Independent Study: Parallel Programming

Sandesh Poudel

July 2023

1 Introduction

Traditionally, programs/problems are broken into series instructions and executed on a single processor sequentially. Parallel Programming is the process of splitting a problem into smaller tasks that can be executed at the same time parallelly on multiple processors.

To understand this, let's take an example:

Finding the sum of elements in an Array from 1-15.

The traditional way to do this problem would be to initialize a for loop and add over each element consecutively to get the final sum.

$$for(i = 0; i < 15; i++)\{ \\ j+ = array[i]; \}$$

In Parallel Programming, we can divide the arrays into smaller parts and then find the sum of elements in the partitions and add the sum of those partitions to get the final sum.

```
array[15] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
//..diving array into three partitions..//
array1[5] = {1,2,3,4,5}
array2[5] = {6,7,8,9,10}
array3[5] = {11,12,13,14,15}
//..implementing Parallel Computing..//
sum_Array1 = 15
sum_Array2 = 40
sum_Array3 = 65
final_Sum = 120
```

1.1 History Of Parallel Programming.

Before 2003, the primary way to make computers faster was to have more transistors and increase the clock speed of computers. As time went by, CPU's became so dense because of transistors, increasing clock speed in order to increase performance was no longer feasible because it will just become too hot. So in an effort to improve CPU performance, manufacturers started making more cores in order to compute more problems, keeping clock speed steady. Thus this transformation lead to the improvement of parallel programming.

1.2 Threading.

A thread is a small sequence of instructions that can run independently by our processor. A program's instruction is divided into processes and each process is carried out by either one thread or multiple threads. In our CPU, one core can only run 1 thread at a time. If we have a Quad-core processor, that means our processor can run 4 threads at a time in 4 cores. Threading refers to the technique of running multiple threads concurrently within a process.

1.3 Concurrency.

Concurrency refers to running multiple tasks at the same time. In a concurrent program, tasks could be sharing the execution thread rather than executing in parallel. Concurrency and Parallelism might seem to be the same but they are different. In a concurrent program, a task might have to wait for another task to complete like fetching network data or user commands.

1.4 Parallelism.

Parallelism means different tasks running at the same time parallelly using multiple cores or processors. It involves dividing a problem into smaller sub-problem and running those sub-problem in parallel in multiple cores.

We might think that parallel programming can make a program run faster but it is not always true. Parallel programming focuses on maximizing throughput rather than latency. With parallel programming, we can perform several tasks in a short amount of time. Hence Parallel computing reduces the overall execution time.

Not every program can be parallelized and there is a limit to the speedup achieved by parallel computation. Amdahl's Law is a fundamental principal in parallel computing that quantifies the potential speedup achievable by parallelizing a computation. This law is given by:

$$OverallSpeedup = \frac{1}{1 - P + \frac{P}{N}}$$

where:

- speedup is the improvement in performance achieved by parallelizing a program,
- P represents the fraction of the program that can be parallelized,
- N is the number of processors or cores used.,

There are different programming language and libraries available for parallel computing. Some of them are:

- OpenCl
- OpenMP
- CUDA

2 Sieve of Eratosthenes

We all are familiar with prime and composite numbers. Prime numbers are natural numbers greater than 1 whose factors are only one and itself. We can find prime numbers by basic hand computation until certain small numbers. But to find prime numbers upto larger numbers we need to perform certain computational algorithms. One of the basic algorithms is Sieve of Eratosthenes which can calculate all prime numbers upto a given limit. This algorithm works by marking multiples of each prime as a composite or not prime.

We can visualize this algorithm with the help of example below:
Finding prime numbers upto N=25.

Listing out all natural numbers from 2 to 25.

2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24 25

Taking the first smallest number 2 and marking out its multiples greater than or equal to 2^2 as composite.

2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24 25

Next, take the next smallest unmarked number which is 3 and mark out its multiples greater than or equal to 3^2.

2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24 25

Take the next smallest unmarked number which is obviously a prime number 5 and mark out its multiple greater than or equal to 5^2.

2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24 25

Finally, The unmarked numbers in the list are our prime numbers up to 25. Note that we only have to take multiples of prime numbers from 2 to $\sqrt{N}$.

A sample code to run Sieve of Eratosthenes is presented below:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

void SieveOfEratosthenes(int n)
{

    //initializing an array
    bool prime_list[n + 1];

    //setting each element in the array to true
    memset(prime_list, true, sizeof(prime_list));

    //initalizing a for loop
    for (int i = 2; i * i <= n; i++) {
        // If prime[p] is true
        if (prime_list[i] == true) {
        //initializing another for loop where j will
        //be the multiples of i from i^2
            for (int j = i * i; j <= n; j += i){
                // marking multiples as false
                prime_list[j] = false;

            }
        }
    }


    // Print all prime numbers
    for (int p = 2; p <= n; p++)
        if (prime_list[p])
            printf("%d ",p);
}

int main()
{
    int n = 25;
    printf("The prime numbers upto %d is: \n", n);
    SieveOfEratosthenes(n);
    return 0;
}
```

Time Complexity: $O(n * \log(\log(n)))$

Understanding Time Complexity of Sieve of Eratosthenes:

Lets start with the previous example of finding all prime numbers up to $N = 25$.

The number of steps we would use to mark composite numbers up to 25 is given by:

$$\frac{25}{2} + \frac{25}{3} + \frac{25}{4} = 25(\tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{5})$$

which can be written as:

$$n \textstyle\sum_{i=2}^{n} \frac{1}{p}$$

where:

- **n** is the number upto which we need to find primes.
- **p** is prime numbers upto n.

So the time complexity can be given by:

$$\text{no. of steps} \leq n \textstyle\sum_{i=2}^{n} \frac{1}{p} < n * \log(\log(n))$$

Hence a proper upper-bound time complexity of Sieve of Eratosthenes is **O(n*log(log(n)))**

3 Parallelizing Sieve of Eratosthenes

There can be two ways to parallelize this algorithm. One of the method is using simple parallelizing technique and the other is using threads in CUDA application which will be the one that I will be using for this paper.

Let's start with the first method of parallelizing. Below are the methods to implement Sieve of Eratosthenes: (The steps which can be parallelized are marked in bold typeface):

1. Listing out all the numbers from 2 to N.
2. Finding smallest unmarked number starting from 2.
3. **Marking all multiples of smallest unmarked prime numbers up to $\sqrt{N}$.**

4. Printing out the unmarked prime numbers.

In the above steps, You can see that step 1 can also be parallelized for very large N but for mid-range N I don't think there is any benefit of parallelizing it. Step 3 can be parallelized by dividing the number set into different partitions and each thread can work on marking multiples of smallest unmarked prime number in each partition. This way we dont have to wait to mark 8 because we are marking 4 currently. This marking can be done parallelly.

Another way of implementing this Sieve parallelly in CUDA is simply passing the boolean array to GPU function code and then assigning each thread with its unique thread index to mark the multiples false.

The code that I implemented in cuda is given below. As this is my first try running sieve parallel, this is not a perfect code and is missing error handlers and can only print prime numbers up to four hundred thousand.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

__global__ void seive( bool* d_in) {
//get unique thread index in the deployed thrad blocks from kernel.
int idx = threadIdx.x + blockIdx.x * blockDim.x;



//we willl be using thread index instead of initializing for loops because
//each thread has its own unique index that runs parallely
if (idx >= 2) {

if (((idx ) * (idx )) < 400000) {
if (d_in[idx ] == true) {
for (int i = ((idx ) * (idx )); i <= 400000; i += (idx )) {

d_in[i] = false;

}
}
}
}
}
```

```
int main(){

const int Array_size = 400000;
const int Array_bytes = Array_size * sizeof(bool);
bool prime[Array_size];
memset(prime, true, Array_size);

bool prime_out[Array_size];

//pointer to device array
bool* d_in;


//allocating memory for device array in gpu
cudaMalloc((void**)&d_in, Array_bytes);


//moving contents from host array prime to device array d_in
cudaMemcpy(d_in, prime, Array_bytes, cudaMemcpyHostToDevice);

//launching kernel with 40 blocks of 1024 threads.
seive<<<40, 1024>>>( d_in);

//moving contents from device array to our host output array to print
cudaMemcpy(prime_out, d_in, Array_bytes, cudaMemcpyDeviceToHost);

for (int p = 2; p <= 400000; p++) {
if (prime_out[p])
printf("%d ", p);
}


//freeing device memory.
cudaFree(d_in);



return 0;



}
```