string minus the where in the substring the partial multiplicand ends. For example if our longer multiplicand is 10 digits long, and our first thread used the first 4 digits for its partial multiplicand, then we must shift our partial product on said thread by 6 digit places left. In order to avoid having to multiply by powers of ten or use a complicated system of bit shift if we were to use decimal integer, our implementation instead uses hexadecimal numbers initializing the input string, where in order to shift a hexadecimal integer to the left one digit, it is equivalent to four bit shifts. As for the example above, the partial product we have before would be bit shifted using the mpz_mul_2exp function, by the length of the longer string minus the last index of our substring as found in our original long string. This is less complicated than it

sounds as the values necessary for finding said bit shift is already calculated when copying the substring from the original in each thread.

5. Using a mutex lock around a mpz_add function, add each partial product, now properly shifted over by the correct number of digits, to our final product.

Karatsuba Algorithm

Karatsuba's algorithm, first published 1962 Anatoly Karatsuba's essay "Multiplication of Many-Digital Numbers by Automatic Computers" was the first published algorithm for multiplication faster than $O(n^2)$.[1] For one to understand Karatsuba's algorithm, one must understand that any integer can be represented as a polynomial, as a value times a base to some power plus some constant. As

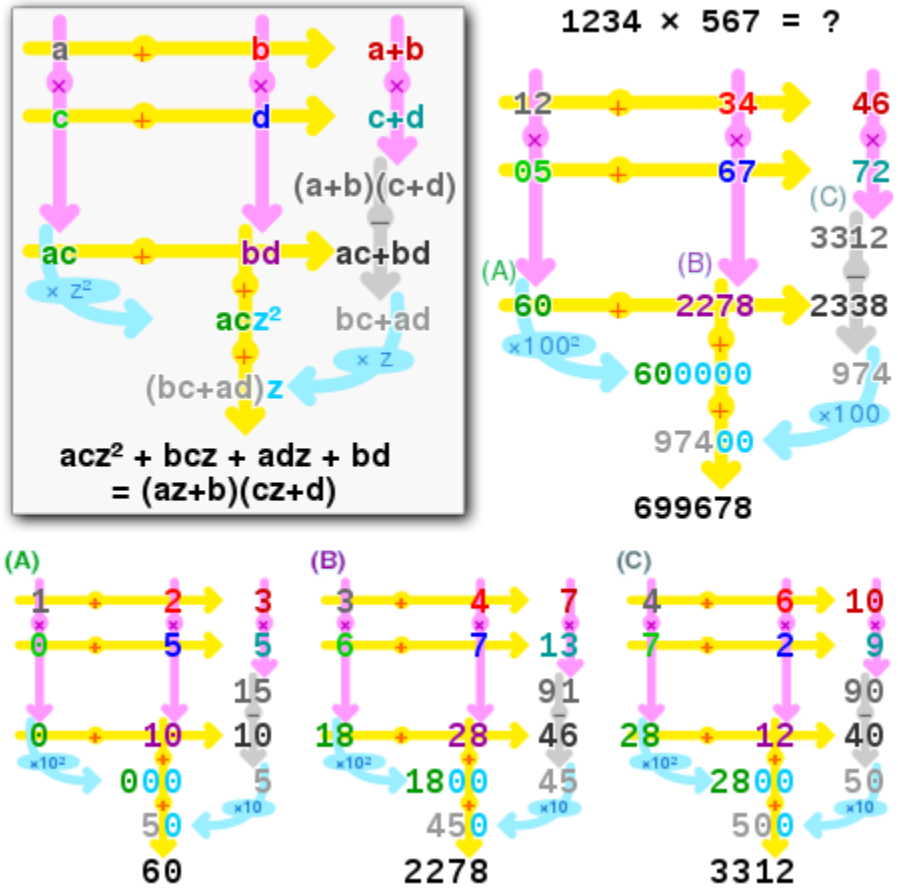
Charles Babbage noted in 1864:

$C = c_1*B^n + c_0$

$D = d_1*B^n + d_0$

Then the product of C * D, would be equal to $(c_1d_1)*(B^{2n})+ (c_1d_0 + c_0d_1)(B^n) + c_0d_0$. As such, we would say that this factorization of two integers consists of four smaller multiplications. Karatsuba however identified that we can use the c_1d_1 and c_0d_0 partial products in order to reduce the total number of multiplications necessary as $(c_1 + c_0)(d_1 + d_0) = c_1d_1 + c_1d_0 + c_0d_1 + c_0d_0$. However if we subtract the c_1d_1 and c_0d_0 that we have calculated earlier, then we receive the value of $c_1d_0 + c_0d_1$, which we can then substitute in our original equation. As such by adding a few more additions we can entirely remove one of the four partial products to be calculated by multiplication. This makes Karatsuba's algorithm asymptotically faster than long multiplication from $O(n^2)$ to $O(n^{\log_2 3})$.

Figure (a): Diagram of Karatsuba's algorithm with examples



Provided by the Wikimedia Commons

Parallel Karatsuba Implementation

Unlike the data parallelism of our naive multiplication parallel algorithm, Karatsuba's algorithm gives us the opportunity to perform task parallelism as there are multiple steps of the algorithm that can be done simultaneously. For example, the calculations of the partial product "ac" and "bd" are completely independent of each other, and as such can be both performed at the same time. This makes a parallel Karatsuba's algorithm prima facie more efficient than our naïve parallelization because at each step we are multiplying smaller multiplicands than our naïve program still which makes the split multiplicands multiply by the entirety of the other input. Here are the steps of our parallel Karatsuba algorithm using pthreads and GMP:

Note: From this point forward, the names of the substrings, partial products, and sums will be in relation to figure 1 above. For example instead of c_1 above, the value will instead be called B.

1. After the two integer strings are read into memory, the longer of the two strings will be initialized as the string for AB, and the smaller of the two CDs. Z, or the number of digits that the final answer will be shifted is set equal to half the length of AB.
2. Using four threads, parse out substrings for A,B,C,D respectively and initialize them as integers using the mpz_init_set_str. These substrings are calculated using the value of z which we determined earlier. For example, A is set to the substring starting from index 0 of the AB string to the length(AB) - Z. Subsequently B is set to the substring of AB from length(AB) - Z to length(AB), and similarly for D but instead as the substring of CD. Determining the value of C is the most complicated as there is potential chance that the length of CD is less than half the length of AB, thus there being potential that length(CD)- Z is negative In these cases if length(CD) is less than length(AB)/2, then C is automatically set to 0. As such this particular implementation is not ideal with numbers that are massively different in terms of length, however in future iterations the value of Z could simply be changed to account for this, such as setting Z equal to length(AB) / 3 or length(CD) * 2 / 3.
3. After the values of A,B,C, and D are determined, given that there is no conflict between the calculation of AC, BD, and (A + B)(C + D), these can all be calculated in parallel. In our particular implementation we performed the multiplication of AC and BD in their own separate pthread while main finds the partial sums of (A + B) and (C + D) and subsequently their product. In order to save time on initialization, all mpz_t representing the partial products and partial sums in this step were all initialized in the previous.
4. Given that all the values need to find the substitute of the factoring of AD+BC (this is equivalent to $(c_1d_0 + c_0d_1)$ in the proof above), we can now subtract the values of AC and BD from (A + B)(C + D) while at the same time shifting over AC by 2Z digits. At this time we also perform the shift of Z on AD + BC so that the only remaining parts of the algorithm can only be done in serial.
5. Into one final mpz_t struct, we add the values of BC, $AC*16^{2Z}$, and $(AD + BC)*16^{Z}$. Unfortunately addition of mpz_t values into the same output is not parallel safe, however given that addition is significantly faster than multiplication, the amount of time saved from adding multiple additions for less multiplication is well worth it

Results

Figure 2.a: Time elapsed after read-in overhead for each implementation

	Implementations (time in seconds)			
Number of Digits (Hexadecimal)	Serial	Naïve - 2 Threads	Naïve - 3 Threads	Karatsuba
10^3	0.000056	0.00049	0.00054	0.001038
10^4	0.000344	0.00073	0.000791	0.001073
10^5	0.004146	0.004158	0.004004	0.003292
10^6	0.046211	0.042329	0.042034	0.028992
10^7	0.63776	0.472755	0.490994	0.333678
10^8	7.324334	5.921079	5.38337	3.59996
2*10^8	15.791844	12.568834	-	7.481155
3*10^8	24.198732	19.304917	17.724394	12.030467

Figure 2.b: Table of speedups relative to serial implementation

	Implementations (time in seconds)			
Number of Digits (Hexadecimal)	Serial	Naïve - 2 Threads	Naïve - 3 Threads	Karatsuba
10^3	1	0.1142857143	0.1037037037	0.05394990366
10^4	1	0.4712328767	0.4348925411	0.3205964585
10^5	1	0.9971139971	1.035464535	1.259416768
10^6	1	1.091710175	1.099371937	1.593922461
10^7	1	1.349028567	1.298916076	1.911303712
10^8	1	1.236993122	1.360548132	2.034559828
2*10^8	1	1.256428719	-	2.110883146
3*10^8	1	1.25350096	1.365278384	2.011454086

Machine Used: Dual Intel Xeon Processor E5-2630 v4 (10C, 2.2GHz, 3.1GHz Turbo, 2133MHz, 25MB, 85W) with 56GB RAM and FreeBSD 12.2 operating system

The methodology of determining our times consisted of running our programs on Siena College's private high-performance scientific computer Noreaster (specifications above in figure 2). These programs were compiled using the GCC, a free compiler produced by GNU. For the input values, in order to maximize the total possible number of bits used per digit, each and

every digit in each number was input as the value F in hex, or equivalent to 15 in decimal.. Unfortunately due to runtime overflows, we were unable to test for values above $3 * 10^8$ digits long, or 300 million digits. Noticeable observations include the following:

- Speedup of the parallel Karatsuba algorithm in comparison to serial multiplication far exceeds it beyond 10^5 digits long, where after one million digits, there is over a 1.5 speedup. This is very important because at its most demanding, our parallel Karatsuba implementation uses only four threads to parse for the values of A, B, C, and D. With the largest known prime numbers being all above 5 million digits long in decimal, this means that cryptography keys using large numbers as their basis can be computed significantly faster, especially if multiple keys must be transferred at once.[7]
- Given that efficient serial multiplication algorithms such as FFT can have as low computational complexity as O(N^1.333), adding additional threads seems to be in many cases very overwhelming, where the difference in speedup between two threads and three threads of the naïve implementation compared to serial at even 300 million digits is only about .1 whereas the parallel Karatsuba implementation is over 1, halving the runtime.
- The serial implementation, given that it does not deal with the significant overhead of generating substrings for their respective multiplicands, remains by far the best choice when dealing with numbers below a few thousand digits.

Conclusion

Our result came out significantly more promising than initially expected as we worried that any additional benefit we would get from multiplying smaller multiplicands together would be offset by the overhead of creating mpz_t structs and the massive number of bit shifts still necessary to properly sum the partial products. This was shown not to be true, and with the parallel Karatsuba implementation to have noticeably significant speedup compared to serial implementations over 1 million digits long. Given that some of the largest prime numbers available are over 5 million digits long, an extremely secure key can be computed in some cases half the time. Further advancements in this domain would include attempts at parallelizing more efficient fast multiplication algorithms such as FFT through algorithms such as Schrönage-Strassen algorithm and those like it. If this proves to be too difficult, this particular instance Karatsuba's algorithm can be interpreted as thinking of the multiplicands as polynomials of degree one, however if we are to divide each multiplicand into 3 or more sections, we can

create a similar parallel algorithm as our degree 1 Karatsuba, but using the partial products in order to create substitutions for the coefficients of the terms in the final product.

Citations

1 Karatsuba, Anatoly Alexeyevich. “The Complexity of Computation.” Translated by Interperiodica Publishing. *Proceedings of Steklov Institute of Mathematics* 211 (1995): 169–83.
This is an English article explaining the history of Karatsuba's algorithm, including the background on Kolmogorov's Conjecture.

2 Yedugani, Sai Yedugani Krishna. “Toom-Cook.” Indiana State University. Accessed December 12, 2021. http://cs.indstate.edu/~syedugani/ToomCook.pdf.
Article is an explanation for the efficient multiplication algorithm Toom-Cook and its variations, many of which are used by GMP.

3 Granlund, Torbjörn. et al. “FFT Multiplication.” GNU Multiple Precision Arithmetic Library, 2020. https://gmplib.org/manual/FFT-Multiplication.
Webpage explanation of what Fast Fourier Transform multiplication is as used by the GMP library.

4 Singer, David, and Ari Singer. “Big Numbers: The Role Played by Mathematics in Internet Commerce.” Case Western Reserve University, 2008. https://case.edu/affil/sigmaxi/files/CryptoslidesSinger.pdf.
Presentation explaining why immensely large prime numbers are used in cryptography.

5 “51st Known Mersenne Prime Discovered.” Great Internet Mersenne Prime Search, 2018. https://www.mersenne.org/primes/press/M82589933.html.

6 Granlund, Torbjörn. “The GNU MP Manual.” pp. 30–3.The GNU Multiple Precision Arithmetic Library, 2020. https://gmplib.org/gmp-man-6.2.1.pdf.
The official documentation for version 6.2.1 of GMP MP. This section explains how the integer struct mpz_t works.

7 Caldwell, Chris K. “Largest Known Primes.” The Top Twenty: Largest Known Primes. Accessed December 12, 2021. https://primes.utm.edu/top20/page.php?id=3.
List of the twenty largest prime numbers currently known. Website also contains database of the largest prime of different types such as Mersenne Primes.