Universidade do Porto

FEUP Faculdade de Engenharia

Implementation and Performance Analysis of “The Sieve of Erastosthenes”

Computação Paralela

- Luís Magalhães, 201207224
- Miguel Mendes, 201105535

21/05/2016

Problem description

The Sieve of Erastothenes is an algorithm targeted towards the calculation of all prime numbers until a designed number. The way the algorithm works is through the marking of numbers, resulting in a list of marked numbers which should all be prime. Like a sieve, the algorithm filters numbers which are not prime. Beginning with 2, the algorithm will unmark all of its multiples as non-primes, repeating the same process on all following numbers still considered prime, until all numbers up until the desired target have been processed and deemed to be prime or not.

The algorithm would already seem simple enough. However, there are still some improvements we can put in practice, merely at a sequential point of view. Firstly, it is enough to find the multiples of numbers up until the square root of the target limit, since all primes will have been found by then. Other possible improvements, not undertaken by us, would be to not even consider even numbers, immediately cutting a great portion of required processing, and only finding the multiples of a number k from k^2 to the square root of the target number, also cutting part of the processing.

Sequential solution and performance

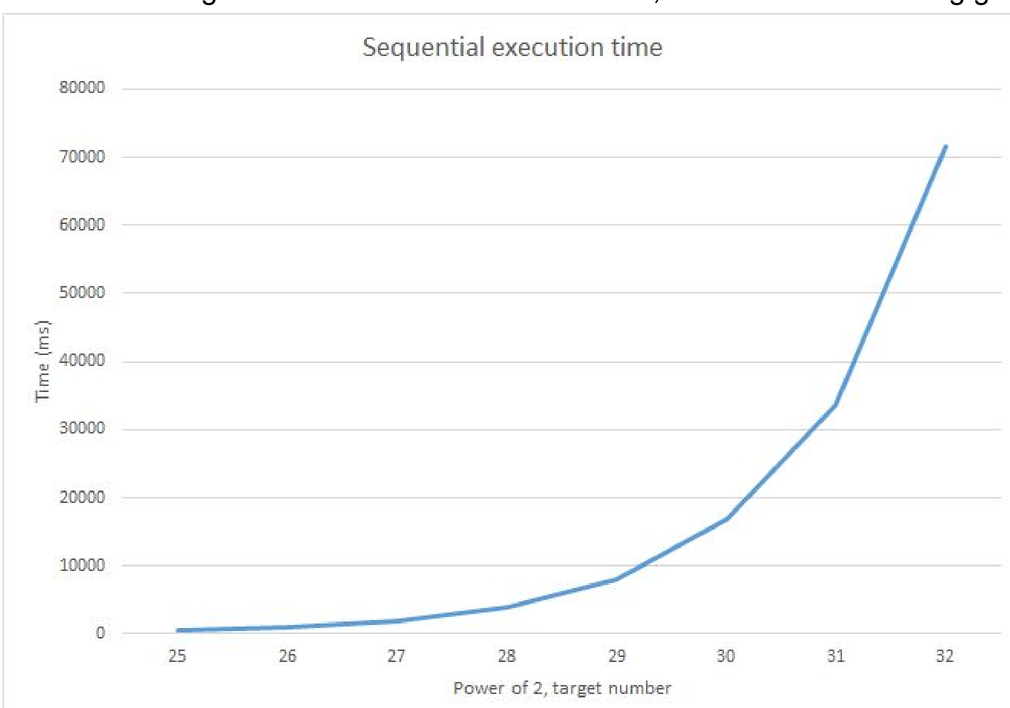
The algorithm employed as a sequential solution processes the multiples of all numbers from 2 to the square root of the target limit. Executing the algorithm 10 times for each limit, we obtained the raw data shown below:

	Sequential Algorithm			
N (2^N)	**25**	**26**	**27**	**28**
Time (ms)	472.138309	922.852171	1929.099992	3932.010815
	462.749537	916.313622	1958.032801	4019.303101
	467.583614	928.449504	1934.295601	4084.223779
	432.343258	916.470529	1852.478192	4087.176066
	454.080835	918.401030	1934.026180	3841.817794
	465.448588	920.644810	1839.997572	3862.499377
	463.752678	885.149346	1932.638135	3890.404469
	426.054753	903.417479	1938.188062	3892.864358

	464.957098	902.533738	1841.988534	3886.406793
	449.385847	904.566763	1886.089708	3830.084269
N (2^N)	**29**	**30**	**31**	**32**
Time (ms)	7927.876312	17294.195033	33656.667568	73807.547654
	7922.583345	17282.166326	33679.778216	73728.803573
	7933.008224	17289.459440	33663.840834	73782.342069
	7931.161656	17286.519150	33678.367685	73922.569076
	8032.511167	16544.686001	33692.839355	69220.883454
	8025.275018	16586.825056	33662.067644	69221.213607
	8050.140609	16349.366047	33801.387780	70400.102425
	8330.634766	16352.688812	33679.651282	70443.557973
	8377.413456	16343.404375	33676.946370	70526.975239
	8487.647993	16342.585522	33673.992848	70497.488087

Table 1. Raw data from sequential algorithm, going from power 25 to 32.

Making use of the average of execution times for each limit, we obtain the following graph:



Graph 1. Representation of the times obtained in the sequential algorithm

Parallel solutions

For a parallel execution of the solution and improvement of execution times, and therefore faster results, two approaches were undertaken.

Firstly, using OpenMP, a parallel version of the algorithm applied in the sequential solution was developed, which utilizes the intended numbers of threads, defined by user input, having it applied to 4 and 8 threads. For each number n for which to process its possible multiples, we divide those between every and each thread.

Secondly, through MPI, another parallel version of the algorithm is used, this time taking into account the availability of even more processor cores, due to the system now being a node cluster, capable of more processing power, depending on the number of used nodes. Again, the possible multiples of a number n to be processed are divided between all the available cores.

Parallel algorithm

The approach taken to divide the problem in several threads/processes (will refer as process, from now on) is the consideration that, for each *i* ($i \in [2, k]$, *k* being the limit for one given iteration), all the multiples of *i* can be divided [approximately] equally by each process into blocks, so that each process gets his “share” of multiples to consider. The down side for this process is that, for the distributed case, one of the processes (in this particular case, the process calculating from 2 to sqrt(*k*)) must broadcast the next sieving prime (*i*) to the others.

Time measurements

As with the sequential solution, we measured execution time for each power of 2 limit 10 times, for a better estimate. The gathered data is as follows (see next page).

	OpenMP Parallel (4 threads)			
N (2^N)	**25**	**26**	**27**	**28**
Time (ms)	220.092923	446.065350	938.620427	1957.795083
	220.669981	445.236748	938.211023	1958.843377
	213.230944	441.674592	942.201556	1952.577268
	216.730602	441.941265	949.535063	1956.194263
	215.333096	443.075426	941.842458	1955.528762
	226.165511	446.401907	929.333798	1953.699995

	205.628660	441.123881	934.836731	1957.687114
	206.044603	448.157616	927.888012	1960.026914
	208.784748	443.618282	936.865023	1962.182714
	205.154325	436.278881	939.292429	2097.232180
N (2^N)	**29**	**30**	**31**	**32**
Time (ms)	4424.468899	8992.170602	18138.387717	35673.941062
	4410.136417	8394.959626	17318.760950	35676.477885
	4395.208718	8392.380123	17438.761949	35756.894695
	4379.041898	8382.529945	17345.299189	35693.895689
	4404.740548	8390.568832	17352.803497	35842.653339
	4318.018886	8405.881250	17359.223584	36196.698955
	4353.200619	8390.329267	17495.966445	36490.518385
	4359.176601	8776.943530	17732.449177	35699.865488
	4368.452122	8352.601516	17333.691170	35629.034938
	4353..787584	8386.140102	17321.961196	37003.578359

Table 2. Time measurement for the parallel version of the algorithm, using OpenMP with 4 threads, from power 25 to 32.

	OpenMP Parallel (8 threads)			
N (2^N)	**25**	**26**	**27**	**28**
Time (ms)	205.997601	434.801363	991.777593	2098.515263
	197.930595	444.931981	1006.044723	1952.262483
	206.194396	444.187672	998.503208	1951.311683
	198.239838	445.347631	992.922246	2085.796990
	204.486928	440.195027	998.490601	2140.141323
	198.619704	476.566951	1005.847786	2084.806392
	198.949066	479.577974	938.956218	2102.678751
	196.486188	447.318598	939.099484	2014.820409

	196.175971	474.063734	941.294117	1939.544139
	198.334764	471.408805	939.565959	1932.902929
N (2^N)	**29**	**30**	**31**	**32**
Time (ms)	4026.971198	9070.967148	17301.558142	35873.124877
	4016.397171	8917.402656	17303.575694	35498.686090
	4016.049847	8919.674214	17678.359381	35536.292131
	4019.808515	8388.742639	18055.444190	35547.350306
	4040.284392	8369.976660	17287.588858	35916.490702
	4049.363183	8338.729925	17271.377286	35590.313636
	4112.986123	8327.785762	17289.863437	35435.125282
	4057.431546	8320.090457	18350.956473	35592.453529
	4415.140173	8384.170310	18391.312715	35950.702190
	4384.840022	8387.957228	18077.268224	35602.663761

Table 3. Time measurement for the parallel version of the algorithm, using OpenMP with 8 threads, from power 25 to 32.

	OpenMPI Distributed Parallel (8 threads, 2 machines)							
N (2^N)	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**
Time (ms)	309.638	664.693	1450.66	3016.00	6219.22	12912.6	26476.1	53700.3
	293.079	660.634	1446.56	2994.50	6267.80	12836.3	26444.5	54184.2
	290.979	663.911	1431.63	3030.63	6228.02	12889.9	26483.4	53901.4
	288.257	666.529	1431.86	3000.82	6260.37	12939.7	26606.3	54116.2
	284.092	659.290	1432.90	2989.80	6249.84	12877.4	26477.2	53778.2
	290.888	667.820	1415.25	2986.41	6226.43	12922.3	26529.0	53763.7
	291.690	683.155	1462.73	2984.70	6234.64	12822.0	26410.7	54346.4
	289.136	682.677	1459.16	2983.21	6213.05	12868.1	26368.3	53985.9
	291.272	672.468	1447.57	3002.31	6210.38	12829.3	26379.8	54190.5
	299.353	660.984	1432.31	2998.05	6228.42	12838.4	26442.2	53873.2

Table 4. Times for the distributed parallel version of the algorithm, using OpenMPI with 8 threads, from power 25 to 32.

	OpenMPI Distributed Parallel (16 threads, 2 machines)							
N (2^N)	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**
Time (ms)	188.633	429.275	927.093	2028.65	4282.01	8656.67	17749.5	36107.2
	158.613	417.733	955.853	2037.68	4224.79	8683.79	17750.5	36149.1
	182.987	434.647	948.496	2020.41	4247.20	8688.30	17790.7	37216.8
	158.756	421.436	977.476	2022.63	4210.88	8673.72	17741.2	35918.7
	158.441	406.240	958.047	2038.16	4223.64	8675.14	17720.3	36369.2
	169.292	422.488	954.891	2032.11	4241.25	8681.84	17774.5	36175.5
	157.686	425.863	959.516	2012.81	4247.12	8702.51	17768.9	36172.8
	161.561	411.499	967.682	2044.65	4235.31	8677.82	17760.5	36280.7
	202.744	436.155	954.835	2037.59	4225.41	8680.57	17772.4	36116.3
	164.996	415.526	956.462	2321.97	4208.49	8667.60	17771.2	36249.0

Table 5. Times for the distributed parallel version of the algorithm, using OpenMPI with 16 threads, from power 25 to 32.

	OpenMPI Distributed Parallel (24 threads, 3 machines)							
N (2^N)	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**
Time (ms)	94.0781	299.575	675.887	1455.95	3090.88	6362.48	13043.5	26576.3
	101.761	279.602	684.289	1445.75	3087.41	6368.67	13045.2	26563.9
	95.9890	286.283	680.990	1467.89	3070.58	6357.40	13050.5	26581.9
	92.9921	288.475	674.400	1463.83	3095.67	6358.32	13041.0	26570.3
	92.5081	303.378	663.986	1471.05	3102.44	6383.82	13039.3	26589.6
	88.0692	279.266	652.043	1467.04	3098.67	6372.06	13063.7	26539.7
	105.393	288.613	669.582	1464.81	3076.34	6376.23	13036.1	26567.7
	134.266	278.871	685.535	1479.03	3074.01	6379.06	13082.9	26537.3
	103.126	288.443	679.218	1480.55	3112.52	6384.31	13049.0	26600.3

	89.8621	283.646	664.544	1454.22	3087.24	6366.36	13031.5	26539.1

Table 6. Times for the distributed parallel version of the algorithm, using OpenMPI with 24 threads, from power 25 to 32.

Performance evaluation and scalability analysis

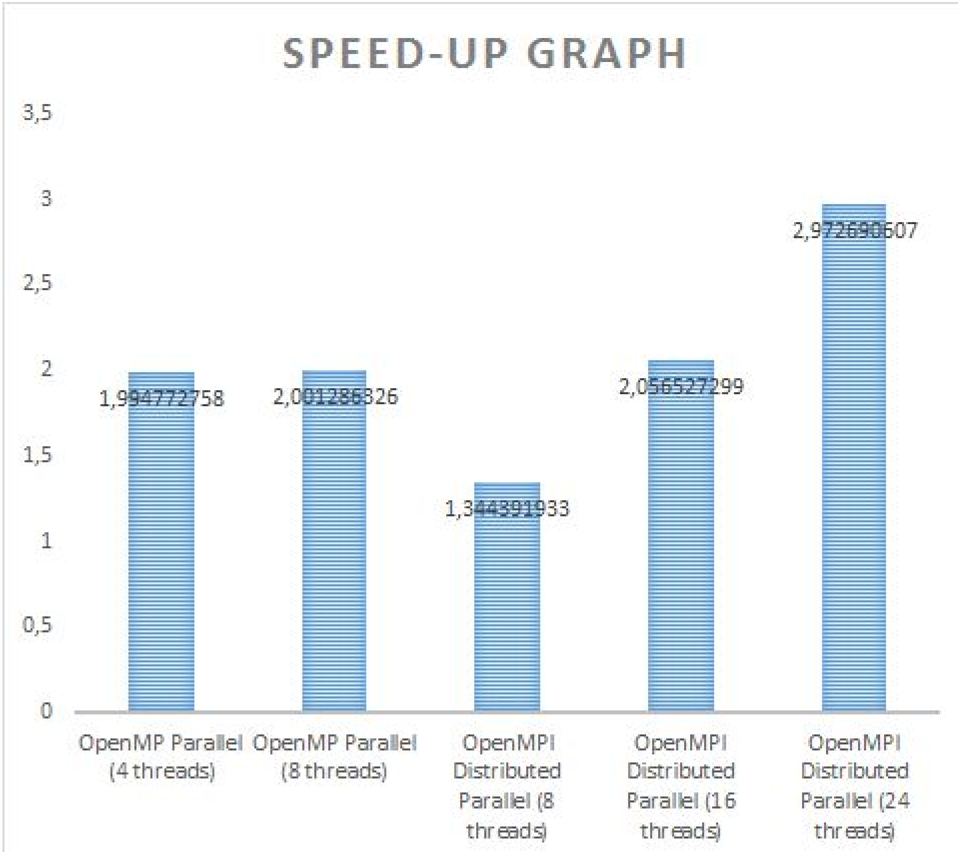
From all this data, we can generate the following graph, which already compares execution times with the previous solution.

Initially, we can immediately recognize how performance degrades exponentially with the increase of N (2^N), in a sequential solution, using a single core.

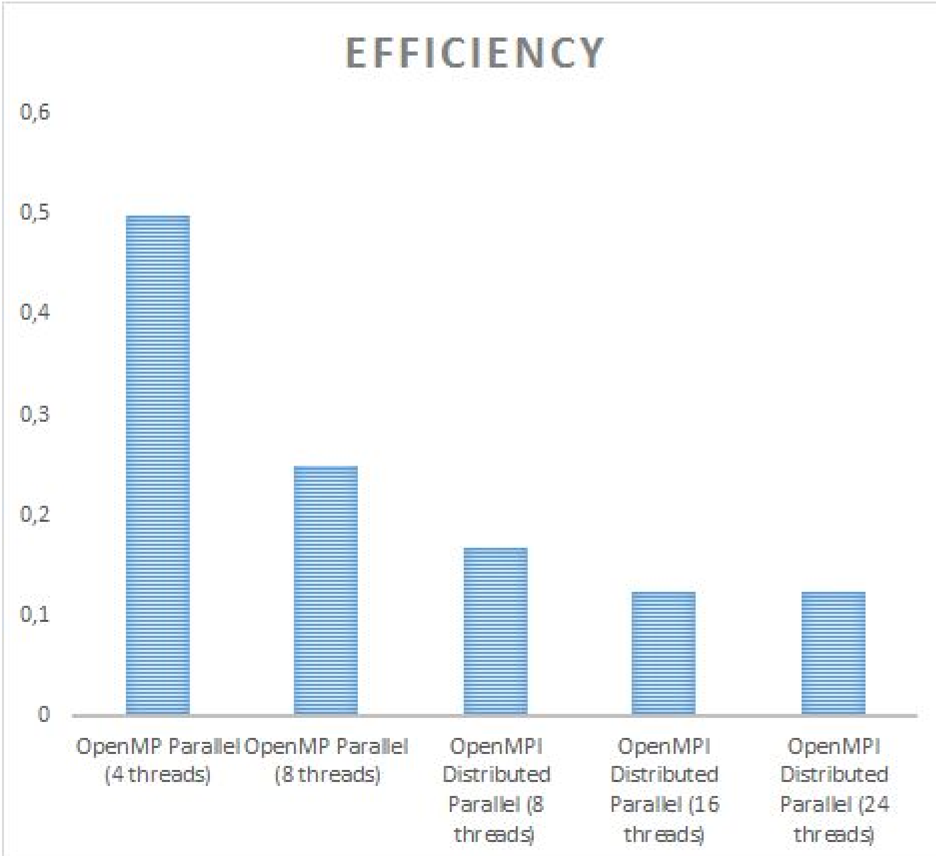
Let us look at the observed speed-up. Average speed-up’s are as follows.
In relation to the sequential solution:

OpenMP Parallel (4 threads): 1.994773
OpenMP Parallel (8 threads): 2.001286
OpenMPI Distributed Parallel (8 threads): 1.344392
OpenMPI Distributed Parallel (16 threads): 2.056527

OpenMPI Distributed Parallel (24 threads): 2.972691



Taking these speed-up values, we can infer the efficiency of the applied solutions through some simple calculations, and obtain the following.


EFFICIENCY
0,6
0,5
0,4
0,3
0,2
0,1
0
OpenMP Parallel (4 threads)
OpenMP Parallel (8 threads)
OpenMPI Distributed Parallel (8 threads)
OpenMPI Distributed Parallel (16 threads)
OpenMPI Distributed Parallel (24 threads)

Result analysis and conclusions:

We can already infer that all methods beyond the sequential solution provide improvements in performance in relation to that initial method. However, using surplus resources in the same machine appears to provide a near constant speed-up, regardless the number of used cores. The distributed solutions show larger promise towards higher and higher numbers of available machines/cores. It's noteworthy how a distributed solution that only makes use of the initial node's cores, through OpenMPI, does not provide the same kind of speed-up as the parallel solution, with OpenMP, and additionally, only appears to be the superior method once using more than 16 cores (in 2 machines, in the examined case). This is will be related to the time spent by OpenMPI sending messages between the several cores for division of tasks, time which is not spent in actual algorithm processing. Surely this would be mitigated by applying both OpenMP and OpenMPI, but such possibility was not undertaken here.

Another noteworthy event is the lack of efficiency in the applied solutions. Possible reasons for such a lower efficiency in the applied methods, regarding the distributed solution, could be the time OpenMPI needs for communication between threads. As the main thread broadcasts required information to all the other threads, and more communication is necessary the more numbers need to be processed. Additionally, splitting the processing between several nodes and several processes means these need to apply further calculations, in order to determine what numbers they should process.

As a final remark, some experiments were made beyond the 2^{32} limit, and in these, the distributed version started performing significantly better, which might be an indicator that for large scalable systems and complexities, if approached correctly, the gain from distributed systems can be significant.