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˜[7] 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 [4], 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 [7]. Section 3 analysis the current performance of our model. Section 4 explores several tradeoffs 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 [7] 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 (*parmap*).

The new layer acting as the OS kernel was implemented in SimGrid to emulate systems calls, called *requests*. Each time a process want to interact with other process, or the engine itself, it raises a *request*. After what is called a 'Scheduling Round' (SR), all the active processes have raised their request and wait for a response, or have finished their work. Then the engine takes control of the program and answer sequentially the *requests* of each process. This way the user processes can be parallelized in a safe manner. More details on the execution model can be found in [7] and the the official website of SimGrid [4].

Experimental results showed that the new design does not hinder the tool scalability, and even the sequential version is 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 [7], being the first time that a parallel simulation of Chord runs faster than the best known sequential implementation.

Another 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 look 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).

All the experiments were run using the facilities provided by Grid'5000 [3].

3. PERFORMANCE ANALYSIS

3.1 Current speedup achieved

To get baseline timings and a speedup plot starting from the development version of SimGrid (3.12), benchmarks to measure the execution time in Precise mode with different amount of threads (1, 2, 4, 8, 16 and 24) were done. For this we used an implementation of the well known Chord protocol [1] as workload.

The absolute times of a normal execution for the Chord simulation are presented in the table 1.

Table 1: Execution times of a normal execution of Chord with different sizes, serial and with 2 and 8 threads. The average memory consumption is reported in GB.

nodes	serial	Mem	2 thr.	Mem.	8 thr.	Mem.
10k	0:01:03	0.25	0:01:20	0.26	0:01:35	0.25
50k	0:06:20	1.24	0:07:39	1.27	0:08:03	1.25
100k	0:13:34	2.47	0:15:36	2.53	0:15:50	2.50
300k	0:50:58	7.38	0:55:18	7.54	0:57:55	7.47
500k	1:38:16	12.30	1:34:15	12.47	1:35:10	12.45
1m	4:05:41	24.53	4:00:42	24.89	3:47:28	24.91

As it can be seen in Figure 1, the memory consumption linearly increases with respect to the number of simulated nodes, and shows that each node is using around 25 KB and 30 KB of memory. A simulation with 1000 nodes, has a peak memory consumption around 30 MB (regardless of the amount of threads launched) and finishes in 4 seconds in a serial execution, and one with 1000000 nodes takes 24-25GB of memory and 3h47m to finish in the best case (parallel execution with 8 threads).

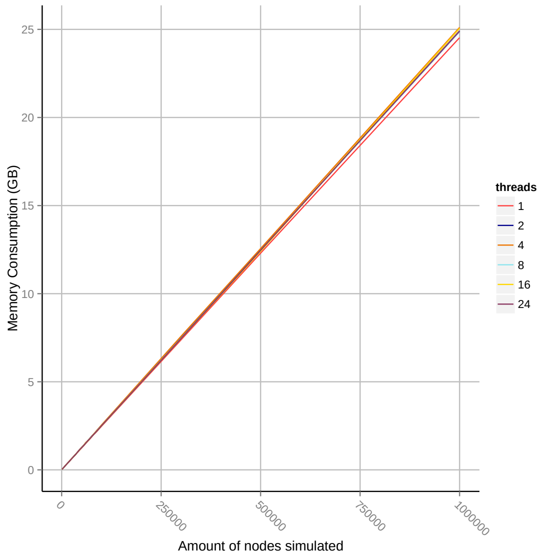

Figure 1: Memory consumptions reported in GB

The actual speedup obtained can be seen in the Figure 2. 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 300000 nodes. This confirms what was proved in [7].

Figure 2 also shows that increasing the number of threads may not be the best option to increase performance, since the best speedups are achieved with 2,4 and 8 threads. Some of the optimizations proposed in section 4 show improvements over the original versions with 16 and 24 threads, but their total times are still behind the ones of the same simulations with lesser amount of threads.

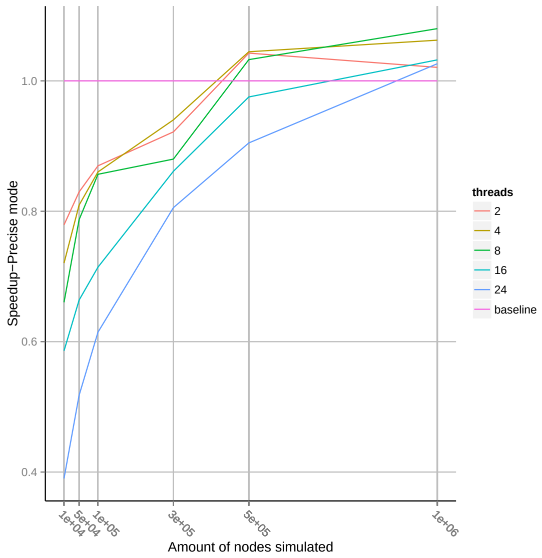

Figure 2: Baseline performance of SimGrid 3.11. Speedups achieved using multithreaded executions against the sequential ones.

3.2 Parallelizable portions of the problem

We want to analyze each SR and find any possible performance problem here, since is the portion of code that is run in parallel in our model. Using the same Chord implementation as workload, we want to gather the following data: ID of each Scheduling Round, time taken by each Scheduling Round and number of process executed in each scheduling round.

As it can be seen in the Figure 3, the amount of SR's having just one process varies between 26% and 48% (the larger the simulated size, the lower the amount of SR's that have only one process) while the others involve two or more processes. These remaining processes are executed in parallel due to the parallel execution threshold already setted up in SimGrid (which can be modified trough a parameter).

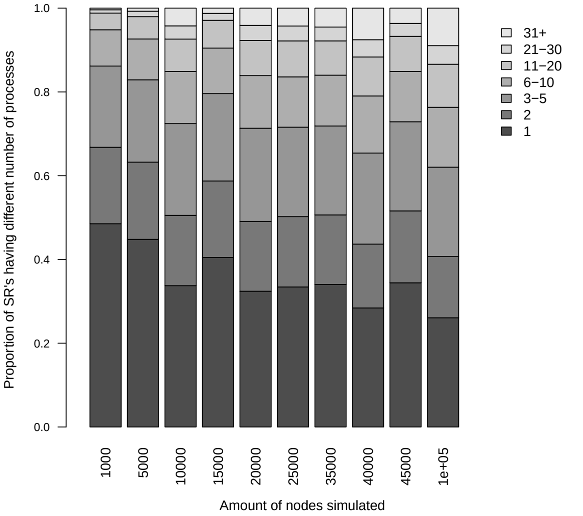

Figure 3: Proportions of SR's having different numbers of processes to compute; according to the size of nodes simulated.

However, launching a small amount of processes is inefficient due to the synchronization costs of threads. Even when Figure 4 shows that the bigger the amount of processes in a SR, the bigger the execution time, there is no speedup obtained from executing small amounts of processes in parallel, as we will see in Section 5. Hence, it would be convenient to know, during a simulation, when to launch SR in parallel and when to do it sequentially. A heuristic to accomplish that is proposed later in this document.

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 choice if done correctly. This approach, used currently by SimGrid, showed a good gain in speed with bigger sizes, as we said in Section [?] But there are still improvements that might reduce the noise and the overhead that inherently comes with threads.

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. 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.

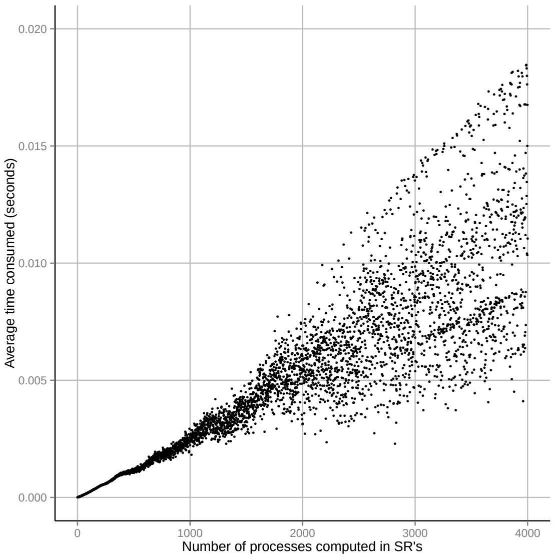

Figure 4: Average times of sequential executions of SR's depending on the amount of processes of each SR.

Regardless of the situation, or the scheduler we are using, the general problem remains: increasing the CPU migrations of threads can be detrimental for the performance.

In order to avoid these CPU migrations produced by a constant context switching of threads, GLibc [2] 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, but the relative speedup was not significant: always lower than x1.5, regardless the amount of threads/sizes. However, the bigger speedups were obtained with sizes less than 100000 nodes, which allow us to conclude that CPU migrations should be avoided when the simulation is small enough, since they introduce an unwanted overhead.

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 showed that no performance gain was achieved. In fact, the creation of one extra thread proved to be 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 that inherently come with synchronization costs. In this section we compare busy waiters and futexes performances, using the Chord example.

As it can be seen in Figure 5, the gain in speed is immediate with small sizes: the elimination of any synchronization call makes the simulation run up to 2 times faster. However, we can see the performance drop and match the one achieved with futexes with bigger sizes.

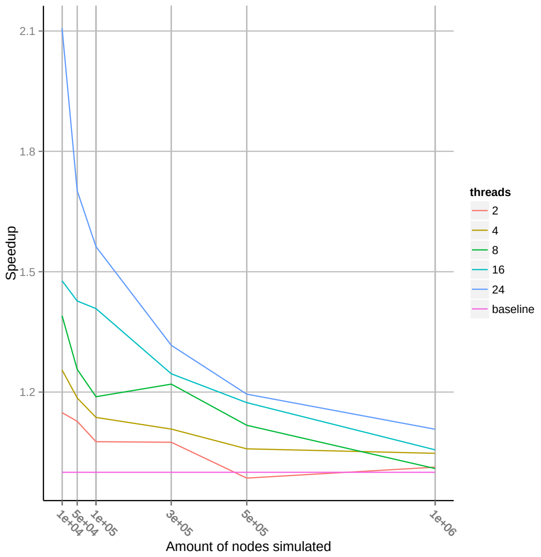

Figure 5: Relative speedup of busy waiters vs. futexes in Chord simulation.

5. OPTIMAL THRESHOLD FOR PARALLEL EXECUTION

5.1 Getting a real threshold over simulations

Plus the optimization of 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 to get each SR execution time for both parallel and serial executions, and calculated the speedup obtained in each SR.

A typical run using Chord and with 10000 examples shows that after 500 processes per SR, the speedup is always bigger than one. It is interesting to note that even for simulations with different sizes the similar limit is reached. Analyzing the data thoroughly tell us that the 83% of SR's with processes between 250 and 300 show a speedup. In consequence, 250 processes will be our base threshold for parallel execution, and the adaptive algorithm proposed in next section will be in charge of increasing or decreasing that threshold according to the needs and characteristics of the simulation.

In Figure 6 we can see the example with 10000 nodes simulated. Although it seems there is an important amount of SR with less than 250 processes that are faster in parallel, they represent only the 5% of that subset of SR's. The remaining 95% of SR's with less than 500 processes showed speedup equal or less than 1.

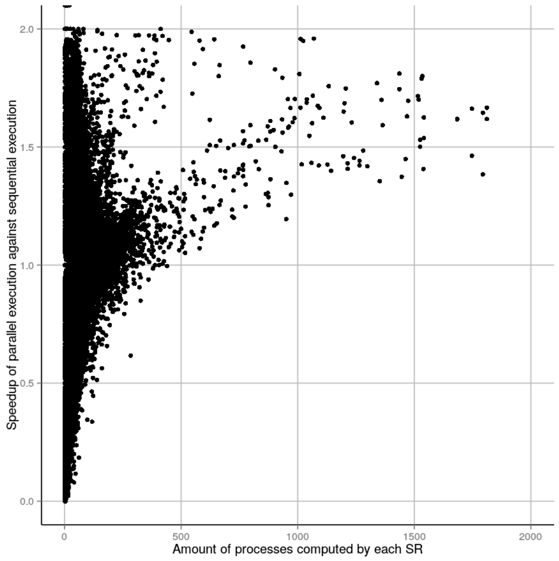

Figure 6: Speedup of parallel vs. sequential executions of SR's, depending in the number of processes taken by each SR

5.2 Dynamic estimation of the optimal 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 heuristic for a dynamic threshold estimation is proposed.

The main idea behind this heuristic (1) is to calculate the optimal number of processes that can be run in parallel during the execution of the simulation.

For that purpose, the time of a certain amount of scheduling round is measured. A performance ratio for the 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.

A naive implementation of this heuristic, showed a small relative improvement in performance. The times were certainly reduced with small sizes, since it chooses to execute the majority of the processes sequentially, while with bigger sizes (more than 100000 nodes), the speedup is insignificant.

In terms of absolute times, we can see that the execution times have been slightly reduced (up to ten minutes less in a one million nodes simulation in the best case, with 8 threads).

This improvements may be small due to the fact that we are calculating the ratio with the times of the latest SR's, and in consequence, using values that may not represent the general situation.

A new approach, using a cumulative ratio (calculated during all the simulation) instead of the one computed with the latest values, proved better in terms of performance. This approach also changes the way we do the timings: instead of benchmarking the SR's each time, we benchmark the SR's that have certain amount of processes, limited by an upper limit for parallel execution and a lower limit for the sequential ones. This is a way to prevent the timing of extreme cases (very big or very small number of processes) which may introduce errors in the estimation of the threshold, and acts like a 'window' to filter the cases we are interested in.

These limits are calculated along the simulation with the average amount of processes that have been run so far in parallel (or serial), plus the standard deviation. Since we never know beforehand the amount of SR's we will have, the average and the standard deviation are computed using the algorithm of Welford [5, 6].

When a prefixed amount of parallel and sequential SR's have been run, we proceed to update the threshold applying a similar rule of thumb: if the sequential executions were better and we have a bigger number of processes than the corresponding average, we increase the threshold, giving a chance to the serial executions to prove they are better. Otherwise, if the parallel executions performed better and the number of processes of the current SR is smaller than the average, we decrease the threshold.

Figure 7 shows that this new implementation proved to be faster than the original parallel version with sizes under 300000 nodes, while with bigger amount of nodes the speedup remains almost the same. However, this approach also avoids the increase of the threshold to unrealistic values (which may happen in the naive version, due to fact that we have a lot of SR's with small amount of processes that are computed sequentially and the fact that we increase the threshold each time a sequential execution performs better than a parallel).

All the experiments were performed setting the initial threshold to 250 processes, which was estimated as an optimal starting threshold in previous section. The heuristic lead to different final thresholds depending on the initial one, of course, since the SR's launched in parallel will not be the same from the beginning. However, experiments showed that it behaves quite stable, and there is a tendency to increase/decrease the threshold in the same simulation regardless the one at the beginning.

Regarding the memory consumption, the values remain the same in general, as it can be seen in Table 2.

Heuristic: Adaptive Threshold

```
// Amount of parallel/sequential SRs that ran
$parallel\_SRs, sequential\_SRs \leftarrow 1$
// Sum of times of par/seq SR's
$seq\_time, par\_time \leftarrow 0$
// Number of processes computed in par/seq
$process\_seq, process\_par \leftarrow 0$
// Average amount of processes parallel and sequential
$avg\_par\_proc, avg\_seq\_proc$
// Standard deviation of processes parallel and sequential
$sd\_seq\_proc, sd\_par\_proc$

procedure RunSchedulingRound
    if computed five par/seq SR's then
        $ratio\_seq \leftarrow seq\_time/process\_seq$
        $ratio\_par \leftarrow par\_time/process\_par$
        $sequential\_is\_slower \leftarrow ratio\_seq > ratio\_par$
        if $sequential\_is\_slower$ then
            if $processes\_to\_run < avg\_par\_proc$ then
                decrease($parallel\_threshold$)
        else
            if $processes\_to\_run > avg\_seq\_proc$ then
                increase($parallel\_threshold$)

    if $processes\_to\_run >= parallel\_threshold$ then
        if $processes\_to\_run < par\_window$ then
            $parallel\_SRs{+}{+}$
            start($timer$)
            execute_SR_parallel()
            stop($timer$)
            $par\_time \leftarrow par\_time$+elapsed($timer$)
            $process\_par \leftarrow process\_par + processes\_to\_run$
            $avg\_par\_proc \leftarrow$calculate_current_avg_of_par_processes()
            $sd\_par\_proc \leftarrow$calculate_current_sd_of_par_processes()
            $par\_windows = avg\_par\_proc + sd\_par\_proc$
        else
            execute_SR_parallel()
    else
        if $processes\_to\_run < seq\_window$ then
            $sequential\_SRs{+}{+}$
            start($timer$)
            execute_SR_serial()
            stop($timer$)
            $seq\_time \leftarrow seq\_time$+elapsed($timer$)
            $process\_seq \leftarrow process\_seq + processes\_to\_run$
            $avg\_seq\_proc \leftarrow$calculate_current_avg_of_seq_processes()
            $sd\_seq\_proc \leftarrow$calculate_current_sd_of_seq_processes()
            $seq\_windows = avg\_seq\_proc - sd\_seq\_proc$
        else
            execute_SR_serial()
```

6. CONCLUSION

We have shown in this work several ways to optimize large scale distributed simulations in a specific framework, namely, binding threads to physical cores, choosing a better threshold for parallel execution or choosing between different synchronization modes between threads. The optimizations were done over the open-source multi-purpose SimGrid sim-

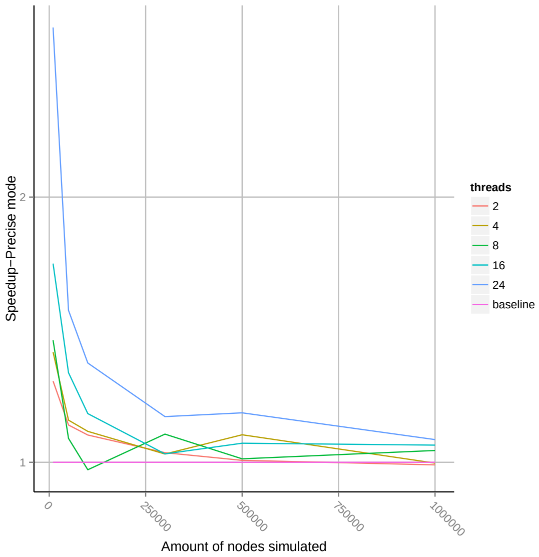

Figure 7: Speedups achieved with Adaptive threshold heuristic. Chord simulation.

Table 2: Execution times (seconds) of the Adaptive threshold heuristic, with 2,4 and 8 threads. The average memory consumption is reported in GB.

nodes	2 thr.	Mem	4 thr.	Mem	8 thr.	Mem
10k	0:01:19	0.26	0:01:20	0.26	0:01:27	0.25
50k	0:07:21	1.27	0:07:28	1.27	0:07:30	1.26
100k	0:15:16	2.53	0:15:04	2.55	0:14:48	2.51
300k	0:54:48	7.55	0:54:05	7.52	0:53:44	7.48
500k	1:38:52	12.47	1:35:19	12.56	1:31:50	12.45
1m	3:59:12	24.89	3:47:22	25.19	3:37:12	24.91

ulation framework, in its development version (3.12). The changes proposed worked in some scenarios better than others (for instance, the binding threads to cores optimization showed a real speedup in simulations using bigger amount of threads, such as 16 or 24, while using busy waiters proved to be better than futexes in simulations with small sizes and small amount of threads). Also, some of the modifications did not affect the overall performance, or even made it worst, like the parmap changes proposed in Section 4.

Most of the changes proposed gained performance with small sizes simulations (under 300000 nodes), but remained almost the same with larger ones, showing the difficulty of optimizing a complex multi-threaded system.

We certainly arrived to a point where optimization depends heavily on reducing the synchronization costs and playing with low level features of the code. An intelligent choice of when to launch processes in parallel and when to do it in a serial way proved to help with small cases but it was unnecessary with bigger ones, where there is already speedup achieved using threads to simulate user processes.

In a final note, the present work was done with the reproducible research approach in mind. Hence, the steps and scripts needed to run the experiment can be found in the appendix section.

7. ACKNOWLEDGMENTS

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

8. REFERENCES

[1] Chord (peer-to-peer). http://en.wikipedia.org/wiki/Chord_%28peer-to-peer%29.

[2] The gnu c library (glibc). http://www.gnu.org/software/libc/.

[3] Grid'5000, efficient and scalable operating system provisioning. https://www.grid5000.fr/.

[4] Simgrid. http://simgrid.gforge.inria.fr/.

[5] T. F. Chan, G. H. Golub, and R. J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

[6] R. F. Ling. Comparison of several algorithms for computing sample means and variance. *Journal of the American Statistical Association*, 69(348):859–866, 1974.

[7] M. Quinson, C. Rosa, and C. Thiery. Parallel simulation of peer-to-peer systems. In *CCGrid 2012 - The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 668–675, Ottawa, Canada, May 2012. IEEE.

APPENDIX

A. DATA PROVENANCE

This section explains and show how to run the experiments and how the data is saved and then processed. Note: 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 'parapluie' node at 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 Modifiable Parameters

Some of the parameters to run the experiments can be modified, like the amount of nodes to simulate and the amount of threads to use. Note that the list of nodes to simulate have to be changed in both the python session and the shell session. This sessions are intended to last during all your experiments/analysis.

This sizes/threads lists are needed to run the simulations, generate platform/deployment files, and generate tables after the experiments. Hence, is mandatory to run this snippets.

Bash:

```
BASE_DIR=$PWD
sizes=(1000 5000 10000 15000 20000 25000 30000 35000 40000 45000 50000 55000 60000 65000 70000 75000 80000 85000 90000

threads=(1 2 4 8 16 24)
```

Python:

```
SIZES = [1000]
SIZES += [elem for elem in range(5000,100000,5000)]
SIZES += [100000,300000,500000,1000000]
THREADS = [1, 2, 4, 8, 16, 24]
# All the benchmarks can be done using both modes, but note that this
# paper uses only precise
MODES = ['precise']
nb_bits = 32
end_date = 10000
```

A.2 Setting up the machine

Install required packages to compile/run SimGrid experiments. If you are in a cluster (such as Grid5000) you can run this file remotely in a deployed node and still be able to setup your environment. Run this two code chunks one after other in order to create folders, install packages and create required deployment/platform files.

If the `setup_and_install` snippet was run before, or everything is already installed and set up, then check/modify the parameters of the shell session with the snippets `check_args` and `go_to_chord`

`setup_and_install`:

```
# Save current directory where the report is
BASE_DIR=$PWD
apt-get update && apt-get install cmake make gcc git libboost-dev libgct++ libpcre3-dev linux-tools gdb liblua5.1-0-dev
mkdir -p SimGrid deployment platforms logs fig
cd $BASE_DIR/SimGrid/
# Clone latest SimGrid version. You may have to configure proxy settings if you are in a G5K node in order to clone thi
git clone https://gforge.inria.fr/git/simgrid/simgrid.git .
SGPATH='/usr/local'
# Save the revision of SimGrid used for the experiment
SGHASH=$(git rev-parse --short HEAD)
cmake -Denable_compile_optimizations=ON -Denable_supernovae=OFF -Denable_compile_warnings=OFF -Denable_debug=OFF -Denab
make install
cd ../../
```

`generate_platform_files`:

```
# This function generates a specific platform file for the Chord example.
import random
def platform(nb_nodes, nb_bits, end_date):
  max_id = 2 ** nb_bits - 1
  all_ids = [42]
  res = ["<?xml version='1.0'?>\n"
  "<!DOCTYPE platform SYSTEM \"http://simgrid.gforge.inria.fr/simgrid.dtd\">\n"]
  res.append("<!-- nodes: %d, bits: %d, date: %d -->\n"%(nb_nodes, nb_bits, end_date))
  res.append("<platform version=\"3\">\n"
  "  <process host=\"c-0.me\" function=\"node\"><argument value=\"42\"/><argument value=\"%d\"/></process>\n" % end_dat
  for i in range(1, nb_nodes):
    ok = False
    while not ok:
      my_id = random.randint(0, max_id)
      ok = not my_id in all_ids
    known_id = all_ids[random.randint(0, len(all_ids) - 1)]
    start_date = i * 10
    res.append("  <process host=\"c-%d.me\" function=\"node\"><argument value=\"%d\" /><argument value=\"%d\" /><argume
    all_ids.append(my_id)
  res.append("</platform>")
  res = "".join(res)
  f  = open(os.getcwd() + "/platforms/chord%d.xml"%nb_nodes, "w")
  f.write(res)
  f.close()
  return

# This function generates a specific deployment file for the Chord example.
# It assumes that the platform will be a cluster.
def deploy(nb_nodes):
  res = """<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
<AS  id="AS0"  routing="Full">
  <cluster id="my_cluster_1" prefix="c-" suffix=".me"
radical="0-%d"  power="1000000000"    bw="125000000"     lat="5E-5"/>
</AS>
</platform>"""%(nb_nodes-1)
  f = open(os.getcwd() + "/deployment/One_cluster_nobb_%d_hosts.xml"%nb_nodes, "w")
  f.write(res)
  f.close()
  return

# Remember that SIZES was defined as a global variable in the first python code chunk in [[Modifiable Parameters]]
for size in SIZES:
  platform(size, nb_bits, end_date)
  deploy(size)
```

Optional snippets to check arguments and go to chord folder:

check_args:

```
echo $sizes
echo $threads
echo $BASE_DIR
#sizes=(1000)
#threads=(1 2)
#BASE_DIR=$PWD
echo $sizes
echo $threads
echo $BASE_DIR
```

go_to_chord:

```
cd $BASE_DIR/SimGrid/examples/msg/chord
echo $BASE_DIR
echo $sizes
echo $threads
make
```

A.3 Scripts to run benchmarks

This are general scripts that can be used to run all the benchmarks after the proper modifications were done.

testall:

```
# This script is to benchmark the Chord simulation that can be found
# in examples/msg/chord folder.
# The benchmark can be done with both Constant and Precise mode, using
# different sizes and number of threads (which can be modified).
# This script also generate a table with all the times gathered, that can ease
# the plotting, compatible with gnuplot/R.
# By now, this script copy all data (logs generated an final table) to a
# personal frontend-node in Grid5000. This should be modified in the near
# future.

################################################################################
# MODIFIABLE PARAMETERS: SGPATH, SGHASH, sizes, threads, log_folder, file_table
# host_info, timefmt, cp_cmd, dest.

# Path to installation folder needed to recompile chord
# If it is not set, assume that the path is ’/usr/local’
if [ -z "$SG_PATH" ]
then
    SGPATH=’/usr/local’
fi

# Save the revision of SimGrid used for the experiment
SGHASH=$(git rev-parse --short HEAD)

# List of sizes to test. Modify this to add different sizes.
if [ -z "$sizes" ]
then
    sizes=(1000 3000)
fi

# Number of threads to test.
if [ -z "$threads"]
then
    threads=(1 2 4 8 16 24)
fi

# Path where to store logs, and filenames of times table, host info
if [ -z "$log_folder"]
then
    log_folder=$BASE_DIR"/logs"
else
    log_folder=$BASE_DIR"/logs/"$log_folder
fi

if [ ! -d "$log_folder" ]
then
    echo "Creating $log_folder to store logs."
    mkdir -p $log_folder
fi

# Copy all the generated deployment/platform files into chord folder
cp $BASE_DIR/platforms/* .
cp $BASE_DIR/deployment/* .

file_table="timings_$SGHASH.csv"
host_info="host_info.org"
rm -rf $host_info

# The las %U is just to ease the parsing for table
timefmt="clock:%e user:%U sys:%S telapsed:%e swapped:%W exitval:%x max:%Mk avg:%Kk %U"

# Copy command. This way one can use cp, scp and a local folder or a folder in
# a cluster.
```

```
sep=’,’
cp_cmd=’cp’
dest=$log_folder"/." # change for <user>@<node>.grid5000.fr:˜/$log_folder if necessary
###############################################################################

###############################################################################
echo "Recompile the binary against $SGPATH"
export LD_LIBRARY_PATH="$SGPATH/lib"
rm -rf chord
gcc chord.c -L$SGPATH/lib -I$SGPATH/include -I$SGPATH/src/include -lsimgrid -o chord

if [ ! -e "chord" ]; then
    echo "chord does not exist"
    exit;
fi
###############################################################################

###############################################################################
# PRINT HOST INFORMATION IN DIFFERENT FILE
set +e
echo "#+TITLE: Chord experiment on $(eval hostname)" >> $host_info
echo "#+DATE: $(eval date)" >> $host_info
echo "#+AUTHOR: $(eval whoami)" >> $host_info
echo " " >> $host_info

echo "* People logged when experiment started:" >> $host_info
who >> $host_info
echo "* Hostname" >> $host_info
hostname >> $host_info
echo "* System information" >> $host_info
uname -a >> $host_info
echo "* CPU info" >> $host_info
cat /proc/cpuinfo >> $host_info
echo "* CPU governor" >> $host_info
if [ -f /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor ];
then
    cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor >> $host_info
else
    echo "Unknown (information not available)" >> $host_info
fi
echo "* CPU frequency" >> $host_info
if [ -f /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq ];
then
    cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq >> $host_info
else
    echo "Unknown (information not available)" >> $host_info
fi
echo "* Meminfo" >> $host_info
cat /proc/meminfo >> $host_info
echo "* Memory hierarchy" >> $host_info
lstopo --of console >> $host_info
echo "* Environment Variables" >> $host_info
printenv >> $host_info
echo "* Tools" >> $host_info
echo "** Linux and gcc versions" >> $host_info
cat /proc/version >> $host_info
echo "** Gcc info" >> $host_info
gcc -v 2>> $host_info
echo "** Make tool" >> $host_info
make -v >> $host_info
echo "** CMake" >> $host_info
cmake --version >> $host_info
echo "* SimGrid Version" >> $host_info
```

```
grep "SIMGRID_VERSION_STRING" ../../../include/simgrid_config.h | sed 's/.*"\(.*\)"[^"]*$/\1/' >> $host_info
echo "* SimGrid commit hash" >> $host_info
git rev-parse --short HEAD >> $host_info
$($cp_cmd $host_info $dest)
################################################################################

################################################################################
# ECHO TABLE HEADERS INTO FILE_TABLE
rm -rf $file_table
tabs_needed=""
for thread in "${threads[@]}"; do
thread_line=$thread_line"\t"$thread
done
thread_line=$thread_line$thread_line
for size in $(seq 1 $((${#threads[@]}-1))); do
tabs_needed=$tabs_needed"\t"
done
echo "#SimGrid commit $SGHASH"     >> $file_table
echo -e "#\t\tconstant${tabs_needed}precise"     >> $file_table
echo -e "#size/thread$thread_line" >> $file_table
################################################################################

################################################################################
# START SIMULATION

test -e tmp || mkdir tmp
me=tmp/`hostname -s`

for size in "${sizes[@]}"; do
    line_table=$size
    # CONSTANT MODE
    for thread in "${threads[@]}"; do
filename="chord_${size}_threads${thread}_constant.log"
rm -rf $filename

if [ ! -f  chord$size.xml ]; then
./generate.py -p -n $size -b 32 -e 10000
fi

if [ ! -f  One_cluster_nobb_${size}_hosts.xml ]; then
./generate.py -d -n $size
fi


echo "$size nodes, constant model, $thread threads"
cmd="./chord One_cluster_nobb_"$size"_hosts.xml chord$size.xml --cfg=contexts/stack_size:16 --cfg=network/model:Constan

/usr/bin/time -f "$timefmt" -o $me.timings $cmd $cmd 1>/tmp/stdout-xp 2>/tmp/stderr-xp

if grep "Command terminated by signal" $me.timings ; then
    echo "Error detected:"
    temp_time="errSig"
elif grep "Command exited with non-zero status" $me.timings ; then
    echo "Error detected:"
    temp_time="errNonZero"
else
    temp_time=$(cat $me.timings | awk '{print $(NF)}')
fi

# param
cat $host_info >> $filename
echo "* Experiment settings" >> $filename
echo "size:$size, constant network, $thread threads" >> $filename
echo "cmd:$cmd" >> $filename
#stderr
echo "* Stderr output" >> $filename
cat /tmp/stderr-xp >> $filename
# time
echo "* Timings" >> $filename
cat $me.timings >> $filename
line_table=$line_table$sep$temp_time
$($cp_cmd $filename $dest)
rm -rf $filename
rm -rf $me.timings
    done

    #PRECISE MODE
    for thread in "${threads[@]}"; do
echo "$size nodes, precise model, $thread threads"
filename="chord_${size}_threads${thread}_precise.log"

cmd="./chord One_cluster_nobb_"$size"_hosts.xml chord$size.xml --cfg=contexts/stack_size:16 --cfg=maxmin/precision:0.00

/usr/bin/time -f "$timefmt" -o $me.timings $cmd $cmd 1>/tmp/stdout-xp 2>/tmp/stderr-xp

if grep "Command terminated by signal" $me.timings ; then
    echo "Error detected:"
    temp_time="errSig"
elif grep "Command exited with non-zero status" $me.timings ; then
    echo "Error detected:"
    temp_time="errNonZero"
else
    temp_time=$(cat $me.timings | awk '{print $(NF)}')
fi
# param
cat $host_info >> $filename
echo "* Experiment settings" >> $filename
echo "size:$size, constant network, $thread threads" >> $filename
echo "cmd:$cmd" >> $filename
#stderr
echo "* Stderr output" >> $filename
cat /tmp/stderr-xp >> $filename
# time
echo "* Timings" >> $filename
cat $me.timings >> $filename
line_table=$line_table$sep$temp_time
$($cp_cmd $filename $dest)
rm -rf $filename
rm -rf $me.timings
    done

    echo -e $line_table >> $file_table

done

$($cp_cmd $file_table $dest)
rm -rf $file_table
rm -rf tmp
```

testall_sr:

```
# This script is to benchmark the Chord simulation that can be found
# in examples/msg/chord folder.
# The benchmark is done with both Constant and Precise mode, using
# different sizes and number of threads (which can be modified).
# This script also generate a table with all the times gathered, that can ease
# the plotting, compatible with gnuplot/R.
# By now, this script copy all data (logs generated an final table) to a
# personal frontend-node in Grid5000. This should be modified in the near
# future.

###############################################################################
# MODIFIABLE PARAMETERS: SGPATH, SGHASH, sizes, threads, log_folder, file_table
# host_info, timefmt, cp_cmd, dest.

# Path to installation folder needed to recompile chord
# If it is not set, assume that the path is ’/usr/local’
if [ -z "$SG_PATH" ]
then
    SGPATH=’/usr/local’
fi

# Save the revision of SimGrid used for the experiment
SGHASH=$(git rev-parse --short HEAD)

# List of sizes to test. Modify this to add different sizes.
if [ -z "$sizes" ]
then
    sizes=(1000 3000)
fi

# Number of threads to test.
if [ -z "$threads"]
then
    threads=(1 2 4 8 16 24)
fi

# Path where to store logs, and filenames of times table, host info
if [ -z "$log_folder"]
then
    log_folder=$BASE_DIR"/logs"
else
    log_folder=$BASE_DIR"/logs/"$log_folder
fi

if [ ! -d "$log_folder" ]
then
    echo "Creating $log_folder to store logs."
    mkdir -p $log_folder
fi

# Copy all the generated deployment/platform files into chord folder
cp $BASE_DIR/platforms/* .
cp $BASE_DIR/deployment/* .

file_table="timings_$SGHASH.csv"
host_info="host_info.org"
rm -rf $host_info

# The las %U is just to ease the parsing for table
timefmt="clock:%e user:%U sys:%S telapsed:%e swapped:%W exitval:%x max:%Mk avg:%Kk %U"

# Copy command. This way one can use cp, scp and a local folder or a folder in
# a cluster.
sep=’,’
cp_cmd=’cp’
dest=$log_folder # change for <user>@<node>.grid5000.fr:˜/$log_folder if necessary
###############################################################################

###############################################################################
```

```
echo "Recompile the binary against $SGPATH"
export LD_LIBRARY_PATH="$SGPATH/lib"
rm -rf chord
gcc chord.c -L$SGPATH/lib -I$SGPATH/include -I$SGPATH/src/include -lsimgrid -o chord

if [ ! -e "chord" ]; then
    echo "chord does not exist"
    exit;
fi
###############################################################################

###############################################################################
# PRINT HOST INFORMATION IN DIFFERENT FILE
set +e
echo "#+TITLE: Chord experiment on $(eval hostname)" >> $host_info
echo "#+DATE: $(eval date)" >> $host_info
echo "#+AUTHOR: $(eval whoami)" >> $host_info
echo " " >> $host_info

echo "* People logged when experiment started:" >> $host_info
who >> $host_info
echo "* Hostname" >> $host_info
hostname >> $host_info
echo "* System information" >> $host_info
uname -a >> $host_info
echo "* CPU info" >> $host_info
cat /proc/cpuinfo >> $host_info
echo "* CPU governor" >> $host_info
if [ -f /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor ];
then
    cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor >> $host_info
else
    echo "Unknown (information not available)" >> $host_info
fi
echo "* CPU frequency" >> $host_info
if [ -f /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq ];
then
    cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq >> $host_info
else
    echo "Unknown (information not available)" >> $host_info
fi
echo "* Meminfo" >> $host_info
cat /proc/meminfo >> $host_info
echo "* Memory hierarchy" >> $host_info
lstopo --of console >> $host_info
echo "* Environment Variables" >> $host_info
printenv >> $host_info
echo "* Tools" >> $host_info
echo "** Linux and gcc versions" >> $host_info
cat /proc/version >> $host_info
echo "** Gcc info" >> $host_info
gcc -v 2>> $host_info
echo "** Make tool" >> $host_info
make -v >> $host_info
echo "** CMake" >> $host_info
cmake --version >> $host_info
echo "* SimGrid Version" >> $host_info
grep "SIMGRID_VERSION_STRING" ../../../include/simgrid_config.h | sed ’s/.*"\(.*\)"[^"]*$/\1/’ >> $host_info
echo "* SimGrid commit hash" >> $host_info
git rev-parse --short HEAD >> $host_info
$($cp_cmd $host_info $dest)
###############################################################################
```

```
################################################################################
# ECHO TABLE HEADERS INTO FILE_TABLE
rm -rf $file_table
tabs_needed=""
for thread in "${threads[@]}"; do
thread_line=$thread_line"\t"$thread
done
thread_line=$thread_line$thread_line
for size in $(seq 1 $((${#threads[@]}-1))); do
tabs_needed=$tabs_needed"\t"
done
echo "#SimGrid commit $SGHASH"     >> $file_table
echo -e "#\t\tconstant${tabs_needed}precise"     >> $file_table
echo -e "#size/thread$thread_line" >> $file_table
################################################################################

################################################################################
# START SIMULATION

test -e tmp || mkdir tmp
me=tmp/‘hostname -s‘

for size in "${sizes[@]}"; do
    line_table=$size
    # CONSTANT MODE
    for thread in "${threads[@]}"; do
filename="chord_${size}_threads${thread}_constant.log"
output="sr_${size}_threads${thread}_constant.log"
rm -rf $filename

if [ ! -f  chord$size.xml ]; then
./generate.py -p -n $size -b 32 -e 10000
fi

if [ ! -f  One_cluster_nobb_${size}_hosts.xml ]; then
./generate.py -d -n $size
fi


echo "$size nodes, constant model, $thread threads"
cmd="./chord One_cluster_nobb_"$size"_hosts.xml chord$size.xml --cfg=contexts/stack_size:16 --cfg=network/model:Constan

/usr/bin/time -f "$timefmt" -o $me.timings $cmd $cmd 1>/tmp/stdout-xp 2>/tmp/stderr-xp

if grep "Command terminated by signal" $me.timings ; then
    echo "Error detected:"
    temp_time="errSig"
elif grep "Command exited with non-zero status" $me.timings ; then
    echo "Error detected:"
    temp_time="errNonZero"
else
    temp_time=$(cat $me.timings | awk '{print $(NF)}')
fi

# param
cat $host_info >> $filename
echo "* Experiment settings" >> $filename
echo "size:$size, constant network, $thread threads" >> $filename
echo "cmd:$cmd" >> $filename
#stdout
echo "* Stdout output" >> $filename
cat /tmp/stdout-xp | grep Amdahl >> $filename
#stderr
```

```
echo "* Stderr output" >> $filename
cat /tmp/stderr-xp >> $filename
# time
echo "* Timings" >> $filename
cat $me.timings >> $filename
line_table=$line_table$sep$temp_time
# Gather SR data from logs
echo -e '#id_sr\ttime_taken\tamount_proccesses' >> $output
grep 'Total time SR' $filename | awk '{print $7 "\x09" $9 "\x09" $10}' | tr -d ',' >> $output
$($cp_cmd $output $dest)
$($cp_cmd $filename $dest)
rm -rf $filename $output
rm -rf $me.timings
    done

    #PRECISE MODE
    for thread in "${threads[@]}"; do
echo "$size nodes, precise model, $thread threads"
filename="chord_${size}_threads${thread}_precise.log"
output="sr_${size}_threads${thread}_precise.log"

cmd="./chord One_cluster_nobb_"$size"_hosts.xml chord$size.xml --cfg=contexts/stack_size:16 --cfg=maxmin/precision:0.00

/usr/bin/time -f "$timefmt" -o $me.timings $cmd $cmd 1>/tmp/stdout-xp 2>/tmp/stderr-xp

if grep "Command terminated by signal" $me.timings ; then
    echo "Error detected:"
    temp_time="errSig"
elif grep "Command exited with non-zero status" $me.timings ; then
    echo "Error detected:"
    temp_time="errNonZero"
else
    temp_time=$(cat $me.timings | awk '{print $(NF)}')
fi
# param
cat $host_info >> $filename
echo "* Experiment settings" >> $filename
echo "size:$size, constant network, $thread threads" >> $filename
echo "cmd:$cmd" >> $filename
#stderr
echo "* Stderr output" >> $filename
cat /tmp/stderr-xp >> $filename
# time
echo "* Timings" >> $filename
cat $me.timings >> $filename
line_table=$line_table$sep$temp_time
# Gather SR data from logs
echo -e '#id_sr\ttime_taken\tamount_proccesses' >> $output
grep 'Total time SR' $filename | awk '{print $7 "\x09" $9 "\x09" $10}' | tr -d ',' >> $output
$($cp_cmd $output $dest)
$($cp_cmd $filename $dest)
rm -rf $filename $output
rm -rf $me.timings
    done
    echo -e $line_table >> $file_table
done

$($cp_cmd $file_table $dest)
rm -rf $file_table
rm -rf tmp
```

A.4 Baseline Performance

The benchmark can be run from this org-mode file, or simply by running `./scripts/chord/testall.sh` inside the folder `examples/msg/chord` of your SimGrid installation. Inside that script, the number of threads to test, as well as the amount of nodes, can be modified

The script generates a .csv table, but just in case it is done in different stages, the resulting logs can be processed with `./scripts/chord/get_times.py` (located in the same folder as testall.sh). This generates a .csv that can easily be plotted with R/gnuplot.

The script is self-documented.

A.5 SR Distribution

To enable Scheduling Rounds benchmarks, the constant `TIME_BENCH_ENTIRE_SRS` has to be defined. It can be defined in `src/simix/smx_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/testall_sr.sh`. It gathers data about the id of each SR, time of each SR and num processes of SR, in stores it in table format.

A.6 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.7 Binding threads to physical cores

The constant `CORE_BINDING` has to be defined in `include/xbt/xbt_os_thread.h` in order to enable this optimization. The benchmark is then run in the same way as the Amdahl Speedup experiment.

A.8 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 and the standard benchmark can be used.
2. maestro sleeps with N-1 threads To avoid that maestro works with the threads, comment out the line: `xbt_parmap_work(parmap);` from the function `xbt_parmap_apply()` in `src/xbt/parmap.c`
3. maestro sleeps with N threads To avoid that maestro works with the threads, comment out the line: `xbt_parmap_work(parmap);` from the function `xbt_parmap_apply()` in `src/xbt/parmap.c` Then the function `src/xbt/parmap.c:xbt_parmap_new` has to be modified to create one extra thread. It is easy: just add 1 to `num_workers` parameter.

A.9 Busy Waiters vs. Futexes performance

Enable the use of busy waiters running chord with the extra option: `-cfg=contexts/synchro:busy_wait` The experiment was run with testall.sh using that extra option in the chord command inside the script. The tables were constructed using `get_times.py`. The data regarding the futexes times is the same gathered in Baseline Performance experiment.

A.10 SR parallel threshold

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

A.11 Adaptive threshold

The benchmark is done using testall.sh. The algorithm is the one described in section 5.2, and it can be enabled by defining the constant `ADAPTIVE_ALGORITHM` in `src/simix/smx_private.h`