CS5300 - Parallel & Concurrent Programming

Comparing Different Parallel Implementations for Identifying Prime Numbers

Assignment Report

Sagar Jain - CS17BTECH11034

September 25, 2020

Contents

Program Design

The program calls the three algorithms DAM, SAM1 & SAM2, sequentially and records the time taken for each of them. Elements common to all the three algorithms are:

- `pthreads` have been used for multithreading.
- `chrono` had been used to record time.
- Conditional compilation has been used for writing to output files to avoid taking printing time into consideration when calculating the time taken for the algorithms.
- The same method (`isPrime`) has been used to check primality for all the algorithms to ensure the differences in times are purely based on the load distribution.
- The `isPrime` function uses a precomputed list of all the primes till $\sqrt{n}$ and for any given input `k` checks if there is any prime less than `k` in the list which divides it.

Dynamic Allocation Method

The following is the description of the DAM algorithm:

- The method for the DAM algorithm is `dam`.
- This method creates and launches `M` threads which execute the `dam_thread` method.
- The `dam_thread` method uses the `getAndIncrement` method for an instance of the `Counter` class to be allocated numbers to check for primality.
- Based on the test it either writes to the output file or continues looping until finally being allocated a value greater than 10^N.
- The `Counter` class uses a `mutex` lock to make the `getAndIncrement` function mutually exclusive.

Static Allocation Method I

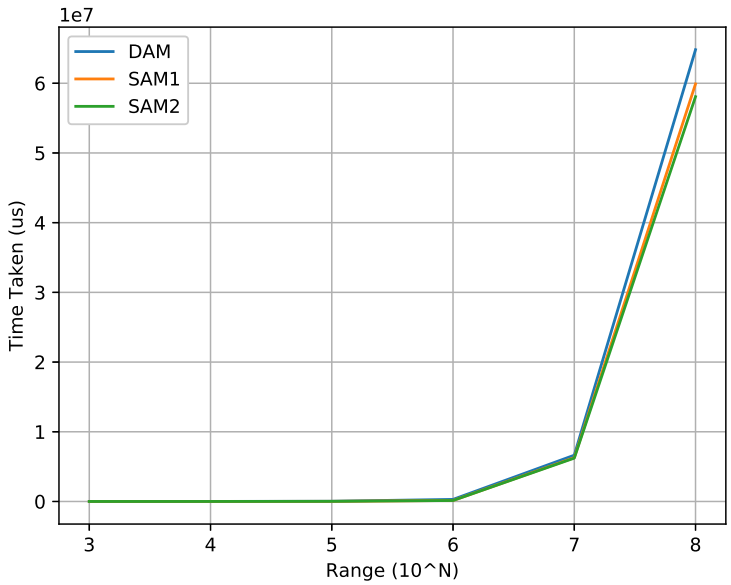
- The method for the SAM I algorithm is `sam1`.
- This method creates and launches `M` threads and gives an index to each one of them between `1` to `M`. These threads execute the method `sam1_thread`.
- The `sam1_thread` simply checks the primality of all the numbers $M * i + index \; \forall i \; \in [0, N/M]$. Here, `index` is allocated to threads when they are launched.

Static Allocation Method II

- The method for the SAM II algorithm is `sam2`.
- This method creates and launches `M` threads and gives an index to each one of them between `1` to `M`. These threads execute the method `sam2_thread`.
- To avoid checking for even numbers, the optimisations used here is that we distribute the odd numbers evenly among the threads. To do this we use the same loop as in `SAM I` but instead of checking the primality of the $M * i + index$ we check for the primality of $2 * (M * i + index) - 1$. In simple words, in `SAM I` we were checking for the primality of the i^{th} number and in `SAM II` we are checking for the primality of the i^{th} odd number.
- **Point to Note**: The distribution of odd numbers between the threads is fair. Since we use the index to choose the odd numbers, we ensure that every thread gets only one odd number between the $M * i^{th}$ odd number and $M * (i + 1)^{th}$ odd number.

Results & Graphs

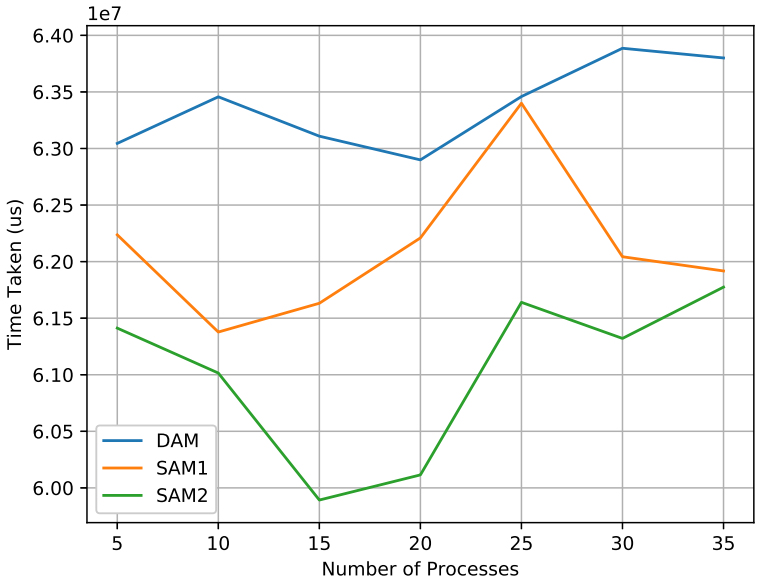
Time Taken vs Value of N



Inference

- Since the x axis of the graph is in logarithmic scale we cannot appreciate the difference in the curves for the lower values of N.
- We expect `DAM` to take the longest time since dynamically allocating numbers requires the threads to wait for aquiring the locks. `SAM II` uses the optimsation of ignoring the even numbers so it is expected to perform better than `SAM I`.
- For values 3, 4, 5 the time taken by `DAM` was 2 to 3 times that of `SAM I` which was 1.5 to 2 times `SAM II`.
- As the value of N increases, we see that the relation of time taken by `DAM > SAM I > SAM II` is still according to expectation.
- Asymptotically, the complexity of all the three algorithms is the same so it is not surprising that for large values of N, the time taken is in the same order of magnitude.

Time Taken vs Number Of Threads



Inference

- This graph would be differ based on the system where the code was run, since increasing the number of threads is only beneficial until the number of threads launched is $<$ number of hardware threads available.
- Firstly we can see that we still have `DAM > SAM I > SAM II`. This is as expected.
- We also notice that the graphs of all the three algorithms have a U shape. Which essentially tells that initially increasing the number of threads led to lower time taken but not on increasing beyong 15.
- The above observation can be explained by the fact that I have an 8 core system with 2 threads in each core, which makes it a total of 16 hardware threads. So before we hit 16 threads we notice improvement in the time taken when increasing the number of threads. But after that increasing the number of threads would lead to more number of context switches on the same physical thread this would lead to more time taken and can be seen in the graph.