Project 1: Divide and Conquer Analysis Report

October 22, 2017

Sabbir Ahmed & Zafar Mamarakhimov

1 Description

A recursive, divide-and-conquer algorithm was developed and analyzed to multiply together lists of complex numbers. Two different multiplication methods were used to compute the same products to analyze the crossover point.

1.1 Background

A complex number z is given by a real part x and an imaginary part y,

$$z = x + iy,$$

where *i* is the imaginary unit $\sqrt{-1}$.

Multiplying two complex numbers are similar to multiplying polynomials. Let $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ be two complex numbers. Then their product is

$$z_1 z_2 = (x_1 + iy_1)(x_2 + iy_2) = (x_1 x_2 - y_1 y_2) + i(x_1 y2 + y_1 x_2)$$

That is, the real part of $z_1 z_2$ is $x_1 x_2 - y_1 y_2$ and the imaginary part is $x_1 y_2 + y_1 x_2$. The computation of a single complex product requires four real products and two real additions (subtraction is just addition with one operand negative and the "$+i$" does not count as an addition as this is really just notation to separate the real and imaginary parts).

As it turns out, there is a way to reduce the number of real multiplications needed to compute a complex product. It is based on the following observation, which is similar to how Karatsuba's method for multiprecision multiplication was derived:

$$(x_1 + y_1)(x_2 + y_2) = x_1 x_2 + (x_1 y_2 + y_1 x_2) + y_1 y_2$$

Let t denote the product $(x_1 + y_1)(x_2 + y_2)$; then if the real products $r = x_1 x_2$ and $s = y_1 y_2$ are computed, the complex product is just

$$z_1 z_2 = (r - s) + i(t - r - s)$$

Now, this computation only requires three real multiplications, but increases the number of additions to five. Since multiplication is the more expensive operation, it is expected for the reduction in the number of multiplies to pay-off, at least if the numbers are large enough.

Simply multiplying two complex numbers would not require a divide-and-conquer solution. However, to multiply a list of n complex numbers, there is a natural recursive divide-and-conquer solution, which is to recursively multiply the left and

right halves of the list, each of length approximately $\frac{n}{2}$, and then multiply together the results of the two recursive calls. The base case is a list of length one, for which the function simply returns the single value in the list.

When multiplying a list of numbers, the difference between three or four real multiplications per complex multiplication can make a significant difference in the running time, especially if individual real multiplications are expensive. Multiprecision arithmetic is required to handle the growth of the product as the numbers get multiplied. Since the cost to perform a single real multiplication will increase per iteration, the use of the "three-multiply" complex multiplication is expected to be faster than the "four-multiply" version. However, since the three-multiply version requires more additions, it may not pay-off until the numbers are large or the list of numbers is long.

2 Crossover Point

The point at which the asymptotically better algorithm becomes faster is called the *crossover point*.

2.1 Theoretical Results

3 Implementation

The project was written in C++11 and built with GCC v5.4.0. The GMP library, along with its C++ wrapper, GMPXX, were used to handle the multiprecision arithmetic. The recursive divide-and-conquer functions for both the three- and four-multiplication methods were implemented identically. The algorithm of the implementation is described in the pseudocode snippet provided in Algorithm 3.1.

Algorithm 3.1: Divide and Conquer Multiplication

```
1  /*
2  Uses a divide and conquer method to recursively multiply all the elements in
3  the complex array using either of the multiplication methods
4
5  Inputs:
6      - complex_array: vector of GMP integer pairs
7      - first, last: first and last indices of the subarray
8
9  Outputs:
10     - cmulx() outputs: final complex product
11 */
12 function cmulx_list(complex_array, first, last):
13
14     // if length of the array is 1
15     if (first == last):
```

```
16          return complex_array(first)
17
18      mid = (first + last) / 2
19      left_half = cmulx_list(complex_array, first, mid)
20      right_half = cmulx_list(complex_array, mid + 1, last)
21
22      return cmulx(left_half, right_half)
```

4 Testing and Timing

4.1 Platform Specifications

Before generating the statistics used in the document, information on the CPU and memory usage of the hosting machine were obtained. The following tables details the specifications captured before initializing the testing procedure.

Table 1: Information about the CPU Architecture, Generated by *$ lscpu*

Component	Specification
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	2
Core(s) per socket:	2
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	142
Model name:	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Stepping:	9
CPU MHz:	1824.398
CPU max MHz:	3100.0000
CPU min MHz:	400.0000
BogoMIPS:	5423.89
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	3072K
NUMA node0 CPU(s):	0-3

Table 2: Information about the Memory, Generated by *$ free -gh*

Component	total	used	free	shared	buff/cache	available
Mem:	3.7G	2.0G	344M	291M	1.4G	555M
Swap:	7.8G	1.1G	6.7G			

4.2 Testing Methodology

The divide and conquer functions of individual multiplication methods were timed. An example of the output time is provided in Figure 1.

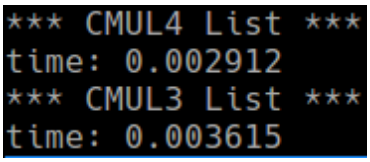

Figure 1: Sample Output of The Program Generated by cmplx_numbers_32_8bit.txt

The program, however, generates its output after a single iteration of the function call. The output times may vary on each execution due to the hosting machine fluctuating on its core and memory usage from other running processes. Therefore, further steps were taken to compute additional statistics on the timings.

An additional Makefile has been provided to generate timings on multiple iterations. 200 iterations were used to compute the current statistics. Furthermore, each iteration recompiles before running the program. A clean compilation prevents any external influences and overhead from the system on the individual timings. All the outputs for a single input are dumped to a buffer file, which is later parsed by an external helper script to generate the following statistics:

- Mean
- Median
- Standard deviation
- Differences of the timing of the two methods

4.3 Results

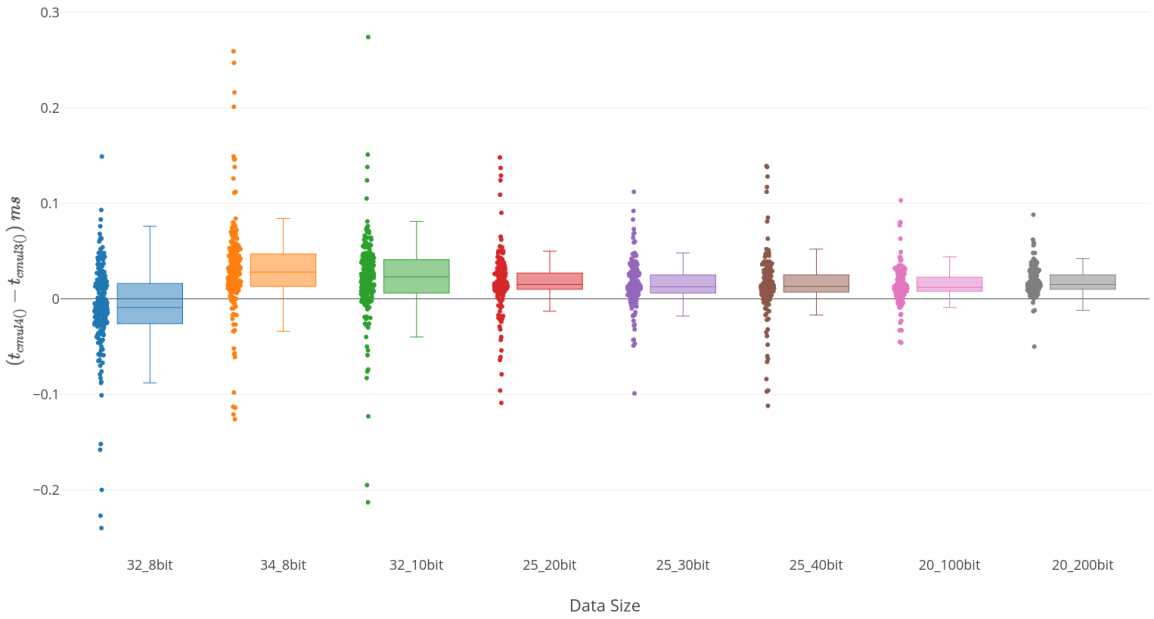

Figure 2: Distribution of Differences in The Multiplication Methods