**Report Oblig 3 IN3030**

Haiyuec

**Processor:**

Intel I7-8550u

Base frequency: 1.80 GHz

Max Turbo frequency: 4.00 GHz

4 cores 8 Threads

**How to run the program:**

javac \*.java

java Main {N} {number of threads} {test-flag}(optional)

**test-flags:**

* test-primes:

run both the sequential and the parallel version of the sieve to compare their results.

* test-seq:

Run the sequential factorization implementation and output the results using the Oblig3Precode, as well as printing the factorizations to the terminal.

This can be used together with the **check.py** script to verify the correctness with the following command in a Linux terminal:

java Main {N} {number of threads} test-seq | python check.py

or

java Main {N} {number of threads} test-seq | ./check.py

* test-para:

Run the parallel factorization implementation and output the results using the Oblig3Precode, as well as printing the factorizations to the terminal.

This can be used together with the **check.py** script to verify the correctness with the following command in a Linux terminal:

java Main {N} {number of threads} test-para| python check.py

or

java Main {N} {number of threads} test-para | ./check.py

**Implementation explanation:**

**Parallel Sieve:**

* To generate all the prime numbers up to N, first generate all the prime numbers up to sequentially, using the code published on the IN3030 GitHub. Then distribute the primes evenly among the threads, e.g.
  + Thread 1: 2, 11……
  + Thread 2: 3, 13……
  + Thread 3: 5, 17……
  + Thread 4: 7, 19……
* The threads would then cross off the multiples of the primes in a common table.
* The threads were also assigned a section of the value from 3 to N of length N/number of threads. When all the threads have done crossing off numbers in the table, they would go and count the number of primes that are in their section of the table and add the result to a counter in the monitor.
* When this phase is done, the main thread would call the **get\_primes()** method of the monitor. Using the number of prime numbers counted by the threads, an array is created and the main thread loops over the table and get all the prime numbers.
* I have chosen to implement the table as a Boolean array, because the crossing of in the table is not synchronized and would cause error if a byte array implementation was used.

**Sequential factorization:**

* To find the prime factorization of N, first generate the prime numbers up to .
* Try to find a number p in in the prime numbers that was generated, that can cleanly divide the N. If p was found, p is stored as a factor and N is divided by p.
* This process continues until N becomes 1 or no more p can be found in the list of prime numbers.
  + If N becomes 1:
    - All the prime factors of N is found.
  + If no more p is found:
    - The current value of N is also a prime number, and it will be stored.
* To make this more efficient, after a p is found, the search would continue from p, not from the beginning of the table, e.g. if 7 is found as a factor, then the search for the next p would begin from 7, skipping 2, 3 and 5.

**Parallel factorization:**

* To find the prime factorization of N, the prime numbers up to is generated using the parallel implementation of the sieve as described above.
* The prime numbers are then evenly distributed to the threads in the same manner as the prime number distribution in the parallel sieve implementation.
* Each thread would then fetch the N from the monitor, and try to find all prime factors of N in their **local** table. This is the same for-loop inside a while-loop as the sequential factorization implementation. When the loop is done, the individual threads adds the factors it found to the monitor in an ArrayList. There are two scenarios when adding factors to the monitor:
  + This is the last thread that returns:
    - It will add the factors it found to the result bucket.
    - Then multiply the factors to see if it adds up to N.
      * If it adds up:
        + Set up the next task
      * If it does not adds up
        + Add the missing factor to the result bucket
        + Set up the next task
    - Signals the other threads
  + This is not the last thread that returns:
    - It will add the factors it found
    - Wait for signal
* This process continues until 0 is fetched from the monitor, signaling that there are no more tasks.

**Runtime data:**

Runtime measured in ms

|  |  |  |
| --- | --- | --- |
| n | Sequential Sieve | Parallel Sieve |
| 2,00E+06 | 64,11 | 17,86 |
| 2,00E+07 | 456,55 | 115,49 |
| 2,00E+08 | 3586,44 | 1393,17 |
| 2,00E+09 | 32939,77 | 14102,11 |
|  |  |  |
| n | Sequential factorization | Parallel factorizaion |
| 2,00E+06 | 291,81 | 161,98 |
| 2,00E+07 | 2673,80 | 679,21 |
| 2,00E+08 | 19765,39 | 5251,93 |
| 2,00E+09 | 161344,81 | 50346,51 |

**Charts:**

**Conclusion:**