Parallel computation of the trajectory of a Markov process

Jean-Michel Fourneau   Maël Guiraud   Yann Strozecki

Abstract

In this short article, we describe several methods to compute the trajectory of a Markov process in parallel. Our algorithms rely on the fact that the states of the process are partially ordered. We explain under which assumption we can obtain a speedup using a parallel algorithm and we give experimental evidences that the method works better than sequential or classic parallel simulation.

1 Introduction

Let $\mathcal{S}$ be a finite set of states and let f be a computable function from $\mathcal{S} \times [0,1]$ to $\mathcal{S}$. The aim of this short article is to propose practical methods to compute in parallel the sequence $T(s, x_1, \ldots, x_n) = (s_0, \ldots, s_n)$ where $s_i = f(s_{i-1}, x_i)$ and $s_0 = s$. The elements $(x_0, \ldots, x_n)$ are realizations of a sequence of independent and identically distributed random variables $X_0, \ldots, X_n$. This models a random process on a finite set of states, which depends only on the current state, in other words a *Markov process*.

In general, s_n depends on s and all x_i's and thus cannot be computed without reading all x_i's, which makes parallelization impossible in general. Hence, we consider additional structure on $\mathcal{S}$ to make the problem tractable. The set $\mathcal{S}$ is equipped with a partial order $\leq$ and it has a smallest element $\perp$ and a greatest element $\top$. We assume that f is monotone, that is if s_1 and s_2 are elements of $\mathcal{S}$ such that $s_1 \leq s_2$ then for all $x \in [0,1]$, $f(s_1, x) \leq f(s_2, x)$. Given a function f, a state s and a sequence $x_0, \ldots, x_n$ of realizations of $X_0, \ldots, X_n$, we want to compute the sequence $T(s, x_1, \ldots, x_n)$ when $\mathcal{S}$ is a partially ordered set. We call this problem POTRASIM, for partially ordered trajectory simulation.

In practice, the values $x_1, \ldots, x_n$ are given by a pseudo random generator initialized by some random seed. These values can be generated efficiently in parallel without changing their distribution and thus the distribution of the generated trajectories: An integer k is chosen and the values x_{ki} are generated using a pseudo random generator and a random seed, then in parallel a pseudo random generator is used to generate $x_{ki+1}, \ldots, x_{ki+k-1}$ using x_{ki} as a seed. When $k = \sqrt{n}$, this computation uses a parallel time of $O(\sqrt{n})$. This efficent parallelization justifies our choice of considering the sequence $x_1, \ldots, x_n$ as an input in the problem POTRASIM.

To formalize our parallel algorithms, we choose a simple PRAM model – exclusive read and exclusive write (EREW)– with shared memory to neglect

synchronization and communication problems (see [1]). This simplification is reasonnable in a multicore machine, since in our algorithms we use very few concurrent accesses to only limited informations, which can be dealt with no or almost no locking. For a distributed computing point of view, the cost of communication is also relevant since it tends to dominate the computing time. Both contexts are investigated in our experimentations.

TODO: Expliquer ici ce qui s'est fait avant (avec moins de suppositions). Ce qui s'est fait en général avec des espaces d'états po (travail de JMF) Expliquer ensuite nos contributions.

2 Parallel computation

The main idea of the method is to divide the sequence T of size n into smaller intervals of t states. If we know the initial state of each interval then we can compute the sequences of iterates independently. In the PRAM model, with an unbounded number of processors and zero cost of communication, it may be optimal to have t small and independent from n, but in practice cores are a scarce ressource and we will set n/t to a value related to the number of machines or cores avalaible.

2.1 Two bounds

In this section, we describe an algorithm which solves POTRASIM in parallel, that is given $s, x_1, \ldots, x_n$ and an algorithm to compute f as inputs, it produces the sequence $T(s, x_1, \ldots, x_n)$. We fix a t, the size of an interval and we denote by $k = n/t$ the number of intervals. We store the status of each interval j in a variable $status_j$ whose value can be TOCOMPUTE, COMPUTED or DONE. We also store for each $0 \leq j < k$ the states s_j^{min} and s_j^{max} which are lower and upper bound on the real trajectory at the beginnig of the jth interval. In all the algorithms we describe, the following invariant is true: $s_j^{min} \leq s_{j*t} \leq s_j^{max}$.

Let us now describe the execution of the algorithm. At the beginning, the status of each interval is TOCOMPUTE, and the lower bounds are set to the minimal elements and the upper bounds to the maximal except for the first. For all $0 < j < k$, $status_j = $ TOCOMPUTE, $s_j^{min} = \perp$ and $s_j^{max} = \top$. Moreover $status_0 = $ TOCOMPUTE and $s_0^{min} = s_0^{max} = s$.

While there is a free processor P and a j such that $status_j = $ TOCOMPUTE, the index j is selected. The variable $status_j$ is then set to COMPUTED and the processor P computes the two sequences $T(s_j^m in, x_{jt}, \ldots, x_{(j+1)t})$ and $T(s_j^m ax, x_{jt}, \ldots, x_{(j+1)t})$. The last value of the two computed sequences, let us denote them by $s_m in$ and $s_m ax$, are then compared to the bounds of the next interval if there is one. If $s_m in > s_{j+1}^{min}$ or $s_m ax < s_{j+1}^{max}$ then better bounds have been found and P sets $s_{j+1}^{min} = s_m in$, $s_{j+1}^{max} = s_m ax$) and $s_{j+1} = $ TOCOMPUTE. Finally, when a processor simulate the interval j and at the beginning $s_j^{min} = s_j^{max}$, the result of the simulation is stored as a part of the trajectory and $s_j = $ DONE. The algorithm we just described is called SPECULATIVE SANDWICH and we give its pseudocode in Figure 1.

Algorithm 1 Speculative Sandwich

Input: Size of the intervals t

  // *Initialisation*

  for $i < \min(\text{nb\_machines}, \text{nb\_inter-1})$ **do**

    Send I_{j+1} to the server i

  end for

  // *Main loop*

  while All the intervals are not DONE **do**

    Wait for a server to answer the results of *current_interval*

    if The server was computing a trajectory **then**

      set $I_{current\_interval}$ to DONE

    end if

    if $f^t(s^{min}_{current\_interval}) > s^{min}_{current\_interval+1}$ or $f^t(s^{max}_{current\_interval}) < s^{max}_{current\_interval+1}$ // *Better bounds have been found* **then**

      $s^{min}_{current\_interval+1} = f^t(s^{min}_{current\_interval})$

      $s^{max}_{current\_interval+1} = f^t(s^{max}_{current\_interval})$

      set $I_{current\_interval}$ to TOCOMPUTE

    end if

    next_interval ← search the first interval which is to TOCOMPUTE

    if $s^{min}_{next\_interval+1} = s^{max}_{next\_interval+1}$ // *The bounds are coupled* **then**

      Wait a trajectory for $I_{next\_interval}$

    else

      Wait some bounds for $I_{next\_interval}$

    end if

    Send $s^{min}_{next\_interval+1} and s^{max}_{next\_interval+1}$ to the current server

  end while

Theorem 1. SPECULATIVE SANDWICH *solves the problem* POTRASIM.

Proof. First, we prove that SPECULATIVE SANDWICH terminates and always compute the right answer. Recall that we have stated that for all j, $s^{min}_j \leq s_{j*t} \leq s^{max}_j$. We can prove that this is true during all the algorithm by induction on the computation time. Indeed, s^{min}_j and s^{max}_j are changed when a simulation from s^{min}_{j-1} and s^{max}_{j-1} find tighter bounds. By induction hypothesis $s^{min}_{j-1} \leq s_{(j-1)*t} \leq s^{max}_{j-1}$ since the have been computed before. The monotony of f guarantees that $j, s^{min}_j \leq s_{j*t} \leq s^{max}_j$. This guarantee that when $s^{min}_j = s^{max}_j$, the trajectory we simulate is correct.

Let us prove that the when the algorithm stops it has computed all the trajectory. Assume for the sake of contradiction that the algorithm stops and that at least an interval with status COMPUTED and let j the smallest index such that $status_j =$ COMPUTED. The interval $j > 0$ since $status_0 =$ DONE because $s^{min}_0 = s^{max}_0 = s$. Hence $j - 1$ is positive and $status_{j-1} =$ DONE by definition. But then SPECULATIVE SANDWICH has updated the bounds s^{min}_j and s^{max}_j to s_{jt} after the last simulation of the interval $j - 1$. As a consequence SPECULATIVE SANDWICH must have simulated the interval j and marked it DONE, a contradiction.

Finally let us prove that SPECULATIVE SANDWICH terminates. To do that, it is enough to remark that each time a new interval is simulated, it implies that the beginning bounds are tighter. Since the number of states is finished,

it implies that at some points the bounds are equal for each interval and the interval is marked DONE. □

Remark that the way we select an index j when there are several $status_j$ equal to TOCOMPUTE is not specified. Each policy to select it gives rise to a different implementation of the algorithm SPECULATIVE SANDWICH. Note that if we have an unbounded number of cores, we can always affect each avalaible interval to a different processor, therefore the policies are all the same. Hence they make sense in a context where the number of processors is limited.

We propose two different policies. The first is the ORDERED policy: choose the avalaible interval with the smallest index. The second policy is called BALANCED as we try to make the processors work on all parts of the trajectory more evenly. The k intervals of size t are grouped in l larger meta-interval and we store how many intervals have been simulated in each meta interval. Then we select an avalaible interval in the meta-interval with the less computed intervals. If there are several such intervals, we chose the one with the smallest index.

For different policies, we would like to have a guarantee on the running time of SPECULATIVE SANDWICH. In particular, we would like it to be faster than the sequential time to solve POTRASIM.

Proposition 1. SPECULATIVE SANDWICH *with policy **Ordered** solves* POTRASIM *in parallel time $O(Kn)$ where K is a bound on the time to compute f.*

Proof. We prove that at any point in time of algorithm SPECULATIVE SANDWICH with policy **Ordered**, there is a processor computing the trajectory in an interval with status DONE. From that fact, the time bound is clear since each interval is marked DONE exactly once and the cost to simulate it is $O(tn)$ with t the size of the interval.

The proof is by induction on the computation time of the algorithm. Assume a processor P finishes to compute the sequence on an interval. Then consider the smallest interval I_j which is not marked DONE. If it is in state INPROGRESS, then a processor is computing the solution on I_j. If it is in state TOCOMPUTE, it will be selected by the free processor P and since I_{j-1} is in state DONE, $x_j^{min} = x_j^{max}$ and P will compute the solution on I_j. □

With policy **Ordered**, the problem is solved in parallel time bounded by the sequential time that is $O(cn)$ where c is a bound on the time to compute f. Remark first that the time bound $O(cn)$ is the same as for the *sequential* algorithm which just applies f repeatedly. It is not worst, and it can be better, if at some point we have $x_j^{min} = x_j^{max}$, we say that the two sequences are coupling. If this phenomena happens frequently, the speed-up can be important. For instance, in the best case, we have a coupling on each interval the first time it is simulated, then the sequential time is bounded by $O(ct)$.

TODO: Faire une proposition sur le temps de calcul de l'algo Faire un petit modèle probabiliste qui pour une proba de coupler donnée combien on va faire de calculs. Ça serait bien de se servir de ça pour couper des intervalles de la bonne taille. Peut être faire un calcul simple pour un tradoff entre nombre de processeurs et temps de couplage.

Remark that all processors do the simulation using only their private memory and the two states at the beginning of the interval. This scheme is thus

reasonably adapted to a distributed computing environment where the cost of transmitting information between processors is high.

In our application to the simulation of a random process, the states of the process are often equipped with a partial order, but the random integers are not. However, we have the following property in our random system, if $s_1 \leq s_2$ then $f(s_1, x) \leq f(s_2, x)$. In fact, the random value alone is used to select an action to apply to the system and the action does not depends on the state of the system. Rather than simulating a given sequence $(s, f(s), \ldots f^n(s))$, we want to compute a random sequence that is to obtain a sequence with the right probability, which is a problem slightly different of POTRASIM. We use the following trick to transform our problem: instead of using a single seed to generate all pseudo random values, one seed for each interval I_j is used. We have a collection of seeds $x_0, \ldots x_{n/t}$ and the i-th random number will be equal to $R^{i \mod t}(x_{i/t})$ instead of $R^i(x)$. Since a random integer in the sequence depends only on the previous one, the random process is the same, that is all realizations of the random process still occur with the same probability as when only one seed is used. However, we can now abstract away the random numbers from our sequence, even when we divide the sequence into intervals, and use the algorithms we have described which solve POTRASIM.

2.2 A simple lockless implementation

We can slightly improve the algorithm BALANCED PARALLEL SANDWICH. First we make it simpler by cutting $[n+1]$ into l intervals if we have l processors, that is at first intervals and meta-intervals are the same. Then when a processor a processor has correctly simulated its interval, we cut the remaining interval to simulate into intervals of similar sizes and affect one processor to each one.

2.3 One bound

TODO: description of the algorithm and complexity bounds

3 Experimental evaluations

3.1 Settings

TODO: Mettre ici la description de la chaîne et des outils communs

3.2 Multicore machine

Closer to the abstraction of the previous part. Mettre aussi les optimisations pour ce cas (on ne recalcule pas après couplage).

3.3 Distributed computing

3.4 Random process

The random process we use in our experiments is the following one. We have a system composed of m finite queues of capacity BUFF_MAX in tandem. A

queue is characterized by its number of client C_i. For each queue i, three events can occur:

- An arrival; C_i is increased by one, if it is not already to BUFF_MAX.

- A service; if there is at least one client in i, a client leave the queue i and goes in the queue $i + 1$. C_i is decreased by one and C_{i+1} is increased by one. If C_{i+1} is equal to BUFF_MAX, the client is lost.

- A departure: the client leave the system. The number of client of the queue is decreased by one if $C_i > 0$.

For the queue i, every event have a probability denoted by respectively a_i, s_i and d_i. The last queue has no service, thus $s_m = 0$. There is thus a total of $3m - 1$ different events that can change the system.

The sequence that we want to parallelize is a succession of drawing of one of those $3m-1$ events. Every event is drawn randomly, following a pseudo random generator.

In the following experiments, the parameters a_i, s_i and d_i of the queues are set by the following rules:

- $P = 0.75$

- $\mu = 300$

- load $= 1.1$

- $a_0 = \mu \times$load

- $a_i = (1 - P) \times a_0, i \in [1 : m]$

- $s_i = P \times \mu, i \in [0 : m]$

- $d_i = (1 - P) \times \mu, i \in [0 : m]$

All those probabilities are then normalized such that $\sum_{i=0}^{m} a_i + s_i + d_i = 1$.

With those parameters, every queue have an average load of 1.1. This load has been experimentally calculated as the load for which the coupling time of the two bounds in the largest.
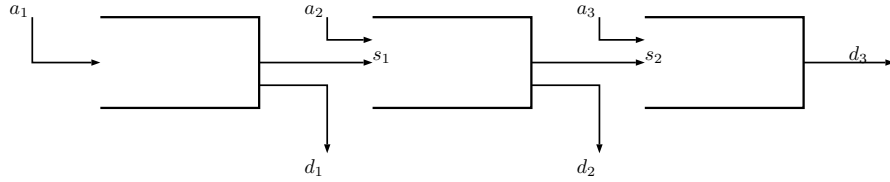

Figure 1: A system with 3 queues

3.5 Hardware and software used

3.6 Experimental settings

Our experimentations are made with the following set-up. The main program run on a MacBook Air, with a 2.2 GHz Intel Core i7, and 8 Go of RAM DDR3 at 1600 MHz. The operating system of the machine is macOS High Sierra v10.13.1. The source code is compiled with gcc version 7.1.0 (Homebrew GCC 7.1.0 – without-multilib).

The simulations of each interval are dispatched on up to 7 Raspberry Pi 3 , Model B, with $1GB$ of RAM. Their operating system is Raspbian GNU/Linux 8.0, installed on a micro SD card element14 with a size of 8GB. The source code is compiled with gcc version 4.9.2 (Raspbian $4.9.2 - 10$). All the machines are connected to a local network through an HP 14-10 8G switch.

¡¡¡¡¡¡¡ HEAD All the machines are connected to a local network through an HP 14-10 8G switch.

During our experiments, we use the socket with the protocol TCP/IP to allow the servers and the master to communicate. All the machines were wired to a local network without access to any external networks. The source code of the simulation is written in C and available on the webpage of the authors. TODO: lien vers site yann avec code

3.7 Network impact

Since our network uses a local network with TCP/IP, one must investigate the cost of the network in our simulations. In our source code, we fixed the size of a standard master/server message to 24 integers and the size of the server/master messages depends of the number of queues in the simulation. One must focus on the cost of sending a trajectory, which is by far the largest message, indeed, the size of a trajectory is equal to $t \times m(t$ is the size of the interval, and m is the number of queues). On the following experiment, we tried to compare the evolution of the cost of the computing time and the network cost for different number of queues going from 1 to 10. We arbitrary set the size of the interval t to $10,000$, which is a good order of magnitude considering the following experiments. The points on fig— **??** are some average times computed on 100 different simulations.

As we can see, both the communication an computing times of a trajectory increases linearly with the number of queues, but the communication time is increasing faster. A linear regression allowed us to calculate the slope of those two point clouds; the slope of the communication time is 3.390585 while the slope of the computing time is only of 0.885585. Both of those two correlations coefficients are greater 0.999. Thus, to avoid an over domination of the network time, which would hide every interesting results, in the following experiments, we will consider a little number of queues in tandem.

3.8 Algorithm performance evaluation

We now want to measure the performance of the SMALLEST PARALLEL SAND-WICH and the BALANCED PARALLEL SANDWICH, compared to the One bound algorithm (cf sec 2.3). In the following experiment we thus take a random process with 5 queues, which allows us to have a reasonable amount of computation
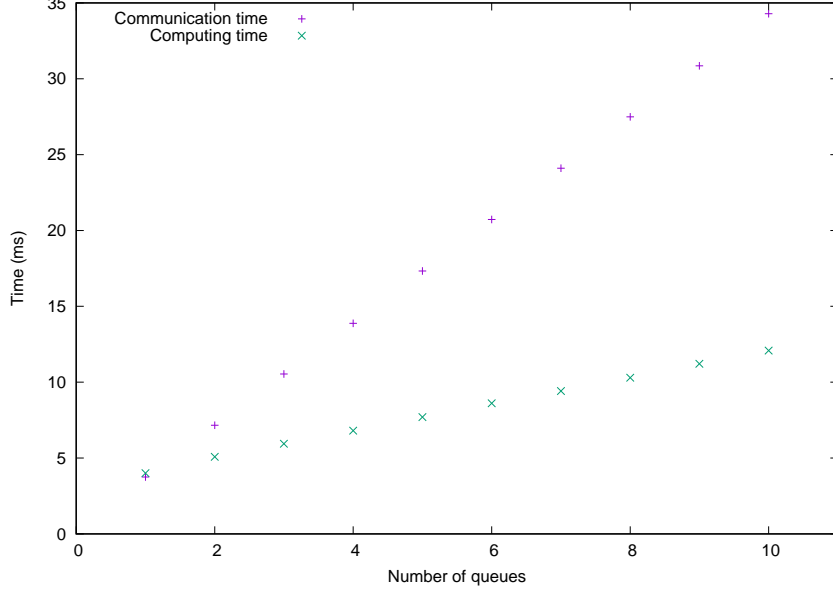
Figure 2: Evolution of the computing and communication time of a trajectory of $10,000$ events

in the servers, and a not too high $\frac{\text{Network time}}{\text{Computing time}}$ ratio. In order to choose the size n of the simulation, we made an upstream experiment to determine the average time between the coupling of the two bounds, for two queues, with the parameters we give in sec 3.4. Thus, this average coupling time is to $28,049$, calculated over 100 simulations.

3.8.1 Long simulation

We first want to look at the behavior of our algorithms in a simulation in which there is an high probability to have a coupling during the computing of a single interval. We then set the size of the simulation to $n = 210,000$. Fig. 3.8.1 and 3.8.1 show the results of respectively, the average sum of the lengths of the intervals computed, and the average total time if simulation in regard of the number of intervals, over 100 instances. By the way, the size of the interval directly depends of the number of intervals. The total size of the simulation is always the same, that is, $210,000$. We used 7 servers in those experiments.
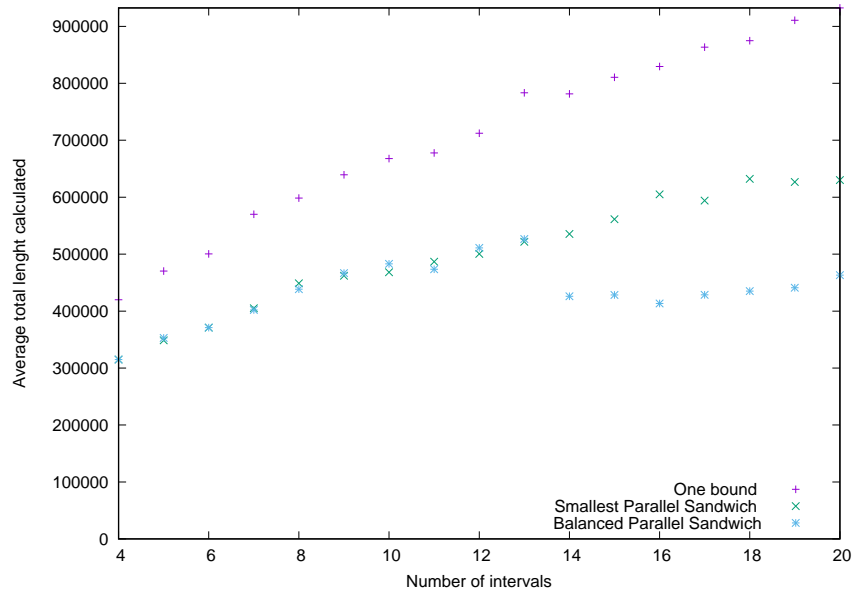
Figure 3: Evolution of the average of the sum of the lengths of the intervals computed for 100 simulations of $210,000$ events.
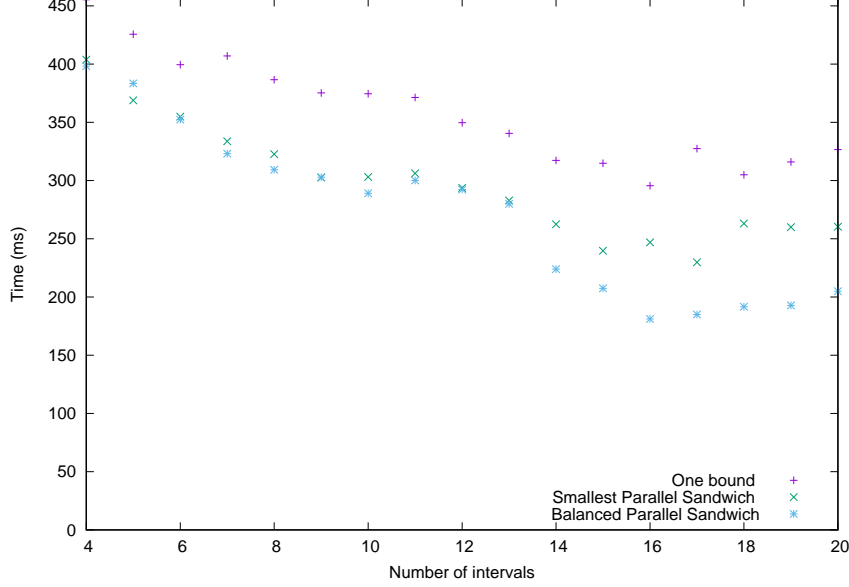
Figure 4: Evolution of average for 100 simulations of $210,000$ events.

First, we remark that the SMALLEST PARALLEL SANDWICH and the BALANCED PARALLEL SANDWICH are always better than the One Bound algorithm. The weak increasing trend for One Bound and SMALLEST PARALLEL SANDWICH on fig. 3.8.1 might be due to the decreasing probability of coupling in one interval when the number of intervals increases. Nevertheless the BALANCED PARALLEL SANDWICH seems to have a different behavior after an undetermined threshold. We will study this phenomenon later.

TODO: pas vraiment en fait, j'utilise le fait que il y ait ce point du rupture pour comparer les performances en fonction du nombre de servers mais je ne sais pas du tout pourquoi ca fait ca??

In contrast to those results, as we can see on fig. 3.8.1 , it looks like having few long intervals is more expensive in time than having more shorter intervals. This might be due to the network. Indeed, as we observed in the results of sec. 3.7, the more the interval is long, the more network increases. Moreover, the experiment in sec. 3.7 were made on a single server communicating with the master. Here, we have 7 servers which probably create some contention, and thus, some additional latency. On a single processor, the average time on 100 simulations for $210,000$ events is to 162.13 ms.

To make sure that the network is dominating the simulation time, we investigate the activity of the servers during the experiments. Fig. 3.8.1 shows the average repartition of the computation, network and waiting time of the servers, during the previous experiments, for the computation of the BALANCED PARALLEL SANDWICH simulations.
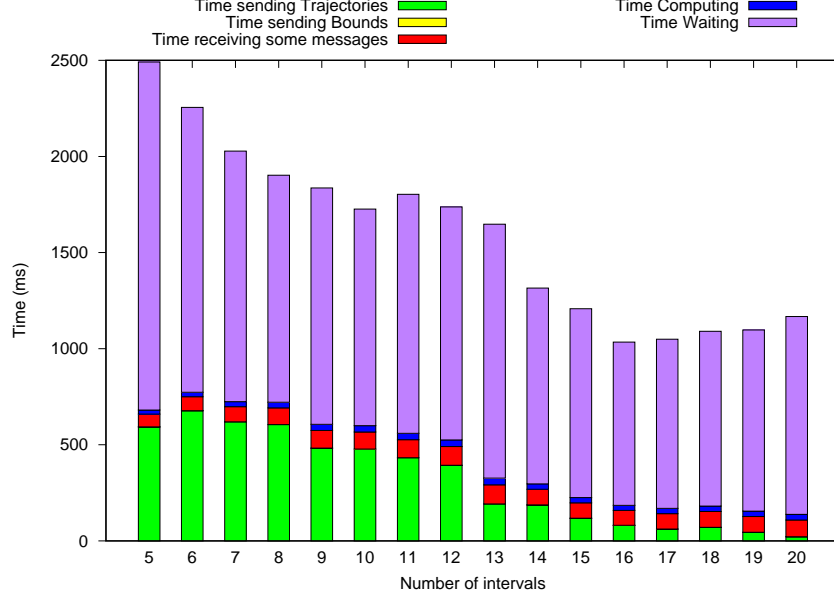
10

Figure 5: Repartition of the activities of the processors during the simulation.

First, we can see that, the most of the time, the servers are waiting for something to do. Since the master algorithm is not very complicated, we suppose that the network highly slows the communication between the master and the servers. The rest of the time, the servers are mainly computing some network operations (sending, receiving).

3.8.2 Short simulation

We then tried to look if there is a difference when the simulation is short. The following experiment are made with exactly the same parameters than upward, except the simulation length, which is now of $30,000$. In this situation, the probability of coupling is very low.
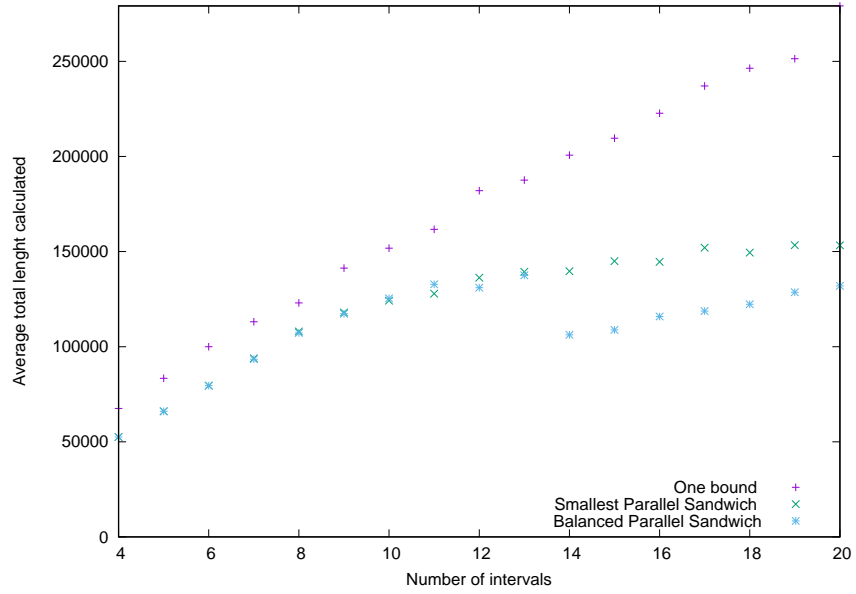
Figure 6: Evolution of the average of the sum of the lengths of the intervals computed for 100 simulations of $30,000$ events.
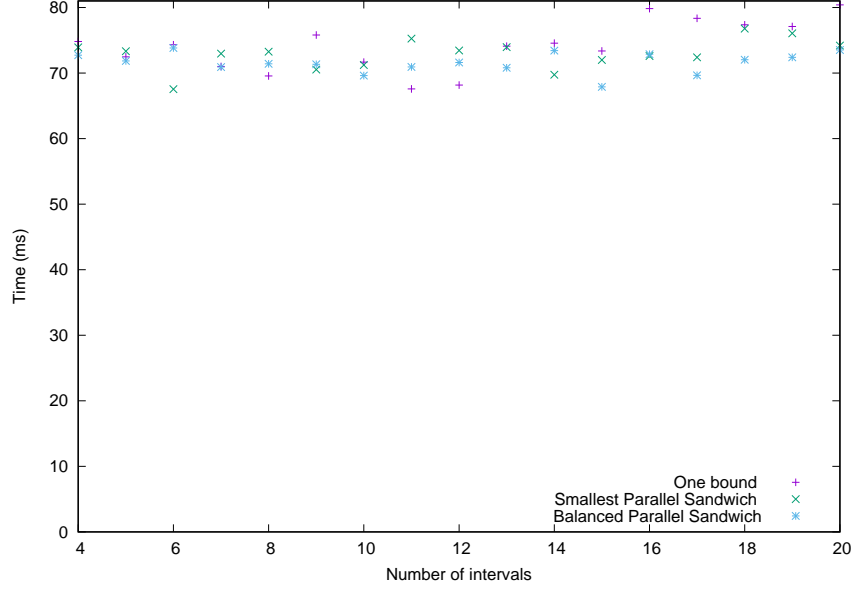
Figure 7: Evolution of average for 100 simulations of 30, 000 events.

The trend of the curves on fig. 3.8.2 are notably similar to the curves of fig. 3.8.1. On the other hand, the three algorithm seems to run in a similar time on fig. 3.8.2. Once again, this is due to the domination of the network time, which is the same for all algorithms, since the messages are all short. On a single processor, the average time on 100 simulations for 30, 000 events is to 23.06 ms. TODO: expliquer un peu plus, de toute facon on calcule tout les intervals en attendant le gars du debut vu que ca a tres peu de chance de coupler

=======

TODO: Dire ce qui est utilisé comme techno entre les raspberry et donc la latence induite (à mesurer) ¿¿¿¿¿¿¿ 43f1c719ccab17cdd0288dfaf16a9cd2cea56835

3.9 Number of server used

We now want to study the impact of the number of server used. As we noticed in fig. 3.8.1, there is an interessant point when the number of intervals is twice the number of servers (this threshold was observed for any number of servers between 2 and 7). We then look at the execution time of 100 simulation from 2 to 7 servers, whith the BALANCED PARALLEL SANDWICH algorithm. The number of event n of the simulation is set to 210, 000, and the number of intervals is always set to $2 \times$ number of servers, since it is the first number at which the execution time of the algorithm starts to stagnate to it's lower value, for any number of processors.
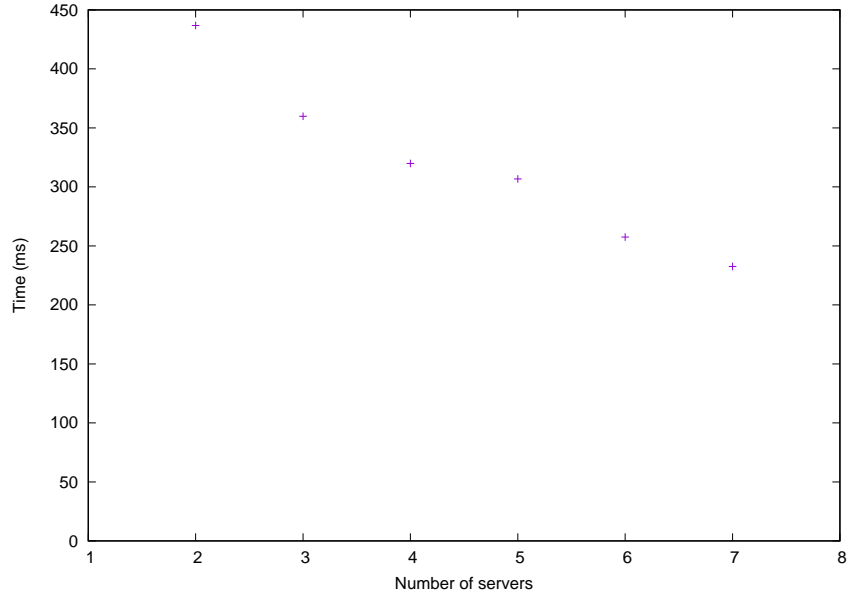
Figure 8: Impact of the number of servers on the execution time.

As we can see on fig.3.9, the more we have servers, the faster the algorithm runs. TODO: bof cette conclusion, vu que c'est le reseau qui masque tout

Practical problems: cost of the network transmission, especially for transmitting long sequences. -¿ measure the time of a two way trip for a small message and the time of sending an interval. To say that in practice we will not compute the whole sequence but statistic on it which could help reduce the use of the network.

References

[1] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.