Comparing Different Parallel Implementations for Identifying Prime Numbers
CO21BTECH11004

In the output file, time is printed in microseconds which is the difference between the time measured at the instant and the start time.

Design

- A global array of size N+1 (10^n) is created and initialized to 0.
- Threads check the number assigned to them, and if it is prime, then update that index in the array to 1.
- A particular thread updates a specific index in the array. Hence no two threads will update the same index in the array, so there is no need for a lock for updating the array.
- At last, if the ith element of the array is 1, print i in the output file as it is prime.

SAM

- The numbers are divided in the following order:
    - T_0:- 1, 2, 1+2m, 1+4m, …
    - T_1:- 3, 3+2m, 3+4m, …
    - T_i:- 2*i+1, 2*i+1+2m, 2*i+1+4m,...
- Here odd numbers are divided, and only 2 is checked by thread 0, as we can reduce the task by not checking even numbers.
- Threads check the number assigned; if it is prime, it updates the same in the global array.

DAM

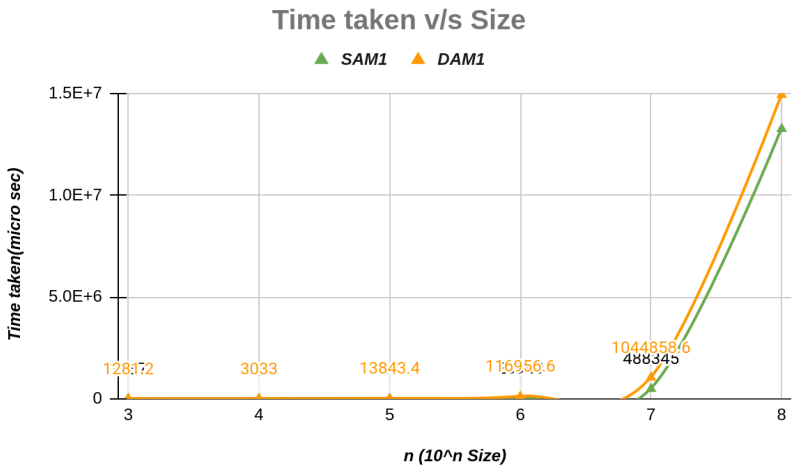
- Each thread calls the increment function with the argument global int idx to get the number to check whether it is prime.
- Lock must be acquired before calling the increment function and released afterward.
- Increment function returns (idx+1) and is called once if idx<=2; otherwise, 2 times. We must check 2 for prime and avoid all even numbers after that.
- Threads check the number assigned; if it is prime, it updates the same in the global array.

Implementation

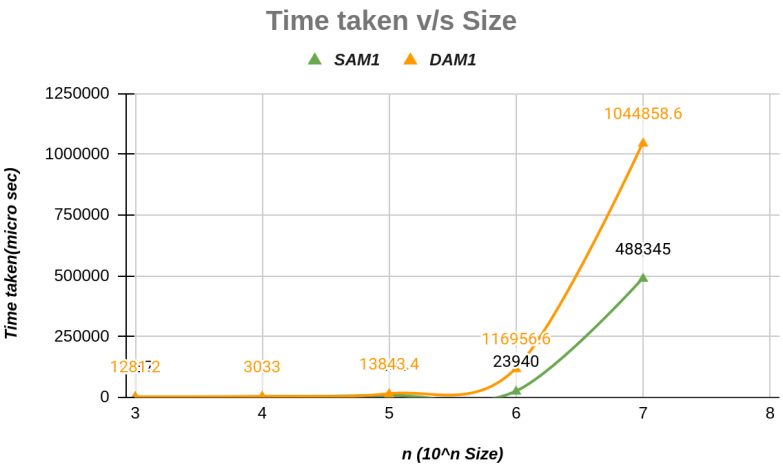
- Global variables n, N, m are created. Also, add idx in DAM.
- n and m are read from the input file, and N = 10^n is calculated.
- Global array arr is dynamically allocated size N+1 to keep track of the prime numbers.
- C++ threads are used, created m threads, which call the threadFunc with the argument thread number.
- Each thread checks numbers allocated by SAM/DAM and updates the same in the global arr.
- Time is stated before the creation of threads and stopped after all threads join.
- Then, all the prime numbers are printed in the output file.
- Files are closed, and the global array is free.

Graphs

Time vs. Size



- For both SAM1 and DAM1, time increases as size (n) increases as more numbers have to be checked, so the computation task increases, which is reflected in time.

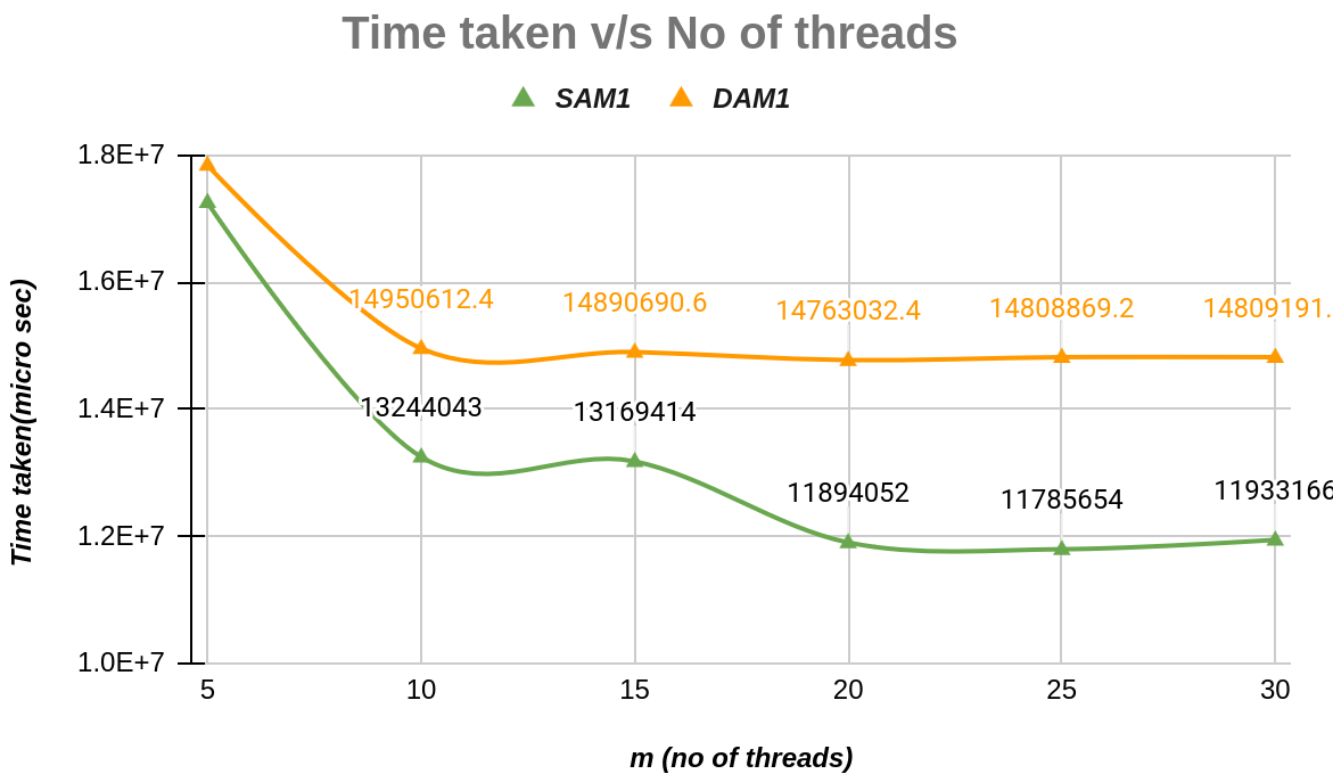


- The time taken by DAM1 is greater than SAM1. Each time a thread asks for the number using lock (only one thread can access that) in DAM1, it increases the time compared to SAM1, where numbers are already allocated to the thread, and no lock is used.

(SAM1) n	Time_taken in each iteration (microseconds)					Time average
	1st	2nd	3rd	4th	5th	
3	1279	1446	1170	1115	1177	1237
4	1512	1426	1356	1426	1621	1468
5	2766	2487	26146	2901	3121	7484
6	24055	25075	22700	24289	23579	23940
7	489532	488283	496170	480791	486950	488345
8	13177192	13383129	13113681	13272131	13314127	13252052

(DAM1) n	Time_taken in each iteration (microseconds)					Time average
	1st	2nd	3rd	4th	5th	
3	1011	1531	1346	1093	1425	1281.2
4	3151	2899	3290	3094	2731	3033
5	13921	14123	13812	13571	13790	13843.4
6	116015	116846	115994	117872	118056	116956.6
7	1033222	1056440	1034680	1055273	1044678	1044858.6
8	14908476	14967422	14913128	14942791	14863842	14919131.8

Time vs. No of Threads



- The time taken for both SAM1 and DAM1 decreases as no of threads increases and then become constant above a certain threshold and slightly starts increasing.
- For SAM1, the high decrease was from 5 to 10, then 10 to 15, then 15 to 20. From 20-25, a slight decline was there and from 25-30 a slight increase in time. This is because context switching time increases as threads exceed the machine's thread.
- For DAM1 significant decrease in time is from 5 to 10, then time remains almost constant and starts slightly increasing. Here, acquiring the lock and getting the number to check for each thread is overhead. Also, context switching time increases as threads exceed the machine's thread.

(SAM1) m	Time_taken in each iteration (microseconds)					Time average
	1st	2nd	3rd	4th	5th	
5	17234913	17228083	17222649	17264036	17281230	17246182
10	13030493	13456931	13219573	13245181	13268039	13244043
15	13175545	13132432	13338410	13047659	13153025	13169414
20	11505131	12184968	11599595	12170038	12010526	11894052
25	11583049	11939636	11759652	11818568	11827363	11785654
30	11858649	11668148	12002538	11936693	12199804	11933166
35	11336394	11458789	11713691	11757872	11799796	11613308

(DAM1) m	Time_taken in each iteration (microseconds)					Time average
	1st	2nd	3rd	4th	5th	
5	17711227	18107731	17869503	17853663	17627854	17833995.6
10	14865321	14955240	14956551	14973058	15002892	14950612.4
15	14837237	15161246	14854579	14813503	14786888	14890690.6
20	14751485	14765527	14760925	14772210	14765015	14763032.4
25	14778006	14762761	14782977	14807620	14912982	14808869.2
30	14765267	14775252	14849948	14825867	14829625	14809191.8
35	14794437	14778537	14753146	14822308	14743713	14778428.2