

Making Multiplication Faster

...

By Joseph Capper, Zachary Gilman, and Luke Jennings

Introduction

- ★ Long multiplication
- ★ Parallelization
- ★ Karatsuba
 - Algorithm created to multiply numbers faster than the classic long division algorithm

What we used

★ Pthreads

★ GMP

- The GNU Multiple Precision Library
- This library allows the use of operations on numbers of varying precisions
 - The length is limited only by memory
 - This was used to make the incredibly large integers to be multiplied together

GMP Functions

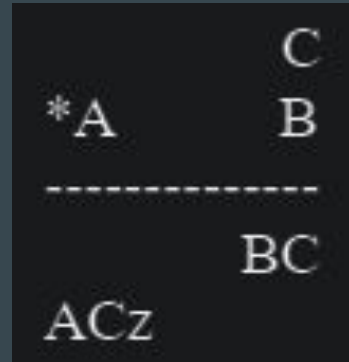
- ★ `mpz_init(mpz_t x)`
- ★ `mpz_init_set_str(mpz_t x, const char *string, int base)`
- ★ `mpz_mul(mpz_t rop, mpz_t x, mpz_t y)`
- ★ `mpz_add(mpz_t rop, mpz_t x, mpz_t y)`
- ★ `mpz_mul_2exp(mpz_t rop, mpz_t x, mp_bitcnt_t op2)`

Serial

- ★ Integers are scanned in from .txt files
- ★ This was used to measure the base time to see speedups
 - Timers begin after the numbers have finished scanning in
- ★ All integers used will be in hexadecimal
 - This helps with bit shifting
- ★ Buffer of size 1,073,741,824
 - One Gigabyte of text

Naïve Parallelization

- ★ Pthreads
- ★ Dividing an integer into multiple substrings
- ★ Multiply each of those substrings with the other multiplicand
- ★ Shift the number of digits such that each partial product then matches the digits places



The diagram illustrates the layout for Naïve Parallelization in multiplication. It shows a vertical arrangement of labels: $*A$ on the left, C and B on the right, a dashed line below $*A$, BC on the right, and ACz at the bottom left. This represents the components of a multiplication operation where A and B are the multiplicands, C is a shift factor, BC is the partial product, and ACz is the final result.

5 Steps

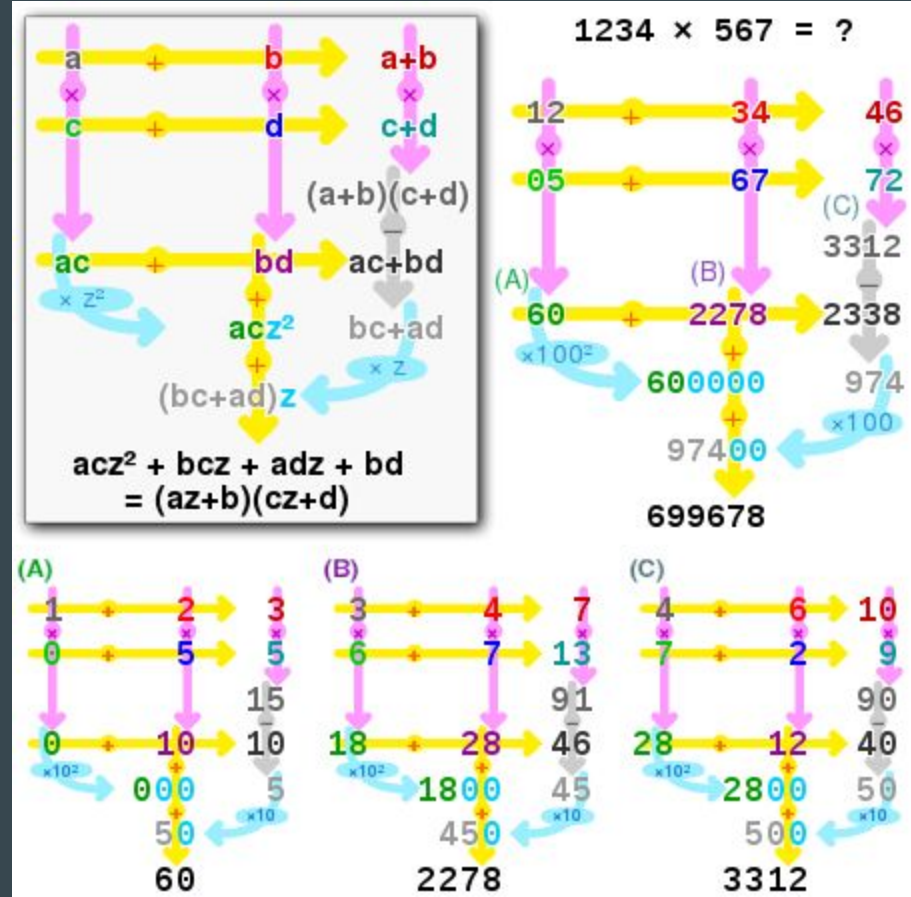
1. Finding the length of both strings
2. Based on the length of the longer string, we block partition the corresponding amount of digits on each thread
3. Using the `mpz_mul` function, we multiply each thread's local multiplicand with the global, shorter integer

5 Steps

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
5. Add each partial product, now properly shifted over by the correct number of digits, to our final product.

Karatsuba's Algorithm

- ★ The first published algorithm for multiplication faster than $O(n^2)$.
- ★ See diagram on the right
- ★ Karatsuba's algorithm gives us the opportunity to perform task parallelism as there are multiple steps of the algorithm that can be done simultaneously



Parallel Karatsuba Implementation

1. The longer of the two strings will be initialized as the string for AB, and the smaller of the two will be CD. Z will be set to 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
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

Parallel Karatsuba Implementation

4. 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. Add the values of BC , $AC \cdot 16^{2Z}$, and $(AD + BC) \cdot 16^Z$

Results

	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
$2 \cdot 10^8$	15.791844	12.568834	-	7.481155
$3 \cdot 10^8$	24.198732	19.304917	17.724394	12.030467

Time elapsed after read-in overhead for each implementation

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
$2 \cdot 10^8$	1	1.256428719	-	2.110883146
$3 \cdot 10^8$	1	1.25350096	1.365278384	2.011454086

Results

- ★ Speedup of the parallel Karatsuba algorithm in comparison to serial multiplication far exceeds it beyond 105 digits long, where after one million digits, there is over a 1.5 speedup
- ★ 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

- ★ We were 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

