Internship Report

Ezequiel Torti Lopez

August 7, 2014

Contents

basicstyle=

1 Motivation and Problem Statement

Simulation is the third pillar of science, allowing to study complicated phenomenons through complex models. When the size or complexity of the studied models becomes too large, it is classical to leverage more resources through Parallel Discrete-Event Simulation (PDES).

Still, the parallel simulation of very fine grained applications deployed on large-scale distributed systems (LSDS) remains challenging. As a matter of fact, most simulators of Peer-to-Peer systems are sequential, despite the vast literature on PDES over the last three decades.

dPeerSim is one of the very few existing PDES for P2P systems, but it presents deceiving performance: it can achieve a decent speedup when increasing the amount of logical processes (LP): from 4h with 2 LPs down to 1h with 16 LPs. But it remains vastly inefficient when compared to sequential version of PeerSim, that performs the same experiment in 50 seconds only. This calls for a new parallel schema specifically tailored to this category of Discrete Event Simulators.

Discrete Event Simulation of Distributed Applications classically alternates between simulation phases where the models compute the next event date, and phases where the application workload is executed. We proposed in~[**?**] to not split the simulation model across several computing nodes, but instead to keep the model sequential and execute the application workload in parallel when possible. We hypothesized that this would help reducing the synchronization costs. We evaluate our contribution with very fine grained workloads such as P2P protocols. These workloads are the most difficult to execute efficiently in parallel because execution times are very short, making it very difficult to amortize the synchronization times.

We implemented this parallel schema within the SimGrid framework, and showed that the extra complexity does not endangers the performance since the sequential version of SimGrid still outperforms several competing solutions when our addition are present but disabled at run time.

To the best of our knowledge, it is the first time that a parallel simulation of P2P system proves to be faster that the best known sequential execution. Yet, the parallel simulation only outperforms sequential one when the amount of processes becomes large enough. This is because of the pigonhole principle: when the amount of processes increases, the average amount of processes that are ready to run at each simulated timestamp (and can thus run in parallel) increases. When simulating the Chord protocol, it takes 500,000 processes or more to amortizing the synchronization costs, while the classical studies of the literature usually involve less processes.

The current work aims at further improving the performance of our PDES, using several P2P protocols as a workload. We investigate the possible inefficiency and propose generic solutions that could be included in other similar simulators of large-scale distributed systems, be them P2P simulators of cloud, HPC or sensornets ones.

This paper is organized as follows: Section~**??** recaps the SimGrid architecture and quickly presents the parallel execution schema detailed in~[**?**]. Section~**??** explores several trade-offs for the efficiency of the parallel sections. Section~**??** analysis the theoretical performance bound, and discusses the previous work at the light of the Amhdal law. Section~**??** proposes an algorithm to automatically tune the level of parallelism that is adapted to the simulated application. Section~**??** concludes this paper and discusses some future work.

2 Context

In the previous work ~[**?**] we proposed to parallelize the execution of the user code while keeping the simulation engine sequential. This is enabled by applying classical concepts of OS design to this new context: every interaction between the user processes (from now on, user processes and processes mean the same thing) and the simulated environment passes through a specific layer that act as an OS kernel.

A novel way to virtualize user processes (*raw contexts*) was crafted to improve efficiency and avoid unnecesary system calls, but other ways to do this can be found for the sake of portability, such as full featured threads, or POSIX ucontexts. A new data structure to store the shared state of the system and synchronize the process execution was implemented as well.

A new specific layer that acts as the OS kernel was implemented in SimGrid to emulate systems calls, called *requests*, and each time a user process want to interact with other process, or the kernel itself, it raises a *request*. After that, the engine takes control of the program and answer the *requests* of each process. This way the user processes can be parallelized in a safe manner.

Experimental results showed that the new design does not hinder the tool scalability. In fact, the sequential version of SimGrid remains orders of magnitude more scalable than state of the art simulators. The difficulty to get a parallel version of a P2P simulator faster than its sequential counterpart was also revealed in ~[**?**], being the first time that a parallel simulation of Chord runs faster than the best known sequential implementation.

An interesting result showed in the previous work is that the speedups only increased up to a certain point when increasing the amount of working threads. We also have proved that for small instances, parallelism actually hinders the performance, and that the relative gain of parallelism seems even strictly increasing with the system size.

Now we are closer to the optimal Amdahl's law threshold, that means that we have reach a limit on the parallelizable portions of the code in our proposed model. The remaining optimizations seek for a final speedup, trying to get a better parallel threshold dynamically depending on the simulation, and better performance of the threads taking in count their distribution on the CPU cores and the different synchronization modes (futex, POSIX primitives or busy waiters).

3 Performance Analysis

3.1 Current speedup achieved

We want to find the maximum speedup achieved with our current parallel model. For that, a test a benchmark test is run to get the timings of a typical sequential and parallel executions. After that, and using the Amdahl's law, we can retrieve the real speedup achieved with our system.

But first we want to prove that our benchmarks are not intrusive, that is, our measures do not really affect the overall performance of the system. For that, the experiments are run with and without benchmarking, using the

Precise mode, and then a comparation of both is made to find if there is a significative breach in the timings of both experiments.

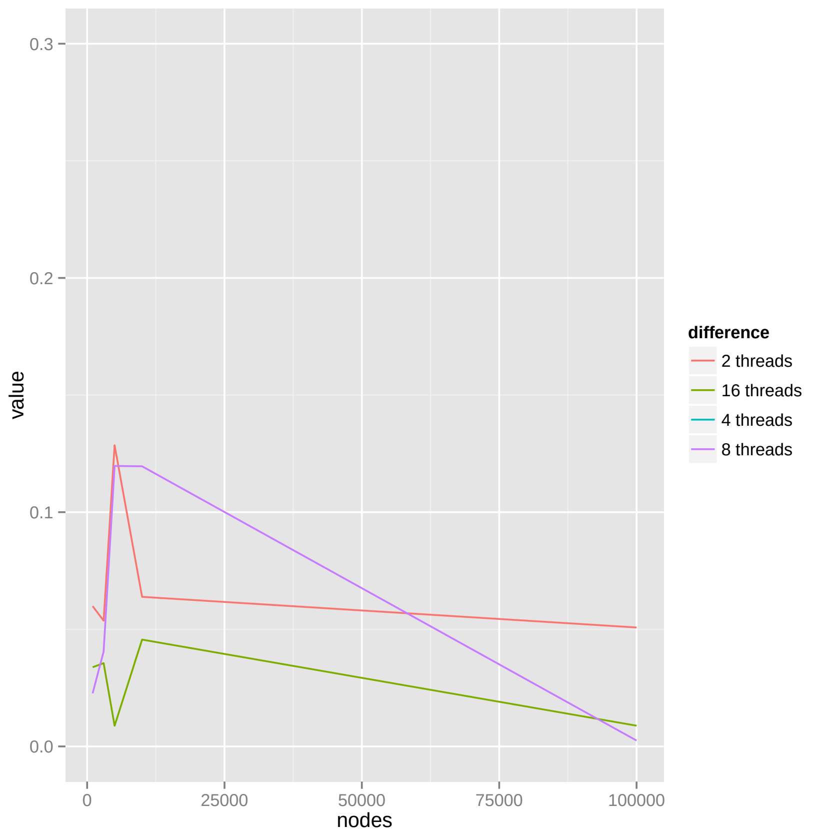


Figure 1: Percentage difference of time between benchmarked and original version.

As it can be seen in the Figure **??**, the maximum difference in the execution time of both versions is lesser than 15%, and even lower with sizes bigger than 100000 nodes. That allow us to conclude that the benchmarking done to calculate the speedup is not intrusive since it does not affect the parallel nor sequential executions of the simulation in a significant way.

The experiment to calculate speedups involves the Chord simulation, using the Precise model of our engine, and running it with 2,4,8,16 and 24 threads. The actual speedup obtained can be seen in the Figure **??**. It is clear from that graph that the real speedup with our parallel model is obtained when the size of the problem is bigger than 100000 nodes. This confirms what we have proved in ~[?].

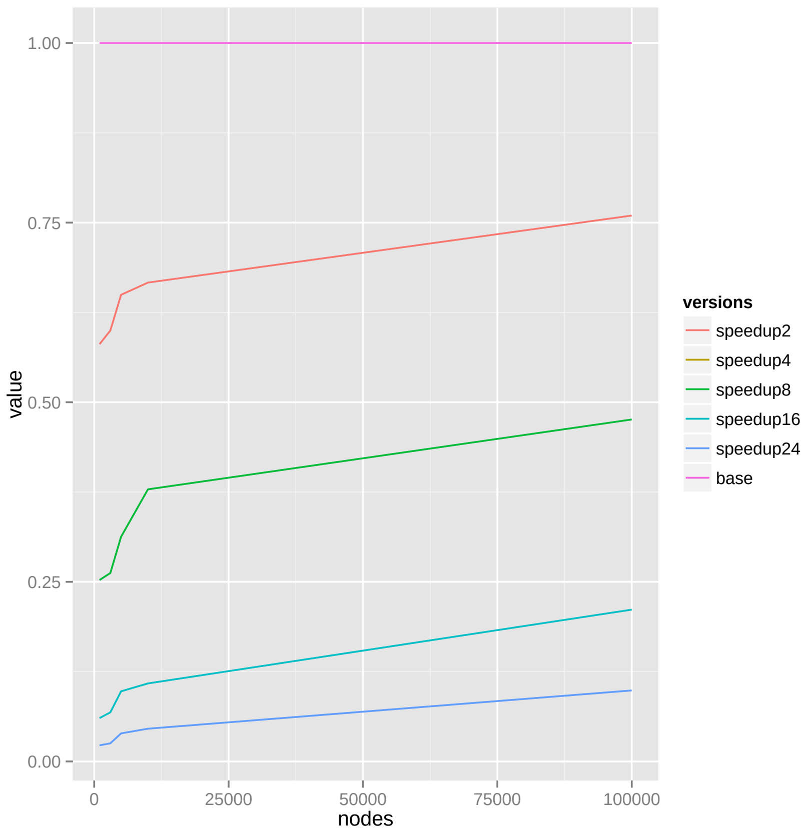


Figure 2: Real speedup achieved using parallell mode in Chord simulation.

3.2 Parallelizable portions of the problem

This experiment is based on a typical Chord simulation, and the data wanted is the following: ID of each Scheduling Round, time taken by each Scheduling Round and number of process executed in each scheduling round.

What we want to prove is that the limit on the speed up reached is due to the fact that we are very closer to the line that define what is parallelizable in our model and what is exeuted sequentially. As it can be seen in the Figure **??** , the amount of processes computed by each scheduling round is only one most of the times, so the parallel execution is not possible in that instances.

The remaining processes are executed in parallel due to the parallel execution threshold already setted up in SimGrid (which can be modified), but it only represents the 31% of the total amount of user processes in a typical run.

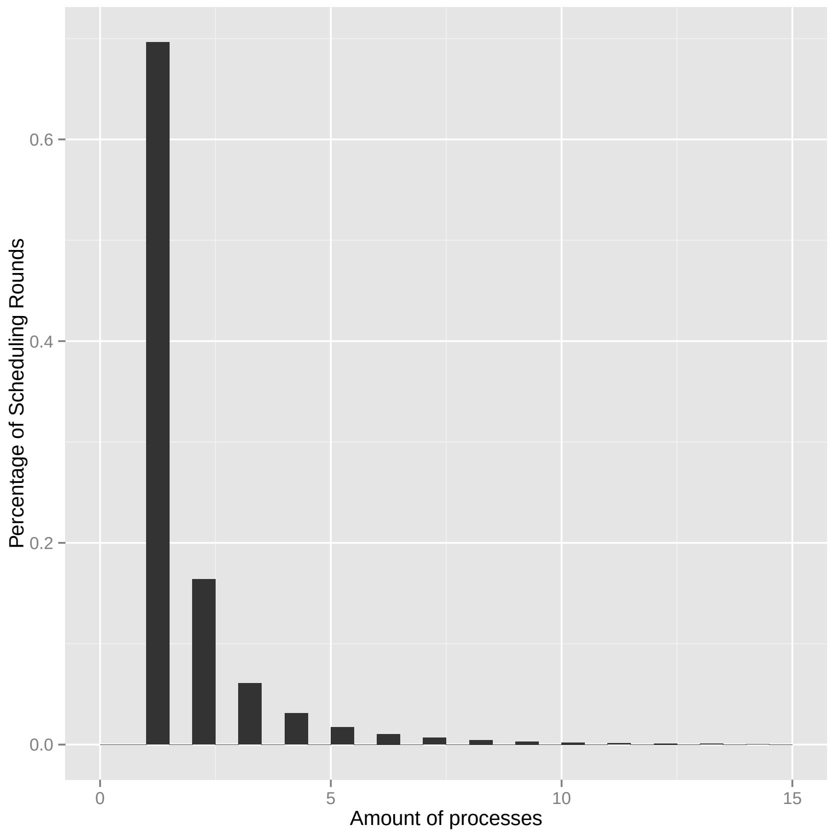


Figure 3: Proportion of scheduling rounds computing processes.

Besides that, the Figure **??** show that when the amount of processes is biggger, then the sequential execution time is bigger. That means that parallelizing that remaining 31% of processes is what gets makes the achieved speedup.

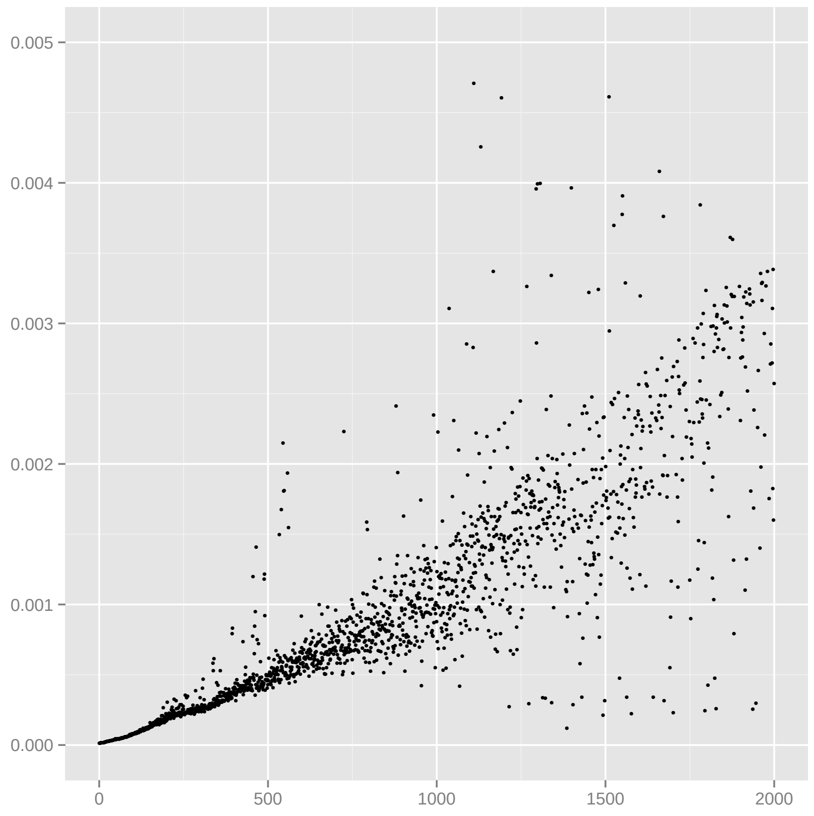
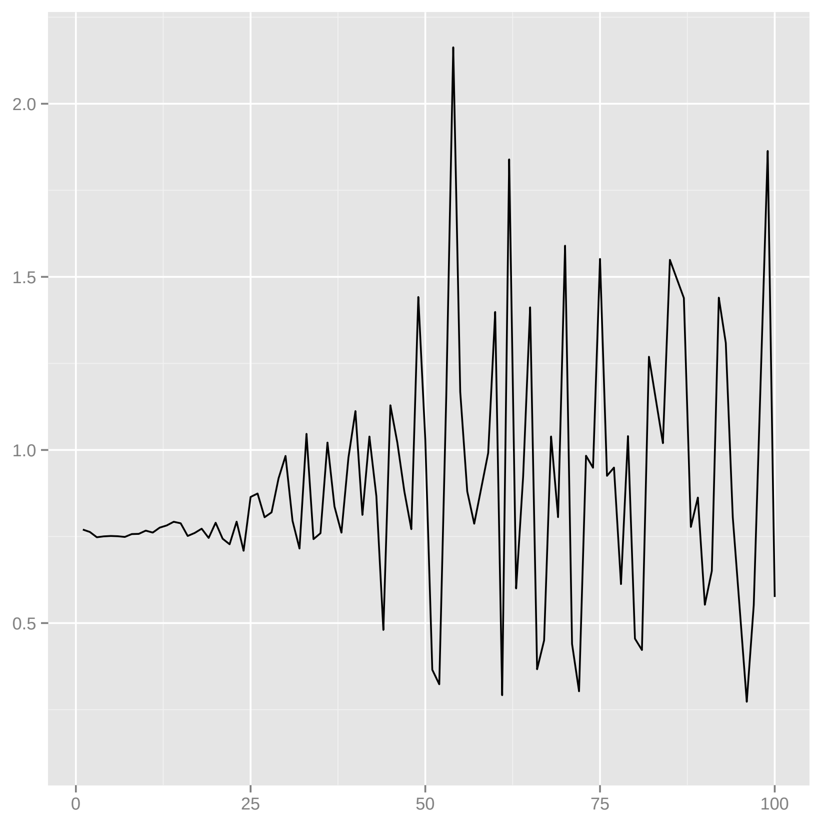


Figure 4: Mean of times depending on the amount of processes of each scheduling round.

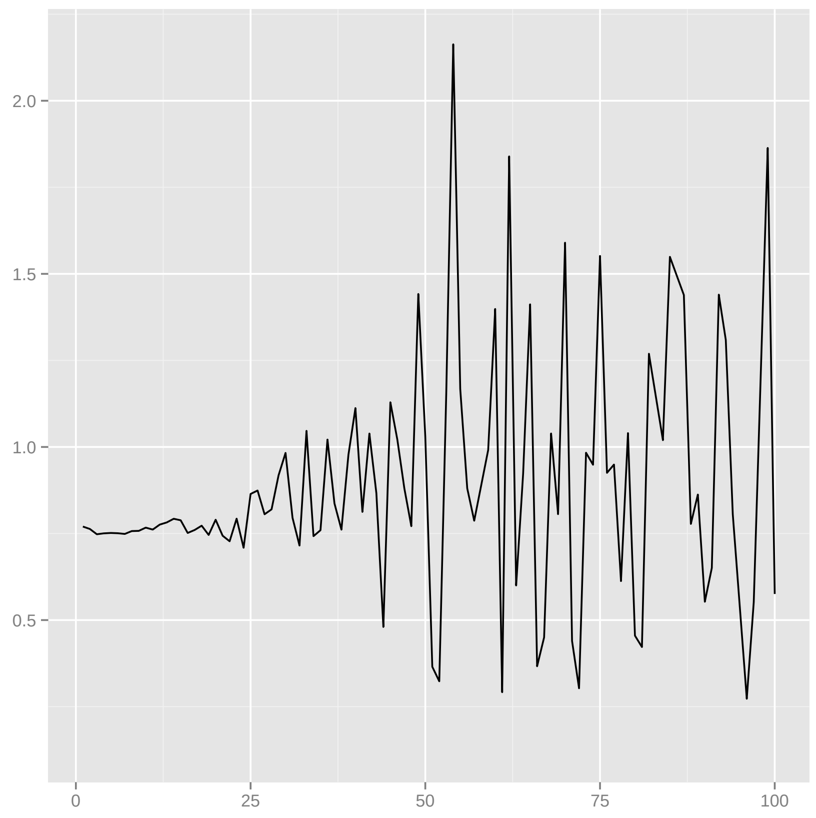
4 Optimal threshold for parallel execution

4.1 Getting a real threshold over simulations

The threshold wanted is how many processes are the right amount to be executed in parallel when it is necessary, and when is it better to execute them in a sequential way. Initially, what we want is to find an optimal threshold for the beginning of any simulation. For that, a series of experiments have to be run using <version> of SimGrid. That is why we test the performance of the engine in an exhaustive way, benchmarking the scheduling rounds timings in parallel and sequential executions, and finding the best average option for a simulation.



2.0
1.5
1.0
0.5
0
25
50
75
100



As it can be seen in the Figure **??**, despite being in a constant or precise model, the speedup can be found starting from 45 user processes.

5 Optimizations

??

5.1 Binding threads to physical cores

Talking about multicores CPUs, parallelization through threads is well proved to be a good optimization, as we said in the previous section **??**. But there is still some improvements that can be done.

Thread execution depends heavily on the operative system scheduler: when one thread is *idle*, the scheduler may decide to switch it for another thread ready to work, so it can maximize the occupancy of the cpu cores, and probably, run a program in a faster way. Or it may just want to switch threads because their execution time quote is over.

And when the first thread is ready to work again, the cpu core where it was before might be occupied, forcing the system to run the thread in another core. Of course this depend on which scheduler implementation we are using.

Regardless of the situation, migration of threads between cores entails an increase of cache misses, which in a big size simulation can be detrimental to the performance.

Moreover, in a dedicated computer with many cores (like a cluster node, for example), thread migration between the cores might be senseless and might introduce unnecessary cache misses.

This is actually our case: the amount of cache misses is XX% in a typical Chord simulation while with threads binding it is reduced until XX%. TODO: a quick study of cache misses using perf with small amount nodes. Just write the percentage, dont plot/table nothing

In order to avoid these cache misses produced by regular context switching of threads, Glib offers a way to bind each thread to a physical core of the CPU. Note that this is only available in Linux platforms.

A Chord simulation was run in a parapluie node, with 24 cores, and the speedup gained binding the threads to cores was XXX in the best case, as it can be seen in the Figure **??**.

5.2 Parmap between N cores

5.3 Busy Waiters

5.4 Performance Regression Testing

5.5 Adaptive algorithm to calculate threshold

Finding an optimal threshold and keep it during all the simulation might not always be the best option: some simulations can take more or less time in the execution of user processes. If a simulation has very efficient processes, or processes that don't work too much, then the threshold could be inapropiate, leading to parallelize scheduling rounds that would run more efficiently in a sequential way. That's why an algorithm for a dynamic threshold calculation is proposed.

TODO: explanation of the heuristic... bla bla is the amount of time taken by each scheduling round, and calculate on the fly a dynamic threshold to fit better the simulation. Pseudocode

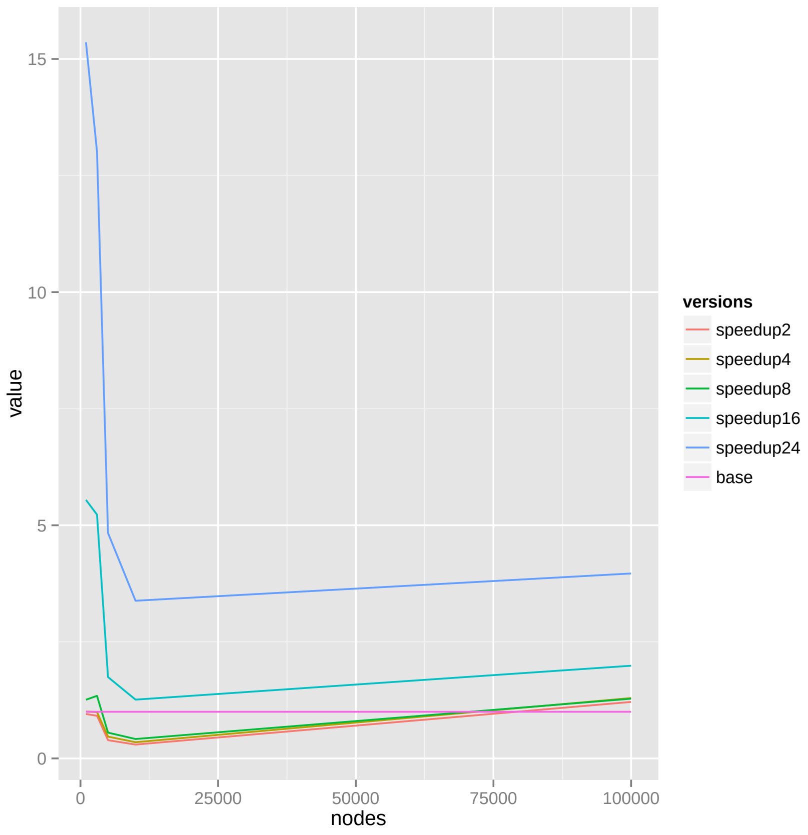


Figure 5: Speedup achieved binding threads to physical CPU cores.

```
Models used in the chord simulations
\begin{itemize}
\item Workstation model: Default vm workstation model (as it appears on ./chord --help
\item Network Model: LV08 (or Constant)
\item Cpu Model: Cas01
\end{itemize}
```

6 Conclusion

7 References

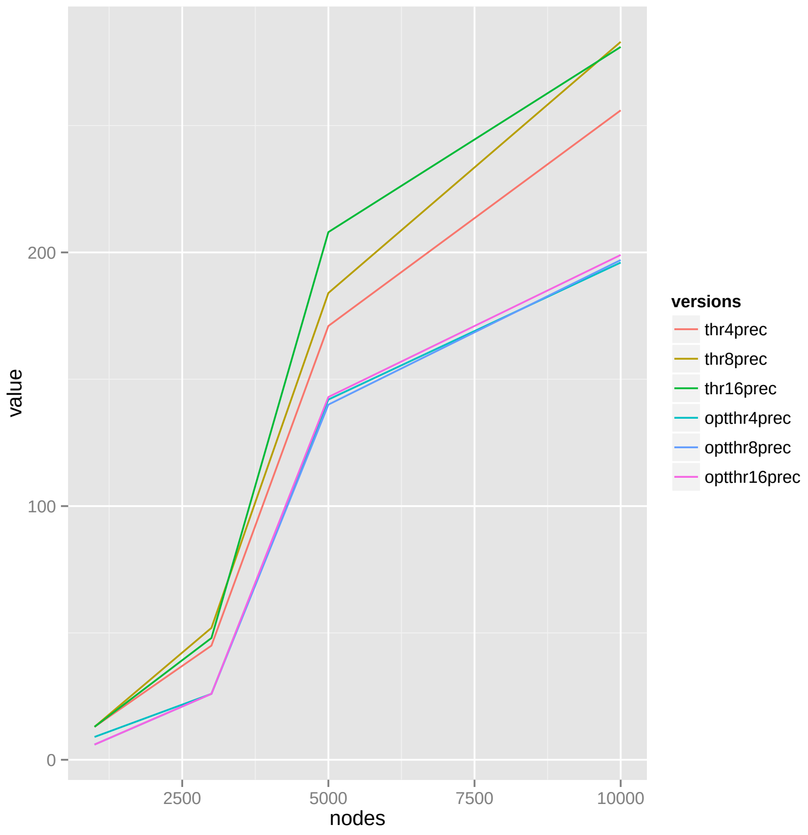


Figure 6: Chord simulation, Precise Model. Original version vs. Adaptative algorithm.