Parallel Concurrent Programming: Fall 2023

—

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

Souvik Sarkar(CS23MTECH02001)

Report

Objective:

The goal of this assignment is to implement different parallel implementations for iden- tifying prime numbers and compare their performance. Implement these algorithms in C++.

Implementation:

Majority of the function in both SAM1 and DAM1 are same. But ***checkPrimeWithCounter function()*** is exclusive for DAM1 and ***checkPrimeRange function()*** is exclusive for SAM1.

SAM1 **:**

Below is the code snippet for the `checkPrimeInRange()` function:

```
void checkPrimeInRange(int n, int threadId, int numThreads) {
    for (int numToCheck = 1 + threadId; numToCheck < n; numToCheck += numThreads) {
        if (numToCheck % 2 == 0)
            continue; // Skip even numbers except for 2
        if (isPrime(numToCheck)) {
            mtx.lock();
            fprintf(log_file, "%d(Tid:%d) \n", numToCheck, threadId);
            mtx.unlock();
        }
    }
}
```

The `checkPrimeInRange` function is a critical component within a multi-threaded program dedicated to assessing numbers for their primality. Each thread is distinguished by a unique identifier (`threadId`) and focuses on evaluating a specific range defined by the parameters `start` and `end`. The function operates through the following steps:

1. A loop iterates over numbers starting from `start` and continuing up to `end`.
2. For each number, the function employs the `isPrime` function to ascertain whether the number is prime.
3. In cases where a prime number is identified, the corresponding thread obtains exclusive access to a mutex (`mtx`). This mutex ensures that concurrent threads don't encounter conflicts while performing logging operations.
4. Subsequently, the thread records the prime number and its associated `threadId` to a log file using the `fprintf` function. The format utilized for logging is `"%d(Tid:%d)"` to denote the prime number and thread identifier.

Significantly, the `start` and `end` parameters define the range of numbers that each thread is responsible for evaluating. This strategy enables multiple threads to concurrently examine potential prime numbers within their assigned ranges, thereby harnessing parallelism to potentially enhance program efficiency.

DAM1 **:**

Below is the code snippet for the `checkPrimeWithCounter()` function:

```
void checkPrimeWithCounter(int num, int threadId) {
    while (true) {
        int current = counter.fetch_add(1);
        if (current > num) {
            break;
        }
        if (isPrime(current)) {
            mtx.lock();
            fprintf(log_file, "%d(Tid:%d) \n", current, threadId);
            mtx.unlock();
        }
    }
}
```

The `checkPrimeWithCounter` function is a pivotal component within a multi-threaded program designed to identify and log prime numbers within a specified numerical range. In the context of this program, multiple threads execute this function concurrently, each carrying a unique thread identifier (`threadId`). The function's core steps can be outlined as follows:

1. Utilizing an atomic operation, the function increments a shared counter, synchronously across all threads. The value returned by this operation is stored in the variable `current`.

2. The value of `current` is compared against the upper limit (`num`) that signifies the endpoint of the prime number search. If `current` surpasses this threshold, the loop terminates, indicating that all pertinent numbers have been inspected.

3. When `current` is recognized as a prime number, the corresponding thread employs a mutex (`mtx`) to ensure exclusive access to a logging mechanism. Consequently, the thread's discovery is meticulously documented within the program's log file. The prime number itself and its associated `threadId` are recorded using the `fprintf` function.

This concurrent approach leverages parallelism to efficiently explore potential prime numbers. It's crucial to emphasize that the mutex guarantees proper synchronization and mitigates potential conflicts during the process of logging prime numbers from diverse threads.

Comparison 1:

Test Conditions: i)No. of Threads=10. ii) n= 3,4...,8.

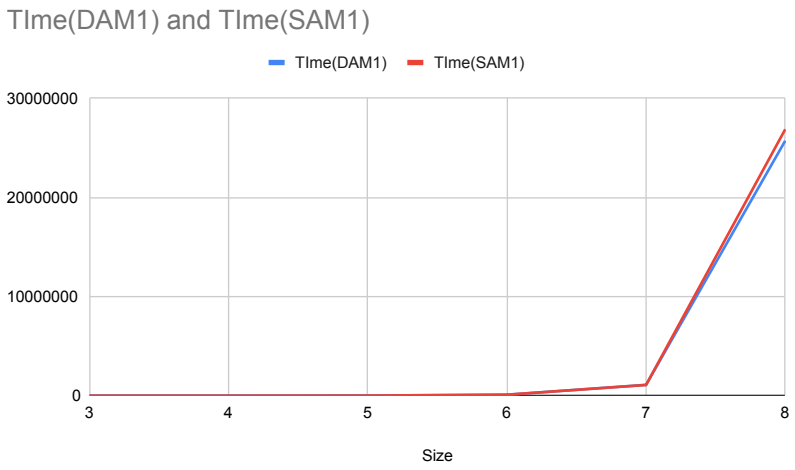

Figure 1: Time Vs Size of input

Comparison 2:

Test Conditions: i)No. of Threads=5,10,...35. ii) n= 7(N=10^7).

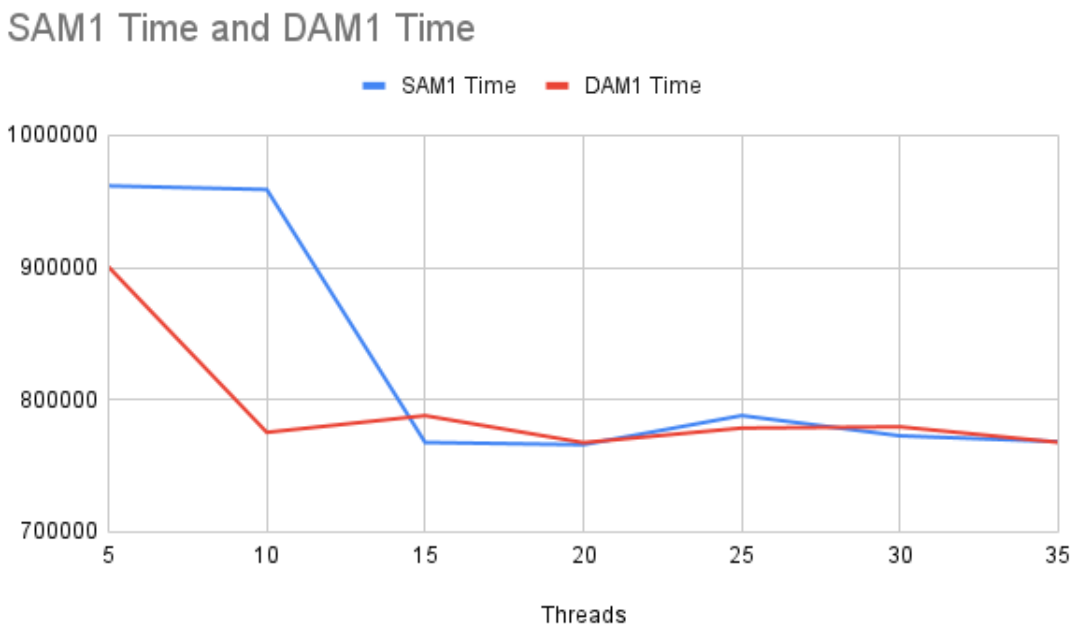

Figure 2: Time vs No. of Threads

Observation:

According to my implementation DAM1 perform better than SAM1 where N is very large (likely n>7). The main reason is that in SAM1 where we allocate a predefined range to each thread , though the numbers in each range is same. But the higher range thread is doing lot more work than the lower threads. So basically the workload balancing is not same across the threads.Where as in DAM1 we are dynamically allocating numbers to the threads so the load balancing is linear across all the threads.In my code I used mutex to lock the output file so that all the threads can access it with proper synchronization.