Wednesday, 20 November 2019
Ungur Nicoleta 937

Laboratory 6 - Documentation

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.

Computer Specification

- MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports)
- Processor 2,3 GHz Quad-Core Intel Core i5
- Memory 16 GB 2133 MHz LPDDR3 512 SSD

Short Description of the Implementation

- Regular polynomial multiplication
- Karatsuba algorithm
- Parallelization - Used C++ threads

Regular polynomial multiplication

Complexity: $O(n^2)$

Step 1: Distribute each term of the first polynomial to every term of the second polynomial. Remember that when you multiply two terms together you must multiply the coefficient (numbers) and add the exponents

Step 2: Combine like terms.

The Idea: The simple and naive implementation of multiplication of polynomials.

Karatsuba algorithm

Complexity: $O(n^{\log_2 3})$

A fast multiplication algorithm that uses a divide and conquer approach to multiply two numbers.

The Idea: I have an initial number of threads. For each polynomial I will split it in a half and I will reapply the mult() function to each part. This will be the task for each thread(To call the

mult method on each part of the polynomial). When a polynomial is splited, I'll add 2 more threads to the threads vector. When the size of a polynomial is less than 1, I'll calculate the solution.

The synchronization used in the parallelized variants: I'm using a vector of threads and at each call of mult() method, 2 new threads will start working synchronised until every polynomial will be splited and the solution computed.

Conclusion : For the most part, the parallelized versions of the algorithms run faster. Karatsuba's is clearly superior to the regular algorithm and for large numbers it would be preferred

The performance measurements

Karatsuba sequencial

Running test tests0.in 10

Karatsuba algorithm took me 160 cycles (0.00016 seconds)

Running test tests1.in 50

Karatsuba algorithm took me 527 cycles (0.000527 seconds)

Running test tests2.in 100

Karatsuba algorithm took me 1344 cycles (0.001344 seconds)

Running test tests3.in10000

Karatsuba algorithm took me 1989644 cycles (1.98964 seconds)

Karatsuba parallelized

Running test tests0.in 10

Karatsuba algorithm (threaded) took me 432 cycles (0.000432 seconds)

Running test tests1.in 50

Karatsuba algorithm (threaded) took me 650 cycles (0.00065 seconds)

Running test tests2.in 100

Karatsuba algorithm (threaded) took me 1181 cycles (0.001181 seconds)

Running test tests3.in 10000

Karatsuba algorithm (threaded) took me 1403235 cycles (1.40323 seconds)

RPM sequencial

Running test tests0.in 10

Naive algorithm took me 83 cycles (0.005 seconds)

Running test tests1.in 50

Naive algorithm took me 500 cycles (0.0500 seconds)

Running test tests2.in 100

Naive algorithm took me 1400 cycles (0.014 seconds)

Running test tests3.in 10000

Naive algorithm took me 23124 cycles (2.3124 seconds)

RPM parallelised

Running test tests0.in 10

Naive algorithm (thread) took me 71 cycles (0.003 seconds)

Running test tests1.in 50

Naive algorithm (thread) took me 300 cycles (0.0500 seconds)

Running test tests2.in 100

Naive algorithm (thread) took me 1200 cycles (0.000109 seconds)

Running test tests3.in 10000

Naive algorithm (thread) took me 11618 cycles (1.11618 seconds)

Wednesday, 20 November 2019
Ungur Nicoleta 937

Laboratory 7 - Documentation

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

Requirement

Solve both problems below:

1. Given a sequence of *n* numbers, compute the sums of the first *k* numbers, for each *k* between 1 and *n*. Parallelize the computations, to optimize for low latency on a large number of processors. Use at most 2**n* additions, but no more than 2*log(*n*) additions on each computation path from inputs to an output. Example: if the input sequence is 1 5 2 4, then the output should be 1 6 8 12.

Computer Specification

- MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports)
- Processor 2,3 GHz Quad-Core Intel Core i5
- Memory 16 GB 2133 MHz LPDDR3 512 SSD

The Idea: I have a function compute sum, I have a vector of futures and I store the inputs in a vector from 1 to N. I store the results into the vector of futures and after that I call the get method for each of the. After that I'm printing the result and I print the time.

N=100000	25979ms
N=10000	355ms
N= 1000	17ms
N=100	1ms