Erathostenes and the Raspberries
Performance and energetic efficiency analysis of a Beowulf cluster

Sidney Bovet

École Polytechnique Fédérale de Lausanne
sidney.bovet@alumni.epfl.ch

Abstract

The goal of this report is to analyse the performance and power consumption of a Beowulf cluster made of Raspberry Pi, a low-cost, credit card-sized computer. It uses a distributed implementation of the famous Sieve of Eratosthenes. This report begins by performing a theoretical analysis of Eratosthenes' algorithm for finding prime numbers, including a theoretical speedup prediction for distributing it, and then compares those results to empirical ones measured on the cluster. Then the power- and cost-effectiveness of the Raspberry Pi is analysed and compared to other type of machines.

Figure 1: *A picture of the cluster with 16 nodes, along with a schematic representation of the cluster's networking and powering components.*

I. Introduction

Raspberry Pi 32 is a Bachelor semester project made at EPFL in 2013 [2]. It is constituted of 32 Raspberry Pi, a small computer which release price was USD 35 in 2012. The nodes communicate with each other using MPICH2, an implementation of the Message Passing Interface (MPI), and are physically linked using Ethernet 10/100 Netgear switches. Figure 1 shows the cluster and a schematic representation of its configuration.

The algorithm used to carry out the analysis is the well-known Sieve of Eratosthenes. Its basic sequential implementation is very efficient and runs in $\mathcal{O}(n \log \log n)$. A detailed explanation and analysis of the sequential and parallel version of this algorithm follows.

II. The algorithms

The Sieve of Eratosthenes consists of two main phases that we will call *sieving* and *gathering*. The sieving phase is the central one of the algorithm: given a prime number p, eliminate every number n_i in the list such that $n_i = kp, k \in \mathbb{N}$. The gathering phase is pretty straightforward in a sequential implementation and consists in identifying the next prime number in the list (the smallest n_i that was not eliminated). This then runs until the square root of some provided upper bound n has been reached. All unsieved numbers are then gathered and constitute the set of prime numbers smaller than n.

The sequential implementation of the algorithm is quite simple. The C code can be found in this report's annex, and the pseudocode is as follows:

```
$n \leftarrow$ size of the search
$A \leftarrow$ boolean array of $n$ **true** values
**for** $i = 2$ to $\sqrt{n}$ **do**
    **if** $A[i]$ is **true then**
        **for** $j = 0$ to $n$ **do**
            $A[i^2 + j * i] \leftarrow$**false**;
        **end for**
    **end if**
**end for**
```

One of the great strength of this algorithm is that it does not require to compute the modulus for each sieving pass through the array (observe how the inner `for` loop simply sieve according to `i` and `j`). Doing so would otherwise dramatically increase the computation time.

This simple algorithm may however not fit entirely in memory for large values of n. It then becomes necessary to use a segmented sieve, as proposed by Crandall & Pomerance [3]. This however goes beyond the scope of this report.

The parallel implementation that will be analysed in this report is based on the observation that prime numbers are not evenly distributed among $\mathbb{N}$. First of all, we chose to directly eliminate all even numbers. This does not change the asymptotic running time of the algorithm but doubles however its speed by already eliminating half of the solution space. Each process –with rank r– has a set to sieve locally. The i^{th} number of this set S_r is defined as $S_{r,i} = 2 \cdot r + 1 + 2 \cdot i \cdot (w - 1)$. Notice how the local sets can be entirely computed by the processes themselves, based on its rank and the world size w. This saves a lot of initialization time because the master does not have to send long integer lists to all the processes.
This means for example that for $n = 20$ and $nb\_workers = 2$, process P_1 will be assigned to $\{3, 7, 11, 15, 19\}$, while P_2 will get the set $\{5, 9, 13, 17\}$. Note that the maximum size difference between any two sets cannot exceed one.

Then begins the usual two-phase algorithm, which can be assimilated to some sort of explicit pipelining:
The master node broadcasts the smaller yet known prime. Each process then sieves according to the received number. This is the *sieving* part. The nodes then send to the master the smallest unsieved number in their list, and the master node proceeds to select the smallest number among those it received. This is the *gathering* part and it implies by definition a synchronisation barrier, since the master must await the values of all the nodes before deciding which is the new prime to sieve multiples of. Then a new *sieving* part happens, etc.

Once the node has reached $\sqrt{n}$ it broadcasts a final, special message telling the nodes to send him the rest of their unsieved numbers. It then prints all the numbers in a file. Note that it doesn't sort the received numbers in any manner, which does not violates the requirements of the algorithm, namely *finding all primes below n*. If one however wanted to implement this feature it could be included in the receiving pipeline if the numbers received by the worker nodes are guaranteed to be in order by using some variant of an External sorting algorithm [4].

Let us now look at the running time of these parts:

Initialization This takes constant time, every node can deduce from the world size and its rank the whole set of numbers it has to take care of. $\mathcal{O}(1)$

Sieving The time complexity of this part is that of going through each numbers in the local list, which is $\mathcal{O}(\frac{n}{nb\_workers}) = \mathcal{O}(n)$.

Gathering The master node has to find the minimum among all received numbers, which takes $\mathcal{O}(nb\_workers) = \mathcal{O}(1)$

There are also additional durations added because of the message passing scheme: the master broadcasts some messages, and the worker nodes send back to the master some other numbers. The measured latency and transfer time on the cluster is that of a 100 Mb/s Ethernet network using switches, namely 0.9 ms of latency and 10.8 MB/s of throughput Figure 4 shows these results graphically.
The latter does not really interests us since the nodes will mostly transfer 32-bit integers, with an exception for the last phase where all the nodes send a potentially large amount of integers to the master. However this suddenly massive traffic is negligible from the point of view of the master; the incoming messages can be streamed into the sorting pipeline, thus hiding the transfer time. The former value however is of great concern: the latency is quite big and given that a synchronization is needed after each *sieving* part, it will have to be very huge in order for the latency not to be too big compared to the computation.
This condition is further discussed at the end of the following section.

III. Performance predictions

Let us first define t_s and t_p as the serial and parallel execution time, respectively. The

Figure 2: *Timing diagram for 4 workers and $n = 32$. The critical path is highlighted in yellow.*

speedup is defined as $S = \frac{t_s}{t_p}$.

We now proceed to use Amdahl's law for determining a maximum achievable speedup. From time complexities of section II, and by observing that both the initialization and the gathering parts must be sequential, we can devise a value for f, the fraction of the code that cannot be parallelized:

$$f = \frac{\frac{1+nb\_workers}{1+n/nb\_workers+nb\_workers}}{\frac{nb\_workers}{n/nb\_workers+nb\_workers}} \approx$$

And the theoretical speedup then is:

$$S(nb\_workers) \leq \frac{1}{f+1/nb\_workers(1-f)} = \frac{1}{\frac{1}{n+nb\_workers^2} + \frac{n+nb\_workers^2+1}{nb\_workers(n+nb\_workers^2)}}$$

This yields the following theoretical speedup limit for $n = 2^{30}$ and $nb\_workers = 32$:

$$S(32) \leq 31.999$$

Which obviously is a very optimistic bound that is not even taking into account the networking costs, which are quite high.

Figure 2 shows the timing diagram of an execution for $n = 31$ and 4 workers, with the critical path highlighted. This critical path allows us to devise another, more accurate theoretical speedup prediction.

Given that the workers all have sets of same length (± 1), we can assume they will all send their smaller unsieved number at the same time. Now this operation takes 0.9 ms, and the sieving operation took four times less time to be computed since we distributed it on four nodes. The size of each worker's local set must thus be such that sieving it takes more than $\frac{4}{3}0.9 = 1.2$ ms to be computed. Given that the Raspberry Pi, while having a cheap CPU, still has acceptable performances, this will be an issue regarding the 512 MB of memory it has.

Note that this discussion ignores the cost of the final retrieval and printing. This cost however has a small impact on the overall running time because the writing I/O time overlaps with the I/O time of the network.

IV. Results and Discussion

This section presents and discusses the experimental results observed on the cluster using the parallel implementation described above.

Table 1 shows the numerical values found for $n = 4 \cdot 10^7$. Figure 3 shows the corresponding speeup (or more accurately the speed-down). The difference between one and two workers is quite promising, but then the performances drop dramatically. The reason for this drop is discussed further below.

nb_workers	**time** [s]
Sequential	46.727
1	69.284
2	63.738
4	157.133
8	163.738
16	278.169

Table 1: *Power consumption of the cluster's components.*

Figure 3: *The absolute and relative speedup measured on the cluster.*

During the run of the algorithm on 16 nodes, the power consumptions of the components was measured and the average values found are shown in Table 2 below.

Component	**Power** [W]
Raspberry Pis	39.2
Network Switches	14.4
Total	53.6

Table 2: *Power consumption of the cluster's components.*

Let us first have a look at why the performances are getting worse. As observed before, the network plays a big role in the computation time. Since the drop happens when using 4 worker nodes (for a total of 5 nodes), and given the network configuration of the cluster, it is possible that the extra hop the packets must perform to reach the 5th node (i.e. going through the head-end switch) harms the performances. The figure 4 invalidates this theory by showing no difference between single and multiple hops transfers. Broadcast time however is different but this is not surprising since all the nodes must receive the message.

The network switch has thus been removed from the suspect list. Only conjectures can be done at this point because the real cause of this drop would need a much deeper analysis. However if the algorithm itself is the cause of the drop in performance, which is most likely the case, the fact that between each sieving part the master acts as a synchronisation barrier is not a good thing for such a small task. Also it aims for quick, small messages which do not perform well in an Ethernet network.

Those are the two issues that should be addressed first if one wanted to enhance this parallel algorithm.

Figure 4: *Average transfer time for broadcasting, single hop, and three hop messages and varying data size. The function $f(x)$ is the trend line for single hop messages.*

The total power consumption of the cluster is comparable to that of some of the state-of-the-art available CPUs, such as Intel's Core i7. The following section further discusses this aspect of the question.

V. Cost- and Power-effectiveness

We now proceed to analyse the cost and power efficiency of the Raspberry Pi, and compare the values to other machines.

In order to get a common frame of reference, a well-known benchmark was run on a single Rasberry Pi and its result is compared to the scores of four different devices, namely a MacBook Pro with a 2.6GHz Intel Core i5, a bare 1.73GHz Intel Core i7 820QM and a 2.13GHz Intel Core2Duo E6400. These numbers have already been found by [1] but the cost and power analysis proposed here is new.

The benchmark runs 15 tests such as Random Number Sort, Matrix Transpose, or Discrete Fourier Transform. The score of the four compared machines are shown in Table 3, along with their price and power consumption[1][2].

	Score	**Price** [USD]	**Power** [W]
MacBook Pro	0.86	1299	200
1.73GHz i7	0.73	546	45
2.13GHz E6400	0.36	299	65
Raspberry Pi	0.01	35	2.5

Table 3: *Mathematica score, Device price, and Power consumption reference values.*

In order to compare these machines let us first look at the score per USD, which is shown in Figure 5. As one can observe the Raspberry Pi is beaten by all the machines, but the MacBook Pro only does twice as good. On Figure 6, which shows the score per Watt, the difference is smaller and only the Intel Core i7 does significantly better. This is not very surprising, given that the Raspberry Pi homes an ARM processor, which is known to be quite power-efficient.

The fact that the bare CPUs perform so well comes from the fact that the real price and power consumption of the machine running the benchmark is necessarily higher than that of the CPU alone. But since Mathematica gives no other details than the OS running on the machines that have achieved these scores it is hard to give a good estimate. In that sense the "MacBook Pro" value below is in both case the closest one to reality.

Figure 5: *Performance - price ratio for various machines.*

Figure 6: *Performance - power ratio for various machines.*

VI. Conclusion

This study has revealed some interesting factors in the performance of a cluster. The first

[1]Power values from `http://en.wikipedia.org/wiki/List_of_CPU_power_dissipation_figures`
[2]Price values from `http://www.cpubenchmark.net/`

thing to take out of this report is that Ethernet has quite a big latency and parallel algorithms must be designed such that this latency does not have a big impact on their performance.

Such design weaknesses may more easily avoided by using tools such as DPS [5], which allow to visualize the parallel implementation and data flows.

The cluster itself has proven being not that bad regarding its price and power consumption, and can therefore constitute a good parallel algorithm analysis tool. On the other hand, with its great computation over communication ratio, it is quite different from other Beowulf clusters with current desktop machines homing for instance Intel i7 CPUs.

References

[1] Adereth (2014, January 6).
Benchmarking Mathematica on the Raspberry Pi.
Retrieved 13:27, December 29, 2014, from `http://adereth.github.io/blog/2014/01/06/benchmarking-mathematica-on-the-raspberry-pi/`

[2] Bovet (2013).
Raspberry Pi 32
LAP & LSR.
`http://aspi32.ch/about.html`

[3] Crandall and Pomerance (2005).
Prime Numbers: A Computational Perspective, second edition
Springer, pp. 121-24.

[4] External sorting. (2014, February 23).
In Wikipedia, The Free Encyclopedia. Retrieved 14:13, December 29, 2014, from `http://en.wikipedia.org/w/index.php?title=External_sorting&oldid=596774086`

[5] Schaeli, Hersch, et al. (January 2009).
Dynamic Parallel Schedules Documentation
`http://dps.epfl.ch/documentation.php`