Lab 5 - Parallelizing techniques

Georgiana Loba, 934/2

Goal

The goal of this lab is to implement a simple but non-trivial parallel algorithm.

Requirement

Perform the multiplication of 2 polynomials. Use both the regular $O(n^2)$ algorithm and the Karatsuba algorithm, and each in both the sequential form and a parallelized form. Compare the 4 variants.

The documentation will describe:

- the algorithms,
- the synchronization used in the parallelized variants,
- the performance measurements

Bonus: do the same for big numbers.

Proposed Solution:

For representing the polynomial, I created a class that receives in the constructor a n, which is the degree and a list of numbers which represents the coefficients. Thus, in the list, on the 0 position, we have the coefficients of x^0 and so on so forth.

Sequential Forms:

1. Regular O(n^2) algorithm

2 nested for loops that add to the resulting polynomial the multiplications of the coefficients.

```
@staticmethod
def multiplySequencially(A: Polynomial, B: Polynomial) -> Polynomial:
    size = A.n + B.n - 1
    productList = [0] * size
    for i in range(A.n):
        for j in range(B.n):
            productList[i + j] += A.coefficients[i] * B.coefficients[j]
    return Polynomial(size, productList)
```

Performance:

- Really small polynomials: 0ms
- 6 digits coefficients and degree = 5000: Elapsed time: 5.272801876068115ms

2. Karatsuba Algorithm

Divide and conquer algorithm. The point of the Karatsuba algorithm is to break large numbers down into smaller numbers so that any multiplications that occur happen on smaller numbers.

```
@staticmethod
def multiplyKamasutra(A: Polynomial, B: Polynomial) -> Polynomial:
    if A.n < 2 or B.n < 2:
        return PolynomialOperations.multiplySequencially(A,B)
    m = int(max(A.n, B.n) / 2)
    lowA = Polynomial(len(A.coefficients[:m]), A.coefficients[:m])
    highA = Polynomial(len(A.coefficients[m:]), A.coefficients[m:])
    lowB = Polynomial(len(B.coefficients[:m]), B.coefficients[:m])
    highB = Polynomial(len(B.coefficients[m:]), B.coefficients[m:])

    result1 = PolynomialOperations.multiplyKamasutra(lowA, lowB)
    result2 = PolynomialOperations.multiplyKamasutra(PolynomialOperations.add(lowA, highA), PolynomialOperations.add(lowB, highB)
    result3 = PolynomialOperations.multiplyKamasutra(highA, highB)

    r1 = PolynomialOperations.shift(result3, 2*m)
    r2 = PolynomialOperations.shift(PolynomialOperations.subtract(PolynomialOperations.subtract(result2, result3), result1), m)
    return PolynomialOperations.add(PolynomialOperations.add(r1, r2), result1)
```

Performance:

- Really small polynomials: 0ms
- 6 digits coefficients and degree = 5000: Elapsed time: 9.10444974899292ms

Parallel forms:

1. Regular O(n^2) algorithm - PARALLEL

Simple threads with 2 indexes. A thread computes the multiplications of the polynomials in that range. Multiple threads in parallel fastens computations.

Performance:

- Really small polynomials: 0.0024962425231933594ms
- 6 digits coefficients and degree = 5000: Elapsed time: 0.3419158458709717ms

2. Karatsuba algorithm - PARALLEL

Use an executor service for creating separate working threads for the recursive call of Karatsuba. Wait until all tasks done before computing.

Performance:

- Really small polynomials: 0.008572578430175781ms
- 6 digits coefficients and degree = 5000: Elapsed time: 1.6449389457702637ms