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 (*parmap*).

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, 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 ~[?], 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

To get a baseline timings and a speedup plot starting from the current version of SimGrid (3.11), benchmarks to measure the execution time of a typical Chord simulation in Precise mode with different amount of threads (1, 2, 4, 8, 16 and 24) were done.

The total times of a normal execution for the Chord simulation in the precise mode 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.
1k	0:00:04	0.03	0:00:07	0.03	0:00:09	0.03
5k	0:00:28	0.13	0:00:40	0.13	0:00:49	0.13
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).

The actual speedup obtained can be seen in the Figure 2. It is clear from that graph that the real speedup with our

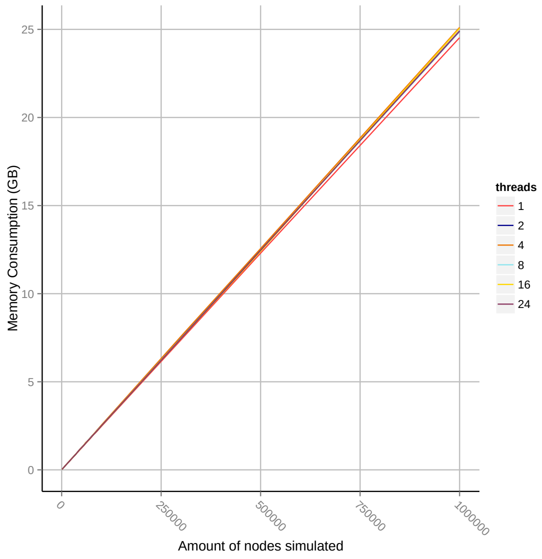

Figure 1: Memory consumptions reported in GB

parallel model is obtained when the size of the problem is bigger than 300000 nodes. This is a proof that what was proved in our previous work ~[?] is still valid.

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.

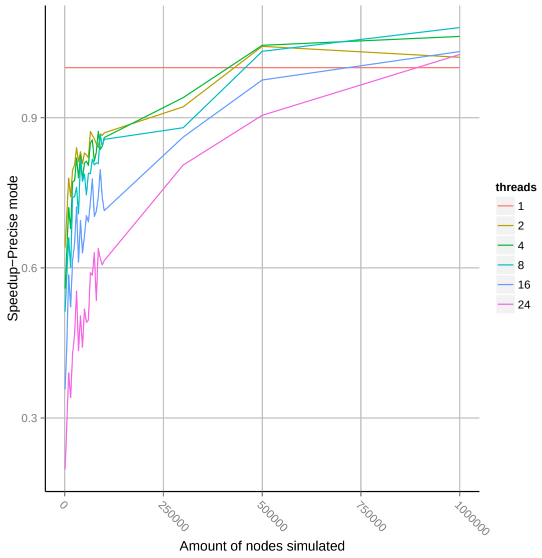

Figure 2: Baseline performance of SimGrid 3.11. Speedups achieved using multithreaded executions against the sequential ones.

3.2 Parallelizable portions of the problem

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.

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 s 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).

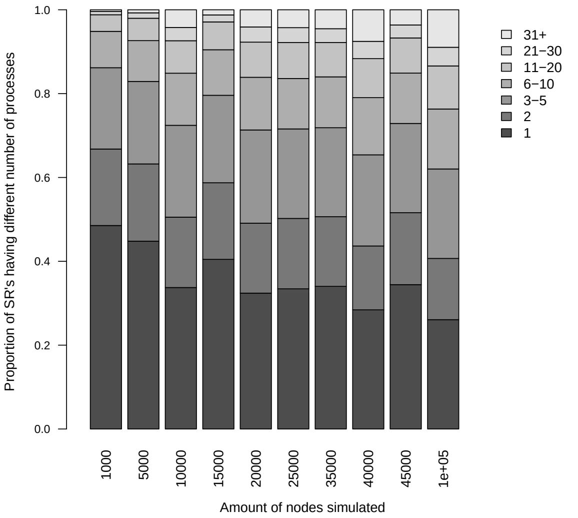

Figure 3: Proportions of SR's having different numbers of processes to compute; according to the size of nodes simulated.

However, launching a small amount of processes is inefficient due to the costs of synchronization 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. Also, in that section we will try to find what is the optimal threshold between serial and parallel executions of SR's.

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 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. 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.

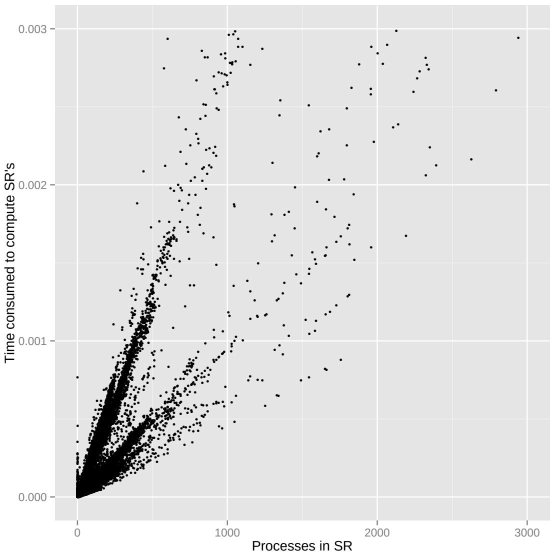

Figure 4: Times of SR's sequential executions depending on the amount of processes of each SR.

Regardless of the situation, migration of threads between cores entails an increase of CPU migrations, which 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 relative speedup 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 Chord Protocol with Precise mode, but 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

Algorithm 1 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
$process\_seq, process\_par \leftarrow 0$

procedure RunSchedulingRound
    if computed five par/seq SR's then
        $parallel\_SRs \leftarrow 1$
        $sequential\_SRs \leftarrow 1$
        $ratio\_seq \leftarrow seq\_time/process\_seq$
        $ratio\_par \leftarrow par\_time/process\_par$
        $sequential\_is\_slower \leftarrow ratio\_seq > ratio\_par$
        if $sequential\_is\_slower$ then
            decrease($parallel\_threshold$)
        else
            increase($parallel\_threshold$)
        $seq\_time, par\_time \leftarrow 0$
        $process\_seq, process\_par \leftarrow 0$

    if $processes\_to\_run >= parallel\_threshold$ then
        execute_SR_parallel()
        $process\_par \leftarrow get\_number\_of\_processes()$
        $parallel\_SRs++$
    else
        execute_SR_serial()
        $process\_seq \leftarrow get\_number\_of\_processes()$
        $sequential\_SRs++$
```

execution time is decreased and approximates its sequential counterpart.

Regarding the memory consumption, the values remain the same in general, as it can be seen in Table /ref{tab:two}

TODO: instead of calculate the last ratio, better calculate an average value of ratios during the simulation. That would solve the problems of noise at the end, right?

TODO: this table is not useful at all as it is (is just a test). We need the logs of the experiments w/bigger sizes to show the real speedup.

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

nodes	2 thr.	Mem	4 thr.	Mem	8 thr.	Mem
1000	7	0.03	8	0.03	9	0.03
5000	40	0.13	45	0.13	49	0.13
10000	80	0.26	87	0.26	95	0.25
50000	459	1.27	470	1.27	483	1.25
100000	936	2.53	947	2.54	950	2.50

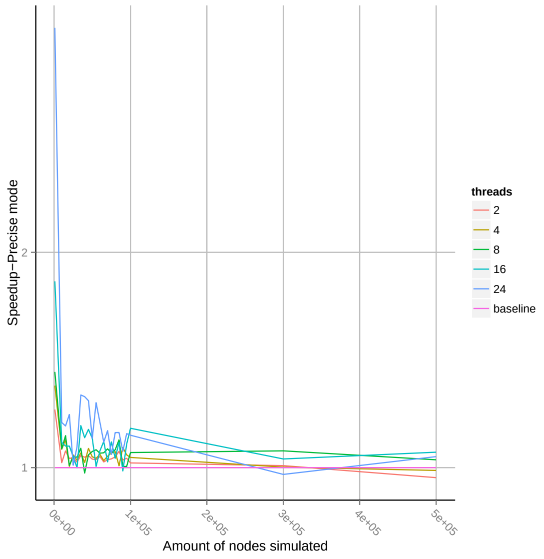

Figure 5: Speedup achieved with Adaptive Algorithm. Chord simulation, Precise mode.

6. CONCLUSION

We have showed in this work several ways to optimize large scale distributed simulations, namely, binding threads to physical cores, choosing a better threshold for parallel execution or choosing between futexes or busy waiters. The optimizations were done over the open-source multi-purpose SimGrid simulation framework, in its development version (3.12). Some of 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 outperforms 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.

It is important to say, the majority of the speedups achieved were done with big size simulations (up to 500000 nodes), which shows the difficulty of parallelize user code efficiently with small sizes.

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 within threads and when to do it in a serial way proved to help, if the initial parallel threshold is good enough. Currently, our adaptive algorithm proposed is sensitive to that initial threshold, and the final value will strongly depend on the one set up at the beginning.

Due to the fact that we found that executing Scheduling Rounds in parallel is efficient only when the amount of processes surpass 300000, is understandable that with small simulations the adaptive algorithm will not find the very optimal threshold, but it will stand a chance of making the simulation a bit faster.

As future work, the adaptive algorithm should be improved to determine a better and unique threshold over different simulations regardless of the initial one at the beginning.

In a final note, the present work was done with the reproducible research approach in mind. Thus, the steps and scripts needed to run the experiment (as well as the data of the benchmarks done) can be found at `link to somewhere` , as well 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).

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 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 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. Thus, is mandatory to run this snippets.

```
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)
```

```
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`

```
# 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 ../../
```

```
# 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:

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

```
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.

```
# 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
```

```
# 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 Amdahl speedup

The benchmark can be run from this org-mode file, or simply by running ./scripts/chord/testall.sh (in this repository) 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 constant $\text{TIME}_{\text{BENCHAMDAHL}}$ 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/$\text{smx}_{\text{private}}$.h

The script generates a .csv table, but just in case it is done in different stages, the gathered logs can be processed with $\text{get}_{\text{times}}$.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$_{\text{BENCHPERSR}}$ has to be defined. It can be defined in 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/testall\$_{\text{sr}}$.sh, the id of SR, time of SR and num processes of SR, in a table format.

This can be run from here or just by running testall\$_{\text{sr}}$.sh in in the examples/msg/chord folder of your SimGrid install.

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$_{\text{BINDING}}$ 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.

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\$_{\text{parmap}\backslash\text{work}}$(parmap); from the function xbt\$_{\text{parmap}\backslash\text{apply}}$() in src/xbt/parmap.c
3. maestro sleeps with N threads To avoid that maestro works with the threads, comment out the line: xbt$_{\text{parmapwork}}$(parmap); from the function xbt$_{\text{parmapapply}}$() in src/xbt/parmap.c Then the function src/xbt/parmap.c:xbt$_{\text{parmapnew}}$ has to be modified to create one extra thread. It is easy: just add 1 to num$_{\text{workers}}$ parameter.

A.9 Busy Waiters vs. Futexes performance

Enable the use of busy waiters running chord with the extra option: –cfg=contexts/synchro:busy\$_{\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 get$_{\text{times}}$.py The data regarding the futexes synchro times is the same gathered in Amdahl Speedup experiment.

A.10 Performance Regression Testing

A.11 SR parallel threshold

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

A.12 Adaptive Algorithm

The benchmark is done using testall.sh. The algorithm is the one described in section 5.2