Making Multiplication Faster: Struggles of 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 pthreads. 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 initial attempts, it appears that parallelizing multiplication does not reap any benefit over using serial efficient multiplication algorithms such as Fast Fourier Transforms (FFT) or Toom-Cook. 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 2 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. 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 are particularly relevant to a few fields, notably cosmology, where calculations of massively large distances, masses, and highly precise numbers are necessary. More relevant to computer science however is its application to cryptography, where multiplication of prime numbers is particularly relevant for encoding messages. 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. 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. 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 pthread program 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.

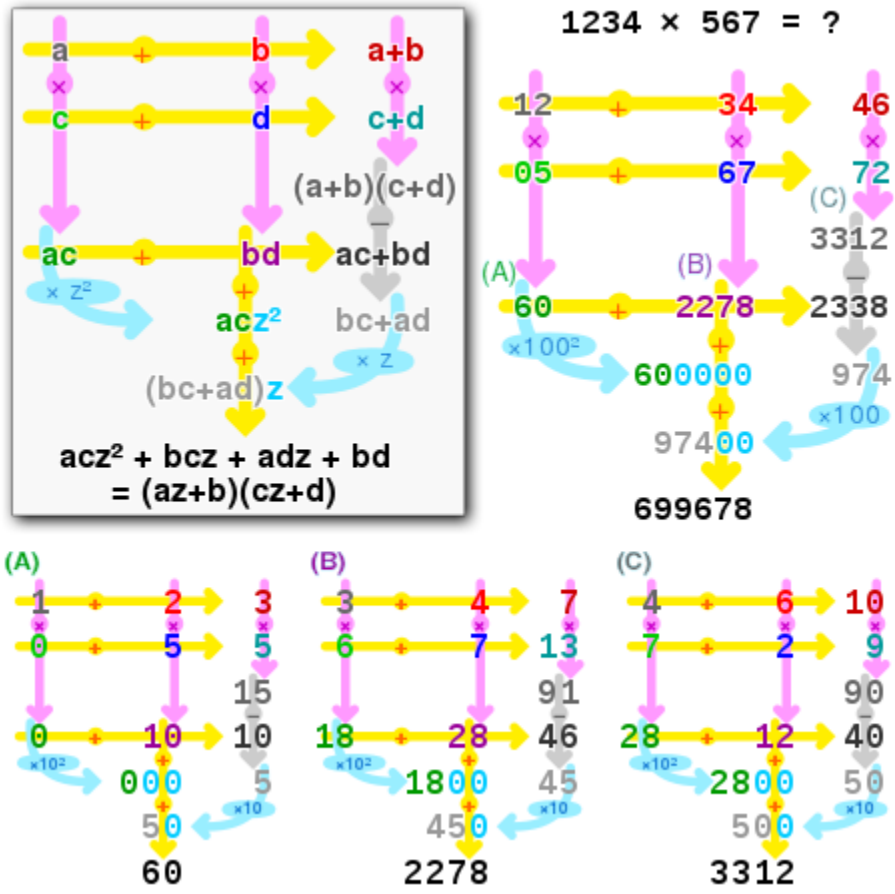
1. Finding the length of both strings of each hexadecimal integer. This step is necessary in order to calculate how many digits of the longer integer should be given to each thread.

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 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 such for the example above, the partial product we have before would be bit shift 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" and later expanded on in his 1965 essay "The Complexity of Computations", was first published algorithm for multiplication faster than $O(n^2)$. 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 $O(n^{\log_2 3})$.

Parallel Karatsuba Algorithm