SEIVE OF ERATOSTHENES

Team Members

Murarishetty Santhosh Charan (2019287)

Repala Nagaraju (2019127)

Eraganaboina Venkata Srujan (2019250)

Veerisetty Subramanyam (2019172)

Instructor

Dr. Manish Kumar Bajpai

Introduction

The aim of this project is to parallelize sieve of Eratosthenes algorithm. The Sieve of Eratosthenes is an ancient mathematical algorithm of finding prime numbers between two sets of numbers.

Sieve of Eratosthenes models work by sieving or eliminating given numbers that do not meet a certain criterion. For this case, the pattern eliminates multiples of the known prime numbers. A prime number is a positive integer or a whole number greater than 1, which is only divisible by 1 and itself. The Prime number algorithm is a program used to find prime numbers by sieving or removing composite numbers. The algorithm makes work easier by eliminating complex looping divisions or multiplications.

However, the sieve of Eratosthenes is not practical for identifying large prime numbers with hundreds of digits. The algorithm has complexity O(n ln ln n), and n is exponential in the number of digits. A modified form of the sieve is needed. Use Parallel computation – the elements representing multiples of a particular prime k are marked as composite.

Serial Implementation Algorithm

The **Sieve of Eratosthenes** is a mathematical tool that's used to discover all possible prime numbers between any two numbers. Eratosthenes was a brilliant Greek thinker who, among many other important discoveries and inventions, was deeply interested in

mathematics. His best known contribution to mathematics is his sieve used to easily find prime numbers. A mathematical sieve is any pattern or algorithm that functions by 'crossing off' any potential numbers that don't fit a certain criteria. In our case, the sieve of Eratosthenes works by crossing off numbers that are multiples of a number that we already know are prime numbers. While this all sounds quite complicated, in practice it's quite simple

Steps involved in Serial implementation

- List all consecutive numbers from 2 to η, i.e. (2, 3, 4, 5, ……, η).
- Assign the first prime number letter p.
- Beginning with p^2, perform an incremental of p and mark the integers equal or greater than p^2 in the algorithm. These integers will be $p(p + 1)$, $p(p + 2)$, $p(p + 3)$, $p(p + 4)$ …
- The first unmarked number greater than p is identified from the list. If the number does not exist in the list, the procedure is halted. p is equated to the number and step 3 is repeated.

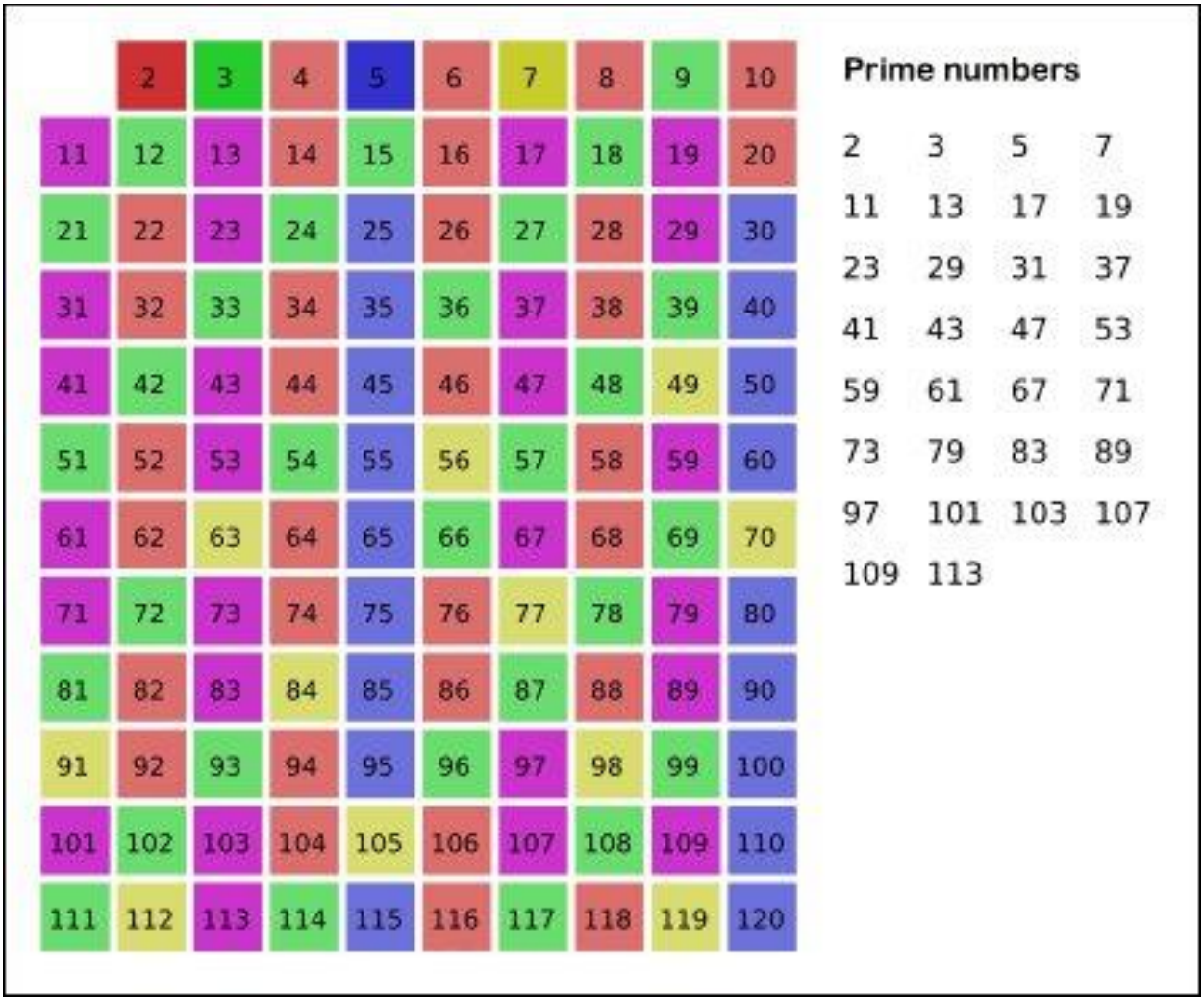
- The Sieve of Eratosthenes is stopped when the square of the number being tested exceeds the last number on the list.
- All numbers in the list left unmarked when the algorithm ends are referred to as prime numbers.

Pseudo Code

```
input: an integer n > 1
output: All prime numbers from 2 to n


 Eratosthenes(n) {
     a[1] := 0
     for i := 2 to n do {
         a[i] := 1
     }
     p := 2
     while p2  ≤  n do {
         j := p2
         while (j  ≤  n) do {
             a[j] := 0
             j := j+p
         }
         repeat p := p+1 until a[p] = 1
     }
     return(a)
```

Basic Illustration



Sieve of Eratosthenes: algorithm steps for primes below 121 (including optimization of starting from prime's square).

Parallel Implementation

In parallel processing, rather than having a single program execute tasks in a sequence (like the tasks of the algorithm above), parts of the program are instead split such that the program is executed concurrently (i.e. at the same time), by multiple entities.
The entities that execute the program can be called either threads or processes depending on how memory is mapped to them.
In shared memory parallelism, threads share a memory space among them.

Threads are able to read and write to and from the memory of other threads. The standard for shared memory considered in this module is OpenMP, which uses a series of pragmas, or directives for specifying parallel regions of code in C, C++ or Fortran to be executed by threads.

In contrast to shared memory parallelism, in distributed memory parallelism, processes each keep their own private memories, separate from the memories of other processes. In order for one process to access data from the memory of another process, the data must be communicated, commonly by a technique known as message passing, in which the data is packaged up and sent over a network. One standard of message passing is the Message Passing Interface (MPI), which defines a set of functions that can be used inside of C, C++.

Steps involved in Parallel implementation

- If a primitive task represents each integer, then two communication are needed to perform the repeat part each iteration of the repeat, until loop.
- Reduction needed each iteration in order to determine the new value of k.
- Then we broadcast to inform all the tasks of the new value of k.
- This will take many reduction and broadcast operations.
- New version of the parallel algorithm that requires less computation and less communication than original parallel algorithm.

Parallel Implementation Code in CUDA lang

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define THREADS 1024
#define MAX 10240000

#define cudaCheckErrors(msg) \
    do { \
        cudaError_t __err = cudaGetLastError(); \
```

```
        if (__err != cudaSuccess) { \
            fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n", \
                msg, cudaGetErrorString(__err), \
                __FILE__, __LINE__); \
            fprintf(stderr, "*** FAILED - ABORTING\n"); \
            exit(1); \
        } \
    } while (0)

__global__ void kernel(int *global, int threads) {
    extern __shared__ int cache[];

    int tid = threadIdx.x + 1;
    int offset = blockIdx.x * blockDim.x;
    int number = offset + tid;

    if ((blockIdx.x != (gridDim.x-1)) || (threadIdx.x != (blockDim.x-
1))){
      cache[tid - 1] = global[number];
      __syncthreads();

      int start = offset + 1;
      int end = offset + threads;

      for (int i = start; i <= end; i++) {
        if ((i != tid) && (tid != 1) && (i % tid == 0)) {
            cache[i - offset - 1] = 1;
        }
      }
      __syncthreads();
      global[number] = cache[tid - 1];}
}


int cpu_sieve(int n){
    int limit = floor(sqrt(n));
    int *test_arr = (int *)malloc(n*sizeof(int));
    if (test_arr == NULL) return -1;
    memset(test_arr, 0, n*sizeof(int));
    for (int i = 2; i < limit; i++)
      if (!test_arr[i]){
        int j = i*i;
        while (j <= n){
```

```
          test_arr[j] = 1;
          j += i;}}
    int count = 0;
    for (int i = 2; i < n; i++)
      if (!test_arr[i]) count++;
    return count;
}

int main(int argc, char *argv[]) {
    int *array, *dev_array;
    clock_t start, end;
    double cpu_time_used;
    int n = 100000;
    if ((n < 1) || (n > MAX)) {printf("n out of range %d\n", n);
return 1;}
    int n_sqrt = floor(sqrt((double)n));

    size_t array_size = n * sizeof(int);
    array = (int*) malloc(n * sizeof(int));
    array[0] = 1;
    array[1] = 1;
    for (int i = 2; i < n; i++) {
        array[i] = 0;
    }

    cudaMalloc((void**)&dev_array, array_size);
    cudaMemcpy(dev_array, array, array_size, cudaMemcpyHostToDevice);

    int threads = min(n_sqrt, THREADS);
    int blocks = n / threads;
    int shared = threads * sizeof(int);
    printf("threads = %d, blocks = %d\n", threads, blocks);
    kernel<<<blocks, threads, shared>>>(dev_array, threads);
    cudaMemcpy(array, dev_array, array_size, cudaMemcpyDeviceToHost);
    cudaCheckErrors("some error");
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (array[i] == 0) {
                        printf("%d ",i);
            count++;
        }
    }
```

```
        printf("\n");
    printf("Count: %d\n", count);
    printf("CPU Sieve: %d\n", cpu_sieve(n));
    return 0;
}
```

Complexity Analysis of the Algorithm

Time Complexity in Serial Implementation:

O(N*log(log N))

Algorithm's running time is O(N*log(log(N)). The algorithm will perform n/p operations for every prime p ≤ N the inner loop.

- The number of prime numbers less than or equal to n is approximately N / ln(N).
- The k-th prime number approximately equals k*ln(k) (that follows immediately from the previous fact)

We extracted the first prime number 2 from the sum, because k=1 in approximation k * ln(k) is 0 and causes a division by zero.

Now, returning to the original sum, we'll get its approximate evaluation is O(N*log(log(N))).

Space Complexity :

O(**N**)

As we are using N sized Boolean array for storing the prime number values as True and composite number values as False, the space complexity becomes o(n).

Time Complexity in Parallel Implementation:

The Sieve of Eratosthenes is impractical for testing primality of numbers with hundreds of digits.

We need to consider the time spent communicating the value of the current prime from processor 1 to all other processors. Assume it takes X time units for a processor to mark a multiple of a prime as being a composite number. Suppose there are k primes as before, less than or equal to sqrt(n). Computation Time: The total time a processor spends striking out composite numbers is:

$$\left(\left\lceil\frac{\lceil\left(\frac{n}{p}\right)\rceil}{2}\right\rceil+\left\lceil\frac{\lceil\left(\frac{n}{p}\right)\rceil}{3}\right\rceil+\left\lceil\frac{\lceil\left(\frac{n}{p}\right)\rceil}{5}\right\rceil+\cdots+\left\lceil\frac{\lceil\left(\frac{n}{p}\right)\rceil}{\pi_k}\right\rceil\right)\chi$$

Communication Time: Assume each time processor 1 finds a new prime it communicates the value to each of the (p-1) processors in turn. If processor 1 spends ʎ amount of time it passes a number to another process, total communication time for k primes is k*(p-1)* ʎ.

Space Complexity :

O(**N**)

As we are using N sized Boolean array for storing the prime number values as True and composite number values as False, the space complexity becomes o(n).

And there is no difference in the space complexity as both utilizes same amount of space.

Reasons for parallelizing Sieve Of Eratosthenes

The three motivations we will discuss here are speedup, accuracy, and weak scaling. These are all compelling advantages for using parallelism, but some also exhibit certain limitations that will also be discussed.

- Speedup is the idea that a program will run faster if it is parallelized as opposed to executed serially. The advantage of speedup is that it allows a problem to be solved faster. If multiple processes or threads are able to work at the same time, the work will theoretically be finished in less time than it would take a single instruction stream.

- Accuracy is the idea of forming a better model of a problem. If more processes or threads are assigned to a task, they can spend more time doing error checks or other forms of diagnostics to ensure that the final result is a better approximation of the

problem that is being solved. In order to make a program more accurate, speedup may need to be sacrificed.

- Weak scaling is perhaps the most promising of the three. Weak scaling says that more processes and threads can be used to solve a bigger problem in the same amount of time it would take fewer processes and threads to solve a smaller problem. A common analogy to this is that one person in one boat in one hour can catch much fewer fish than ten people in ten boats in one hour.

Advantages of Algorithm

- For cryptography, you mostly need larger primes than what you can get via sieving.
- Pseudo-random number generators use Sieve of Eratosthenes.
- Generation of hash tables use prime numbers which are easy to compute using Sieve of Eratosthenes
- This algorithm is also used to find all the prime factors of factorial of a number.
- In practical uses, prime numbers are used in cyphers and codes - including your credit card numbers which can be generated using this algorithm

- Prime numbers are extensively used in cryptography and network security. One of the public key encryption algorithm called RSA actually relies on the factorization of product of large primes.

Link for Complete Implementation of Project (both Serial and Parallel)

GITHUB : https://github.com/santhoshcharan-001/Parallel-Sieve-Of-Eratosthenes

THANK YOU