Parallel simulation of the trajectory of a Markov chain with partially ordered states

Jean-Michel Fourneau Maël Guiraud Yann Strozecki

February 2, 2018

Abstract

In this short article, we describe several methods to compute the trajectory of a Markov chain 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 simulations.

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 $T(s, x_1, \dots, x_n) = (s_0, \dots, s_n)$ where $s_i = f(s_{i-1}, x_i)$ and $s_0 = s$. The elements $(x_0, \dots, x_n)$ are realizations of a sequence of independent and identically distributed random variables $X_0, \dots, X_n$. This models a random process on a finite set of states, which depends only on the current state, in other words a *discrete-time Markov chain.*

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 $\bot$ 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 $x_0, \dots, x_n$ of realizations of $X_0, \dots, X_n$, we want to compute the sequence $T(s, x_1, \dots, x_n)$ when $\mathcal{S}$ is a partially ordered set. We call this problem PoTraSim, for partially ordered trajectory simulation.

In practice, the values $x_1, \dots, x_n$ 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 $x_{ki+1}, \dots, x_{ki+k-1}$ using x_{ki} as a seed. When $k = \sqrt{n}$, this computation uses a parallel time of $O(\sqrt{n})$. This efficient parallelization justifies our choice of considering the sequence $x_1, \dots, x_n$ 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 reasonable 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 resources and we will set n/t to a value related to the number of machines or cores available.

2.1 Two bounds

In this section, we describe an algorithm which solves PoTraSim in parallel, that is given $s, x_1, \dots, x_n$ and an algorithm to compute f as inputs, it produces the sequence $T(s, x_1, \dots, x_n)$. We fix 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 beginning 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: $status_0 =$ ToCompute, $s_0^{min} = s_0^{max} = s$ and for all $0 < j < k$, $status_j =$ ToCompute, $s_j^{min} = \perp$ and $s_j^{max} = \top$. 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 $T(s_j^{min}, x_{jt}, \dots, x_{(j+1)t})$ and $T(s_j^{max}, x_{jt}, \dots, x_{(j+1)t})$. The last value of the two computed sequences, let us denote them by s_{min} and s_{max}, are then compared to the bounds of the next interval if there is one. If $s_{min} > s_{j+1}^{min}$ or $s_{max} < s_{j+1}^{max}$ then better bounds have been found and P sets $s_{j+1}^{min} = s_{min}$, $s_{j+1}^{max} = s_{max}$) 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 at the end. The algorithm we have described is called Speculative Sandwich and we give its pseudocode in Figure 1.

TODO: Update the pseudo code

Algorithm 1 Speculative Sandwich

```
Input: Size of the intervals $t$
  // Initialisation
  for $i <$ min(nb_machines, 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, $s_j^{min} \leq s_{j*t} \leq s_j^{max}$. We can prove that this is true during all the algorithm by induction on the computation time. Indeed, s_j^{min} and s_j^{max} are changed when a simulation from s_{j-1}^{min} and s_{j-1}^{max} find tighter bounds. By induction hypothesis $s_{j-1}^{min} \leq s_{(j-1)*t} \leq s_{j-1}^{max}$ since they have been computed before. The monotony of f guarantees that $j, s_j^{min} \leq s_{j*t} \leq s_j^{max}$. This guarantee that when $s_j^{min} = s_j^{max}$, 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 when the algorithm stops there is an interval with status Computed and let j the smallest index such that $status_j =$ Computed. We have $j > 0$ since $status_0 =$ Done because $s_0^{min} = s_0^{max} = s$. Hence $j-1$ is positive and $status_{j-1} =$ Done by definition. But then Speculative Sandwich has updated the bounds s_j^{min} and s_j^{max} 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 its beginning bounds are tighter. Since the number of states is finished, it follows that at some point in the algorithm, for each interval its bounds are equal and its status is set to 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 available interval to a different processor simultaneously, therefore the policies are all the same. Hence they make sense in a context where the number of processors is limited as shown in Section **??**.

We propose two different policies. The first is the Ordered policy: choose the available 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 for each meta interval i an integer variable $intervals_i$ which counts how many time an interval of the meta interval has been simulated. Then we select j the available interval with the the smallest $(intervals_{j/l}, j/l, j)$ in the lexicographic order.

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 Kn where K is a bound on the time to compute f.*

Proof. We prove the proposition by induction on the time to compute the trajectory on the first j intervals. Assume, by induction hypothesis, that a processor P has finished in time T less than Kjt to simulate an interval so that the first j intervals are correctly computed. There is a time less or equal to T, such that a processor has finished the simulation of the interval $j-1$ and its status has been set to Done. Then by construction of Speculative Sandwich, the interval j had its status set to ToCompute and $s_j^{min} = s_j^{max}$ before time T. If $status_j =$ Computed, by the previous remark, the interval is being simulated with the correct starting bound and will finish in less than Kt time. If $status_j =$ ToCompute, since the processor P is free, it will simulate the interval j, since the Ordered select the smallest available interval. This simulation will be finished in less than Kt time. Thus the $j+1$ first intervals will be simulated in time less than $T + Kt \leq K(j+1)t$, which proves the induction. □

We have proved that the **Ordered** policy is always as fast as the sequential algorithm. It would be interesting to understand when it is faster and by how much. If at some point of the algorithm, we have $x_j^{min} = x_j^{max}$, we say that the two sequences are **coupling** on the interval j. It is when this phenomena happens fast and frequently that the speed-up can be important. We illustrate that in the next proposition by assuming a simplified deterministic coupling time. We could derive the same kind of bounds for any random process given the mean and the variance of the coupling time.

Proposition 2. *Let f be such that, there is a m such that for any $x_1, \dots, x_m$, the last element of $T(\perp, x_1, \dots, x_m)$ and the last element of $T(\perp, x_1, \dots, x_m)$ are equal. Then the parallel time is $O(Kn/p * \lceil m/t \rceil)$ where t is the size of an interval, K a bound on the evaluation of f and p is the number of processors and satisfies $p < n/t$.*

Proof. Each time an interval is simulated again, it means that its initial bounds have been improved. By a simple recurrence, it implies that the bounds obtained after simulating an interval i times can be seen as the simulation of i consecutive intervals. It means that we will always have a coupling in an interval when it has been simulated $\lceil m/t \rceil$ times, which implies that each interval are simulated at most $\lceil m/t \rceil$. Moreover, in this simple model, when bounds have been improved at the beginning of the interval, we obtain better bound at the end. It means that there are always p interval being simulated, d interval done and $n - d - p$ intervals to do during the simulation. □

TODO: Preuve un peu rapide et ça serait intéressant de la faire en probabiliste -¿ demander à JMF

2.2 One bound

TODO: description of the algorithm and complexity bounds

3 Experimental evaluations

In this part, we experiment the proposed algorithms in the context of a multicore machine and of distributed computing. The codes used for the experimentations can be found on the author webpage .....

3.1 Settings

In this section, we present the Markov chain we simulate in our experiments. We choose a network of queues, classically used to model buffers in telecommunication networks. The example is complicated enough that theoretical results are hard to get and it needs to be simulated for a long time to understand its behavior, justifying the parallel simulation approach.

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 $C_i < BUFF\_MAX$.
- A service: if $C_i > 0$ 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 $C_{i+1} < BUFF\_MAX$ otherwise the client is lost.
- A departure: the client leaves the system. The number of client of the queue is decreased by one if $C_i > 0$.

For the queue i, the probability of an arrival, a service and a departure are denoted by respectively a_i, s_i and d_i. The last queue has no service, thus $s_{m-1} = 0$. There are a total of $3m-1$ different events that can change the state of the system. In the following experiments, we chose the parameters $P = 0.75$, $\mu = 300$ and $load = 1$ and fix the probabilities as follows:

- $a_0 = \mu \times load$
- $\forall 0 < i < m$, $a_i = (1 - P) \times a_0$

- $\forall i < m$, $s_i = P \times \mu$
- $\forall i < m$, $d_i = (1 - P) \times \mu$

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

With those parameters, every queue have an average load of 1. This load has been experimentally calculated as the load for which the coupling time of the two bounds in the largest. TODO: J'ai changé le 1.1 en 1 mais bon cc'est surtout cette remarque que je trouve chelou et que j'ai besoin de comprendre mieux.

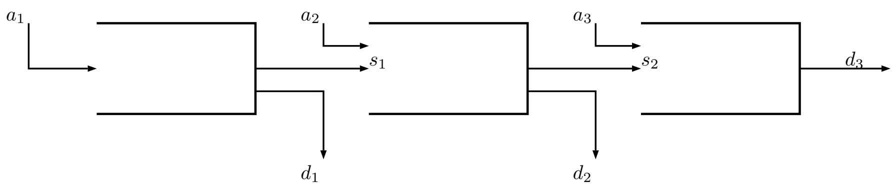

Figure 1: A system with 3 queues

3.2 Multicore machine

In this section, we present the results of our simulation on a multicore machine. This is close to the PRAM model we have presented since the communications cost between cores are small and there is a large shared memory which can be used to write the trajectories as they are computed. Hence we made a simple optimization to our algorithms: when the lower bound and the upper bound are computed in an interval, we test whether there is some coupling. If we detect a coupling at step T in the interval, we compute the real trajectory from this moment on and not the *two bounds*. Moreover, we store the step T so that the next time we simulate the interval, the simulation is stopped at T.

To ensure that two cores do not write or read the status of an interval at the same time we need to use locks, which can cause some performance penalties which are not visible when writing the ideal algorithm for a PRAM. That is why we introduce a simple variant of Speculative Sandwich that we call Fixed Interval. If we have p cores available, then $t = n/p$ and each processor is affected to a specific interval. The algorithm is the following: each processor simulates its interval and modify the bounds of the next and continue to do so until it has computed the right trajectory for its interval. This methods may waste time, in particular because we need to set $t = n/p$ which means that as soon as the first interval is computed, a processor will go unused. On the other hand, it can be implemented without locks since each processor simulate a different part of the trajectory and that we do not need to read and set some status for the processors.

TODO: Dire la machine + code en C avec OpenMP + le générateur de nombre aléatoire

3.3 Distributed computing

Dire pourquoi ça se passe pas trop mal en distributed computing. Expliquer qu'on ne va pas envoyer la trajectoire (pas souvent utilisé en pratique) et qu'on va faire gaffe au temps de transfert. 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. TODO: Dire les machines + le réseau + C et sockets over IP pour la com 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.

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.

3.4 Network impact

TODO: Pas besoin de faire une sous section, mais le but c'est de voir le cout d'un aller-retour par rapport à une étape de sim pour bien régler la taille des intervalles 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 linerarly 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.

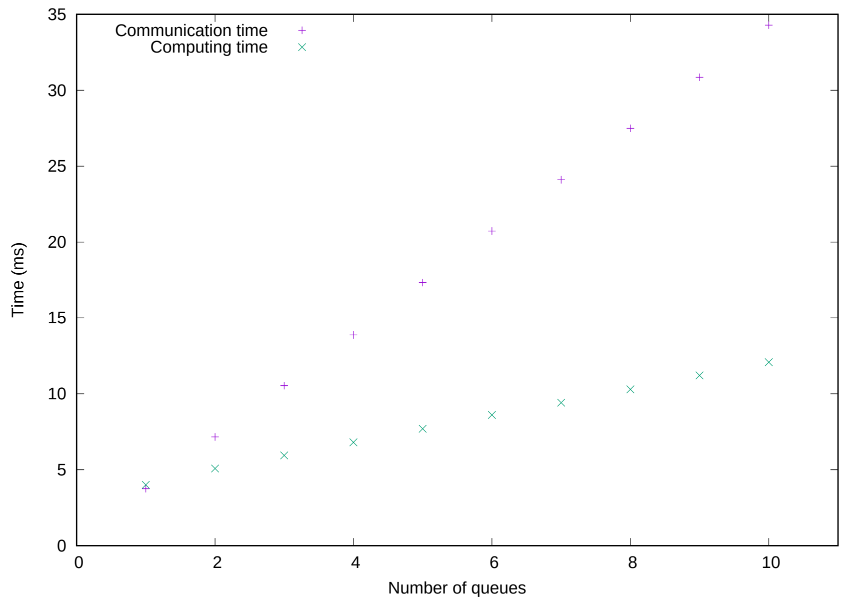

Figure 2: Evolution of the computing and communication time of a trajectory of $10,000$ events

3.5 Algorithm performance evaluation

We now want to measure the performance of the Smallest Parallel Sandwich and the Balanced Parallel Sandwich, compared to the One bound algorithm (cf sec 2.2). In the following experiment we thus take a random process with 5 queues, which allows us to have a reasonable amount of computation 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 3.1. Thus, this average coupling time is to $28,049$, calculated over 100 simulations.

3.5.1 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. 3.5.1 and 3.5.1 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.

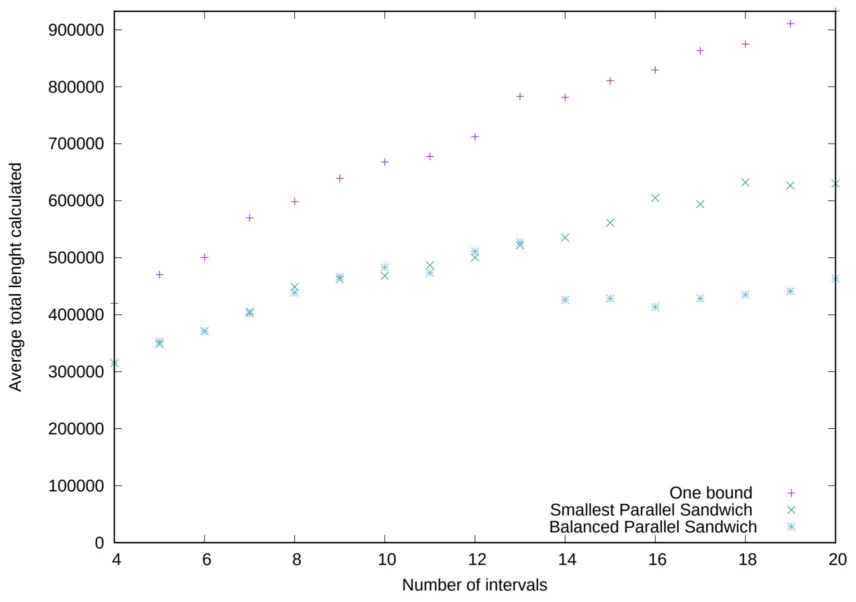

Figure 3: Evolution of the average of the sum of the lengths of the intervals computed for 100 simulations of $210,000$ events.

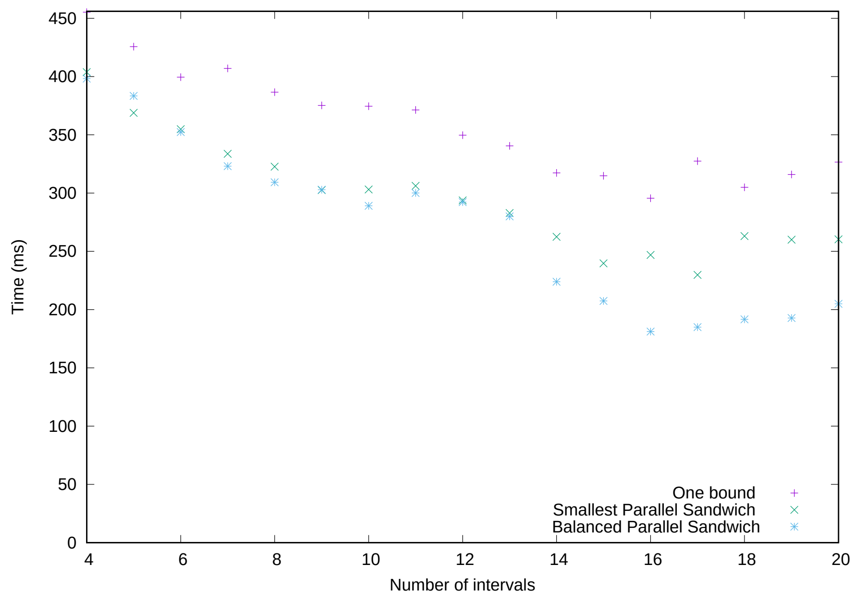

Figure 4: Evolution of average for 100 simulations of 210, 000 events.

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. 3.5.1 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. 3.5.1 , 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. 3.4, the more the interval is long, the more network increases. Moreover, the experiment in sec. 3.4 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. 3.5.1 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.

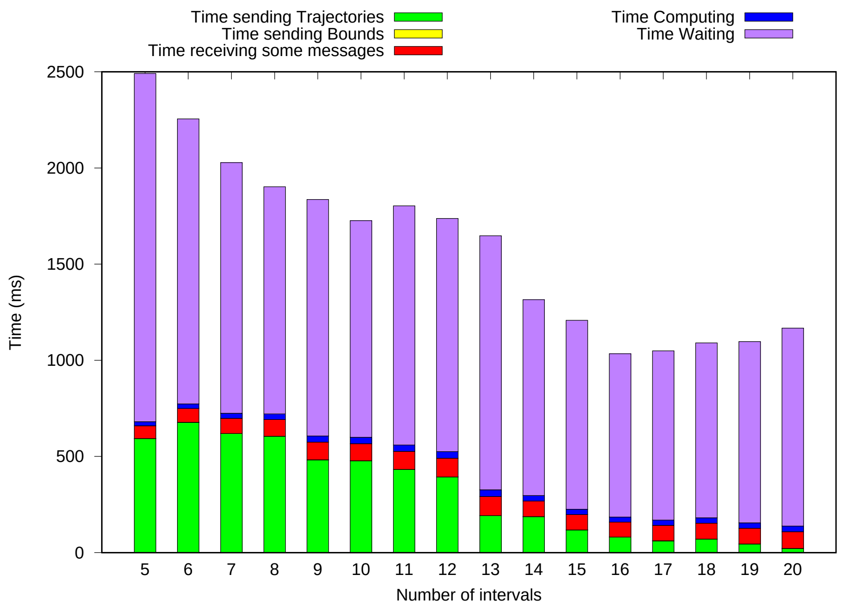

Figure 5: Repartition of the activities of the processors during the simulation.

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).

3.5.2 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.

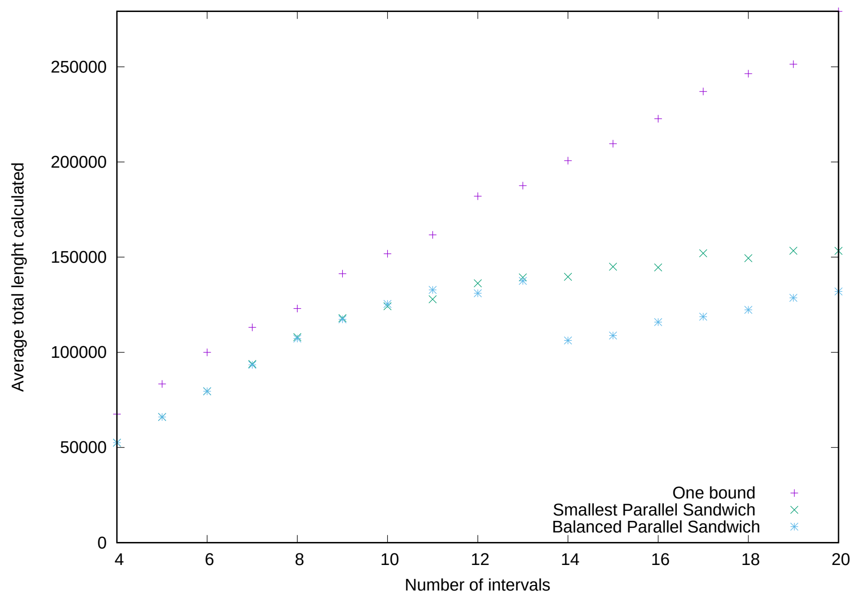

Figure 6: Evolution of the average of the sum of the lengths of the intervals computed for 100 simulations of $30,000$ events.

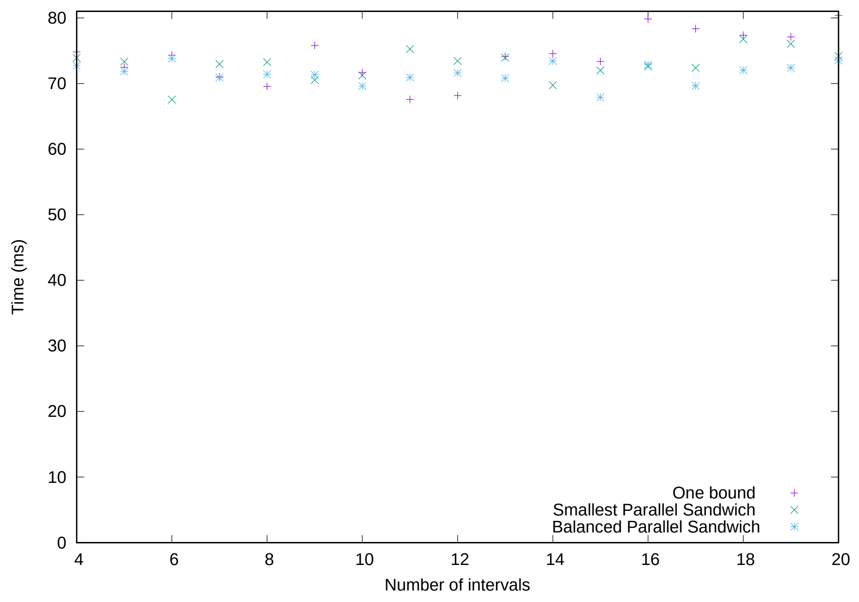

Figure 7: Evolution of average for 100 simulations of 30, 000 events.

The trend of the curves on fig. 3.5.2 are notably similar to the curves of fig. 3.5.1. On the other hand, the three algorithm seems to run in a similar time on fig. 3.5.2. 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

3.6 Number of server used

We now want to study the impact of the number of server used. As we noticed in fig. 3.5.1, 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.

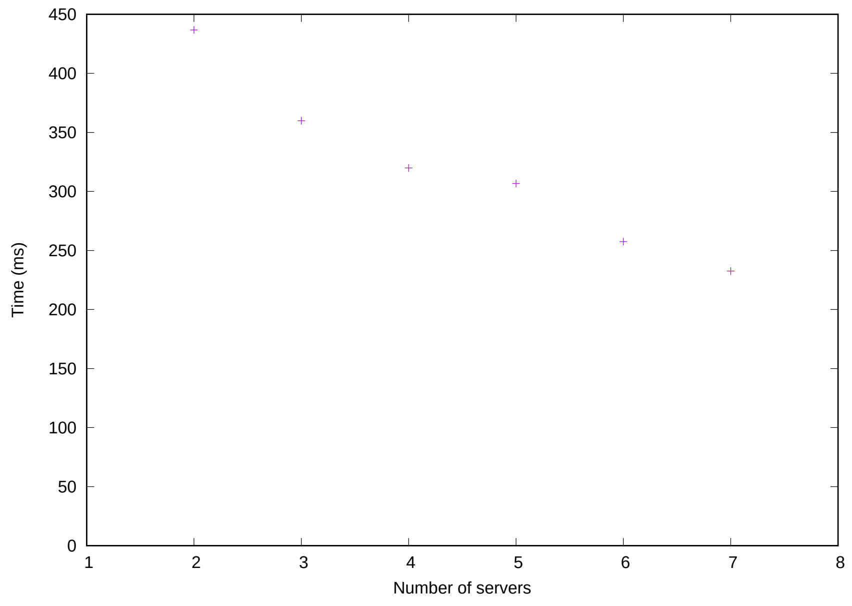

Figure 8: Impact of the number of servers on the execution time.

As we can see on fig.3.6, 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.