Project 1: Divide and Conquer Analysis Report

October 23, 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*" doesn't 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.

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

2 Implementation

The project was written in C++11 and built with GCC v5.4.0. The GMP library, along with its C++ wrapper, GMPXX, was used to handle the multiprecision arithmetic. The recursive divide-and-conquer functions for both the three- and four-multiplication methods were implemented identical.

```
/*
Uses a divide and conquer method to recursively multiply all the elements
the complex array using either of the multiplication methods

Inputs:
    - complex_array (std::vector<mpz_class>):
        vector of GMP integer pairs
    - first, last (const size_t): first and last indices of the subarray

Outputs:
    - cmulx() outputs (std::vector<mpz_class>):
        final complex product
*/
std::vector<mpz_class> cmulx_list(
        std::vector<std::vector<mpz_class>> complex_array,
        const size_t first,
        const size_t last
) {

    // if length of the array is 1
    if (first == last) {
```

```
        return complex_array[first];
    }

    size_t mid = (first + last) / 2;

    return cmulx(
            cmulx_list(complex_array, first, mid),
            cmulx_list(complex_array, mid + 1, last)
    );

}
```

3 Testing and Timing

3.1 Testing Platform Specifications

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

4 References

Some links for references