CS5300: PARALLEL AND CONCURRENT PROGRAMMING, Fall 2020

Programming Assignment 1: Comparing Different Parallel Implementations for Identifying Prime Numbers

SAI HARSHA KOTTAPALLI

CS17BTECH11036

GOAL

The goal of this assignment is to implement different parallel implementations for identifying prime numbers and comparing their performance.

DESIGN

COMMON DESIGN PART FOR ALL 3 METHODS

We parse input-params.txt from input to obtain the size and number of threads. We then create m threads, each responsible for some work as decided by DAM, SAM1, or SAM2.

Before we start a method(DAM/SAM1/SAM2), we store the start time.

Each thread at a particular iteration generally tries to check if a number is a prime number or not and if so to print it.

To check if a number is prime or not, we use a standard algorithm **isPrime()**, wherein given number i, we check if i is divisible by any of the numbers from 2 to sqrt(i). If so it is not prime, else it is prime.

Each of the print calls is responsible for printing into their respective files whose filenames are defined as per the assignment.

Initially, In the case of printing to file parallelly, I was stuck because "**fstream**" on **write** mode was facing some race conditions the final output was not in the required format. To tackle this, I have used **fprintf()**. This solves the issue of threads writing to the file parallelly because, for any particular **FILE** object, all **fprintf()** calls to it are **atomic**. This is due to the fact that internally a lock is present for the **FILE** object so that, after the fprintf() call ends, the lock is released.

Finally, when the chosen method completes and all the threads are joined, we store the end time.

Note: It was intentional to separate out the print functions for all three methods as it wasn't clear if all three methods were supposed to be submitted together in one source file or all together. Hence, I separated them out so that it's clear which functions belong to which method.

DAM - Dynamic Allocation Method

We extend from the common design provided above.

We use **Counter** class which is responsible to store a **counter** and implement the **getAndIncrement()** method. We can use a **mutex** to make this method thread-safe, increasing the counter value when a thread enters the critical section and returning the original value, so the next thread which enters the critical section will access the updated counter value. It uses the **isPrime()** method as defined above to check if a number is prime or not.

The algorithm for DAM given in the book is incorrect, as it checks extra m numbers above the limit, if they are prime and if so also prints them, but this is not what we require. To solve this problem, I call the **getAndIncrement()** towards the end, so that

before the loop body we can check every time if the counter is indeed less than the limit or not.

For example, if the limit was 100, the algorithm in the book will also check for numbers 100 to 109. This is because threads having a value from {90-99} are still less than the limit and then get the next number from the counter which is {100-109}. The updated algorithm correctly solves this problem, stopping when counter >= 100.

The exact algorithm is given below.

SAM1 - Static Allocation Method 1

We extend from the common design provided above.

We pre-assign the numbers to verify to each of the m threads using the following formula: It uses the **isPrime()** method as defined above to check if a number is prime or not.

```
// Let thread numbers are index 1 based
// Total number of threads = m
kth thread evaluates number from: k, k + m, k + 2m, … until 10^n
```

Since we already decided on the numbers each thread can take, this is a static allocation method. This can be improved upon which is what SAM2 does.

The exact algorithm is given below.

SAM2 - Static Allocation Method 2

We extend from the common design provided above.

A thread given an even number to test for prime can immediately return as opposed to a thread given an odd number(2 being the exception).

Basically, we know that odd numbers except 1 can be candidates for **isPrime()** evaluation. All even numbers except 2 are not candidates for **isPrime()** function. So, let the first thread handle the exceptions. The rest of the threads will evaluate all numbers from using:

```
// Let thread numbers are index 1 based and k > 1
// Total number of threads = m
kth thread evaluates number from: 2k-1, 2k-1 + 2m, 2k-1 + 4m, … until 10^n
```

As for the first thread, it will first evaluate 2 and then evaluate all odd numbers which are not included in the above formula, namely when k = 1.

```
// Let thread numbers are index 1 based and k = 1
// Total number of threads = m
kth thread evaluates number from: 2, 2k-1 + 2m, 2k-1 + 4m, … until 10^n
```

This algorithm is correct because it covers all odd numbers except 1 and manually checks for 2 as candidates for **isPrime()** method. This is exactly what we require as stated above. We use **2*m** for incrementing because of simple logic, the next m - 1 alternating odd numbers are already chosen by the other m-1 threads, while the next m - 1 alternating even numbers are skipped.

The exact algorithm is given below.

ALGORITHMS

Standard prime number checking algorithm

```
bool isPrime(int n) {
  if (n <= 1)
      return false;

  // Check from 2 to n-1
  for (int i = 2; i*i <= n; i++)
      if (n % i == 0)
      return false;

  return true;
}
```

COUNTER

```
class Counter {
private:
  int i = 1;
  std::mutex incrementer; // for ME
public:
  int getAndIncrement() {
      int val;
      incrementer.lock(); // cs starts here
      i++;
      val = i-1; // prev value will be returned
      incrementer.unlock(); // cs ends here
      return val;
  }
};
```

DAM

```
void primePrintDAM(int n) {
  long i = counter.getAndIncrement();
  long limit = pow(10, n);
  while (i < limit) {
      // loop until all numbers taken
      if (isPrime(i))
      print_dam(i);
      i = counter.getAndIncrement(); // take next untaken number
  }
}
```

SAM1

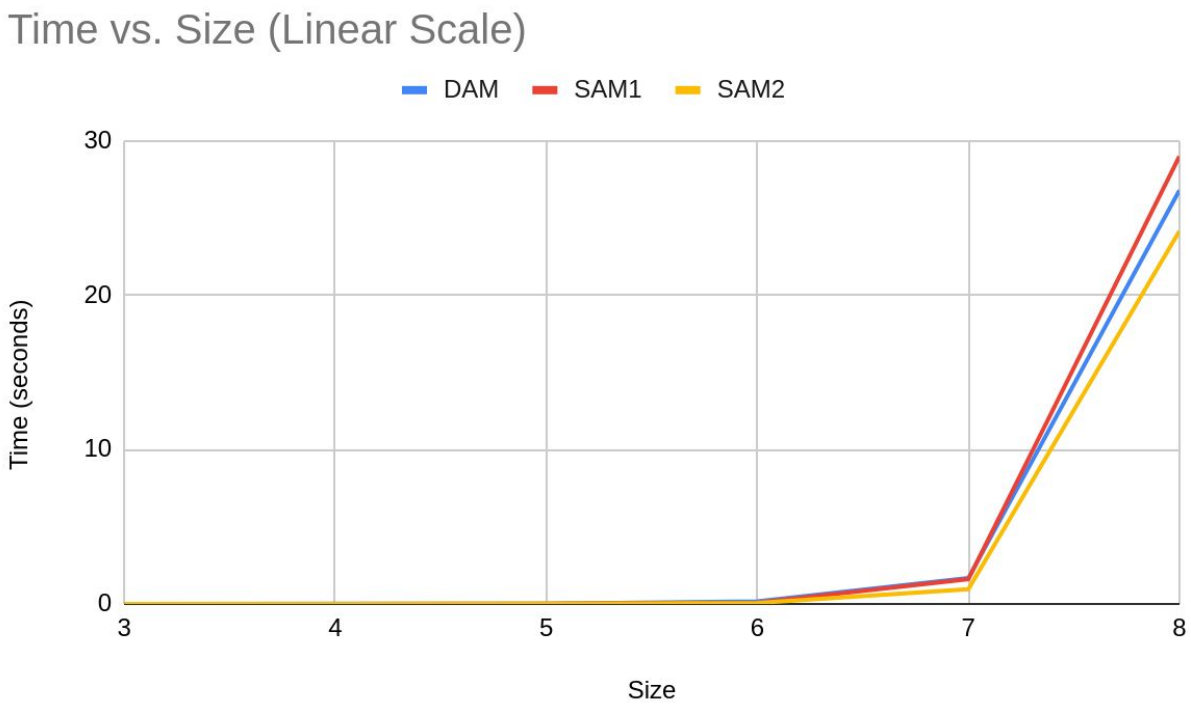
```
// th_id is the thread id (index 0 based)
// m is total number of threads
void primePrintSAM1(int th_id, int n, int m) {
  long i = 1 + th_id; // starting number
  long limit = pow(10, n);
  for (int j = i; j <= limit; j += m) {
      if(isPrime(j))
        print_sam1(j);
  }
}
```

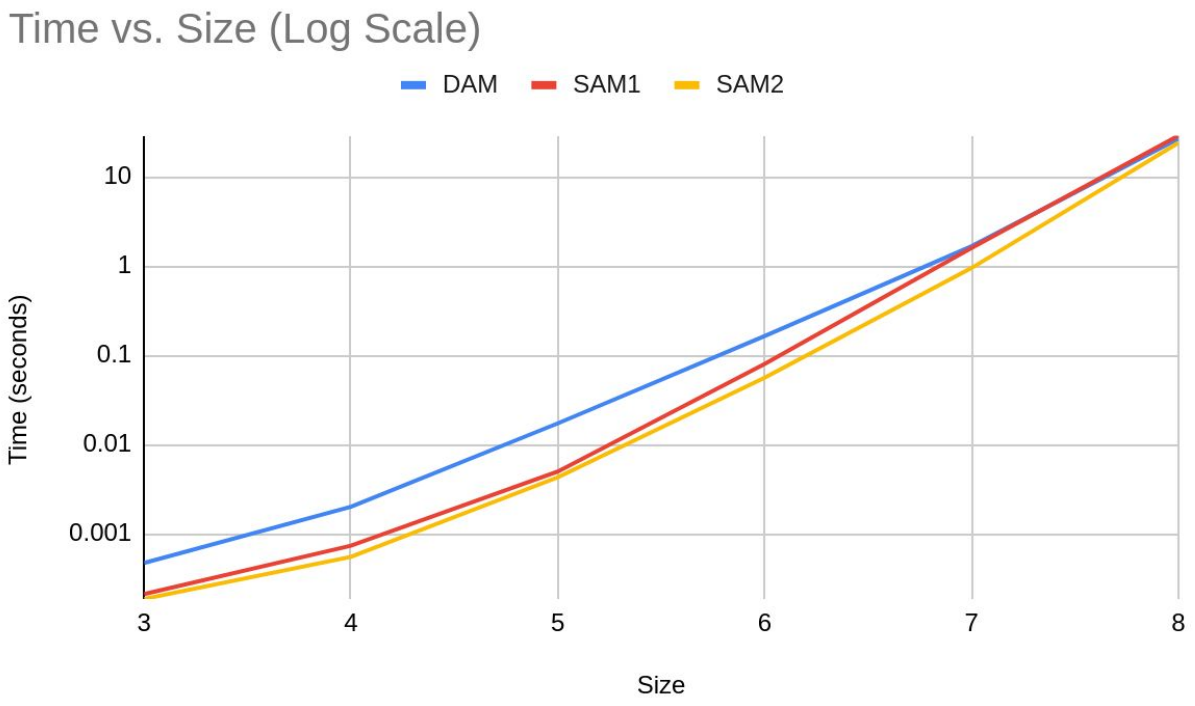
SAM2

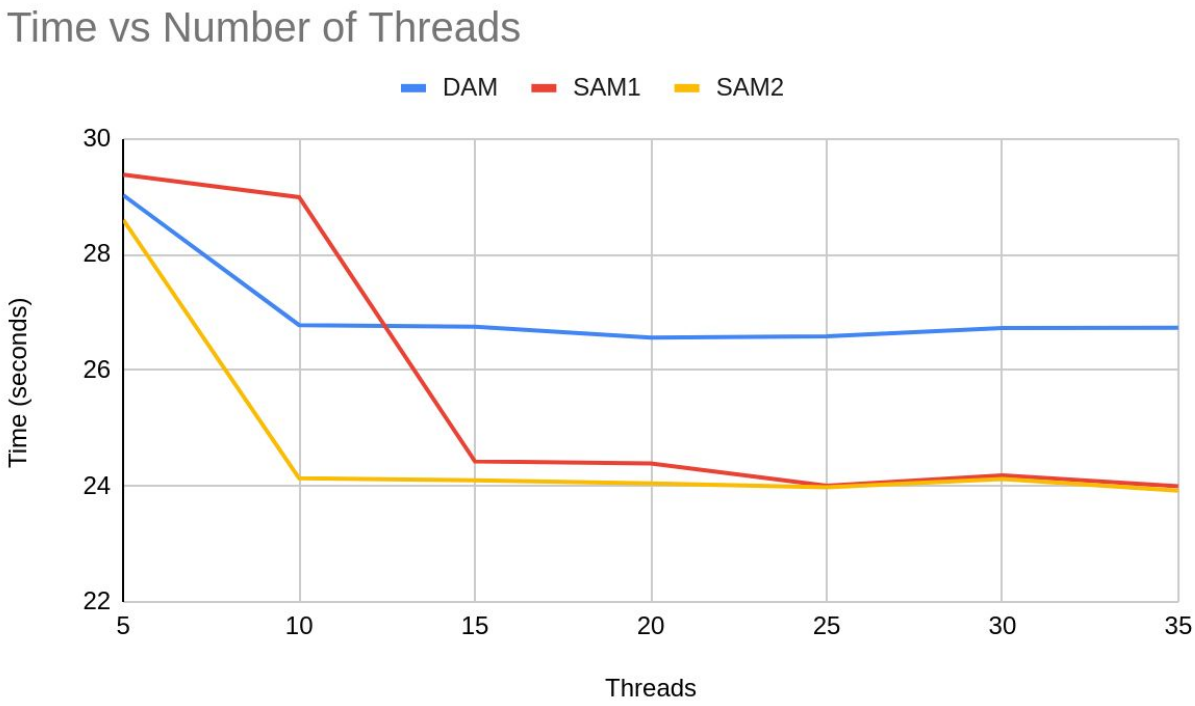
```
// th_id is the thread id (index 0 based)
// m is total number of threads
void primePrintSAM2(int th_id, int n, int m) {
  th_id += 1; // starting number
  long i = 2*th_id - 1;
  long limit = pow(10, n);
  int j = i; // starting point for all threads
  if(th_id == 1) { // we know that 1 is not prime but 2 is. so 1st thread
                   // starts with 2, rest from odd numbers(>= 3)
      if(isPrime(2))
        print_sam2(2);
      j = 1 + 2*m; // for first thread, next number will be 1 + 2m
  }
  for (; j <= limit; j += 2*m) {//m odd numbers handled by other threads
                                //m-1 even numbers are skipped
      if(isPrime(j))
        print_sam2(j);
  }
```

```
}
```

GRAPHS


Time vs. Size (Linear Scale)
DAM
SAM1
SAM2
30
20
10
0
Time (seconds)
3
4
5
6
7
8
Size


Time vs. Size (Log Scale)
DAM
SAM1
SAM2
10
1
0.1
0.01
0.001
Time (seconds)
3
4
5
6
7
8
Size


Time vs Number of Threads
DAM
SAM1
SAM2
30
28
26
24
22
Time (seconds)
5
10
15
20
25
30
35
Threads

OBSERVATIONS

- Since, as we can see with the linear scale of time vs size we can't really make out any inferences except the last test case as the values are too close.
- Only the last test case(in time vs size) seems to tell us some significant difference, rest are almost close. Their values might also be affected due to the context switches at that time.
- In the log scale of time vs size, it seems that DAM takes more time than that of SAM1 and SAM2. This can be explained due to the fact that on every getAndIncrement() call, each thread has to wait for any thread which is already in the CS to leave before getting a chance to enter. Also, there is no guarantee that mutex is fair, so other threads might end up waiting longer than expected. Essentially, the mutex here is a bottleneck.
- In the above inference, we see that, as size increases, this disadvantage is of less importance, this is because, as size increases, each thread has more work to do, so they get stuck waiting on mutex lock as often the times with less size.
- When the number of threads is even, even indexed threads will always get an even number and their workload is evaluated much faster than odd indexed threads. This observation is less obvious for a smaller size but for larger size, this is more noticeable. In this case, I would say DAM would perform better as it does a much better job at evenly distributing the work.

- SAM2 is obviously better than SAM1 but for smaller sizes, the advantage SAM2 has over SAM1 is very minute as the even numbers in the range are very less.
- As the size increases, it is more clear that the time taken by SAM2 is less than that of SAM1 as it should be. As it is already proven above that SAM2 is a better-optimized algorithm than SAM1.
- At this point, it is clear that SAM2 will perform in our observation range as its workload is not as big as that of DAM and SAM1.
- We can see that time taken has reduced significantly when the number of threads is increased from 5 to 10. Personally, for me, the number of cores per socket(1) is 4, and with hyperthreading enabled, the total number of CPUs is 8. Hence, we can see a significant decrease in time when threads are increased from 5 to 10.
- From 10 to 35, even though we increase the number of threads, we don't notice much difference. This is because in my case, the total number of CPUs is 8. Hence, Threads more than 8 won't show much improvement.
- But in the above observation, there is one anomaly when m = 10. This can be explained by the fact(as explained above) that when the number of threads is even, even indexed threads will always get an even number and their workload is evaluated much faster than odd indexed threads. So SAM1 does not do a good job of fully utilizing the threads when compared to DAM, in this case, hence the time taken by DAM, in this case, is less than SAM1.