# Comparing Different Parallel Implementations for Identifying Prime Numbers

## Design

- Threads created each perform a piece of task depending on the type of program.
- Each thread stores its result in its own vector, all the results from separate threads are then combined to get the final result. We don't use a common data structure to store all the results as then this common data structure would also be needed to be protected from concurrent access and that would hinder the performance of the algorithm.
- The final result is sorted and the prime numbers are output in a file.

**DAM**

- The shared counter is protected from concurrent access by using a lock.
- First the counter value is read and checked if it equal to the value till which prime numbers are to calculated.
- It is ensured that only one thread at a time can access the shared counter variable.
- Only one lock is used in the entire program.

**SAM1**

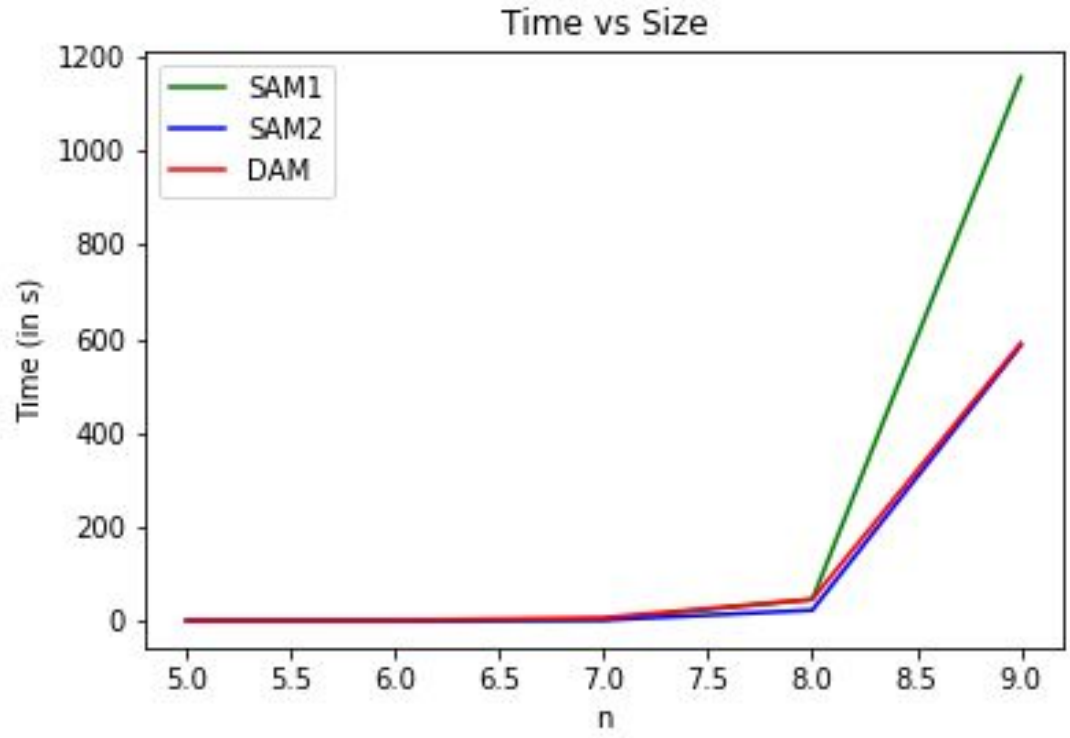
- Numbers 1, m + 1, 2 ∗ m + 1, ..... will be assigned to thread 1; Numbers 2, m + 2, 2 ∗ m + 2, ..... will be assigned to thread 2 and so on.

**SAM2**

- The design is similar to SAM1 but only odd numbers are given to the threads in this case.
- Consider: 1, 3, ….. 2*r + 1, these are the numbers we need to check for primality, here 2*r+1<N or r<(N-1)/2
- Now consider the series of values which r can take : 0, 1, … ceil((N-1)/2).
- Now apply the same logic as SAM1 to the above series to divide the work among the threads and check the primality for 2*r + 1 in each thread.

★ Note: No locks were used in the implementation of SAM1, SAM2

# Graphs and Analysis



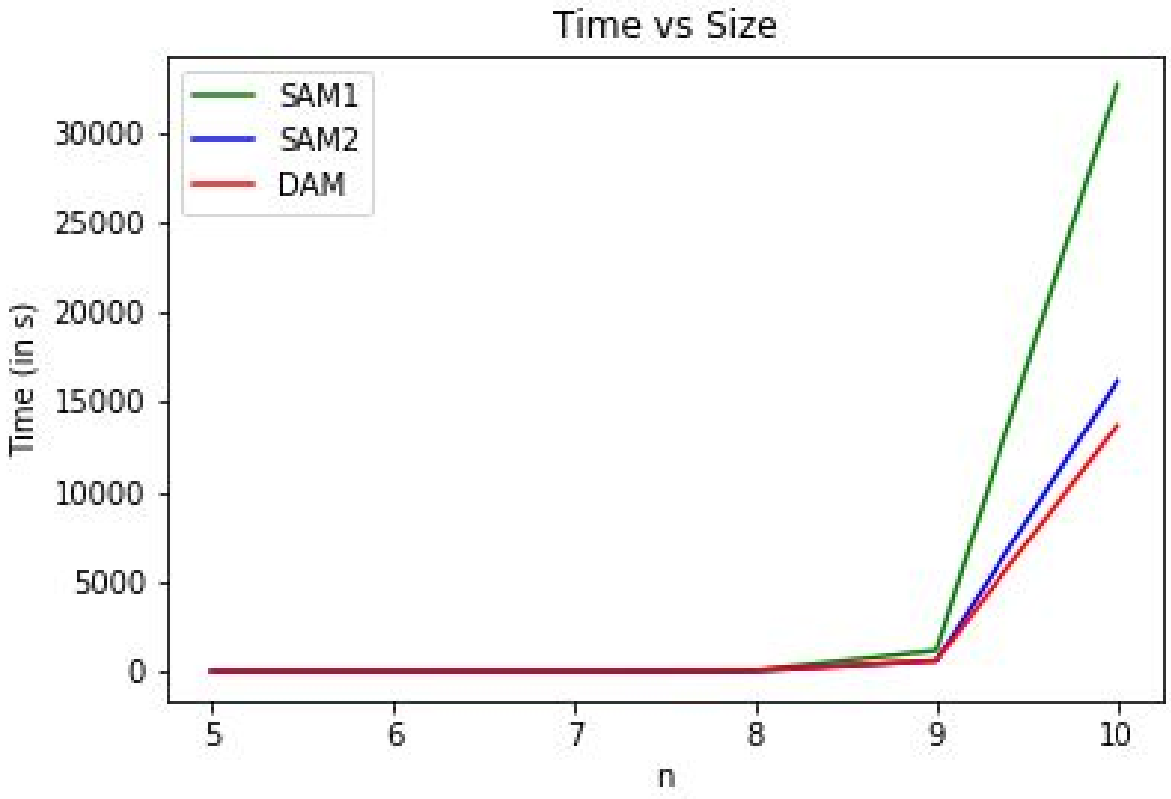

★ For the above graph m = 10 threads were used in a 28 core machine
★ An average of 3 reading were taken to compute the graph

- It can be clearly seen that DAM, SAM2 outperform SAM1, this is because SAM1 do not efficiently distribute the work across the threads and a lot of threads complete their task very quickly and exit.
  For example: N = $10^9$, m = 10
  Thread with ids 0, 2, 4, 5, 6, 8 complete the task very quickly as no primes(except 2, 5 where only 1 prime is present, i.e, 2 & 5 respectively) are present in their respective sets and they exit very soon.
- SAM2 and DAM seems to perform equally well suggesting that the lock used to protect the counter variable does not degrade the performance, this can be explained by the fact that less time is spent by the threads waiting for the lock and more time is spent by them to check whether a number is prime, which relatively takes a lot of time for higher numbers compared to waiting for the lock.

- For N<=$10^8$, m=10
    - Performance is SAM2>SAM1>DAM
    - DAM performs bad here because it needs to wait for the lock frequently as the amount of time spent to check whether a number is prime is very low, SAM1 and SAM2 don't have any locks and hence complete the entire task quicker.

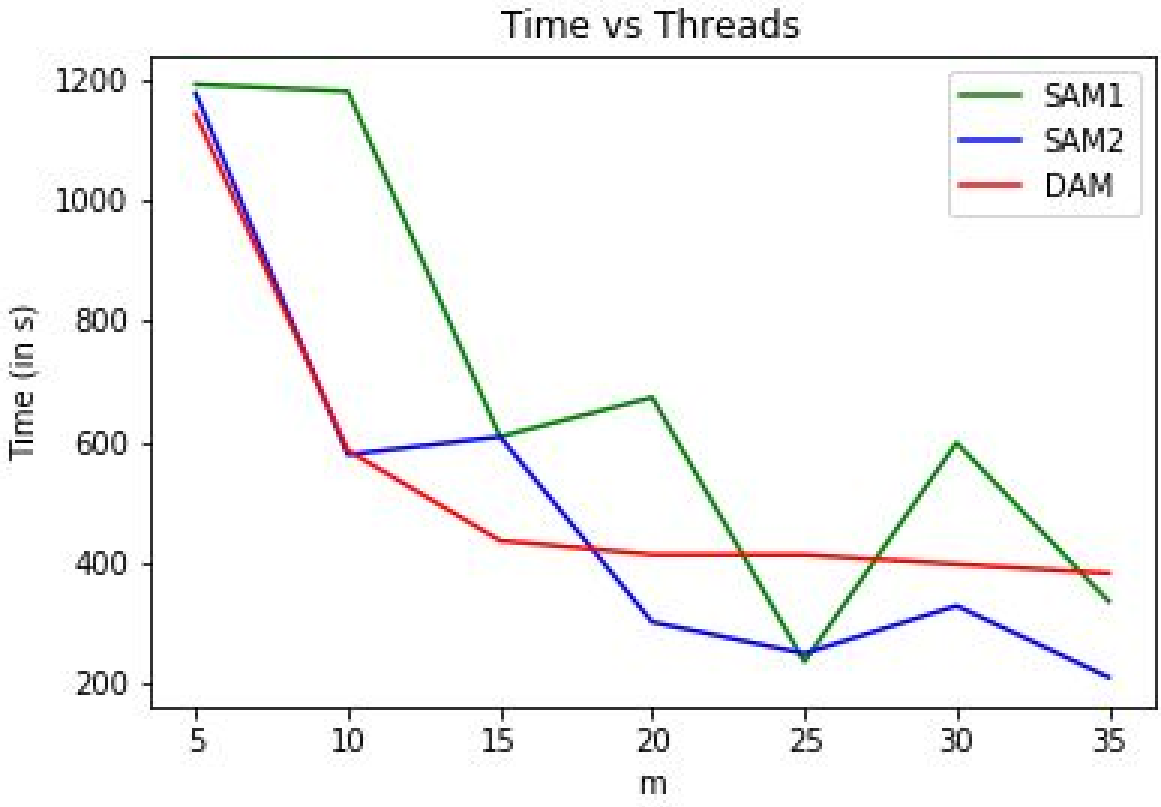
★ Note: The bottleneck here is the function in which the primes are calculated, therefore we see almost the same performance in low values of N.

★ Note: SAM2 performs better because only odd numbers are given to the threads to work on and there is a very good probability (this depends on N chosen) that all the threads are busy all the time and no thread completes faster than the remaining threads.




- It can be seen that for even higher values of N, DAM outperforms even SAM2.
- This is expected, as suggested earlier more time is spent by the threads to check whether a number is prime for higher numbers and more number of threads doing that work helps.
- But even in SAM2 few threads complete their task very fast due to unfair distribution of the work, this number is better than SAM1 atleast as was

discussed before, whereas in DAM all the threads do work till the last and no thread completes their entire task way faster than others.




★ For the above graph N = $10^9$ was used in a 28 core machine
★ An average of 3 reading were taken to compute the graph

- SAM2, DAM outperform SAM1 in most of the cases, this can be explained by the fact that not all threads are given equal work in SAM1 and since even numbers are present in SAM1, few threads might end very soon since they have very few primes in their computational set.
- SAM2 performs better than DAM for higher values of m, this is explained by the fact that only odd numbers are present in the computational set of the threads in SAM2 which is statically allocated beforehand, and most of the threads do useful work till the completion of the program without having to wait in a spin lock as in DAM.

- For m = 25, N = $10^9$
    - SAM1 performs as good as SAM2, this can be explained by that, in this scenario very few threads might have ended early due to less number of primes in their computation set, while most of the threads were doing useful work.

- We see almost a constant time after m = 25, due to the maximum resources available with us being 28 cores for DAM, the same performance statement cannot be made for SAM1, SAM2 due to the uneven distribution of work being made to the threads according to the algorithm, we note here that the extra time spent due to context switching and waiting for the locks after m = 25 compensates for the speedup due to extra number of threads working for DAM.

★ Note: $N=10^9$ was used instead of $N=10^{10}$ as it was taking a combined ~12 - 17 hours per m for the 3 programs to run on a 28 core machine for $N=10^{10}$.