Research Reports on Computer Science
https://ojs.wiserpub.com/index.php/RRCS/

UNIVERSAL WISER PUBLISHER

Research Article

Evaluating Simultaneous Multi-threading and Affinity Performance for Reproducible Parallel Stochastic Simulation

Benjamin Antunes*, David Hill

Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, Clermont-Auvergne-INP, LIMOS UMR 6158 - ISIMA - F-63000 Clermont-Ferrand, France
E-mail: benjamin.antunes@uca.fr

Received: 31 May 2023; **Revised:** 25 July 2023; **Accepted:** 26 September 2023

Abstract: This paper investigates whether simultaneous multi-threading (SMT) can improve performance on modern computing clusters with reproducible results on four types of applications, focused on stochastic simulations with different memory bound and compute bound constraints. We manually set the affinity of processes to compare its efficiency with the computing time obtained by the automatic assignment of the operating system. To measure SMT and affinity impact on a modern multicore processor, we parallelize up to 128 processes of the four types of applications. We expect repeatable numerical results between the sequential and parallel versions of simulations. For the three applications that are not memory bound, SMT is more effective by up to 30%. This represents an interesting increase up to 10% more performance for compute bound applications when compared to the initial papers discussing the efficiency of SMT. However, for the memory-bound application, SMT is less effective and can even decrease performance. The manual setting of core affinity does not show an increase in performance compared to the automatic assignment. All code and data used in the study are available to help reproducible research.

Keywords: simultaneous multi-threading, multicore, hyper-threading, performance

1. Introduction

Reproducible research is one of the cornerstones of science. However, as we can see in many studies, even computer science papers are hardly reproducible [1, 2]. In this paper, we study in a reproducible way the impact of simultaneous multi-threading (SMT) and affinity on performance for parallel stochastic simulations. Papers about SMT are quite old now and our goal is to actualize our knowledge in the era of advanced multicores central processing units (CPUs).

More than 80% of the Worldwide Computing Grid CPU cycles are running stochastic simulations with SMT [3]. Multicore processors have become a ubiquitous technology. In fact, in 2002, Intel released a paper [4] that introduced SMT, also known as hyper-threading (HT) for Intel's patented implementation or SMT for other manufacturers. During the two decades after the introduction of this patent, multicore (or also called more recently manycore) chips emerged to lower the die size and power consumption of massive parallelism. On a single CPU, we can now find 128 physical cores (with recent ARM and AMD chips). In addition, if we consider SMT, we can multiply this number by 2 (or sometimes more depending on the SMT architecture) to get the number of logical cores. With this level of density, we could find a single high performance computing (HPC) node with two sockets providing up to 512 logical cores at the end of 2022.

DOI: https://doi.org/10.37256/rrcs.2220233134

The distinction between multicore processors and manycore processors is somewhat arbitrary and there is no strict definition that applies to all cases. We made the choice here to talk about multicore, but we can find in literature, 32 cores CPU considered as manycore processors.

First, let us explain how 2-way SMT works [4]. For the operating system, a single physical processor will appear as two logical processors. While the physical execution resources are shared, the full architecture state is duplicated for the two logical processors. The operating systems and the user programs can use the two logical cores as if they were real physical processors. The first level of parallelism is instruction-level parallelism (ILP); it "*refers to techniques to increase the number of instructions executed each clock cycle*" [4]. We can cite superscalar processors that execute multiple instructions at each clock cycle. However, this implies to use the out-of-order technique to be efficient. The out-of-order technique (dynamic execution) is now implemented in most modern CPUs. A combination of Speculative Execution, Multiple Branch Prediction and Data-flow Analysis can lead to a better usage of core resources (less idle time). In an out-of-order machine, proper exception handling and branch prediction miss imply a repairing mechanism to restore the machine to a previous reliable state [5]. The out-of-order feature can also lead to a loss of repeatability when dealing with floating point operations since they are not associative [6, 7]. Indeed, with out-of-order cores, instructions are not always executed following the order specified in the source program or even by the compiler. The IBM System 360/Model 91 was the first machine to propose this feature in 1967 but it did not propose floating point instructions, the introduction of this approach to Floating Point Units is quite recent. Another level of parallelism is the thread-level parallelism (TLP). In fact, in current systems, many processes are running at the same time (in foreground or background). So multi-threading is also handled for basic computing, not only for HPC.

Multicore chips are one way to facilitate the handling of TLP by providing more 'hardware'. To handle multiple threads, it is rare that you can propose a physical core for each task; you have to switch between them. Different methods exist: switch between threads at a fixed time (or not), when a thread is still runnable, and switch between threads when long latency events happen, like I/O or synchronization, or interruption handling. The operating system policy of switching can vary; this task is done by the scheduler. When a CPU core is scheduled to switch to another thread, we call this process a "Context Switch". Context Switches are expensive and slow down the computing process as the entire state and flow of the highly pipelined CPU must be stopped, saved and swapped out for another state (as well as other caches, registers, lookup tables, etc.). What can cost time is also cache misses. A cache miss appears when data is not stored in the local hierarchy of memory cache for fast access. The CPU will have to get the data in random-access memory (RAM), and this results in a significant time loss. The goal of SMT is to be able to "exploits both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle" [8], without switching between them. To do so, there is one architecture state for each logical core, as said previously. The processor can keep the state of the two threads simultaneously, and can easily swap between them without time loss. Logical cores share most of the physical CPU resource, but the duplication of the architecture enables the autonomy of both logical cores with only 5% more power and chip size (according to Intel's paper). The latter shows that the HT technology offers around 20% performance improvement on database exploitation and web services. However, the information is quite old now. We want to answer here the following questions: is SMT really that useful today, at the large multicore (or manycore) era? Is it suitable for HPC and more particularly for large stochastic simulations? They sometimes present huge memory needs and sometimes massive computation needs. Does the user management of CPU affinity increase performance? What kind of advices can we give to non-expert users of computing clusters? Can we help system administrators in their decisions about enabling or disabling SMT depending on the application profile? In our context of parallel stochastic simulations, we need reproducible and repeatable applications to address the previous questions. In this paper, we will first present related work about SMT and affinity performance impact and show that we experience a reproducibility problem. Then, we will present our work, how we planned it and then discuss our results. Finally, we will discuss some hypotheses to explain the obtained results before planning future work.

2. A few words about the "reproducibility" terminology evolution

While reproducibility can appear as a simple notion, the fact is that we do not have a consensus definition among researchers and research fields. Before 2020, according to the ACM, here was the main terms and definitions:

• Repeatability: Same team, same experimental setup

• Reproducibility: Different team, different experimental setup
• Replicability: Different team, same experimental setup

In these definitions, as Drummond said in [9], reproducibility requires changes, replicability avoids it. Reproducibility is the fact that a different team applying a different method or setup for the same scientific question obtains the same scientific conclusions. This is strengthening the discovery. In the case of replicability, the goal is that a different team, using the first team article artifacts, can obtain the same results with a stated precision. In literature, authors sometimes use the word "reproducibility" to talk about "replicability", and sometimes it was the other way around. Advised by NISO (National Information Standards Organization), ACM (Association for Computing Machinery) changed their definitions after 2020, swapping terms between reproducibility and replicability for a better match with the practice of other research fields. Barba found which scientific field used one definition or the other and obtained the results presented in [10]. With this study, we realize that computer science was one of the rare fields to use the definition given at the beginning of this section. That was an argument to follow the NISO standardization advices; ACM then switched their reproducibility and replicability definitions. Here are the 2020, and current, definition for both terms [11]:

"*Reproducibility* (*Different team, same experimental setup*): *The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artifacts*.

Replicability (*Different team, different experimental setup*): *The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently*."

We can also observe that some authors, active in the reproducibility research field in computer science, and that were using the previous definitions, like Hinsen [12] or Rougier et al. [13], have also swapped their definitions of reproducibility and replicability; it now matches with the ACM 2020 update. Our approach is the same, and in this paper, we will stick to the ACM definitions for "reproducibility" and "replicability". Increasing trust about a scientific conclusion with reproducibility and/or replicability is important for us. Reproducibility is the first step for everyone to regain confidence in published results.

The last term to discuss about is repeatability. It has led to less controversy than the two others. However, in computer science research papers, we sometimes find a confusion between repeatability and reproducibility. In our opinion, the definition of repeatability can still vary among scientific fields. In fact, in computer science, our machines are designed to be deterministic (except for quantum computing of course). We want to obtain run to run bitwise identical results on the same machine for the same program. This point is absolutely essential for debugging and for the trust in the use of our deterministic computers. Repeatability is a real concern for researchers who are still aware of debugging, and this activity is particularly tough in the HPC world. Ensuring reliable parallel debugging requires repeatability with bitwise identical results. And in this sense, we do not fully agree with the ACM definition which adds that results are identical with a "stated precision". This is because the ACM definition of repeatability comes from the International Vocabulary of Metrology. In our opinion, this definition is perfectly correct for quantum computer science, but not for classical deterministic computing where we really need "bitwise identical results" in order to debug properly. In the case of classical deterministic computer science, the "stated precision" should allow no differences. This applies to numerical results of applications. When talking about computing time, "stated precision" of the general definition is fairer. In this paper, we aim to obtain bitwise identical results for each of our experiments and on different platforms, but also, we want the same results when go from sequential execution to parallel execution (which is not so obvious if no proper method is used to manage the different stochastic streams). This last point is important to ensure that different execution mode and execution time are not due to differences in the execution path of the processes.

3. Related work

Several papers exist about SMT and affinity. If the SMT concept is described in [4], paper [14] studies the impact of SMT in specific case of message passing interface (MPI) applications. It gives us some conclusions: SMT technology

allows a better resource utilization of the processor, but, on the other hand, it can also lead to more overhead. Some studies focused on the first SMT (HT) processor, the Pentium 4. Paper [15] shows an amelioration of 20% as expected from Intel's paper [4] and [16] obtained similar results. A good study from 2011 tested the impact of SMT on processor utilization on four real-life NASA applications [17]. It shows that SMT led to a better utilization of processors, and it results in increased global performance when sufficient memory resources are available (cache and memory bandwidth). In the cases of affinity, there are also some papers existing about this subject; in [18], the authors work on transmission control protocol (TCP) connections on a symmetric multiprocessing (SMP) server. This study obtained a good performance impact using affinity, around 20%. This case is specific for communication applications with the TCP protocol. In [19], the study is close to our considerations. They tried to figure out the affinity performance on multicore chips. They measured it on multicore uniprocessors and multicore multiprocessors according to their terminology. The difference is about the physical architecture: in the first case, there is only one socket, and in the second case, there can be two or more sockets. This is why we have multicore uniprocessor (with only one processor), or multicore multiprocessors (with several). This leads to the fact that L2 and L3 cache might be fully shared between all cores and in the second case, several L2 and L3 caches can be shared between a subset of cores, so we can still use affinity at L2 or L3 cache level. Their results are that affinity gives no improvement on multicore uniprocessors (a single processor with multicore inside and a single L2 usually used on a single socket node). Affinity gives a gain of 11% in performance on average for multicore multiprocessors (nodes with 2 or more sockets, each of them providing their own L2 cache for their multicore). An improvement was observed with affinity before multicore existed (assigning tasks to physical processors on a multiprocessor system). This paper was published in 2008, multicore chips have evolved since then. While [18] and [19] were talking about results obtained with affinity on the same motherboard (handling affinity to work better with cache memory), we can also think about affinity at a larger scale for communicating processes. In [20], they are working at a larger scale on computing clusters. This requires knowing exactly the architecture of the computing cluster we are using, where exactly our programs are going to be executed and where the intense communications between processes are. Without a fast and balanced cluster interconnect, the more distance we have between two communicating points, the more time we will need. This paper [20] places the context of a computing cluster that may have the size of a room for example (a small supercomputer for instance). Knowing the exact physical architecture of the cluster can be useful to understand the behavior of some parallel programs, where are physically the racks and the nodes in the room, on which racks are node X and node Y placed for example. When scientists submit their jobs on computing clusters, they rarely have this information and they should not care about this. As said previously, fast and balanced interconnect solves most of the questions raised by this paper. This paper does not study the performance impact of affinity, but it is interesting to see affinity in a different way, how affinity can be used at a larger scale?

In [21], they studied the performance impact of SMT for parallel simulations. They have shown that the SMT impact on performance was influenced by "the characteristics of the runtime software architecture (i.e., mono-thread/multi-thread implementation)" and that SMT slightly increase performance. In [22], they used benchmarks to predict the performance of memory bounded applications on clusters. In [23], they studied the impact of several technologies, including HT, on high energy physics (HEP) simulation on computing grid for CERN (European Council for Nuclear Research). They worked on GEANT4 detector simulations. Their results were that HT caused unnecessary overheads and should not be used because of a loss of performance. And lastly for [24], they study the effect of SMT on MPI-based applications while testing different math libraries. They conclude that SMT can improve or worsen performance depending on hardware configuration and on the applications that we are running. For a math library, that is using high CPU resources, then SMT can decrease performance, because enabling SMT will share cache and lead to more cache misses. This is why we think that studying SMT effectiveness is still pertinent, because we should not assume that it is in all cases beneficial.

Most related works that we presented come from papers published between 2000 and 2010. As technologies are evolving fast, more up-to-date studies are also important, without minimizing the importance of older papers. In the recent published papers about SMT, we cannot find articles that consider the evaluation of SMT performance compared to machines with SMT disabled. Depending on the type of applications running on clusters, the benefice of SMT is not always there. Osborne and colleagues published several papers about SMT for real-time application from 2019 onwards. We can cite [25-27], where they assume that SMT provides an overall performance benefit. Starting from this knowledge, they aim to obtain a better usage of SMT resources by optimizing the scheduling, in the special case of

real-time applications. These more recent papers are not relevant to our study, as we primarily question the benefits of enabling or disabling SMT on every computing cluster, where we should consider the running applications types, or the other hardware configuration (like memory or interconnect), to know if SMT is effective. With the same considerations of previous recent works, [28] also aim to optimize the usage of the resources offered by SMT, they are proposing a solution affecting the compilation phase. Furthermore in 2020 from [29], wanted to optimize fetching policies with machine learning. All these recent papers are assuming the SMT value for enhancing performance. A small recent paper is more connected to our research questions. In [30], the authors study whether users should enable or disable SMT on cloud service provider. They conclude "*Sometimes, disabling SMT is a good option to boost the performance of certain workloads. Therefore, we should always test our workloads with different SMT settings and deploy with the setting that brings better performance to the workloads*."; this is more similar to what [23] obtained, and to what we are studying. With our paper, we want to question the fact that SMT is always beneficial to performance. Indeed, as we see, papers evaluating SMT performance are quite old now, and are also not reproducible (because back in the time, code sharing and good practices was not considered as important as it is now). In addition, multicore chips have evolved since, and we can even call them "manycore" chips. This paper aims to actualize our knowledge, by practical experiment, as many non-expert users could use an actual computing cluster.

In the new set of ACM definitions (2020) dealing with reproducibility [11], as we said, reproducibility is an important notion which helps to trust the published results. To be reproducible, a paper has to give its artifacts. In our case with SMT and affinity, papers should describe the method used, and should give a link to the scripts and codes with the essential documentation in order to be runnable in a reasonable time. We studied the ability to reproduce the results obtained by the papers we cite as references to write this paper about SMT and affinity. In the following table, we check if a paper gives its scripts/codes, if they are at least minimally documented, and finally if they are runnable.

As we can see in Table 1, we could not reproduce the results of any of the articles cited (except one); this a regrettable for the computer science field. No source codes, no binaries available and even no documentation on how to reproduce the experiments. We could justify the bad scores because most of these papers are quite old (around 15-20 years old), and back in time, reproducibility was not a major concern for publishing. It now tends to become more important and many serious journals require artifacts.

Table 1. Reproducibility of SMT and affinity papers

Reference	Source code and scripts available	Data and Documentation	Runnable experiment (binary versions)
[21]	No	No	No
[22]	No	No	No
[14]	No	No	No
[16]	No	No	No
[17]	No	No	No
[23]	No	No	No
[24]	No	No	No
[18]	No	No	No
[19]	No	No	No
[20]	No	No	No
[25]	Yes	Yes	Yes
[26]	No	No	No
[27]	No	No	No
[28]	No	No	No
[29]	No	No	No
[30]	No	No	No

Computing papers about SMT and affinity use most of the time open-source benchmarks, and this is a good point. The major problem is that they do not give their code, analysis scripts and other artifacts, so we cannot reproduce the

published results. We think that our contribution is useful since we actualize the knowledge about SMT at the multicore (manycore) era, in a reproducible way (https://gitlab.isima.fr/beantunes/simultaneousmultithreading-evaluation).

The reproducibility aspect might concern different parts of our study. First, as we mentioned above, reproducibility is the fact that other researchers are able to replay a study conducted in a published paper, using the paper's artifacts. This is one of our concerns, and this is why we took the time to define reproducible research vocabulary. However, reproducibility is also about numerical reproducibility of application results, and about computation times. In this paper, we worked on two different hardware and software configurations, and we ensure the repeatability of numerical results from parallelization (which needs a specific procedure described in [31]), even if numerical results of running applications are not the main aspect of this paper, and that they are not used, bitwise same numerical results suppose the exact same computation, so differences between computing times, which we are studying here, cannot come from a different execution. Concerning reproducibility in terms of computing time, we did 30 replications so we could identify the variability. In fact, some published papers [32-34] are indeed concerned about the negative influence of SMT on computing time reproducibility, because SMT can increase variability of computing times (which cannot be of course "bitwise" as numerical results would do, but we can consider a stated precision, like ACM definitions would provide).

4. Materials and methods

We have designed an experiment to answer the three questions raised at the end of our introduction. We want to know the impact of SMT on performance, and we study AMD SMT and Intel HT. The AMD processor release date is mid-2019, while the Intel processor release date is end of 2013. This allows us to study the mid-term evolution or consistency of SMT performance impact, and to confirm or disprove the results we would obtain on each different clusters configuration with reproducibility in mind. For AMD, we have two nodes. Each node has two AMD EPYC Rome 7452 processors, each of them with 32-Cores, 64 threads, 2 MB of L1 cache, 16 MB of L2, 128 MB of L3 cache and 512GB RAM (DDR4). The maximum memory bandwidth is 190.7 GB/s. Each core can run at 3.35 GHz. The operating system is Ubuntu 20.04, with kernel version 5.4.0.153-generic, and we work with the gcc/g++ compiler version 9.4.0. The two nodes are the same, but on the first node, named here node-NoSMT, SMT is deactivated, and on the second, node-SMT, SMT is activated. This brings node-NoSMT to 64 cores (physical), and node-SMT to 128 logical cores (64 physical cores proposing 128 logical cores). Our experiments are easily balanced since we run replicates of stochastic simulations (embarrassingly parallel load). This means that we have identical tasks but each of them has a different random stream in order to obtain results with statistically independent runs. We have selected the Mersenne Twister pseudo random generator [35] and used it with different "independent" stream statuses [31]. The parallel technique used for all applications is known as the multiple replications in parallel (MRIP) approach [36]. We have considered four cases: We run 4 kinds of experiments: (1) 128 parallel tasks on node-NoSMT without setting the affinity; (2) 128 parallel tasks on node-NoSMT with affinity setting; (3) 128 parallel tasks on node-SMT without setting affinity and (4) 128 parallel tasks on node-SMT with setting affinity.

To see if we can have a stable growth of the speedup, we fixed the problem size to 128 simulation tasks, and we ran this problem on 1, 2, 4, 8, 16, 32, 64 and 128 cores. The speedup value is calculated by dividing the time taken to execute the bag of work (128 processes in our case) by N cores by the time taken to execute the bag of work on one core (Speedup = TimeOnNcores / TimeOnOneCore). Obviously, for the experiments on node-SMT with SMT and 128 logical cores, a task can be assigned to each core. On node-NoSMT with only 64 physical cores, two tasks will have to be processed on each core at full load. The expected result is that node-SMT should obtain better performance, by around 20% as mentioned in related works. The two nodes we use for this test are located in a cluster accessible with Slurm. All the scripts and code used are available on Gitlab (https://gitlab.isima.fr/beantunes/simultaneousmultithreading-evaluation). We reserved the whole capacity of both nodes (--exclusive Slurm access), even if we use only one core for example, so we do not have interference with other jobs that other lab researchers of our laboratory might have launched. The affinity is set with the taskset Linux command. For node-SMT (with SMT and 128 logical cores), we have first set the affinity only on physical cores as long as possible (from 1 to 64 tasks). Then for 128 tasks, the affinity assigns one process to each logical core (from 0 to 127). For node-NoSMT, with taskset, we did the same, one task by physical core until 64, and then two tasks by cores to process the 128 tasks.

For Intel, we also have two nodes. Each node has two Intel Xeon E5-2650 v2, each of them with 8-cores, 16

threads, 32K of L1 cache, 128K of L2 cache, and 10 MB of L3 cache and 128GB RAM. The operating system is CentOS Linux 7.9.2009, with kernel version 7.9.2009, and we work with the gcc/g++ compiler version 4.8.4. Turbo boost is enabled, to go from 1.2 GHz to 3.4GHz. The two nodes are exactly the same, but on the first node (named node-NoHT), HT is deactivated, and on the second (named node-HT), HT is activated. Due to the technical limits of the hardware at our disposal, experiment on Intel HT is smaller than on AMD (32 processes VS 128 processes). The software configurations of clusters are quite similar, but with different versions of operating system, compiler, libraries. Processor brand and generations are also different, with different SMT implementation. Due to these differences, it allows us to study the impact of SMT of performance depending of the type of applications, and to replicate this on two real-life cluster configurations, that users might encounter during their career, to confirm or disprove obtained conclusions.

On the two clusters, this is a local default non-uniform memory access (NUMA) policy. There are two AMD sockets, and two Intel sockets. Each chip is using its memory for its own cores (L1, L2 and L3 caches). By default, when a process runs on a core, the scheduler is likely to keep this process on this core to avoid loss of performance. This is also called affinity. SMT and HT are disabled on corresponding nodes at the basic input/output system (BIOS) level.

To analyze the SMT impact on performance in our computing clusters, we have selected four different kinds of stochastic applications. An Agent-Based Model (ABM) for COVID-19 epidemiological modelling written in C++ and with a large need of RAM, a benchmark for physics simulation at CERN written in Python, a simple compute bound Monte-Carlo simulation to estimate Pi written in C and a PRNG test library named TestU01written in C [37]. Each C or C++ application in compiled using gcc or g++ (versions mentioned above), and the –O2 option.

The ABM is described in [38]. The spatial agents modelled are humans moving on a map. The map is divided into cells and stored in a squared matrix. If some humans are contaminated, when they move, they can infect others using a differential Moore's neighbourhood (order 1 and 2 have different contamination probabilities). To have a realistic HPC load for our tests, we ran the simulations on a famous big city: Paris. The map is of 10,000 m² and needs around 2 GB RAM per simulation. The load will correspond to 256 GB of RAM for 128 parallel simulations on fat nodes with 512 GB of RAM. Such simulations do not fit in cache memory and because of some long-distance random moves of humans on the map, each time step of the simulation may inspect the content of cells scattered everywhere in the map. Because of this modelling constraint, there will be many cache misses.

The Monte-Carlo Pi simulation is a well-known toy case in the field of stochastic simulations. This is not at all the fastest or optimal way to get an estimation of Pi value, but it stands as an easy basis and benchmark for surface estimation (slowly converging at the square root of the number of points drawn). In addition, with this simple application, we have an example of easily reproducible compute bound application where everything may fit in the cache memory. Many real applications need data fetching and memory accesses, it is still a main concern now despite the improvements of DDR5 (Double Data Rate 5) and the increase of cache size. Memory access can often be the main limitation of intensive computing [39].

The third application we use for our experiments is the DB12 benchmark developed at CERN [40]. This benchmark has the objective to replicate the HEPSPEC06 benchmark used for HEP applications. This benchmark is small and written in Python3. It tries to model the load of stochastic simulations run on the World Large Computing Grid (WLCG) for CERN and it is interesting to check the behavior of such a benchmark used in HPC at CERN with SMT (SMT is almost always activated for compute intensive grid computing). Results of the benchmark itself shows that core performance gets lower as we increase the parallelism. Once we reach the full charge on SMT enabled machines, we noticed a difference in performance from physical and logical cores, as we will show in the results section.

The last application is a PRNG test library software called TestU01. To know if a PRNG is giving "random" numbers, we use statistical tests that should determine if there are visible correlations between generated random numbers. TestU01 is currently the most complete test battery we know to assess the quality of random numbers [31], particularly with its famous "BigCrush" battery. For our experiment, we have selected the SmallCrush test battery since it takes only 2 minutes, against 4 hours for BigCrush on modern processors. It would have been a waste of computing time for our massive parallel testing.

To run these experiments, we used Bash scripts for Slurm jobs. For example, Figure 1 below presents the script executing the bag of work of 128 processes on 4 cores, each running 32 sequential tasks, with affinity. As said previously, we used the "—exclusive" option to get the entire node reserved for our experiment. We used "—ntasks"to

set the number of nodes we want to use, and "mem=0" allows us to reserve all the RAM of the node. Each task corresponds to the execution of a command. Here, the task is "time taskset –c $(($i%64)) ./exe [args]". In the script given, we work on node-NoSMT, so we only have physical cores (64). With taskset (affinity), we set each task to work on its processor number (task 0 in core 0, task 1 on core 1, etc.). When we reach 128 tasks with only 64 cores, the "%64" assigns two tasks on each core. With simultaneous multi-threaded machines, we are careful about assigning only physical cores until the number of tasks gets bigger than the number of cores.

```
#!/bin/bash
#SBATCH -exclusive
#SBTACH -ntasks=4
#SBATCH -cpus-per-task=1
#SBATCH -nodelist=node-NoSMT
#SBATCH -mem=0
          #SBATCH -outpour=4processAffinity-SMA-Experiement1-Node34
for I in 'seq 0 3';
do
   time taskset -c $(($i%64)) ./exe $((0 + 32*$i)) $((31 + 32*$i)) configParis
   configNoMesure logConfigParisProc$i &
done
wait
```

Figure 1. Example of Bash script for Slurm jobs

We have retained an embarrassingly parallel workload approach at the operating system level to efficiently utilize multiprocessor nodes. This means we have a workload that can be easily divided into independent tasks, each of which can be processed in parallel without any need for complex coordination. Thus, we do not need classical parallel libraries: OpenMP, MPI or pthread with lightweight threads, and this allows us to focus on SMT performances and to avoid potential synchronization limitations that might arise with more sophisticated parallel processing methods. We want to make the best use of the available multiprocessor nodes in our systems. Rather than dividing each simulation into smaller units, we group them together to simplify the workload distribution. The "bags of work" can then be easily assigned to 128 or 32 parallel tasks (in the case of our AMD or Intel clusters, respectively).

In 1967 [41], Amdahl defined the speedup and its limits depending on the proportion of code which is mandatorily sequential. In our experiment, the size of the problem to process is 128 tasks and there is no sequential part, a linear speedup could be expected. Each task is a heavy process following the high-level parallelization model called SPMD (Single Program Multiple Data). This approach is intensively used on clusters for Experimental Designs and Sensitivity Analysis where huge parallel loads are required to explore the space of results depending on the number of levels for each factor of an experimental design. For such loads, the best we can propose is to assign one task per core, so we could expect to "divide" the computing time by 128 (AMD) or 32 (Intel). As said previously, we have no sequential parts in our tasks, we are in the Amdahl's law optimal case where the speedup is theoretically linear. With such an embarrassingly parallel problem, we want to see if we can reach this optimum. In this work, we did not only measure computing time, but we also checked the numerical results. Bitwise identical results, even if we are not considering them, allow us to be sure that the exact same execution path is done for each process, so gaps in computing time do not result from a gap in the execution. If one does not take care of the parallelization technique used for the stochastic simulations, it is common to obtain non-reproducible results [42]. In addition, if we want to be able to compare the sequential results of a stochastic simulation and the results obtained by a parallel version of this simulation, we have to prepare this carefully with special care to random streams (including a special approach for the sequential version which serves as reference. The procedure is described in [43]. When you execute your simulations sequentially, you have to think parallel and you still have to prepare different statuses of your pseudorandom number generator. This can be easily achieved with pre-computed statuses. Each task will have its own status, will it be executed sequentially or in parallel, and each task will produce the same result. Thanks to this, when you parallelize, you can obtain results identical to a reference sequential program. For example, we have a process with 128 tasks that will be executed sequentially. The corresponding local program is named "exe". If we parallelize the load on 4 processors for example, we want to execute 32 sequential simulations on each of the four cores: 0 – 31, 32 – 63, 64 – 95 and 96 – 127. This is what we are doing here with "$((0 + 32*$i)) $((31 + 32*$i))" in the script given in Figure 1. Inside the C++ code of the simulation, we initialize the Mersenne Twister generator with a status

depending on the number of the simulation. So, each simulation will be initialized with the same status, independently if it runs sequentially or in parallel. This allows us to get repeatable results between sequential and parallel execution. The performance evaluation is based on two metrics: the overall time taken for execution and the speedup achieved.

5. Results

We did 30 replications with a parallelization with 1, 2, 4, 8, 16, 32, 64 and 128 cores on AMD and 1, 2, 4, 8, 16 and 32 for Intel. We are not comparing AMD and Intel; on the contrary, we are validating our results on two processors from different brands and generations that we have at our disposal for this study. Figure 2A gives the result of the small compute intensive program that estimates the value of Pi, we can already get some information. First, SMT is effective, and gives us a 30% performance increase when compared to with the program running without SMT. At the end (128 tasks fully parallelized on 128/64 cores), we can see a slight benefit from affinity, with or without SMT. We did not show the 95% confidence interval error bars because the gap is so small that it does not appear on graphs, measures are statistically strong and the error bar is not significant for this study. With DB12, conclusions we can make from Figure 2B are similar to those obtained with the Pi estimation program. These applications are similar, with high computation, but there is an increasing need of memory for DB12 even if it remains small compared to the RAM available per node and per core (512 GB of RAM for 128 logical cores).

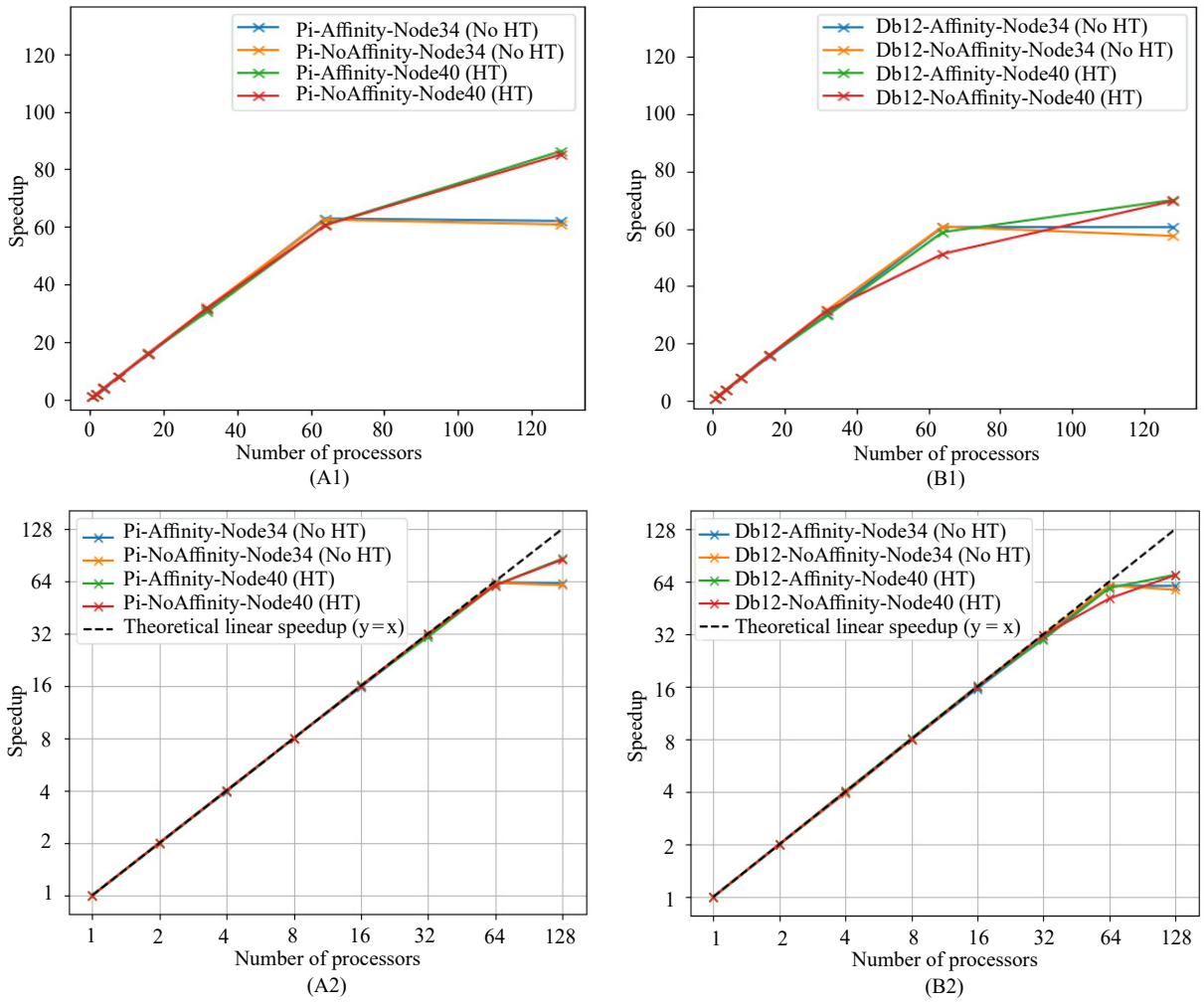

Figure 2. (A1 – A2) Monte Carlo Pi simulation speedup depending on SMT and affinity on AMD (128 cores); (B1 – B2) DB12 benchmark speedup depending on SMT and affinity on AMD (128 cores)

For the SmallCrush of TestU01 (Figure 3A), we have the same conclusion as for DB12 and Pi, where SMT is more efficient with 128 parallel processes. In Figure 3B, for the ABM simulation, the situation is different. First, we can see that, while the speedup was approximating linear for Pi, DB12 and SmallCrush, it is not the case here. Even worse, in Figure 3B we see that if we parallelize with more than 32 cores, performance without affinity starts decreasing. We can also note that the best speedup obtained is without affinity and for a low level of parallelization (32 cores). This means to us that letting the operating system decide could be a better option, if we do not set the affinity according to the exact hardware architecture (which is often not possible for non-expert users). We can also observe a curious phenomenon: after 32 cores, without affinity, there is a decrease in speedup up to 64 cores, before the performance meets a constant plateau close to a speedup of 16 (far from linear). We can also notice that, while affinity is the main difference in performance, the node without SMT is performing slightly better than the hyper-threaded node. We will discuss later the hypothesis of these results.

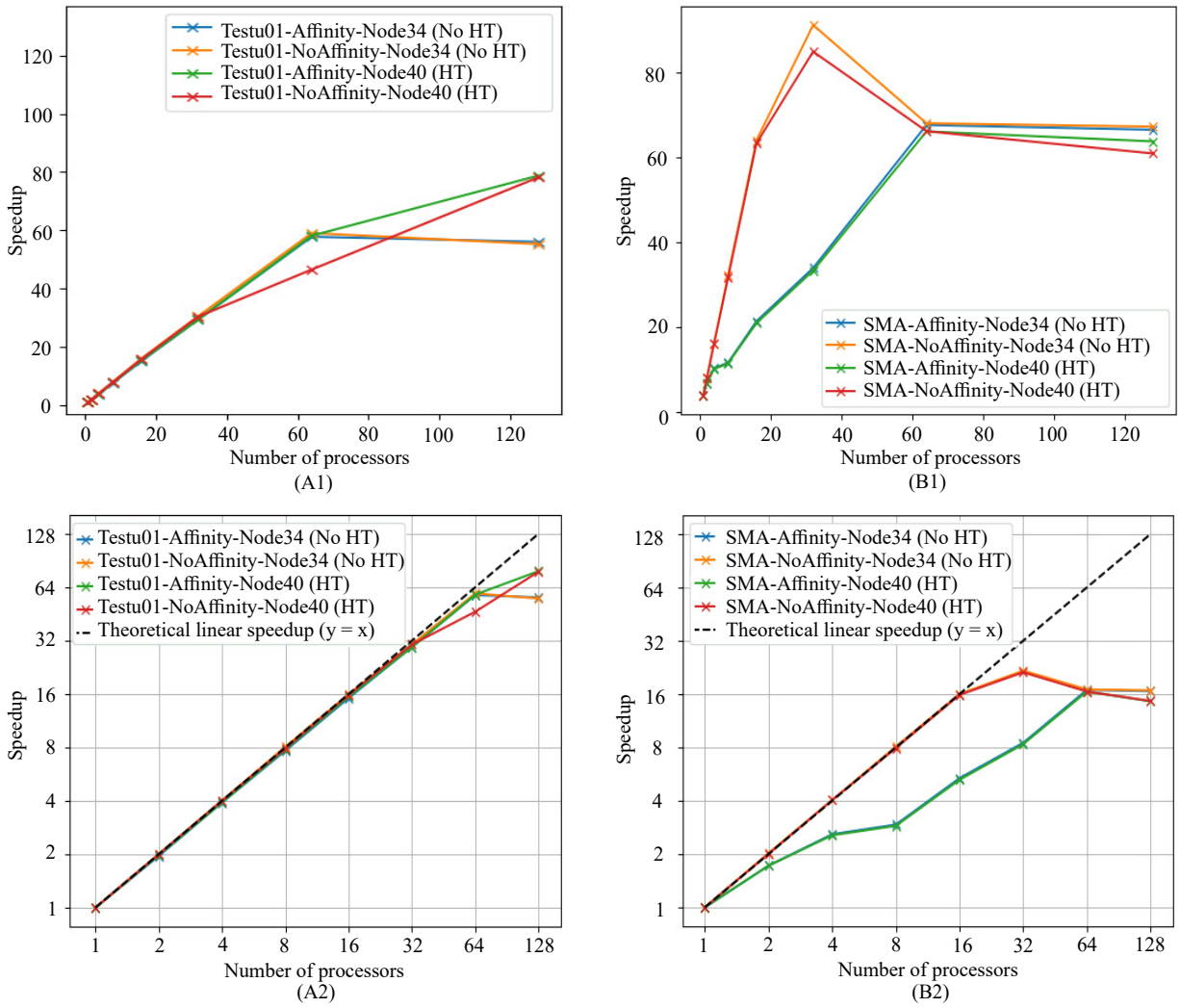

Figure 3. (A1 – A2) TestU01 speedup depending on SMT and affinity on AMD (128 cores); (B1 – B2) ABM speedup depending on SMT and affinity on AMD (128 cores)

Now, we will present results of the same experiment done on an Intel chip (HT From Figures 4 and 5, we can see that the results from Intel node are similar to those obtained with the more "modern" AMD processor. For compute bound applications like Pi Monte Carlo simulation, DB12 benchmark or SmallCrush from TestU01, HT is around 30% more effective (surprisingly not much for DB12 here). For large COVID-19 ABM, we can see that without HT, performance is

slightly better.

These results for the Intel cluster confirm what we obtained on the AMD clusters, with larger multicore chips.

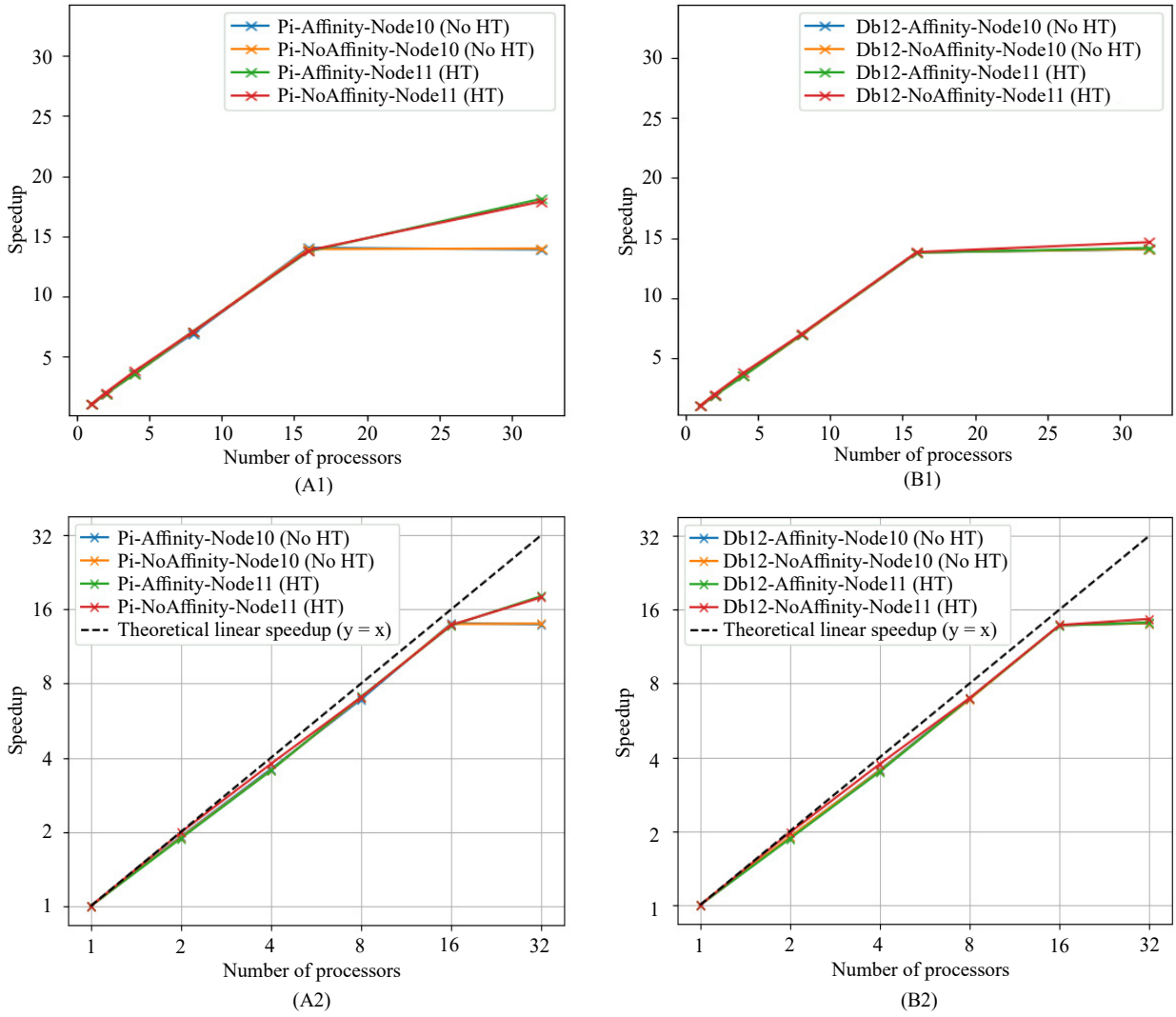

Figure 4. (A1-A2) Monte Carlo Pi simulation speedup depending on SMT and affinity on Intel (32 cores); (B1-B2) Db12 speedup depending on SMT and affinity on Intel (32 cores)

With the execution times given by Tables 2 and 3, we can see with the performance obtained that the use of SMT had a negative impact for the ABM application, which is memory bound. For the compute bound applications, SMT had a positive impact. In fact, on AMD, we can see that, in terms of time, the ABM takes around 12% more time with SMT activated. But for SmallCrush, it takes around 27% less time. On Intel, results are similar, but HT loses less performance for ABM, probably due to less overhead with only 32 processes (compared to 128). However, conclusions remain the same: SMT might decrease performance in the case of huge memory usage simulations.

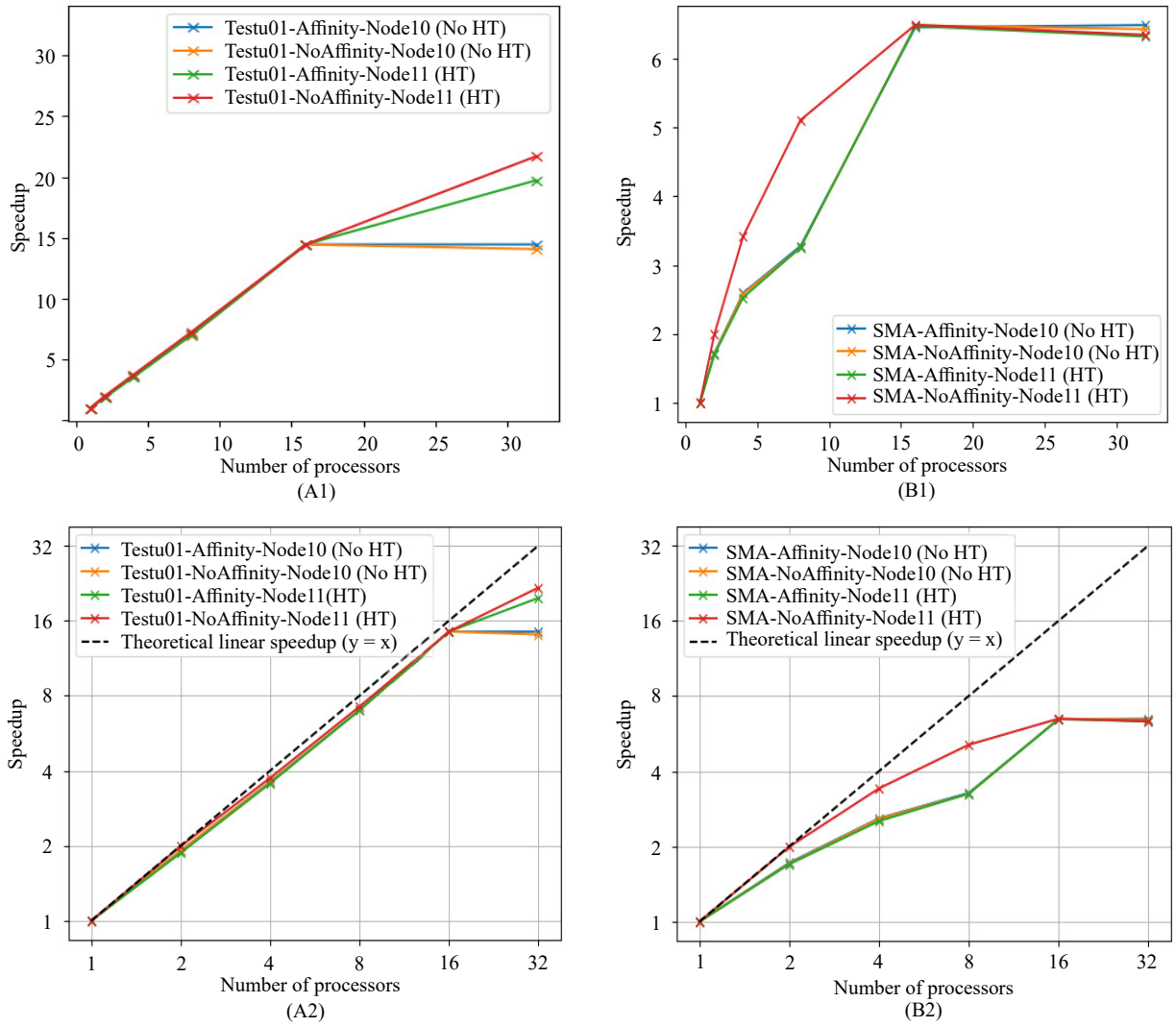

Figure 5. (A1-A2) TestU01 (SmallCrush) speedup depending on SMT and affinity on Intel (32 cores); (B1-B2) ABM speedup depending on SMT and affinity on Intel (32 cores)

Table 2. Mean time and speedup for 128 cores (AMD SMT) and percentage of increase or decrease of performance, on the ABM and TestU01 applications, comparing SMT enabled or not

	ABM time (minutes)	TestU01 time (minutes)	% time increase (SMT)		ABM speedup	TestU01 speedup	% increase speedup	
			ABM	TestU01			ABM	TestU01
AMD Node-NoSMT Affi	248.42	0.2301	11.8%	−28.99%	16.62	56.06	−12.39%	40.62%
AMD Node-SMT Affi	277.74	0.1634			14.56	78.83		
AMD NoSMT NoAffi	248.27	0.221	12.02%	−25.34%	16.79	55.29	−12.03%	39.6%
AMD Node-SMT NoAffi	278.11	0.1650			14.62	78.26		

Table 3. Mean time and speedup for 32 cores (Intel HT) and percentage of increase or decrease of performance, on the ABM and TestU01 applications, comparing HT enabled or not

	ABM time (minutes)	TestU01 time (minutes)	% time increase (HT)		ABM speedup	TestU01 speedup	% increase speedup	
			ABM	TestU01			ABM	TestU01
Intel Node-NoHT Affi	178.3	0.25	2.05%	−26.8%	6.49	14.46	−2.62%	36.45%
Intel Node-HT Affi	181.95	0.183			6.32	19.73		
Intel Node-NoHT NoAffi	178.9	0.257	1.84%	−35.02%	6.43	14.08	−1.4%	54.12%
Intel Node-HT NoAffi	182.2	0.167			6.34	21.7		

In our previous results, we calculated the average execution time for the entire bag of work using all available cores (128 and 32). The baseline time for calculating speedup is the execution time of this bag of work on a single core. We conducted 30 replications, each with 95% confidence intervals. As stated previously, the intervals were too small to be visually represented on the graphs, indicating that the computing time exhibited low variability in our case. This is noteworthy, as many studies focus on the variability introduced by SMT usage.

Concerning the results of the DB12 benchmark itself, it shows that core performance gets lower as we increase the parallelism. Once we reach the full charge on SMT enabled machine, we noticed a difference in performance from physical and logical cores. The higher the score is, the less performance the processor has. We can clearly see from Figures 6 and 7 that without SMT enabled, on AMD or on Intel, the CPU performance lightly decrease as we increase the parallelism. Meanwhile, with SMT enabled, we clearly see the trigger point of doing our computation on logical core instead of physical cores (at 16 for Intel and at 64 for AMD). For the DB12 benchmark, a logical core is just twice worse than a physical core.

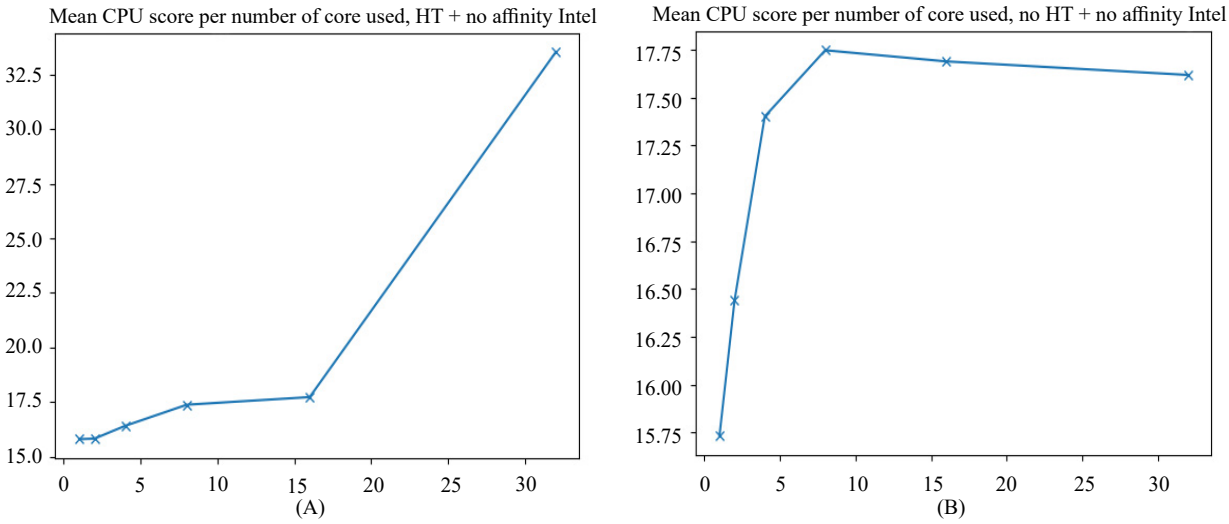

Figure 6. (A) DB12 CPU score with HT on Intel (32 cores); (B) DB12 CPU score without HT on Intel (32 cores)

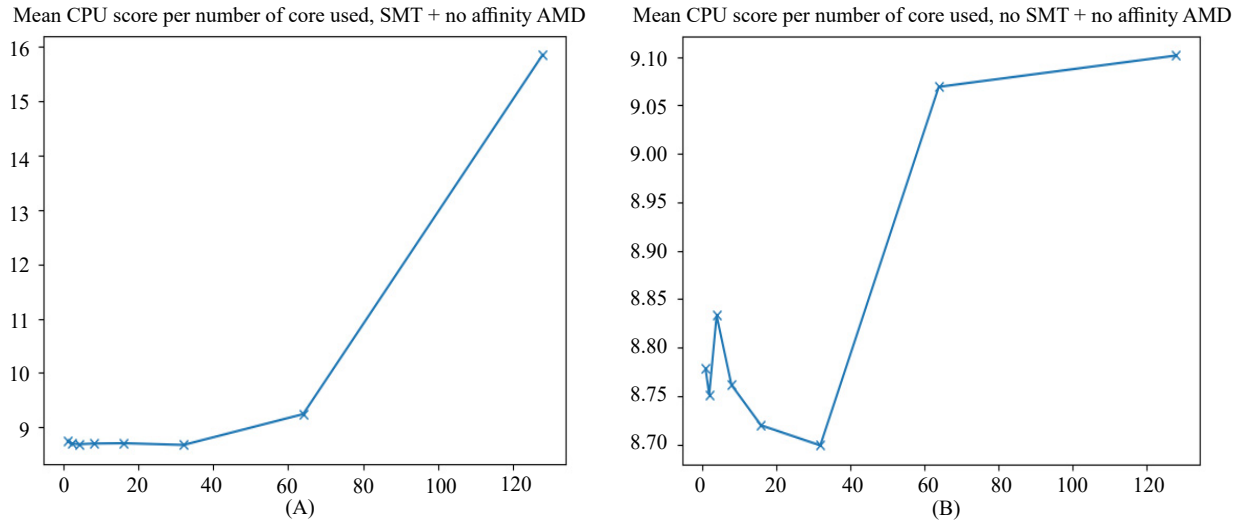

Figure 7. (A) DB12 CPU score with SMT on AMD (128 cores); (B) DB12 CPU score without SMT on AMD (128 cores)

Secondly, to talk about DB12 reproducibility of the measures, we have checked the mean and the standard deviation over 30 replications. We then see how many differences we can observe from run to run in exactly the same configuration. This can of course occur from many CPU benchmarks where we do not expect exactly the same measure. Since time is measured, the lowest results are better. We can notice that the recent AMD processor is significantly faster than the older Intel. In addition, we see that the variance is much smaller with the AMD chip. This is shown in Figures 8 and 9 for respectively AMD and Intel processors.

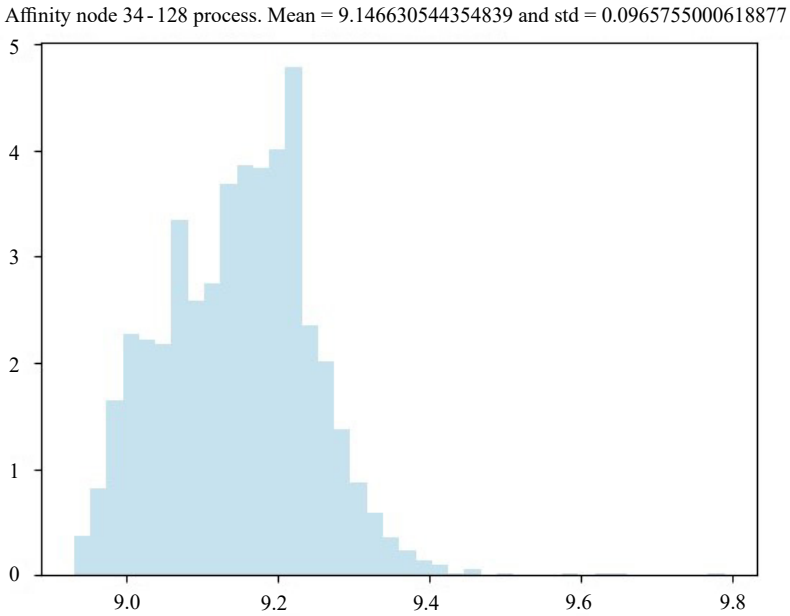

Figure 8. DB12 CPU scores density function on the AMD processor. The scores are presented on the X axis with the mean and standard deviation on AMD over 30 replications (128*30 values) (128 cores) – without SMT

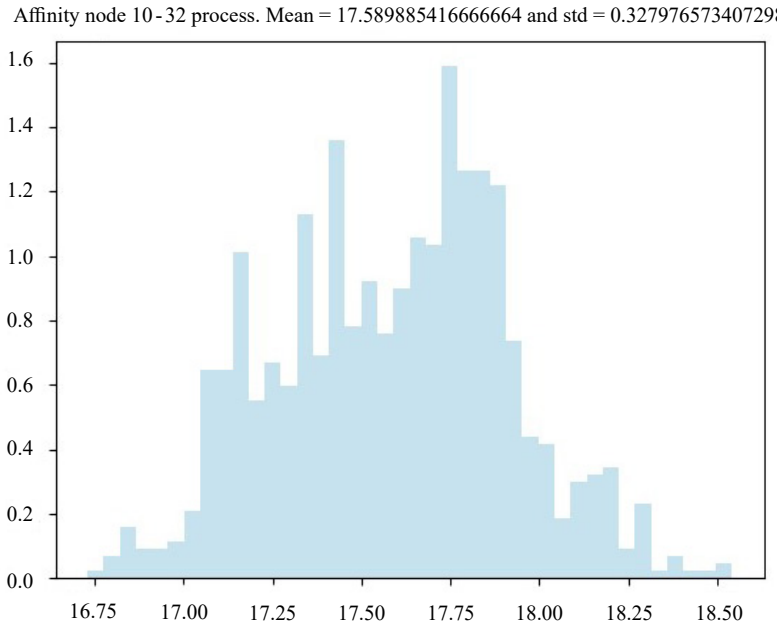

Figure 9. DB12 CPU scores density function on the Intel processor. The scores are presented on the X axis with mean and standard deviation over 30 replications (30*32 values) (32 cores) – without HT

6. Discussion

From the results obtained, there are some hypotheses that we can make. First, on AMD with 128 processes, we can see that setting the affinity with taskset is not always efficient, particularly when the nodes execute heavy loads on all cores. Perhaps, optimizing results by considering NUMA nodes and the CPU hardware architecture when setting affinity might yield better performance. However, it's worth noting that non-expert users are unlikely to possess the knowledge or capability to implement such optimizations. We can see the loss of time when executing the ABM application, which is memory bound. Without setting process affinity, we have obtained better speedup when 1 to 32 cores are used for parallelization. And then, the speedup without affinity is decreasing from 32 to 64 cores. When we took hand on affinity, we could obtain a constant increase of the speedup with the same pace, with or without SMT up to 64 cores. The performance without SMT is a little better for this kind of ABM memory bound application. Above 64 cores, we reach approximately the same plateau. At full charge, SMT did increase the overall time by around 10%. However, the best speedup was obtained with 32 cores only and without affinity. We used the hwloc software from INRIA [44] to find that, on AMD core, L1 and L2 cache are duplicated for each physical core (64 KB and 512 KB respectively), so shared between logical ones, and L3 (16 MB) cache is shared between four physical cores.

Secondly, and most importantly, we can see that for compute bound applications like Pi, DB12 or TestU01, we have obtained an increase in performance around 30% with SMT. This can be expected according to the HT technology as shown in [4]. However, for HPC applications with huge memory needs like the ABM we used, simultaneous multi-threading is less efficient. The main computing difference between ABM and small programs like Pi, DB12 or TestU01 is probably linked to memory bottlenecks and cache misses. The reason why SMT can lead to a decrease of performance for memory consuming applications might be that execution resources are shared between the two hyper-threads of a core. This means that the cache size is halved again this can lead to more cache misses (except for the case of the L1 cache specific to logical cores but it has a very small size). A case in which it might be beneficial is if the working set sizes (if they are disjoint) of the two hyper-threads are small enough that they still fit comfortably within the capacity of the caches. Unless the threads are tightly coupled through sharing, this is unlikely to happen with L1 sizes. In our hypothesis, we get close to the conclusions of the NASA paper [17] which deals with fluid mechanics application. On one of their applications, they did not observe a performance gain with SMT because “SMT increases competition for resources in the memory hierarchy, such as memory bandwidth”. Moreover, SMT performance is affected by increased communication pressure as additional processes compete for network resources such as IB (InfiniBand) HCA (Host Channel Adapter) chips and IB switches.

Therefore, we should care about advantages and drawbacks about SMT. We agree that the competition for cache and for memory in general results in more cache misses, and more bus saturation for realistic HPC memory bound applications, and not for small cases. It questions the usefulness of SMT on computing clusters since we may run large programs that will generate memory contentions.

And lastly, about the speedup, we can see that practical applications do not really meet the theoretical expectations. In fact, for small programs, we can see that the speedup grows quite linearly (around 20/30% slower than linear with SMT, and 50% without). And with SMT activated, we can see a real benefit since SMT processors are able to approximate Amdahl's law as if their logical cores were physical cores [41]. This is a good point for SMT processors. Without SMT, we can see the expected stagnation of the speedup. Nevertheless, for the ABM application, the speedup curve does not look like we expected at all. First, we see a negative impact of affinity that might be surprising. In addition, we can see that the speedup is not only sublinear, but clearly decreases after 32 cores. This difference between theoretical expectation and experimental value is, in our opinion, because Amdahl's law is only considering computing power. We should also consider memory access, and its ratio with the computing power. The result is that increasing computing resources without taking care of the interconnect, the memory speed, caches, etc., might lead to a real losing situation. We lose computing time, and we lose energy consumption. As described in [45], we can see that to optimize energy consumption and computing power ratio, we have to be very careful about associating more computing power to faster memory access. And we can see here, in our ABM case, increasing computing resources is not only a waste of energy, but also a loss of time (and more time equals more energy waste). This shows the limitations of Amdahl's law about estimating speedup for real HPC programs based on the number of cores when there is a need of a rather large memory per core. Our study shows the limits in speedup for memory bound problems with and without affinity, with and without SMT. The bias we might have on this is that the cluster we used is not as optimized as a supercomputer. We also have to keep in mind that the processors we tested here are built and optimized for SMT. Deactivating SMT on these cores might not lead to optimal performance. With single threaded cores, results could have been more significant for huge simulations. In fact, SMT is a cost when we make chips, plus we see now that the tendency is to multiply the number of cores, while decreasing their own clock speed. By doing this, we put more pressure on other components like memory, bus or interconnect, and they become the limiting factor. We show that this can affect the performance when we run memory bounded simulations.

7. Reproducing the results

On a purpose of reproducible research, we found it pertinent to add a reproducibility part to this paper. The original idea to create a "reproducibility" section in a research paper to enhance reproducibility in the writing process and in the review process comes from [46]. All code, script and data can be found at https://gitlab.isima.fr/beantunes/simultaneousmultithreading-evaluation.

Two folders: Intel-hpcnodeX-10 and AMD-Node34-40 are providing code and data. The first one contains experiment done on Intel nodes, and the second one contains experiment done on AMD nodes. In each folder, you will find additional subfolders: Db12-bench, SMA-Covid, testMonteCarlo, testU01 and mts-0000-9999. The first four folders contain all bash scripts to run the experiment, the codes and results file. The mts-0000-9999 folder contains all Mersenne Twister statuses used for repeatable parallel stochastic simulations. There is also a Jupyter Notebook file. This is the one you can use to reproduce all the results. To redo the experiment from beginning, you would need to run all scripts again to obtain data, that are analyzed by the Jupyter notebook. We used the script called "runBash.sh" to run them all, with Slurm. You can make your own choices on how you want to run these scripts and how many replications you want to do. For Db12, the program is a Python script, so it does not need any compilation. For SMA-Covid, use the command "make" to compile the project and generate executable. For testMonteCarlo, it is a small C file that needs to be compiled by yourself such as "gcc calculpi.c –O2 –o exe". Finally, for TestU01, you will have to follow the instructions given by L'Ecuyer on the official TestU01 website.

8. Conclusions

We tried to answer the following questions: is SMT always effective to increase performance, on modern classical computing clusters, when used by non-expert users? Is it suitable for HPC and more particularly for stochastic simulations, which are not solely compute bound? Does the user management of CPU affinity increase performance?

We have shown that knowing if SMT and affinity have a positive impact is not trivial. It depends on many different variables like the profile of the application we want to run, the architecture of the physical machine, etc. With this paper, we presented repeatable parallel stochastic simulation results following the method proposed in [31]. Our results can be reproduced with public access to the artifact. We saw that, for applications with low memory needs, the SMT technology is still very effective at the large multicore era, with up to 30% performance increase compared to SMT disabled. This represents an interesting increase up to 10% more performance for compute bound applications when compared to the initial papers discussing the efficiency of HT/SMT. On the other hand, for applications with large memory needs, we observe a loss of performance with SMT by around 10%. We bring two hypotheses to explain this: First, when we have more processes running simultaneously, we can face the limits of the bus bandwidth that results in a bottleneck to access memory and a loss of time. Secondly, simultaneous multi-threaded cores share 95% of their physical resources, like the cache memory. Therefore, cache size is halved for logical cores. We can then imagine that it results in more cache misses for applications in need of a lot of memory.

Nowadays, supercomputers are all configured with simultaneous multi-threading enabled. With this paper, we discuss the pertinence of that, at an era of dense multicore (up to 256 logical cores on a single CPU). It could become pertinent to split clusters with different configurations depending on the different application profiles. Further work will test the performance of this kind of real ABM application on spare cycles of a supercomputer with fast memory access, a fast interconnect and with larger problems. We might also want to work on the IBM SMT implementation.

Conflict of interest

There is no conflict of interest for this study.

References

[1] Collberg C, Proebsting T. Repeatability in Computer Systems Research. *Communications of the ACM*. 2016; 59(3): 62-69. https://doi.org/10.1145/2812803

[2] Mesnard O, Barba LA. Reproducible and replicable computational fluid dynamics: it's harder than you think. *Computing in Science & Engineering*. 2017; 19(4): 44-55. https://doi.org/10.1109/MCSE.2017.3151254

[3] Boyer AF. *Contributions to Computing needs in High Energy Physics Offline Activities: Towards an efficient exploitation of heterogeneous, distributed and shared Computing Resources*. PhD Thesis. Université Clermont Auvergne; 2022.

[4] Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, et al. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*. 2002; 6(1): 4-15.

[5] Hwu WW, Patt NY. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Transactions on Computers*. 1987; 100(12): 1496-1514. https://doi.org/10.1109/TC.1987.5009500

[6] Goldberg D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*. 1991; 23(1): 5-48. https://doi.org/10.1145/103162.103163

[7] Lutz DR, Hinds CN. High-precision anchored accumulators for reproducible floating-point summation. In: *IEEE 24th Symposium on Computer Arithmetic (ARITH)*. London, UK: IEEE; 2017. p.98-105. https://doi.org/10.1109/ARITH.2017.20

[8] Eggers SJ, Emer JS, Levy HM, Lo JL, Stamm RL, Tullsen DM. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*. 1997; 17(5): 12-19. https://doi.org/10.1109/40.621209

[9] Drummond C. Replicability is not reproducibility: nor is it good science. In: *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*. Montreal, Canada: National Research Council of Canada; 2009.

[10] Barba LA. Terminologies for reproducible research. *arXiv* [Preprint] 2018. Version 1. https://doi.org/10.48550/arXiv.1802.03311

[11] Association for Computing Machinery (ACM). *Artifact Review and Badging – Current*. https://www.acm.org/publications/policies/artifact-review-and-badging-current [Accessed 30th May 2023].

[12] Hinsen K. *Reproducibility, replicability, and the two layers of computational science*. https://khinsen.wordpress.com/2014/08/27/reproducibility-replicability-and-the-two-layers-of-computational-science/ [Accessed 30th May 2023].

[13] Rougier NP, Hinsen K, Alexandre F, Arildsen T, Barba LA, Benureau FC, et al. Sustainable computational science: the ReScience initiative. *PeerJ Computer Science*. 2017; 3: e142. https://doi.org/10.7717/peerj-cs.142

[14] Leng T, Ali R, Hsieh J, Mashayekhi V, Rooholamini R. An empirical study of hyper-threading in high-performance computing clusters. In: *3rd LCI International Conference on Linux Clusters: The HPC Revolution 2002*. Florida, USA: Linux Clusters Institute; 2002. p.1-12.

[15] Tuck N, Tullsen MD. Initial observations of the simultaneous multithreading Pentium 4 processor. In: *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. New Orleans, USA: IEEE; 2003. p.26-34. https://doi.org/10.1109/PACT.2003.1237999

[16] Bulpin RJ, Pratt IA. Multiprogramming performance of the Pentium 4 with simultaneous multi-threading. In: *Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD)*. München, Germany; 2004. p.53-62. https://doi.org/10.1109/PACT.2003.1237999

[17] Saini S, Jin H, Hood R, Barker D, Mehrotra P, Biswas R. The impact of hyper-threading on processor resource utilization in production applications. In: *18th International Conference on High Performance Computing*. Bangalore, India: IEEE; 2011. p.1-10. https://doi.org/10.1109/HiPC.2011.6152743

[18] Foong A, Fung J, Newell D. An in-depth analysis of the impact of processor affinity on network performance. In: *Proceedings 12th IEEE International Conference on Networks (ICON)*. Singapore: IEEE; 2004. p.244-250. https://doi.org/10.1109/ICON.2004.1409136

[19] Kazempour V, Fedorova A, Alagheband P. Performance implications of cache affinity on multicore processors. In: *European Conference on Parallel Processing*. Berlin: Springer; 2008. p.151-161. https://doi.org/10.1007/978-3-540-85451-7_17

[20] Bordage C, Jeannot E. Process affinity, metrics and impact on performance: An empirical study. In: *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. Washington DC, USA: IEEE; 2018. p.523-532. https://doi.org/10.1109/CCGRID.2018.00079

[21] Bononi L, Bracuto M, D'Angelo G, Donatiello L. Exploring the effects of hyper-threading on parallel simulation. In: *2006 Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications*. Malaga, Spain: IEEE; 2006. p.257-260. https://doi.org/10.1109/DS-RT.2006.18

[22] Tikir MM, Carrington L, Strohmaier E, Snavely A. A genetic algorithms approach to modeling the performance of memory-bound computations. In: *SC'07: Proceedings of the ACM/IEEE Conference on Supercomputing*. Reno Nevada, USA: ACM; 2007. p.1-12. https://doi.org/10.1145/1362622.1362686

[23] Gilbert L, Tseng J, Newman R, Iqbal S, Pepper R, Celebioglu O, et al. Performance implications of virtualization and hyper-threading on high energy physics applications in a grid environment. In: *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*. Denver, USA: IEEE; 2005. p.10. https://doi.org/10.1109/IPDPS.2005.338

[24] Celebioglu O, Saify A, Leng T, Hsieh J, Mashayekhi V, Rooholamini R. The performance impact of computational efficiency on HPC clusters with hyper-threading technology. In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. Sante Fe, USA: IEEE; 2004. p.250. https://doi.org/10.1109/IPDPS.2004.1303311

[25] Osborne SH, Bakita JJ, Anderson JH. Simultaneous multithreading applied to real time. *Dagstuhl Artifacts Series*. 2019; 5(1): 8:1-8:2. https://doi.org/10.4230/DARTS.5.1.8

[26] Bakita J, Ahmed S, Osborne SH, Tang S, Chen J, Smith FD, et al. Simultaneous multithreading in mixed-criticality real-time systems. In: *IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Nashville, USA: IEEE; 2021. p.278-291. https://doi.org/10.1109/RTAS52030.2021.00030

[27] Osborne SH, Ahmed S, Nandi S, Anderson JH. Exploiting simultaneous multithreading in priority-driven

hard real-time systems. In: *IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Gangnueng, South Korea: IEEE; 2020. p.1-10. https://doi.org/10.1109/RTCSA50079.2020.9203575

[28] Chen Y, Shi Q, Li X. CSSMT: Compiler based software simultaneous multithreading (SMT). In: 2*6th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Cambridge, UK: IEEE; 2018. p.60-67. https://doi.org/10.1109/PDP2018.2018.00017

[29] Ide Y, Yamasaki N. A learning-based fetch thread gating mechanism for a simultaneous multithreading processor. In: *Eighth International Symposium on Computing and Networking (CANDAR)*. Naha, Japan: IEEE; 2020. p.1-10. https://doi.org/10.1109/CANDAR51075.2020.00011

[30] Hoo KC, Ee OS, Yin TS. Effect of simultaneous multithreading towards the performance of cloud workloads. In: *Embedded World Conference 2022*. Haar, Germany: WEKA FACHMEDIEN GmbH; 2022. p.23-26.

[31] Hill D, Passerat-Palmbach J, Mazel C, Traore MK. Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*. 2013; 25(10): 1427-1442. https://doi.org/10.1002/cpe.2942

[32] Jin X, Zhou Y, Huang B, Yu Z, Zhan X, Wang H, et al. QoSMT: supporting precise performance control for simultaneous multithreading architecture. In: *Proceedings of the ACM International Conference on Supercomputing*. Phoenix Arizona, USA: ACM; 2019. p.206-216. https://doi.org/10.1145/3330345.3330364

[33] Zhang Y, Laurenzano MA, Mars J, Tang L. Smite: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In: *47th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge, UK: IEEE; 2014. p.406-418. https://doi.org/10.1109/MICRO.2014.53

[34] Rosenthal E, León EA, Moody AT. *Mitigating system noise with simultaneous multi-threading*. [Poster] SC'13: The International Conference on High Performance Computing, Networking, Storage and Analysis. 20th-21st November 2013. https://sc13.supercomputing.org/sites/default/files/PostersArchive/post275.html

[35] Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*. 1998; 8(1): 3-30. https://doi.org/10.1145/272991.272995

[36] Passerat-Palmbach, J, Caux J, Siregar P, Mazel C, Hill D. Warp-level parallelism: enabling multiple replications in parallel on GPU. In: *European Simulation and Modelling Conference*. Guimaraes, Portugal: EUROSIS-ETI; 2011. p.76-83.

[37] L'ecuyer P, Simard R. TestU01: AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS.)* 2007; 33(4): 1-40. https://doi.org/10.1145/1268776.1268777

[38] Hill D, Antunes B. Reproductibilité et modèles Covid-un modèle multi-agents. In: *Journées DEVS Francophones-Convergences entre la Théorie de la Modélisation et de la Simulation et les Systèmes multi-agents*. Cargèse, France: Cépaduès; 2022. https://hal.uca.fr/hal-03768175v1

[39] Wulf WA, McKee S. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*. 1995; 23(1): 20-24. https://doi.org/10.1145/216585.216588

[40] Boyer AF, Haen C, Stagni F, Hill D. Porting DIRAC Benchmark to Python3: impact of the discrepancies and solutions. In: *Proceedings of the 41st International Conference on High Energy physics*. Bologna, Italy: PoS; 2022. p.1-4. https://hal.uca.fr/hal-04045256

[41] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the spring joint computer conference*. Atlantic City, USA: ACM; 1967. p.483-485. https://doi.org/10.1145/1465482.1465560

[42] Hill D. Numerical reproducibility of parallel and distributed stochastic simulation using high-performance computing. In: Traoré MK. (ed.) *Computational Frameworks*. London, UK: Elsevier; 2017. p.95-109. https://doi.org/10.1016/B978-1-78548-256-4.50004-1

[43] Hill D. Parallel random numbers, simulation, and reproducible research. *Computing in Science & Engineering*. 2015; 17(4): 66-71. https://doi.org/10.1109/MCSE.2015.79

[44] Broquedis F, Clet-Ortega J, Moreaud S, Furmento N, Goglin B, Mercier G, et al. hwloc: A generic framework for managing hardware affinities in HPC applications. In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. Pisa, Italy: IEEE; 2010. p.180-186. https://doi.org/10.1109/

PDP.2010.67
[45] Szalay AS, Bell G, Huang HH, Terzis A, White A. Low-power amdahl-balanced blades for data intensive computing. *ACM SIGOPS Operating Systems Review*. 2010; 44(1): 71-75. https://doi.org/10.1145/1740390.1740407
[46] Bajpai V, Kühlewind M, Ott J, Schönwälder J, Sperotto A, Trammell B. Challenges with reproducibility. In: *Proceedings of the Reproducibility Workshop*. Los Angeles, USA: ACM; 2017. p.1-4. https://doi.org/10.1145/3097766.3097767