Diana Elisa Noveanu
gr. 935

Documentation PDP

I. *Algorithms*:

1) Sequential naive O(n^2) approach

The naive approach is quite clear, having two polynomials A and B of degree n (for simplicity, without losing generality, we assume that both polynomials have the same degree, let's say n). We multiply the polynomials as we would do on paper, multiplying all coefficients of B with one coefficient of A at a time, summing the coefficients with the same degree.

```
auto A = this->poly;
auto B = other.poly;
auto degree = A.size() + B.size() - 1;
auto result = std::vector<int>(degree, value: 0);
for(int i = 0; i<A.size(); ++i){
    for(int j = 0; j<B.size(); j++){
        result[i + j] += A[i]*B[j];
    }
}

return Polynomial( &: result);
```

2) Parallel naive O(n^2) approach

The parallel naive approach is based on the same idea as the sequential naive approach, but instead of having the main thread multiplying every coefficient of B with every coefficient of A, we divide the work between p threads, every thread multiplying **n/p** coefficients of A with all coefficients of B, thus resulting resulting in a faster (yet not fast enough) method, *parallelization*.

```
void thread_mul_func(std::vector<int>* a, std::vector<int>* b, std::vector<int>* result, int start, int stop){
    for(int i = start; i<stop; i++){
        for(int j = 0; j<a->size(); j++){
            int prod = (*a)[j] * (*b)[i];
            (*result)[i + j] += prod;
        }
    }

}
```

3) Sequential Karatsuba's algorithm

I've implemented Karatsuba's algorithm using **Divide and Conquer**, based on this paper https://eprint.iacr.org/2006/224.pdf.

The basic step of Karatsuba's algorithm is a formula that allows one to compute the product of two large numbers **x** and **y,** (in our case there are polynomials, but the main difference between the multiplication of polynomials and numbers is the fact that we have a carry to transport, so polynomial multiplication is an easier problem), using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and coefficient shifts.

Based on the above observation:

Let x and y be represented as n degree polynomials, with coefficients in base 10. For any positive integer m < n, we can write:

$x = x1 * 10^m + x0$

$y = y1 * 10^m + y0,$ *where* $x0$ *and* $y0 < 10^m,$ *then we can write*

$xy = (x1 * 10^m + x0)(y1 * 10^m + y0) = z2 * 10^{2m} + z1 * 10^m + z0,$

where

$z2 = x1y1,$

$z1 = x1y0 + x0y1,$

$z0 = x0y0$

With z0 and z2 as before we can observe that:

$z1 = (x1 + x0)(y1 + y0) - z2 - z0$ but $(x1 + x0)(y1 + y0)$ may result in an overflow, so we'll rewrite z1 as follows:

$z1 = (x1 - x0)(y1 - y0) + z2 + z0$

4) Parallel Karatsuba's algorithm

As we've already seen the Karatsuba's algorithm consists of 3 steps, so in order to parallelize the algorithm I've splitted each step between the given number of threads.

```
std::vector<int> karatsubaSequential(std::vector<int> A, std::vector<int> B) {
    std::vector<int> product = std::vector<int>( n: 2*B.size(), value: 0);

    if(B.size() == 1){
        product[0] = A[0]*B[0];
        return product;
    }

    int halfSize = A.size() / 2;

    //Arrays for halved factors
    auto aLow = std::vector<int>(halfSize, value: 0);
    auto aHigh = std::vector<int>(halfSize, value: 0);
    auto bLow = std::vector<int>(halfSize, value: 0);
    auto bHigh = std::vector<int>(halfSize, value: 0);

    auto aLowHigh = std::vector<int>(halfSize, value: 0);
    auto bLowHigh = std::vector<int>(halfSize, value: 0);
    //A - multiplicand, B - multiplier
    //Fill low and high arrays
    for(int i = 0; i<halfSize; ++i){
        aLow[i] = A[i];
        aHigh[i] = A[halfSize + i];
        aLowHigh[i] = aHigh[i] + aLow[i];

        bLow[i] = B[i];
        bHigh[i] = B[halfSize + i];
        bLowHigh[i] = bHigh[i] + bLow[i];
    }

    //Recursively call method on smaller arrays
    auto productLow = karatsubaSequential(aLow, bLow);
```

```
    auto productHigh = karatsubaSequential(aHigh, bHigh);

    auto productLowHigh = karatsubaSequential(aLowHigh, bLowHigh);

    //Construct middle portion of the product
    auto productMiddle = std::vector<int>(A.size(), value: 0);
    for(int i = 0; i<A.size(); ++i){
        productMiddle[i] = productLowHigh[i] - productLow[i] - productHigh[i];
    }

    //Assemble the product from the low, middle and high parts
    int midOffset = A.size() / 2;
    for(int i = 0; i < A.size(); ++i){
        product[i] += productLow[i];
        product[i + A.size()] += productHigh[i];
        product[i + midOffset] += productMiddle[i];
    }

    return product;
}
```

For more details: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.7271&rep=rep1&type=pdf

II. Synchronization used in parallel variants

I've found a way to split the work between threads in such a way, that there was no need for any monitors (mutex, semaphore, conditional variable), just plain threads

III. Performance measurements:

Degree	Nr. Threads	Sequential Naive Approach	Parallel Naive Approach	Sequential Karatsuba Approach	Parallel Karatsuba Approach
1024	2	0.0076408	0.0045993	0.169745	0.0068685
1024	8	0.0075652	0.003574	0.169381	0.0068886
1024	20	0.0078281	0.0047299	0.202145	0.0069633
16384	2	1.94652	0.999295	12.7971	1.76273
16384	8	1.92889	0.509178	13.1174	1.71308
16384	20	1.94931	0.524308	15.4762	1.7349
65536	2	30.6031	15.6706	113.192	28.1649
65536	8	30.4665	8.06619	114.212	25.3551
65536	20	30.2512	7.86123	112.141	23.5121