Parallelized Polynomial Multiplication (MultPoly)

Yaxin Chen (yc3995)

November 17, 2021

Introduction

In this project, I will parallelize two algorithms for multiplying two polynomials and compare their runtime. One is the native approach with a time complexity of $O(n^2)$, where n is the degree of the polynomial; the other utilizes fast fourier transform (FFT) and has a time complexity of $O(n \log n)$.

Brute-Force Polynomial Multiplication

A degree-(n-1) polynomial can be represented by an n-element array storing its coefficients. Suppose we have array A representing polynomial $a(x) = \sum_{i=0}^{n-1} A[i]x^i$ and B representing polynomial $b(x) = \sum_{i=0}^{n-1} B[i]x^i$, the array C for the product of $a(x)$ and $b(x)$ can be calculated by

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
      C[i + j] ← C[i + j] + A[i] * B[j];
    end for
end for
```

Parallelization: To parallel the above calculation, I plan to use MapReduce framework. The mapper takes pairs of coefficient of two input polynomials (A[i], B[j]), multiplies them, and sends (key: i+j, value: A[i]*B[j]) to the reducer. The reducer sums the received product and gives output coefficient at index (i+j).

Polynomial Multiplication via FFT

The polynomial multiplication can be speed up to $O(n \log n)$ by fast fourier transforming the input polynomials, multiplying them and the inverse fourier transforming the product.

The discrete fourier transform (DFT) of an n-element sequence A is another n-element sequence P given by

$$P[m] = \sum_{k=0}^{n-1} A[k]\omega_n^{mk},\ m = 0, 1, ..., n-1$$

where $\omega_n = e^{2\pi i/n}$ is the primitive n^{th} root of unity.

For $0 \leq m < n/2$, DFT satisfies

$$P[m] = P_1[m] + \omega^m P_2[m] \tag{1}$$

$$P[n/2 + m] = P_1[m] - \omega^m P_2[m] \tag{2}$$

where

$$P_1[m] = \sum_{k=0}^{n/2-1} A[2k]\omega_n^{2mk}$$

$$P_2[m] = \sum_{k=0}^{n/2-1} A[2k+1]\omega_n^{2mk}$$

FFT utilizes the above property (Eq 1, 2). The algorithm for FFT is shown in Algorithm 1.

Algorithm 1 Fast Fourier Transform

procedure FFT(A, n, ω)
    if n = 1 **then return** A;
    else
        for k ← 0 to n/2 - 1 **do**
            $A_1[k] = A[2k]$
            $A_2[k] = A[2k+1]$
        end for
        $P_1 \leftarrow FFT(A_1, n/2, \omega^2)$
        $P_2 \leftarrow FFT(A_2, n/2, \omega^2)$
        for m ← 0 to n - 1 **do**
            $P[m] \leftarrow P_1[m\, mod\, (n/2)] + \omega^m P_2[m\, mod\, (n/2)]$
        end for
    end if
end procedure

Parallelization: Since the calculation of P_1 and P_2 are independent with each other, their calculation can be parallelized. This part can go parallel to a certain depth and I will investigate the speedup versus the depth. What's more, binary-exchange algorithm and transpose algorithm [1] can parallelize FFT with a better granularity, and I will implement them in Haskell and do some comparisons.

References

1. `https://courses.engr.illinois.edu/cs554/fa2015/notes/13_fft_8up.pdf`
2. `http://www.cs.toronto.edu/~denisp/csc373/docs/tutorial3-adv-writeup.pdf`
3. `https://cse.hkust.edu.hk/mjg_lib/Classes/COMP3711H_Fall14/lectures/DandC_Multiplication_Handout.pdf`