Université Libre de Bruxelles

Faculty of Science

Computer Science Dept.

INFO-F-404 : Real-Time Operating Systems

Project 2: The Ulam spiral

Authors

Chaste Gauvain, Ooms Aurélien

Sunday 1st December, 2013

Academic year 2013 - 2014

Abstract

The following study discribes in details our implementation of the multiprocessing computation and representation of the Ulam spiral. The Ulam spiral is a technique that is used to visualize prime numbers. The main idea is to put all natural numbers on a grid into a spiral and then to highlight all primes. It is beleived this representation could help to understand the distribution of prime numbers.

Contents

1 Generating prime numbers

Several algorithms have been considered for implementation as part of this project. we mainly focused our attention on Eratosthene's and Pritchard's sieves. Although Pritchard's algorithm is part of the source code, we chose Eratosthene algorithm to generate prime numbers. This algorithm is easy to implement and has a time complexity of $O\big(n.log(log(n))\big)$. However we have brought to this algorithm some ameliorations described hereunder.

1.1 Sieve of Eratosthenes

The algorithm is quite simple. To generate prime numbers up to n, we mark all the non-prime numbers like so: for every number i from 2 to $\sqrt{n}$ mark all multiple of i greater or equal than i^2 as non prime. To do that we can simply work with an array of boolean, if *array[i]* is true then we consider i as prime. More precisely: Algorithm 1.1

```
Data: n, primes[$total_count$]
for $i \leftarrow 2$ to $\sqrt{n}$ do
    if prime(i) then
        for $j \leftarrow i^2$ to $n$ do
            primes[$j$]=false;
        end
    end
end
```

Algorithm 1.1: Sieve of Eratosthenes

1.2 Ameliorations

To improve Eratosthene's sieve implementation we have designed some little ameliorations to the base algorithm and the data structure used to store prime numbers.

1.2.1 Reducing memory usage

The larger the Ulam spiral is the easier it is to see it's structure, therefore it is very likely that our software will have to find prime number up to a large bound.

In java a boolean is hard coded on 32 bits. An array of booleans that big can quickly become very memory consuming. For the primes up to 100.000 the array would have a size of 390 Ko. To reduce the size of this array we made sure one boolean was coded on one bit which brings the size down to 97Ko

1.2.2 Removing multiple of two and three

It is obvious that all multiple of two and three are not prime. Knowing that we simply decided no to consider their existance in the array. This means that all index are shifted to the left.
Here is an example representing the six firsts numbers in our sieve.

5	7	11	13	17	19

We noticed that the difference between a number and the next one is always either 2 or 4, depending on this the functions to find a number's index and what number we will find at a given index can be either:

$$index = ((n + 1)/3) - 2;$$

$$number = 6 * ((i/2) + 1) - 1;$$

or

$$index = ((n - 1)/3) - 1;$$

$$number = 6 * ((i/2) + 1) + 1;$$

Using this trick is not only convenient to reduce memory usage by one third but it also means that there are less candidate for primarity and less boolean to toggle.

1.3 Data structures sizes

To avoid having to resize arrays dinamicaly we needed some sort of formula that would ensure an upper bound on the $n - th$ prime. With n beeing a lower bound on the number of primes, chosen by the user. Knowing n, three values need to be calculated: *total_count*, *size* and *last*.

- *total_count* : The number of bits needed to represent all numbers.
  This upper bound is computed with the method shown in Source Code 1.1.
  These formula were found in

Source Code 1.1: computing $total\_count$ upper bound

```
size_t upper_bound(const size_t n){
    if(n < 55){
        return n * (std::log(n) + std::log(std::log(n))) +
   3;
    }
    else if(n < 39018){
        return n * (std::log(n) + std::log(std::log(n)) -
   0.5);
    }
    else{
        return n * (std::log(n) + std::log(std::log(n)) -
   0.9484);
    }
}
```

Olivier Ramaré. 2013. Olivier Ramaré. [ONLINE] Available at: http://math.univ-lille1.fr/ ramare/TME-EMT/Articles/Art01.html. [Accessed 01 December 2013].
They garantee bounds on the *n-th* prime.

- *size* : The size of square containing the ulam spiral.
  This value is easy to get, it is simply $\sqrt{total\_count}$. To make sure the integer part of this value is big enough we calculate it like so.

  $$return(total\_count == 0)?0 : std :: sqrt(total\_count - 1) + 1;$$

- *last* : Number of pixels composing the final picture
  straightforward : $last^2$

Source Code 2.1: sending prime information

```
            MPI_Bcast(&mpi_rank, 1, MPI_SIZE_T, 0, for-
ward[mpi_rank]);
            MPI_Bcast(&j,   1,   MPI_SIZE_T,   0,   for-
ward[mpi_rank]);
            MPI_Bcast(&k,   1,   MPI_SIZE_T,   0,   for-
ward[mpi_rank]);
            MPI_Bcast(&l,   1,   MPI_SIZE_T,   0,   for-
ward[mpi_rank]);
            MPI_Bcast(&right,  1,  MPI_BYTE,  0,  for-
ward[mpi_rank]);
```

2 A multiprocessor system

We have used the Message Passing Interface to generate prime numbers with several processors.
Each processor is assigned a range of number for which he has to find the primes. A processor can be in two ''states'':

- finding primes: Only one processor at the time can be finding primes. This processor go through his range of numbers and identifies the primes. When a prime is found it sends a message to processors taking care of ranges higher than his. The message sent contains required information to discard numbers in higher ranges.
- receiving primes: Those processors are waiting for messages. When they receive one they go through their range of numbers and discard the non-prime numbers.

A message is sent like in Source Code 2.1 with:

- mpi_rank : Value used to determine the sending processor's state
- $i, l, left|right$: Value used to compute the index of a number in the receiver's processor range
- k: The prime found for which multiple need to be discarded.

Two main difficulties have occured while designing the multiprocessor architechture.

2.1 Spliting the work

In order to work on sevral processor the work has to be shared as equaly as possible between them. Each processor is assigned a range of the boolean array used in eratosthene's algorithm.
Normaly with n processors each one should work on a range $\frac{total\_count}{n}$ however it's much easier to calculate indexex in the ranges if they are of a size multiple of 2.

2.2 Writing the files

When the processors finish the last step is to write in the .ppm file. This could be done by two differents strategies.

2.2.1 Random strategy

When a processor finishes he directly writes in the ppm file. This method is not very efficient because it takes a whole disk block, flips one bit and writes it all back on the disck.

2.2.2 Sequencial strategy

When all processors are finished the pixels are written one by one. One pixel is the size of three disk block.

3 Spiral representation

3.1 filters

4 Time analysis

The final time showed on terminal is timed like so:

$if(mpi_rank == mpi_size - 1)std :: cout << Total :< \; < os :: global :: duration.count() << sec < \; < std :: endl;$

because the last processor assigned will always be the last to finish.

5 Analysis

uoyegnrfonexubxfeulhxfleh

5.1 interesting

5.2 veryinteresting

A Code

Here are listed some interesting parts of the source code.

List of Figures

List of Equations

List of Source Code

List of Algorithms