Primes in Parallel Report

Kushagra Indurkhya
(CS19B1017)

Introduction

Implement different parallel implementations for identifying prime numbers less than n and comparing their performance.

Approaches & Designs

- Parallel Writes: Since many threads may be trying to write on the same output file at the same time I have implemented a writer class to facilitate writes by multiple threads using mutual exclusion.

```cpp
class Writer
{
private:
    ofstream *outfile;
    mutex *m;

public:
    Writer(){};
    Writer(string filename)
    {
        outfile = new ofstream(filename);
        m = new mutex();
    };
    // Write is used to append the value of n in the file
    void write(int s)
    {
        // Using mutex to lock the critical section
        m->lock();
        *outfile << s << " ";
        m->unlock();
    };
    // Destructor
    ~Writer()
    {
        outfile->close();
        delete outfile;
        delete m;
    };
};
```

- **DAM (Dynamic Allocation method)**
    - m threads are created
    - A global counter is used, whenever any thread is free it retrieves the number to be checked as the value of the counter and increases it by one.

```cpp
while (i < n)
    {
        i = c.getAndIncrement();
        if (i > n)
            break;
        if (isPrime(i))
            wd.write(i);
    }
```

- This get and increment operation needs to be performed in such a manner that no two threads get the same value, to achieve this I have used a mutex in the counter class

```
class Counter
{
private:
    mutex *m;
public:
    int count;
    Counter()
    {
        count = 0;
        m = new mutex();
    };
    // Returns the value of the counter and increments it by 1
    int getAndIncrement()
    {
        int res = 0;
        m->lock();
        res = count++;
        m->unlock();
        return res;
    }
};
```

- **SAM-1(Static Allocation Method-1)**
    - m threads are created each thread is given a somewhat similarly balanced load
    - Example:
        - if m is 10

          0,10,20,30..... are assigned to thread 0

          1,11,21,31..... are assigned to thread 1

          2,12,22,32..... are assigned to thread 2

          3,13,23,33..... are assigned to thread 3

```
int number = 0;
    for (int i = 0; number < n; i++)
    {
        number = (i * m) + my_id;
        if (number > n)
            break;
        if (isPrime(number))
            ws1.write(number);
    }
```

- **SAM-2(Static Allocation Method-2)**

- Sam-1 is principally the same as sam-1 where each thread checks a set of numbers in such a pattern that load is balanced on each thread with a tweak that we will not be checking for any even number
- number = (i * number_of_threads) + thread_id
- for the number to be odd either they should be ***even, odd*** or ***odd, even*** respectively
- Let's analyze 2 cases:
  - the number of threads is even:
    - (i*number_of_threads) is always even
    - thread id needs to be always odd
  - the number of threads is odd:
    - for odd thread_id , i should be even
    - for even thread_id , i should be odd
- For even cases, we are skipping even thread_ids but creating the same number of threads so a user requesting 16 threads can be thought of as a user requesting 32 threads but skipping even id'd threads.
- So in even cases m sent to thread = actual_m*2 and thread_ids are kept odd

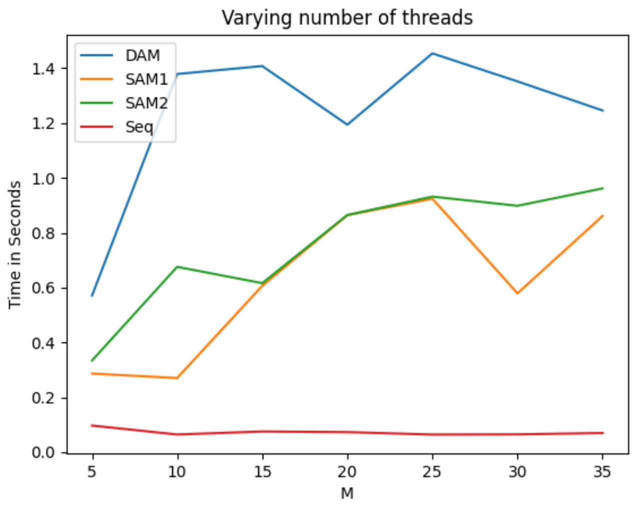
```
if (m % 2 == 0)
    {
        for (int i = 0, j = 1; i < m; i++, j += 2)
        {
            d[i].end = limit;
            d[i].m = m * 2;
            d[i].thread_id = j;
        }
    }
```

- In odd cases depending on the thread_id i should be even or odd
- In thread function i have incorporated both even odd cases with a little tweak

```
    int i = 0;
    int offset;
    if (m % 2 == 0)
        offset = 1; // EVEN CASE NO CHANGE IN THREAD FUNCTION
    else
    {
        offset = 2; // If number of threads is odd all i will be either
even or odd for all (default even) so step will be of 2
        if (my_id % 2 == 0) // IF thread_id is even i should be odd
            i = 1;
    }
    while (number < n)
    {
        number = (i * m) + my_id;
        if (number > n)
            break;
        if (isPrime(number))
            ws2.write(number);
        i += offset;
    }
```

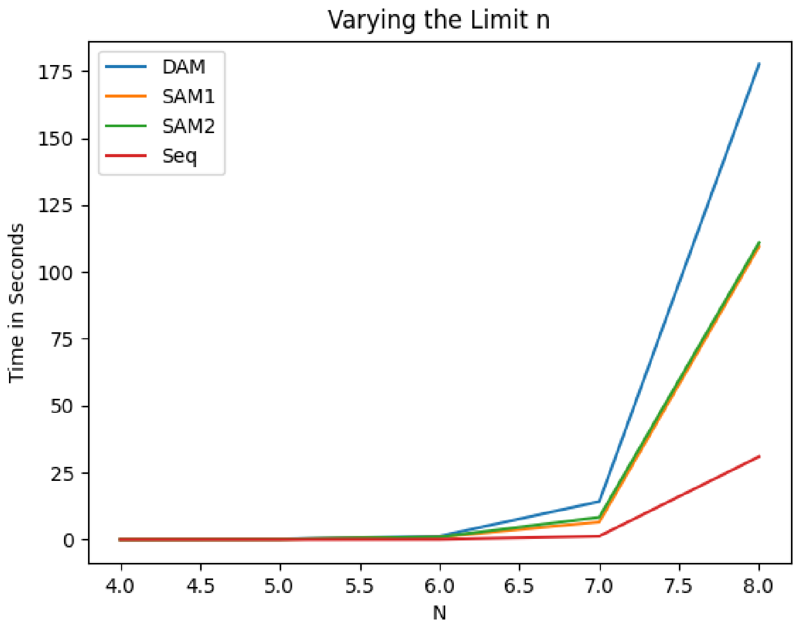
Results

- Comparing Time vs Number of Threads



N fixed to 7 Varying M form 5 to 35 with step 5

- Comparing Time vs Limit



M fixed to 10 Varying N from 4 to 8 with step 1

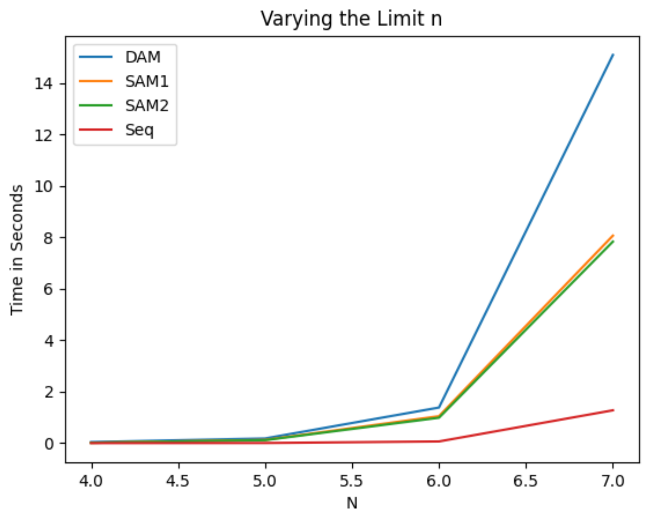
As we can clearly see the sequential is much faster than any of the parallel methods hence it can be asserted the speedup achieved by parallelization cant compensate for the time taken spawning and management of threads and acquiring and releasing mutexes(writing to file or counter).

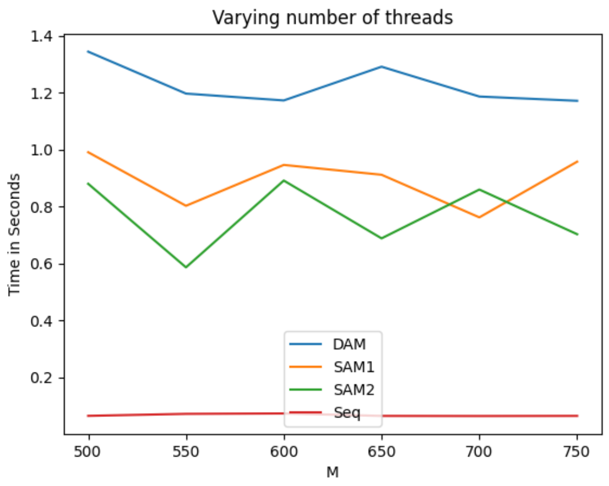
Therefore the speedup achieved if there's any by SAM-2 over SAM1 is not very evident in the results also in the prime number checking function returns immediately when given an even number.

Another observation is SAM-2 is not performing well when the number of threads is even.

Also dynamic takes more than all others as it uses a mutex in the counter too.

For a Better insight into SAM-1 vs SAM-2 I increased the number of threads in the experiments and found these results and on average for both even and odd SAM-2 was performing better than SAM-1





Conclusion

- Prime numbers till a number n may not be an ideal problem for parallelization by these methods since they are not performing better than the sequential algorithm
- DAM is performing worst in all cases
- For a Smaller number of threads, SAM-1 and SAM-2 are performing almost equally
- For a large number of threads (500-1000), SAM-2 Provides better performance.

References

- The Art of Multiprocessor Programming by - Maurice Herlihy and Nir Shavit