The Sieve of Eratosthenes

Parallelization impact on performance

Parallel Computing

Faculty of Engineering of the University of Porto
Integrated Master in Informatics and Computing Engineering

Daniel Marques
up201503822@fe.up.pt

João Damas
up201504088@fe.up.pt

May 19th, 2019

Motivation

Problems can be solved in several ways. However, different implementations and programming strategies for the same problem may result in substantially different performances. With this in mind, the goal of this project is to study the effect of parallelization in the performance of algorithms. For this, different implementations for a well known algorithm, the sieve of Eratosthenes, were developed. This includes a sequential version and parallel versions using shared, distributed and hybrid memory. Their performances were subsequently compared in order to understand the impact of different strategies in the overall execution of the algorithm and its scalability.

Problem Description

The problem at hand is the search for prime numbers. More specifically, it tries to answer the question:

$$\text{What are the prime numbers in the range } [2, n] : n \in \mathbb{N}_{\geq 2}?$$

A number $n \in \mathbb{N}_{\geq 2}$ is considered to be prime if and only if its only divisors are 1 and itself.

Algorithm

The sieve is very simple and intuitive. If we are trying to find all the prime numbers up until n, we start with a list of unmarked integers $[2, n]$. Then, do the following:

1. $k = 2$
2. Repeat until $k^2 > n$:
    - Mark all multiples of k between k^2 and n
    - Define new value of k as the smallest unmarked number $j : j > k$

In the end, all unmarked numbers are guaranteed to be prime. In the figure to the right, this process is illustrated. The algorithm possesses a complexity of $\mathcal{O}(n \log \log n)$.

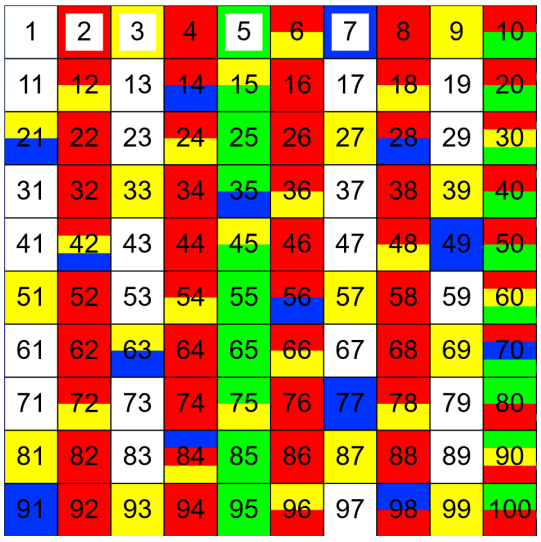

Figure 1: *Sieve of Eratosthenes in action. Notice the multiples of each successive k highlighted in different colors, until the next k (11) is superior to sqrt(100) = 10.*

Implementations

In this section, k represents the current prime number whose multiples are going to be marked (as described in the previous section), *primes* represents the array that contains marking information of all

odd numbers in the range (initially all set to *true*), and n the size of the problem.

When actually implementing the sieve, some *a priori* optimizations can be made:

- It is known that 2 is even and prime, therefore it is useless to check all other even numbers, since they are multiples of 2 (cutting the array size and, consequently, the search space in half);
- When marking non-primes (multiples of k), the increment can be $2k$ instead of k so as to skip iterations over even numbers (e.g., for $k = 5$, first check is 25, then the next can be $25 + 2 * 5 = 35$ instead of the even $25 + 5 = 30$, and so on).

Four implementations are presented in the following subsections (sequential and parallel with shared memory, distributed memory and hybrid memory), all of which have the above optimizations into consideration.

Sequential

The sequential implementation simply follows the algorithm presented, with the optimizations suggested, and is presented below.

Algorithm 1 Sieve of Eratosthenes (Sequential)

```
1       do {
2           // Mark odd multiples of k (hence 2*k increment) as non prime
3           for (long long j = k*k; j < n; j += 2*k)
4               //Since only odd numbers are stored, the array size is halved, and each number is stored
                ↪  in index number/2
5               primes[j>>1] = false;
6           // Calculate next k
7           do {
8               k += 2;
9           }while (k*k <= n && primes[k>>1]);
10      } while (k*k <= n);
```

Parallel

When implementing parallel versions for the algorithm, three main options can be considered. First, there is shared memory, where the process' memory is accessible among a certain number of threads. On the other hand, we can have multiple independent processes, each with their share of the data structure to be parallelized. Finally, both options can be combined to provide the best of all the perspectives.

Another important task is to decide which block(s) will be subject to parallelization, since it cannot be applied to the whole program. In this context, the most sensible choice is the non prime marking loop, shown in ll. 3-5 in the sequential implementation, which can be subject to data parallelization.

A Shared Memory

Through the usage of the **OpenMP** tool, it is possible to indicate, through pragma directives, sections of code to be parallelized by different threads in the same process, which is what is done below. Since

element marking is independent, no communication is required between threads and, therefore, this clause is enough. The implementation does not differ much from the sequential version other than this pragma directive. Notice the call on the first line to set a custom number of threads to be used by the process, which allows for custom benchmarking later.

Algorithm 2 Sieve of Eratosthenes (Parallel w/ Shared Memory)

```
1       omp_set_num_threads(n_threads); //Set number of threads to share memory
2       do {
3           #pragma omp parallel for //Indicate this loop is to be parallelized
4           for (long long j = k*k; j < n; j += 2*k)
5               primes[j>>1] = false;
6           do {
7               k+=2;
8           }while (k*k <= n && primes[k>>1]);
9       } while (k*k <= n);
```

B Distributed Memory

Through the usage of the **OpenMPI** tool, one can distribute the data across different processes, effectively partitioning the problem. In this case, the marking array is distributed, as equally as possible, between all processes involved. Each process then marks only the multiples that fall within its attributed range. Initially, all processes set the sieving prime k to 3. However, after each iteration, it is up to the root process to calculate the next sieving prime and broadcast its value to the other processes. It is guaranteed that the root process will have all the sieving primes in its range, since $p < \sqrt{n}$, where p is the number of processors (for the data range considered, $\sqrt{n} >> p$). Therefore, it is safe to task the root process with the responsibility of sole transmitter.

In order to allocate blocks of the marking array among processes, the macros shown below were used. The process rank (i), problem size (n) and number of processes (p) are the factors taken into account when distributing the blocks.

Algorithm 3 Sieve of Eratosthenes (Parallel w/ Distributed Memory)

```
1  #define BLOCK_LOW(i, n, p) ((i) * (n) / (p))
2  #define BLOCK_HIGH(i, n, p) (BLOCK_LOW((i) + 1, n, p) - 1)
3
4  //Block step of 2 because we're only considering odd numbers
5  //+3 to start at the first odd prime number, 3
6  lower_bound = BLOCK_LOW(rank, (n-1)/2, p) * 2 + 3;
7  upper_bound = BLOCK_HIGH(rank, (n-1)/2, p) * 2 + 3;
8  k = 3;
9  do {
10         //Antecipate uneven block size allocation and calculate offset if necessary
11     if (k*k < lower_bound) {
12         start_value = (lower_bound % k == 0 ? lower_bound : lower_bound + (k - (lower_bound % k)));
           ↪  //Guarantee start value is a multiple of k
13         start_value += (start_value % 2 == 0 ? k : 0); //Guarantee start value is an ODD multiple of
           ↪  k
14     }
15     else
16         start_value = k*k;
17
18     for (unsigned long i = start_value; i <= upper_bound; i += 2*k)
19         primes[(i - lower_bound)>>1] = false;
20
21     if (IS_ROOT(rank)) {
22         do {
23             k += 2;
24         } while (!primes[(k - lower_bound)>>1] && k*k < upper_bound);
25     }
26
27     MPI_Bcast(&k, 1, MPI_LONG, 0, MPI_COMM_WORLD);
28 }while(k*k <= n);
```

Because primes are increasingly apart (and so are their squares), it is important to check beforehand if the block falls nicely within a search range, hence the $if - else$ structure in ll. 11-16. There are mainly two cases to be considered:

1. k^2 is **larger** than the block's first value: normal search case, the search starts safely at k^2 (ll. 15-16; if k^2 is larger than the last value, the block simply won't have non primes to sieve)

2. k^2 is **smaller** than the block's first value: if the first value is a multiple of k, the search could start there (ll. 11-14, when the ternary is true); otherwise (when the ternary is false), there is the need to add an offset of k minus that modulus in order to find the smallest k multiple in the block. Line 13 guarantees that the first k multiple is odd.

C Hybrid Memory

In order to try to achieve the best of both perspectives, this approach mixes distributed with shared memory. The only relevant difference from the previous approach is the addition of a similar pragma directive as to what was shown previously, as can be seen below.

Algorithm 4 Sieve of Eratosthenes (Parallel w/ Hybrid Memory)

```
1       omp_set_num_threads(n_threads); //Set number of threads to share memory
2       ...
3       #pragma omp parallel for //Indicate this loop is to be parallelized
4       for (unsigned long i = start_value; i <= upper_bound; i += 2*k)
5           primes[(i - lower_bound)>>1] = false;
6       ...
```

Evaluation Methodology

In order to perform the necessary experiments, all implementations of the algorithm were coded in C++ programs. All experiences were run on desktops running the Ubuntu Linux distribution, powered by an Intel i7-4790 3.6GHz processor, which contains 4 cores (2 threads per core), each with a 32KB L1 data cache and a 256KB L2 cache (as well as a common L3 8MB cache).

In terms of problem size, the range considered is based on powers of two, more specifically $[2^{25}, 2^{32}]$ in order to assess performance (and its variation) using significant amounts of data.

In the case of shared memory, different number of threads were tested, more specifically 2, 4, 6 and 8. For distributed memory, four different computers, in a cluster, were used for the computation. The amount of processes per computer will be equal in all experiments and varies between 1 and 4. Finally, in the hybrid version, a combination of the previous two will be used: 1 to 4 processes on each computer will be used, as well as from 2 to the maximum number of threads allowed for each specific number of processes (i.e., if 1 process is running on each computer, then each process can use between 2 to 8 threads safely. However, if 3 processes are running on each computer, then each one may only use 2 threads).

In each experiment, the wall time is measured. With this, some metrics can be derived to assess each implementation and variation's performance, more specifically **Speedup** and **OP/s**, which are described in the equations below.

$$Speedup = \frac{T_{sequential}}{T_{parallel}} \tag{1}$$

In other words, the speedup is, for a given instance of the problem with size n, the improvement in execution time due to parallelization.

$$\frac{OP}{s} = \frac{n \log\log n}{T_{measurement}} \tag{2}$$

In order to obtain the number of operations per second for a given measurement (sequential or parallel), simply divide the total number of operations by the wall time. In the numerator, the algorithm's complexity is used.

Finally, in order to assess the parallel implementations' scalability, their efficiency will be measured:

$$E = \frac{Speedup}{P} \tag{3}$$

An implementation is scalable if its efficiency can be kept stable, preferably in the range [0,1], for increasing P and n.

Results and Analysis

Sequential Implementation

Exponent	Time (s)
25	0.090033
26	0.232309
27	0.521267
28	1.11706
29	2.39085
30	5.30123
31	11.4246
32	24.7402

Table 1: *Sequential implementation measurements.*

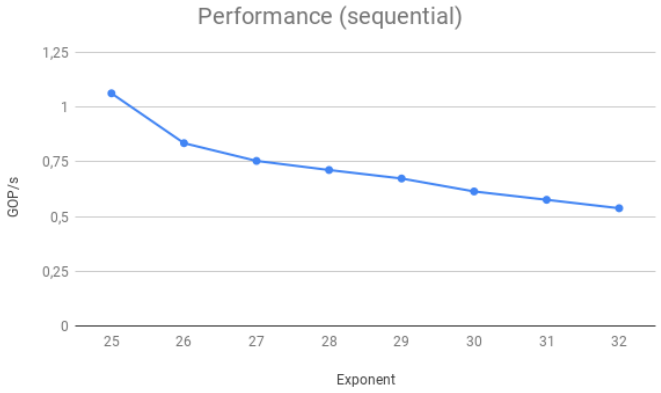

Figure 2: *Performance evolution, in GOP/s, for the sequential implementation.*

The results show that the base sequential version, as expected, has a progressively deteriorating performance. Although the *a priori* optimizations help in getting better base results (in some initial tests with a non-optimized version, i.e., process all numbers and all sieving prime multiples, times were considerably higher: for 2^{32}, it took nearly 50 seconds to complete the algorithm), some improvements can clearly be made.

The time for a given exponent is about a little over twice the time for the previous exponent. This is to be expected, since the problem is, effectively, doubling in size at each new measurement. Performance wise, the decay can be almost approximated by a linear function with a slope of ~0.07 (the average decline in performance at each iteration).

Parallel Implementations

Shared Memory

Exponent	2T Time (s)	4T Time (s)	6T Time (s)	8T Time (s)
25	0.077611	0.0655873	0.0685019	0.0729867
26	0.191009	0.176174	0.185274	0.189532
27	0.454832	0.403866	0.422039	0.43713
28	0.928529	0.869453	0.933395	0.903485
29	1.91352	1.83942	1.93407	1.9265
30	3.97909	3.81725	3.98831	3.94176
31	8.32019	7.94465	8.27619	8.60114
32	17.827	17.1017	17.4897	19.212

Table 2: *Shared Memory implementation measurements, for an even number of threads up until the maximum allowed by the processor (xT = x threads).*

With some parallelism introduced, the differences in performance are already very noticeable. Execution times shrink by about 25%, which shows that the block chosen to be parallelized was probably a good effort in trying to optimize the implementation. Notice that more threads does not necessarily mean better performance. In fact, the peak performance was achieved using a number of thread similar to the number of physical cores available, with execution times from that point on getting worse. This is somewhat to be expected, since, when using more threads than physical cores, there will be an additional overhead due to context switching (not all threads can work simultaneously). Performance wise, any iteration of each variant is visibly better than the sequential implementation (e.g., for 2^{32}, the sequential's performance is ~0.53 GOP/s, while the shared memory's are in the range [0.74, 0.77]).

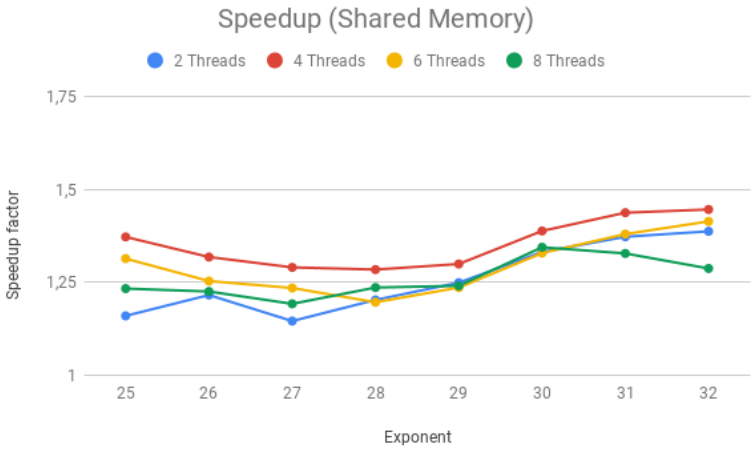

Figure 3: *Speedup factors for the shared memory implementation.*

Figure 3 confirms the superiority of using 4 threads for shared memory parallelization. It is also for this variant that the speedup factor assumes more stable values. Notice the values for 6 and 8 threads, which are never clearly better than the 2 thread variant and, in some cases, are worse by comparison.

Distributed Memory

Exponent	1C/PC Time (s)	2C/PC Time (s)	3C/PC Time (s)	4C/PC Time (s)
25	0.0221415	0.0208432	0.0267027	0.0218286
26	0.0478363	0.0333571	0.0288754	0.0324254
27	0.104445	0.107128	0.10141	0.090707
28	0.294049	0.274096	0.257815	0.253364
29	0.722904	0.628455	0.630332	0.60475
30	1.48692	1.32636	1.35383	1.31641
31	3.01093	3.0974	2.6386	2.5738
32	6.1913	6.0381	5.5379	5.1889

Table 3: *Distributed Memory implementation measurements (1C/PC = 1 Core, i.e., 1 process running per PC).*

With a distributed memory implementation, the algorithm's performance is much better than what was achieved so far, as well as more consistent. Being a more complex and thought implementation (block distribution), this enhancement comes with no surprise. Execution times are down to around a third of the shared memory's and, despite not having a clear advantage, it appears that the usage of more cores produces better results.

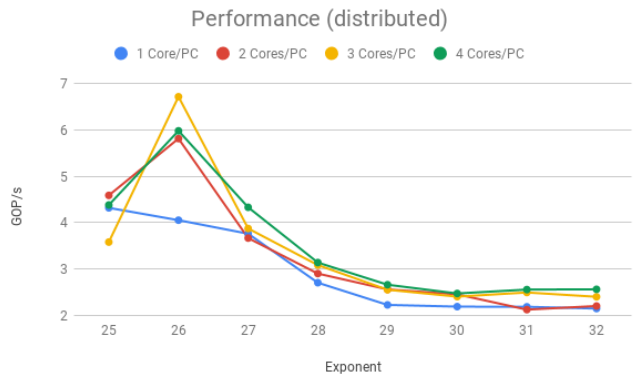

Figure 4: *Performance evolution for the distributed memory implementation.*

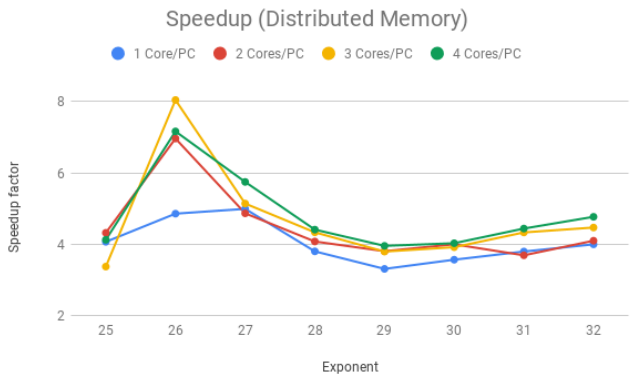

Figure 5: *Speedup factors for the distributed memory implementation.*

Performance and speedup values are, once again, much better than before and tend to stabilize as the problem size grows (stable values of ~2.5GOP/s and ~4.5x, respectively, for 4 cores/PC), which indicates some parallel scalability, which can be further analyzed with efficiency calculation.

Exponent	1C/PC	2C/PC	3C/PC	4C/PC
25	1.0166	0.5399	0.281	0.2578
26	1.2141	0.8705	0.6704	0.4478
27	1.2477	0.6082	0.4283	0.3592
28	0.9497	0.5094	0.3611	0.2756
29	0.8268	0.4755	0.3161	0.2471
30	0.8913	0.4996	0.3263	0.2517
31	0.9486	0.4611	0.3608	0.2774
32	0.999	0.5122	0.3723	0.298

Table 4: *Distributed Memory efficiency values (1C/PC = 1 Core, i.e., 1 process running per PC).*

Most values are in the desired range of [0, 1] (except some measurements for the 1 core/PC version) and, overall, present some stability, which further corroborates that this implementation can scale reasonably well.

Hybrid Memory

Exponent	1C/PC 4T Time (s)	2C/PC 4T Time (s)	3C/PC 2T Time (s)	4C/PC 2T Time(s)
25	0.0232	0.0270	0.0257	0.0285
26	0.0311	0.0315	0.0408	0.0399
27	0.0948	0.1055	0.1030	0.1134
28	0.2120	0.2613	0.2592	0.2569
29	0.5409	0.5737	0.5729	0.6273
30	1.1509	1.2434	1.3423	1.2084
31	2.4110	2.3081	2.3978	2.4785
32	4.8772	4.6934	4.6697	4.6432

Table 5: *Hybrid Memory implementation measurements (xC/PC yT= x Cores y Threads, i.e., x processes running per PC, each running y threads).*

Finally, by combining the previous two strategies, it is possible to achieve a further, even though smaller, boost in execution performance. It is not clear which combination produces the absolute best results, which indicates that maxing out cores and threads does not necessarily lead to a better performance. Only the most relevant measurements are included, with the remaining cases being available in the Appendix section.

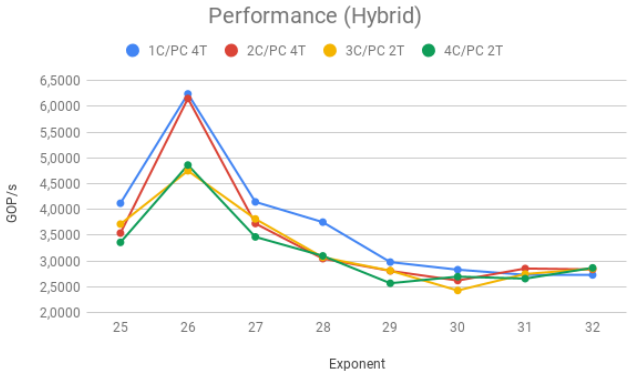

Figure 6: *Performance evolution for the hybrid memory implementation.*

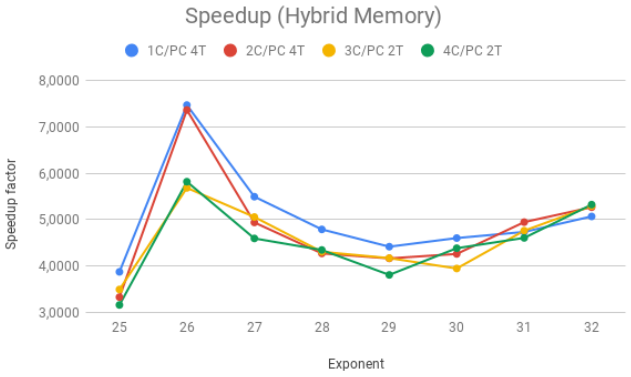

Figure 7: *Speedup factors for the hybrid memory implementation.*

Performances, once again, tend to stabilize, this time at slightly higher values than before, since further parallelism was applied. As for speedups, despite some signs of growth, its values tend mostly to a finite small range. Once again, notice the enhancement from the pure shared and distributed memory implementations.

Exponent	1C/PC 4T	2C/PC 4T	3C/PC 2T	4C/PC 2T
25	0.9692	0.4165	0.2914	0.1977
26	1.8690	0.9212	0.4741	0.3639
27	1.3747	0.6179	0.4217	0.2874
28	1.1982	0.5343	0.3592	0.2718
29	1.1050	0.5209	0.3478	0.2382
30	1.1515	0.5329	0.3291	0.2742
31	1.1846	0.6187	0.3971	0.2881
32	1.2682	0.6589	0.4415	0.3330

Table 6: *Hybrid Memory efficiency values (xC/PC yT= x Cores y Threads, i.e., x processes running per PC, each running y threads).*

Finally, efficiency values appear stable, in spite of the signs of super linear efficiency for 1 core per PC, 4 threads per process. Seeing as it is not a common situation to all measurements, it can inferred that this implementation can probably scale up too (which is corroborated by stabilizing performances).

Conclusions

Parallelization can go a long way in improving an algorithm's performance and creating scalable solutions. With this work, it was possible to explore different types of parallelization, with distributed memory implementations showing greater scalability signs. This comes with no surprise, as the creation of clusters from similar computers (horizontal scalability), partitioning the problem, is less costly and more efficient than constantly upgrading one single computer (vertical scalability) where the problem is partitioned locally. Finally, these two can be combined in order to take advantage of both worlds.

Appendix

A Remaining Hybrid Memory Measurements

Exponent	1C/PC 2T Time (s)	1C/PC 6T Time (s)	1C/PC 8T Time (s)
25	0.0276	0.0290	0.0298
26	0.0375	0.0329	0.0452
27	0.0964	0.0893	0.1033
28	0.2591	0.2495	0.2331
29	0.5764	0.5743	0.5642
30	1.2233	1.2021	1.2419
31	2.5611	2.5792	2.8230
32	4.9510	4.8650	4.8326

Table 7: *Remaining distributed memory measurements, for 1C/PC.*

Exponent	2C/PC 2T Time (s)
25	0.0235
26	0.0379
27	0.1011
28	0.2345
29	0.5810
30	1.2333
31	2.4147
32	4.8067

Table 8: *Remaining distributed memory measurements, for 2C/PC.*