Parallel and Distributed Simulation of Large-Scale Distributed Applications

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 2 recaps the SimGrid architecture and quickly presents the parallel execution schema detailed in [**?**]. Section 3 analysis the theoretical performance bound, and discusses the previous work at the light of the Amhdal law. Section 4 explores several trade-offs for the efficiency of the parallel sections. Section 5 proposes an algorithm to automatically tune the level of parallelism that is adapted to the simulated application. Section 6 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 benchmark test is run to get the timings of typical sequential and parallel executions, so we can get the real speedup achieved with our system using the Amdahl's law.

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.

Using the Chord simulation, the experiment showed us that the maximum difference in the execution time of both versions is lesser than 10% in most of the cases, and is even lower with sizes bigger that 100000 nodes, which allow us to conclude that the benchmarking is, indeed, not intrusive.

The experiment to calculate speedups involves the Chord simulation as well, 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 1. 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 ˜[**?**].

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

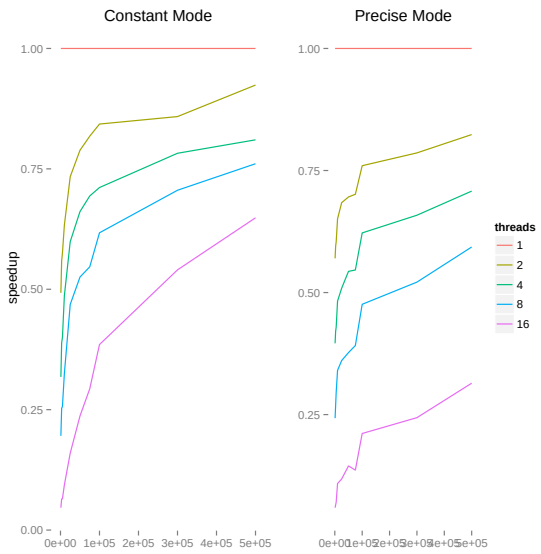


Figure 1: Real speedup achieved running Chord simulation in multithread mode.

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.

Besides that, the Figure 2 show that when the amount of processes is bigger, then the sequential execution time is bigger. That means that parallelizing that remaining 31% of processes is what achieves the current speedup.

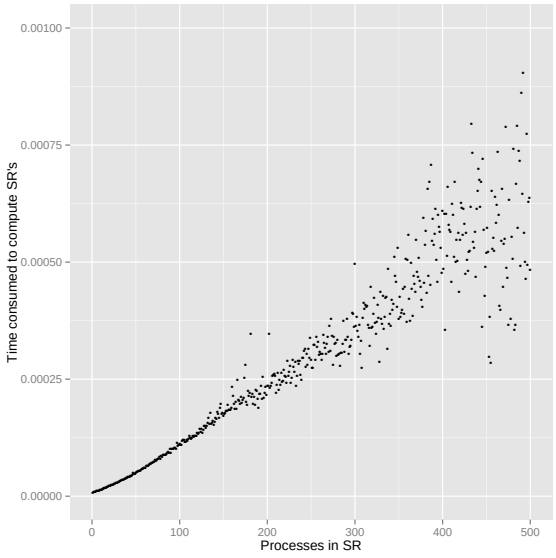


Figure 2: Execution times of SR's depending on the amount of processes of each SR (Chord simulation)

4. OPTIMIZATIONS

4.1 Binding threads to physical cores

Regarding the multicore architectures (like almost every modern CPU), parallelization through threads is well proved to be a good optimization, as we said in the previous section 3. But there are 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, and the amount of CPU migrations in a big size simulation can be detrimental for the performance.

In order to avoid these CPU migrations produced by a constant 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, binding the threads to physical cores. The CPU migration was drastically reduced (almost 97% less migrations) in all the cases. The speedup obtained with few threads (2, 4 and 8) was not big enough: x1.63 in the best case, and x1.23 in average. But when the simulation is run with a bigger amount of threads (16 or 24), the impact of having less CPU migrations is notable, being obtained speedups between x2.44 and almost x15 (depending on the amount of threads and the size of the simulation). This proves that physical binding of threads to CPU cores can be useful when a big amount of threads is needed.

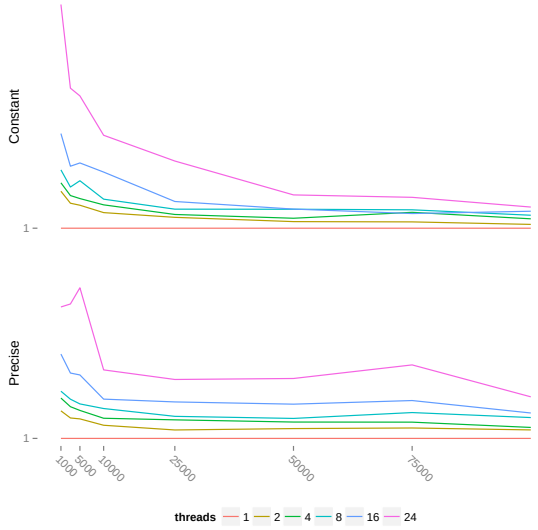
4.2 Parmap between N cores

Several optimizations regarding the distribution of work between threads were proposed: the first option is the default one, where maestro works with its threads and the processes are distributed equitably between each thread; the second one is to send maestro to sleep and let the worker threads do all the computing; the last one involves the creation of one extra thread and make all this N threads work while maestro sleeps.

The experiments were made using up to 75000 nodes in a Chord simulation with Precise and Constant modes, but no performance gain was achieved. In fact, the creation of one extra thread proved to be slightly more slower than the original version of parmap, while sending maestro to sleep and make its N-1 threads do the computation did not show any improvement or loss in performance.

4.3 Busy Waiting versus Futexes

SimGrid provides several types of synchronization between threads: Fast Userspace Mutex (futex), the classical POSIX synchronization primitives and busy waiters. While each of them can be chosen when running the simulation, futexes are the default option, since they have the advantage to implement a fast synchronization mode within the parmap abstraction, in user space only. But even when they are more efficient than classical mutexes, which run in kernel space, they may present performance drawbacks. In this section we compare the busy waiters vs. futexes synchronization types, using the chord simulation in both Constant and Precise modes, and we found that using busy waiters can result in a speedup, against futexes, between 1.08 and 1.53 using few threads (up to 8), and from 1.15 to 3 when the amount of threads is big enough (16 or 24).

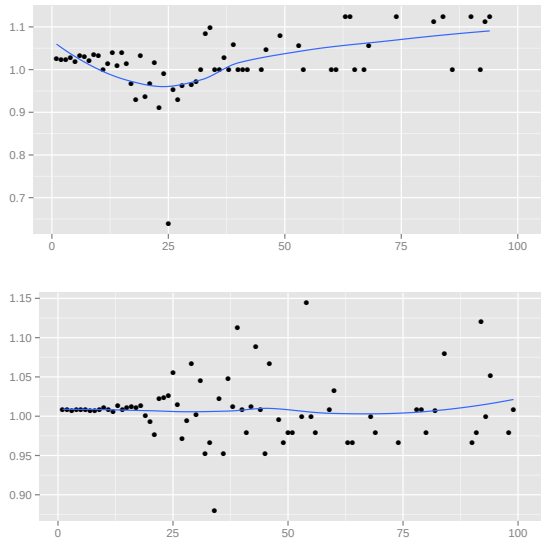


4.4 Performance Regression Testing

5. OPTIMAL THRESHOLD FOR PARALLEL EXECUTION

5.1 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 purpose, we have done a benchmark of the scheduling rounds timings in parallel and sequential executions, and have found the best average option for a simulation.



As it can be seen in the Figure **??**, the optimal threshold starts at 28 user processes for the Precise mode, while with Constant mode, the speedup of parallel vs. sequential execution of scheduling rounds is immediate.

5.2 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 inappropriate, 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.

The main idea behind this heuristic is to calculate the optimal number of processes that can be run in parallel during the execution of the simulation.

For that purpose, the times of five scheduling round are measured. A performance ratio for both of the possible parallel and sequential executions is calculated, simply by dividing the time taken by the amount of processes computed. If the sequential ratio turns to be bigger than the parallel one, then the threshold is decreased, and increased otherwise.

This results in a relative improvement in performance. As it can be seen on Figure **??**, the speedup is important in the 16 and 24 threads cases, reaching levels between 11 and 47 with small sizes (1000, 5000, and 10000 nodes), while with fewer amounts of threads (2,4,8) the speedup is not big, between 1.28 and 4.62. This can be explained by the fact that the algorithm choose more efficiently when to launch a scheduling round in parallel, and while having a lot of threads increases the costs of synchronization, an intelligent choice of when to launch them will reduce that cost.

6. CONCLUSION

APPENDIX

A. DATA PROVENANCE

This section explains and show how to run the experiments and how the data is saved and then processed. Note:

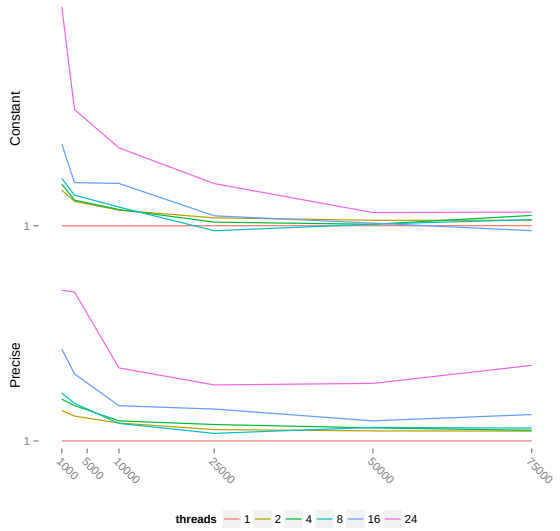


Figure 3: Speedup achieved with Adaptive Algorithm. Chord simulation, Precise and Constant modes.

that all experiments are run using the Chord simulation that can be found in examples/msg/chord folder of your SimGrid install. Unless stated, all the experiments are run using the futex synchronization method and raw contexts under a Linux environment; in a node from 'parapluie' cluster of Grid5000. The analysis of data can be done within this paper itself, executing the corresponding R codes. Note that it is even possible to execute them remotely if TRAMP is used to open this file (this is useful if you want the data to be processed in one powerful machine, such as a cluster).

A.1 Setting up a node in Grid5000

This section is useful only if you have an account in Grid5000 and want to run the experiments there. Once you have a node deployed, you can do something similar to what is in the ./scripts/g5k/setup$_{\text{node}}$.sh script to install packages/compile SimGrid.

1. **DONE** make setup$_{\text{node}}$.sh more portable to other users.

A.2 Amdahl speedup

The constant TIME$_{\text{BENCHAMDAHL}}$ must be defined in SimGrid in order to enable the required logs for this experiment. This variable can be defined in the file src/simix/smx$_{\text{private}}$.h The experiments were done using the Chord simulation that can be found in the SimGrid folder /examples/msg/chord. To run the experiment, simply run the script "testall.sh" inside the folder examples/msg/chord of your SimGrid install. The script can be found in the folder ./scripts/chord of this repository. Inside that script, the number of threads to test, as well as the amount of nodes, can be modified

The logs gathered must be processed with get$_{\text{times}}$.py, located in the same folder as testall.sh. This generates a .csv that can easily be plotted with R/gnuplot. Note that the script is self-documented.

A.3 SR Distribution

To enable Scheduling Rounds benchmarks, the constant $\text{TIME}_{\text{BENCHPERSR}}$ has to be defined. It can be defined in $\text{src/simix/smx}_{\text{private}}$.h The logs give information about the time it takes to run a scheduling round, as well as the amount of processes each SR takes. For this experiment, we are only interested in the amount of processes taken by each SR.

The script to run this experiment is ./scripts/chord/$\text{testall}_{\text{sr}}$.sh, it is mandatory to have also the $\text{get}_{\text{srcounts}}$.py scripts to process output data and save only the id of SR, time of SR and num processes of SR, in a table format.

To run, simply put $\text{testall}_{\text{sr}}$.sh and $\text{get}_{\text{srcounts}}$.py in the examples/msg/chord folder of your SimGrid, and run ./$\text{testall}_{\text{sr}}$.sh

1. **DONE** put the R code of post-processing the tables in the Data Analysis section (separated from the plot generation code)
2. **DONE** this plot has to be changed for a more representative one

A.4 SR Times

The data set used for this plot is the same as the one before. We just use the data of the sequential simulations (1 thread).

A.5 Binding threads to physical cores

The constant $\text{CORE}_{\text{BINDING}}$ has to be defined in src/xbt/parmap.c in order to enable this optimization. The benchmark is then run in the same way as the Amdahl Speedup experiment (using the same script in the same way) and then creating a table with the times using the scripts/chord/$\text{get}_{\text{times}}$.py script

A.6 parmap between N cores

This may be the experiment that requires more work to reproduce:

1. maestro works with N-1 threads This is the default setting. Just run the testall.sh, get the logs and create a table using $\text{get}_{\text{times}}$.py
2. maestro sleeps with N-1 threads To avoid that maestro works with the threads, just comment out the line: $\text{xbt}_{\text{parmapwork}}$(parmap); from the function $\text{xbt}_{\text{parmapapply}}$() in src/xbt/parmap.c

   Then just run the testall.sh benchmark the same as before. Create tables with $\text{get}_{\text{times}}$.py
3. maestro sleeps with N threads This requires to do the same as before. Then the function src/xbt/parmap.c:$\text{xbt}_{\text{parmapnew}}$ has to be modified to create one extra thread. It is easy: just add 1 to $\text{num}_{\text{workers}}$ parameter.

   Then just run testall.sh and then $\text{get}_{\text{times}}$.py the same as before.

A.7 Busy Waiters vs. Futexes performance

Enable the use of busy waiters running chord with the extra option: –cfg=contexts/synchro:$\text{busy}_{\text{wait}}$ The experiment was run with testall.sh using that extra option in the chord command inside the script. The tables were constructed using $\text{get}_{\text{times}}$.py The data regarding the futexes synchro times is the same gathered in Amdahl Speedup experiment. Then both tables can be compared in a plot.

A.8 Performance Regression Testing

1. **TODO** not yet...

A.9 SR parallel threshold

The data set is the same as SR Distribution and SR times experiments.

1. **DONE** put code to post-process tables in Data Analysis, separated from plot code

A.10 Adaptive Algorithm

The benchmark is done using testall.sh and the tables generated using $\text{get}_{\text{times}}$.py