Draft: sOMP: NUMA and cache-aware simulations for task-based applications

Idriss Daoudi, Samuel Thibault, Thierry Gautier

sOMP: NUMA and cache-aware simulations for task-based applications

Idriss Daoudi, Samuel Thibault, Thierry Gautier

RESEARCH REPORT

N° 9400

March 2021

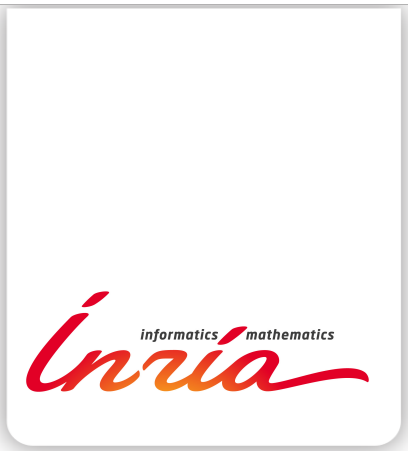
Project-Teams STORM - AVALON

ISSN 0249-6399 ISRN INRIA/RR--9400--FR+ENG

Draft: sOMP: NUMA and cache-aware simulations for task-based applications

IMPORTANT:

This version of research report 9400 is only a draft of a paper currently being submitted for publication, this is not published material.



sOMP: NUMA and cache-aware simulations for task-based applications

Idriss Daoudi, Samuel Thibault, Thierry Gautier

Project-Teams STORM - AVALON

Research Report n° 9400 — version 2 — initial version March 2021 — revised version April 2021 — 25 pages

Abstract: Anticipating the behavior of applications, studying, and designing algorithms are some of the most important purposes for the performance and correction studies about simulations and applications relating to intensive computing. Many frameworks were designed to simulate large distributed computing infrastructures and the applications running on them. At the node level, some frameworks have also been proposed to simulate task-based parallel applications. However, one missing critical capability from these works is the ability to take Non-Uniform Memory Access (NUMA) effects into account, even though virtually every HPC platform nowadays exhibits such effects. We thus enhance an existing simulator for dependency-based task-parallel applications, that enables experimenting with multiple data locality models. We also introduce two locality-aware performance models: we update a lightweight communication-oriented model that uses topology information to weight data transfers, and introduce a more complex communications and cache model that takes into account data storage in the LLC. We validate both models on dense linear algebra test cases and show that, on average, our simulator reproducibly predicts execution time with a small relative error.

Key-words: shared-memory, simulation, NUMA, tasks, modeling

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

sOMP: simulations prenant en charge le cache et les effets NUMA pour les applications à base de tâches

Résumé : Anticiper le comportement des applications, étudier et concevoir des algorithmes sont parmi les objectifs les plus importants des études de performance et de correction sur les simulations et les applications liées au calcul intensif. De nombreux outils ont été conçus pour simuler de grandes infrastructures informatiques distribuées et les applications qui y sont exécutées. Au niveau du nœud, certains outils ont également été proposés pour simuler des applications parallèles à base de tâches. Cependant, une capacité critique manquante à ces travaux est de pouvoir prendre en compte les effets NUMA (Non-Uniform Memory Access), alors que pratiquement toutes les plates-formes HPC présentent aujourd'hui de tels effets. Nous améliorons ici un simulateur pour les applications parallèles à base de tâches avec dépendances, qui permet d'expérimenter plusieurs modèles de localité de données. Nous introduisons également deux modèles de performances: nous améliorons un modèle orienté communication léger, et nous introduisons un modèle de communication et de cache plus complexe qui prend en compte le stockage des données dans le LLC. Nous validons les deux modèles sur des cas test d'algèbre linéaire dense et montrons qu'en moyenne, notre simulateur prédit de manière reproductible le temps d'exécution avec une erreur relative faible.

Mots-clés : mémoire partagée, simulation, NUMA, tâches, modélisation

1 Introduction

Task-based runtimes exist for a long time. They were popularized in 1998 by Cilk [5] and the initial theoretical proof of the performance guarantee of the work-stealing algorithm. The Cilk model of independent tasks on shared-memory machines was further extended in several directions: the capacity to define point-to-point synchronisation between tasks in the dependent task models [18], to run on heterogeneous architectures with accelerators [4, 2, 20] or distributed memory systems [19, 7, 12, 2]. The wide acceptance of the task-based programming model and its capacity for writing portable programs across a large category of architectures was consecrated in 2008 by introducing the independent task model in version 3.0 of the OpenMP standard, further extended in 2013 with the dependent task model and the capacity to target accelerators.

Nevertheless, on a shared-memory machine, the complex memory hierarchy requires precise temporal and spatial localization of the data to obtain good performance. Thus several schedulers for task-based OpenMP programs have been proposed so as to deal with NUMA effects [29, 37], and the standard has integrated the capacity to control affinity of threads to cores and to give affinity hints for tasks with respect to data. Since shared-memory machines are being more and more complex, with a hierarchy of NUMA nodes, it is necessary to understand the impact of architectural decisions on the performance of applications.

Experimenting with such platforms is however hindered by technical constraints (aging hardware, system upgrade, ...) which impair the reproducibility of the results. Some works [34] ironed such issues out for the case of GPU-based platforms thanks to *simulation*. Indeed, it notably allows to understand if the application has been designed properly, if it is getting executed efficiently by the runtime, and to test its limits and sensitivity to hardware and network. This opened the door for scheduling research on such platforms that is both realistic and reproducible [1]. Simulation allowed for quick-prototyping, before actual implementation for real systems.

Achieving the same level of simulation quality for (OpenMP) task-based application on shared-memory will similarly make it possible to quickly design, prototype, and eventually implement the right runtime for the right hardware, thanks to a robust reproducible methodology based on reliable simulations. The previous works [34] have however also shown that accurately predicting the performance of the current complex platforms strongly requires taking NUMA and cache effects into account, and none of the available tools meets this growing need for both NUMA awareness and cache effects of dependent tasks simulation.

The work of Daoudi & al. [14] proposed a preliminary simulator (called sOMP), based on SimGrid [10], to predict the performance of a task-based application on a shared-memory architecture. They modeled the NUMA structure of the platform and studied the impact of data locality on execution times. However, their architecture model does not take cache effects into account, and is too simple to capture complex NUMA architectures, thus making the prediction on some applications less relevant. In this paper, we extend their work:

- We model complex NUMA and cache architectures;
- We refine the task execution simulation to take into account overlapping between communication and computation;
- We introduce L3 caching in the simulation, that strongly improves simulation accuracy;
- We study the cost of the simulation;
- We show that we can easily experiment with a proof-of-concept cache-aware scheduler thanks to the refined models.

After presenting the state of the art, we detail the simulation principles. We recap a previously-proposed model which does not take data locality into account, and we extend another previously-proposed model which takes NUMA locality into account. We then introduce a refined model that additionally takes into account cache locality. We eventually present simulation accuracy results for various application algorithms, matrix sizes, Intel and AMD platforms, discuss the cost of the simulator, and discuss the resulting potential for cache-aware scheduling research.

2 State of the art

Many simulators have been designed for predicting performance in a variety of contexts in order to analyze application behavior. Several simulators have been developed to study the performance of MPI applications on simulated platforms, such as BigSim [39], xSim [16], the trace-driven Dimemas tool [21], or MERPSYS [13] for performance and energy consumption simulations. Some others are oriented towards cloud simulation like CloudSim [9] or GreenCloud [25].

Other studies are oriented towards simulations on specific architectures, such as the work by Aversa and al. [3] for hybrid MPI/OpenMP applications on SMP, and task-based applications simulations on multicore processors [30, 33, 23, 35, 31]. All these studies present approaches with reliable precision, but, as with Simany [24], no particular memory model is implemented.

Many efforts have been made to study the performance of task-based applications, whether with modeling NUMA accesses on large compute nodes [15, 22], or with accelerators [34]. Some studies have a similar approach to our work, whether in the technical sense, like using SimGrid's components for the simulation of parallel loops with various dynamic loop scheduling techniques [27], or in the modeling sense, such as simNUMA [26] on multicore machines (achieving around 30% precision error on LU factorization) or HLSMN [32] (without considering task dependencies). But to our knowledge, no currently available simulator allows the prediction of performance of task-based applications with data dependencies on NUMA architectures while taking into account both NUMA and cache locality effects.

The sOMP simulator [14] leverages the SimGrid framework to simulate the execution of task-based application with NUMA effects. It obtains good predictions for the Cholesky factorization, but relatively large tile sizes had to be used to mitigate cache effects which were not simulated. For the more complex QR factorization, and on AMD platforms, the simulations were also much less reliable. In this paper, we extend sOMP into taking into account the cache effects, but also refine the modeling of the platforms, thus strongly improving prediction precision. We also investigate the cost of the simulation and the potential opened for scheduling research.

3 Context, principles and implementation

This work targets the situation where, for instance, scheduling researchers want to improve the scheduling heuristics of a task-based runtime system for a given application executing on a given platform. Experimenting with heuristics in real executions on the platform however meets various concerns. The measured execution times are subject to potential system noise coming from running software, thermal conditions of the room, etc. The measurements may not be reproducible due to unexpected software or firmware upgrades on the platform, that can strongly change the computation efficiency, 10%-20% variation is not uncommon. The access to the platform may also itself be limited by CPU.hour quotas, corresponding to the high energy costs of running native measurements.

It is thus highly desirable to be able to experiment with heuristics in a *simulated* environment, which can provide perfect reproducibility of the obtained results, and can be run by researchers at will on any commodity platform. The simulation however needs to accurately model the behavior of the platform, so as to confront the scheduling heuristics to the actual performance of the platform. In particular nowadays, on multicore systems the NUMA and L3 cache effects are especially relevant for scheduling heuristics, and thus must be accurately reflected in the simulation. This is why in this paper, we carefully model the NUMA architecture and introduce L3 cache simulation, and then verify that the obtained performance matches native execution. Other effects such as thermal constraints, DVFS, OS noise, etc. could also influence the performance, but are for now out of scope of this paper. Properly modeling NUMA and L3 cache effects will already provide ample reproducible experimentation material for scheduling research purpose.

3.1 Proposed profiling and simulation principle

The overall principle of our profiling and simulation experiments is as follows, given a task-based application to be run on a target platform:

- The platform characteristics are determined thanks to manufacturer documentation and benchmarking: L3 caches, NUMA nodes, and the bandwidths of the architectural links.

- The (unmodified) application is run on the platform with varying parameters to record its behavior in different situations (e.g., the tile size). In this paper, we also record the overall application execution makespan, which will serve as the reference time to be reproduced.
- During the application executions, an execution trace is recorded with, e.g., OpenMP's OMPT support. From this trace we extract the task graph and tasks execution durations.
- With the recorded information, the execution can be simulated at a coarse grain: tasks are replaced by mere virtual time accounting, which makes the simulation very cheap. Each point in the simulated results of this paper is the result of such a *reproducible* simulation run.
- The scheduling researchers can then run such simulations and experiment with their heuristics at will in a reproducible way. They can for instance change their runtime task scheduler, their data placement, or even artificially alter the platform details (to e.g., check for the impact of the various hardware bandwidths on their scheduling heuristic)

In the following subsections we describe these steps in more details for the case of our experiments.

3.2 Target platforms

Our experiments were performed on two platforms:

- a dual-socket **Intel Xeon Gold 6240**: 36 cores, **CascadeLake** microarchitecture (AVX-512) with 2 NUMA nodes, each containing 18 cores and a 24.75 MB L3 cache;
- a dual-socket **AMD EPYC 7452**: 64 cores, **AMD Infinity** microarchitecture (zen-2) with a hierarchy of 16 NUMA nodes, each containing 4 cores and a 16 MB L3 cache.

These are thus largely different platforms: the Intel system, with only two caches and two NUMA nodes, exhibits quite limited locality effects; the AMD system, on the other hand, comprises many NUMA nodes and caches, with a complex interconnect.

We used the OpenBLAS 0.3.10 and the LLVM OpenMP runtime with the *close* thread binding on the *cores* places. We have set the frequency governors to the *powersave* mode for the used machines, to avoid the uneven behavior of the software and hardware governors.

The experiments conducted in this paper, unless specified otherwise, use a matrix side size of 12288 with double precision, i.e. 1GB of data and a total of around 5 000 tasks for the Cholesky case. This matrix size was chosen because the whole matrix cannot fit in the set of the L3 caches, and thus exhibits NUMA effects. The matrix is however also only a few times larger than the set of the L3

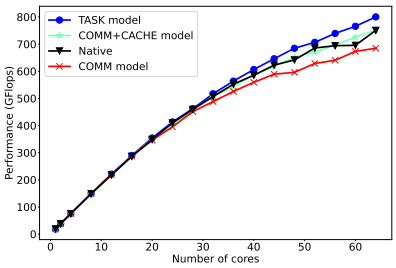


Figure 1: Cholesky performance on the AMD platform according to the number of cores used on the platform.

caches, and thus allows for significant data reuse and cache-to-cache transfers that improve performance, that our simulation will have to reproduce.

For the task tile size, we chose 512x512. This is small enough so that the working sets of the tasks fit in the L2+L3 caches, but do not fit in the L2 cache alone. This is the typical tuning that leads to benefiting the most from the L2 and the L3 caches.

The performance is measured against the number of cores used to execute the application. In order to closely observe the topological effects, the cores are taken in proximity order (the hwloc [6] *logical* order), so that executions on 1 to 18 cores will progressively fill only the first 18-core Intel socket (resp. 1 to 32 for the first 32-core AMD socket), thus observing the intra-socket effects only. Only executions on 19 to 36 cores will progressively execute more and more tasks on the second socket, thus observing the inter-socket effects (resp. 33 to 64 for the two 32-core AMD sockets). The obtained performance for a Cholesky factorization on the AMD platform is shown by the Native curve of Figure 1. We can notice that the performance increase is almost steady up to execution on 32 cores (the end of the first socket), but beyond 32 cores the increase flattens significantly more than in the first part. This is the performance behavior that this paper aims to reproduce in simulation, the "model" curves will be commented in Section 7.1.

3.3 Application case

The KASTORS [38] benchmark suite has been designed to evaluate the implementation of the OpenMP dependent task paradigm, introduced as part of the OpenMP 4.0 specifications. The experiments presented here are based on the PLASMA subset of the suite, which provides dense matrix factorization algorithms extracted from the PLASMA library [8], in double precision. We evaluate three of them.

The Cholesky factorization includes 4 types of tasks: $\theta(n)$ dpotrf, $\theta(n^2)$ dtrsm, $\theta(n^2)$ dsyrk, and $\theta(n^3)$ dgemm. It is thus mostly composed of dgemm tasks, which are very efficient and involve 3 matrix tiles. The algorithm also exhibits a fair amount of data reuse between tasks.

The QR factorization, on the other hand, includes 4 types of tasks: $\theta(n)$ dgeqrt, $\theta(n^2)$ dormqr, $\theta(n^2)$ dtsqrt, and $\theta(n^3)$ dtsmqr. It is thus mostly composed of dtsmqr tasks, which are significantly less efficient than the dgemm tasks, and involve 4 matrix tiles and 1 scratch tile. The algorithm also exhibits less data reuse between tasks, which thus tends to generate more cache evictions.

Lastly, the LU factorization (with pivoting) includes 4 types of tasks: $\theta(n)$ dgetrf, $\theta(n^2)$ dswptr, $\theta(n^3)$ dgemm, and $\theta(n^2)$ dlaswp. It is thus also mostly composed of dgemm tasks. The algorithm exhibits less data reuse between tasks, and the pivoting brings behavior variation.

3.4 SimGrid

The SimGrid[10] framework is originally intended for the simulation of distributed memory platforms, to allow the study of scheduling algorithms on heterogeneous platforms. We use it here to model shared-memory architectures, because their L3 caches and NUMA coherency mechanisms actually make them distributed systems, as discussed in Section 3.6.

It is important to note that SimGrid is not a cycle-accurate simulator: computations are interpreted as overall calculation quantities consuming time according to the performance of the machine (GFlop/s), and communications are overall quantities of data to be transferred according to the bandwidth (GB/s) / latency (ns) of the links getting crossed. Our goal is not to simulate the application cycle by cycle (which would be very costly). It is rather to simulate its overall behavior (thus much less costly) and still be able to observe accurately enough the encountered phenomena (NUMA and cache effects, contention, concurrency, ...).

3.5 sOMP

The sOMP simulator is geared towards task-based applications with data dependencies. As presented in the work of Daoudi & al. [14], this tool is used to predict the performance of these applications on architectures modeled in the SimGrid XML format, using the trace files generated during native execution.

After parsing the trace file, sOMP proceeds by inserting tasks in a submission queue (FIFO) handled by a scheduler. It uses a centralized task queue, which is similar to the one performed by a typical OpenMP runtime [28]. In Section 7.3, we show how other scheduling policies can be tested to improve the application performances relating to that field. sOMP does not use the SimDAG (deprecated) and disk support of Simgrid since they do not allow to finely control data transfers and interactions on the memory bus.

In this paper, we improve that previous version of sOMP by adding support for simulating the L3 cache, and refining the platform modeling to improve the simulation accuracy, which will be discussed in the next sections.

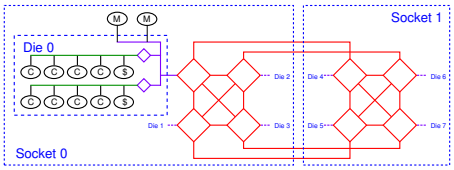


Figure 2: AMD platform model using SimGrid components (the details of only die 0 are shown, other dies are modeled identically)

3.6 NUMA architectures modeling

We see a NUMA architecture as a distributed machine in this work: several computation units are interconnected, forming a NUMA node. Depending on the machine, one or more NUMA nodes (also interconnected) form a socket that can be coupled to one or more other sockets, each having its own memory controller. The sockets are connected with UPI (for Intel) or Infinity Fabric (AMD) links. In the previous version of sOMP, the interconnection of NUMA nodes was modeled only with a single level of communication links. This is enough for the Intel platform, but the AMD platform, based on the zen2 microarchitecture, is much more complex: it comprises 2 sockets of 4 dies connected through an Infinity Fabric network. Each die contains two NUMA nodes, 2 caches, and 8 cores.

In this paper, we thus largely revamp the modeling. Figure 2 shows the proposed SimGrid modeling for the AMD platform. The die-to-die Infinity Fabric network is modeled as a network of routers (shown in red), according to the AMD documentation in terms of topology. Each die is then modeled as shown on the top left of the figure. A SimGrid backbone (shown in purple) represents the in-die Infinity Fabric interconnect between the die-to-die network, the RAM, and two sets of L3 cache + 4 CPU cores. Each set of 1 L3 cache + 4 cores, called CCX in the AMD documentation, embeds its own backbone (shown in green) at a higher speed than the in-die interconnect. This modeling follows the actual zen2 architecture quite closely. As we will see in the results, this level of details is necessary to properly take into account that the cores of a CCX have high-speed access to the corresponding L3 cache, slower access to the L3 cache of the other CCX of the same die, and yet slower access to the L3 caches of other dies. The die-to-die interconnect modeling also allows us to account for bandwidth contention properly.

Last but not least, an essential aspect of our modeling is the measurement of machine parameters, especially the bandwidths of the links between the modeled components. While some bandwidth values are provided directly by the documentation of the manufacturers, they are way too optimistic and not actually representative of the achievable bandwidth because, for instance, they do not take into account the overhead of the coherency protocol. Setting the bandwidths thus requires benchmarking. For this, we used the Intel Memory Latency Checker (MLC) v3.8, which provides the bandwidth of the memory con-

trollers and the inter-socket link. Lowering its buffer size allows us to keep data within the L3 cache, thus measuring the available bandwidth between cores and the shared L3 cache, i.e. the bandwidth of the intra-CCX interconnect (shown in green). This tool however does not allow to measure the bandwidth of the other links of the topology (the purple backbone and the various red links of the dies interconnection). We thus wrote a writer/reader micro-benchmark similar to likwid-bench [36], but that measures the bandwidth that can be achieved by transfers between L3 caches. It confirmed some of the values obtained with MLC and the documented topology of the interconnect links, but it additionally allowed us to directly measure the characteristics of the various links. Notably, SimGrid expresses them with three figures: the overall bandwidth of the link (called *shared*), the unilateral bandwidth of the link (called *splitduplex*), and the per-flow bandwidth of the link (called *fatpipe*). Our micro-benchmark is able to measure all three of them. We obtain the *fatpipe* bandwidth by measuring the bandwidth achieved by a single writer+reader pair. Using more and more writer+reader pairs, with all writers close to one L3 cache and all readers close to another L3 cache, the aggregated bandwidth eventually provides the *splitduplex* bandwidth. With the same approach but half of the writers close to one L3 cache and the other half close to the other L3 cache, and conversely for the readers, we obtain the *shared* bandwidth.

3.7 Methodology

To measure the accuracy of the simulations by comparing simulation time (T_{sim}) with real execution time (T_{native}), we do not consider the absolute values of the metric, but set a metric that defines the relative precision error of sOMP compared to native executions: $PrecisionError = (T_{native} - T_{sim})/T_{native}$. Therefore, when the precision error is positive, it means that we "under-simulate" the actual execution time, in other terms our prediction is optimistic. A negative precision error means that we "over-simulate", hence a pessimistic prediction. Curves closer to 0 are thus better in the precision error figures shown in this paper.

In the following sections, we present three simulation models to provide three levels of refinement: a task model in Section 4, a communication model in Section 5, and a communication+cache model in Section 6. We present the simulation accuracy results for the three models in Section 7.

4 Task model

4.1 Principle

The first approach is as proposed in the previous version of sOMP, which simulates only the task durations and not data transfers, meaning that it only uses SimGrid to replace the task computations with virtual clock accounting and

respects the task dependencies. It considers only non-preemptible tasks bound to CPU cores, which is the case with most task-based runtime systems.

For this simple model, the average durations of the different types of tasks are recorded when the application is executed on a single core. For instance, in the case of the Cholesky factorization on the Intel platform with a 512 tile size, these averages are 2.4 ms for dpotrf, 6.8 ms for dtrsm, 2.4 ms for dsyrk, 3.9 ms for dgemm. sOMP then uses these tasks durations to simulate parallel executions, called *TASK model* in this paper.

4.2 Discussion

Of course, such modeling for parallel executions is quite rough: the tasks' durations get longer when the application is executed on several cores, the main reason being data locality: with more and more cores involved, data exchanges are required between the sockets, which hit at some point the limitation of the socket-socket bandwidth.

That is why the previous version of sOMP proposed a second model which simulates the NUMA data locality effects, which we extend in the next section. This model requires to separate out, in task execution simulation, the computation part from the communication part, so as to be able to *replay* the former (like the *TASK model* does), and to *simulate* the latter. Simulating communications indeed allows sOMP to take into account the varying locality effects produced by the different strategies tried by scheduling researchers, as will be illustrated in Section 7.3. That is why, in the second approach, for the computation part sOMP still uses average tasks durations of executions on a single core only. That indeed replays only the cost of computations without any data locality effects. The communication simulation is then added on top of this.

5 Communication model

The second approach, as proposed in the previous version of sOMP, extends the *TASK model* by modeling the NUMA data communications induced by data dependencies between tasks. They are simply translated into SimGrid transfers over the modeling of the platform, thus modeling the NUMA effects. In this section, we describe how in this paper we revamp that approach, to take into account communication overlap with computation.

5.1 Data transfers modeling

In the previous version of sOMP, the data communication cost was trivially added to the computation cost. A task executing on a CPU core is however essentially executing arithmetic instructions interleaved with memory instructions (typically load/store instructions). Depending on the quality of the implementation and the CPU cores' behavior, the memory instructions' latency may be overlapped by arithmetic instructions or not. This means that over the whole

duration $T(t_i)$ of task t_i, the time to perform the memory instructions of the task (denoted $T_M(t_i)$) is more or less overlapped with the time to perform the arithmetic part of the task (denoted $T_C(t_i)$). Put formally,

$$max(T_C(t_i), T_M(t_i)) \leq T(t_i) \leq T_C(t_i) + T_M(t_i) \quad (1)$$

In our dense linear algebra application cases, tasks are composed of a single call to a BLAS operation. In the case of the dgemm task, the gemm BLAS matrix-matrix multiplication is usually very carefully designed to get ample overlap. For other types of tasks and notably the QR factorization tasks, this is much less true.

Therefore, we determine the amount of overlap, i.e., the amount of computing time covered by communications, through experimentation. After testing multiple values, we introduce an overlap ratio in our simulator and set the amount of overlapped computing time to 60% in the case of the Cholesky factorization, 4% for QR, and 10% for LU. For the Cholesky case for instance, this means for a given task that if the communication time is smaller than 60% of the computation time, it is considered as wholly overlapped by the computation. Otherwise, we add the surplus to the computation time. These values can also be obtained using performance counters, but this is outside the scope of this paper.

The memory instructions that are accounted for in $T_M(t_i)$ are those which read or write the task input and output operand or scratch buffer (we ignore accesses to the local scalar or vector variables, which fit in the L1 cache and are thus already accounted for in $T_C(t_i)$). To make simulation times tractable, we group these instructions by the task operands, i.e., matrix tiles or tiled scratch buffer. This grouping allows matching with SimGrid's programming model, which is oriented towards distributed memory platforms: we model the task memory accesses as data transfers for the task operands, i.e., as SimGrid *communications* between the CPU core and the RAM, one per operand.

Since with dense linear algebra, application tasks usually access the content of all operands in an interleaved pattern, we make these communications concurrent by access mode, i.e., all read-type operations are concurrent, and all write-type operations are also concurrent. However, communications of different access modes are made sequential, since a task usually reads its data, performs the computations, and then writes the result back to memory. $T_M(t_i)$ can thus be written as:

$$T_M(t_i) = \max_{j=1}^{n} T_{CommR}(a_{i,j}) + \max_{j=1}^{n} T_{CommW}(a_{i,j}) \quad (2)$$

where n is the number of memory accesses, $a_{i,j}$ is the j-th operand of task t_i and $T_{CommR}(a_{i,j})$ (resp. $T_{CommW}(a_{i,j})$) the time to read (resp. write) $a_{i,j}$ depending on its NUMA location and the core performing task t_i.

As a result, the set of tasks executing at the same time on the different cores induce a corresponding set of communications that progress concurrently on the platform. SimGrid can then determine, at each timestep of the simulation,

the bandwidth sharing between the communications [11], and thus account for contention on the simulated links.

5.2 Discussion

With the communication model, the quality of the simulations is improved by taking into account data NUMA locality. The different data flows between the architecture components impact the application's execution time and depend on the architecture's parameters and the crossed links (discussed in the Section 3.6). By therefore taking into account the NUMA locality of the data and modeling the transfers with communications, one creates contention and concurrency effects that influence the simulated times.

However, during the application's real execution, the data locality is not static, cache effects come into play in addition to the NUMA effects. When a task executes on a given CPU core, the matrix tiles used by the task remain in the corresponding L3 cache. If another task that executes on the same core needs the same matrix tiles, the core can fetch the tiles from the L3 cache instead of the NUMA node RAM, making the transfer much faster, and saving interconnect bandwidth. Since the communication model of the previous version of sOMP does not take these effects into account, the simulation will be pessimistic. Indeed, it always considers that the data has to be fetched from the NUMA node RAM, even when the data is still available in the L3 cache, thus simulating a higher cost in terms of transfer times.

To remedy this problem, in this paper we improve the communication model by introducing a caching mechanism to take into account the effects of data reuse between tasks.

6 Communication+cache model

We now extend the communication model into a new communication+cache model, which tracks in which L3 caches one can fetch copies of matrix tiles efficiently, and which models the communications between the RAM, the L3 caches, and the CPU cores.

6.1 Implementation of L3 caches

To benefit most from caches, the dense algebra tile size is usually chosen so that the datasets of tasks fit in the L3 cache but not in the L2 cache (as is the case in our experiments), we will thus model only the L3 caches and not the L2 caches. Modeling the L2 caches would significantly increase simulation times for a not actually better precision error, since L2 caches are not shared between CPUs cores (as is most often the case), and thus do not exhibit locality behavior that we would have to model.

In the implementation of the L3 caches, we consider the actual size of the cache in the target architecture. We also consider atomically the whole size of

a given tile. The cache is therefore in the form of slots, with the number of available slots being $CacheSize/TileSize$. When data is inserted in the cache, we implement an LRU-type behavior to evict existing data. This is a simple approximation of the actual associativity of caches by ignoring e.g. conflict misses. In addition, during the execution of a task, the data associated with the task is locked in the cache. This accounts that tasks usually need the tile data during their whole execution. As seen in the results in the next section, this model with a low simulation cost is still accurate enough for dense linear algebra kernels.

6.2 Cache transfers

The previous version of sOMP recorded the locality of the matrix tiles in NUMA node RAM according to the allocation tasks. In this section, we additionally track the copies of tiles in the L3 caches of the platform. The notion of locality is thus now more complex: the tiles needed to execute a task on a CPU can be fetched from the local L3 cache, from a remote L3 cache, from the local or remote NUMA node. In all but the first case, the tile needs to be transferred from the remote location to the local L3 cache before the CPU core can load the data from the local L3 cache and then execute computation.

Therefore, we model this with a *communication* between the remote cache or RAM and the local cache, and another between the local cache and the core.

If a subsequent task, executed on a CPU core next to the same L3 cache, needs the same tile, only a transfer between the local cache and the core will be triggered (provided that the tile has not been evicted from the cache in the meantime). This makes it possible to decongest inter-socket links, thus modeling the actual behavior of the application on the real platform.

When a task modifies a matrix tile, we remove the tile from all other L3 caches, so that the corresponding CPU cores will have to reload it if they execute tasks that need the new value of the tile.

When a modified matrix tile needs to be evicted from an L3 cache, its content has to be transferred back to its corresponding NUMA node RAM. Therefore, we perform a *communication* from the L3 cache to the RAM where the initialization task allocated it initially.

In the case of the QR factorization, one of the tile operands of the tasks is a *scratch* workspace, which is stored on the stack. This means that this workspace is actually stored per CPU core and keeps getting reused by the tasks running on the same core. We model this with one matrix tile per CPU core, that tasks only write to. As a result, we properly model that the L3 caches store the workspaces of their corresponding CPU cores.

6.3 Discussion

While the model of Section 5 used a pessimistic data locality, i.e., it considers that all the data are remote, the communication+cache model refines the locality of data. This makes it possible to improve simulations by modeling as

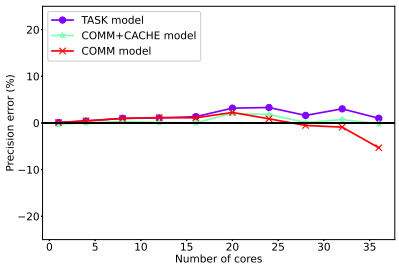


Figure 3: Precision error of Cholesky simulations on the Intel platform.

closely as possible the occupancy of L3 caches and the data transfers between L3 caches and RAM, which will improve the accuracy in predicting the application behavior. As mentioned in Section 6.1, since the sizes of tiles fit in the L3 cache and not in the L2 cache, we choose to model only the first because the second will exhibit little cache sharing effects. A compromise is made here between the precision of the simulation and its cost. The proposed modeling is well-suited to tiled dense linear algebra; sparse linear algebra, for instance, would however require much more involved modeling of the behavior of tasks.

7 Results

We first present the simulation precision results with the metric discussed in Section 3.7, then we show that the time to run simulations is largely shorter than the real execution time of the application. Finally, we show the importance of the simulator for studying various scheduling policies with a cache-aware scheduling example.

7.1 Precision results

The obtained results for the Cholesky case on the Intel platform are shown in Figure 3. As presented in previous sections, the task model takes into account only the task computation time. We observe that this model is less precise beyond 18 cores, which is the number of cores on the first NUMA domain (also a socket). Beyond that, the task model is too optimistic and encounters around +3% precision error, which was expected since data locality and transfers are not considered with this model.

However, we can see that the communication model improves the simulations beyond 18 cores since memory latencies become more and more critical due to platform contention. The communication model starts taking this into account, thus avoiding the task model's optimism. It however ends up being too pessimistic when a large amount of cores is used in the machine, which was again expected since this model does not take into account data reuse in caches.

Finally, the communication+cache model achieves the best overall precision. Thanks to taking into account data movements inside the L3 cache, it shows

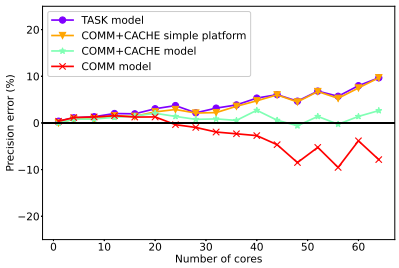


Figure 4: Precision error of Cholesky simulations on the AMD platform.

less than 1% average precision error and is consistently precise for all numbers of used cores. It is important to note that according to the task model's performance, the Intel platform does not display many NUMA-related effects: the task model's precision is already very good, especially on the first socket where we average a 0.8% precision error. This is understandable since this machine has only a single NUMA domain per socket, with all 18 cores inside a socket sharing the same L3 cache. This configuration will obviously not result in many data transfers inside a single socket, compared to the AMD machine. For conciseness we will thus present further results only for the AMD machine.

Figure 4 shows the results obtained for the Cholesky factorization on the AMD EPYC 7452, which comprises a total of 16 NUMA nodes and 16 L3 caches. In this case, the task model is no longer accurate compared to the results on Intel. On the first socket alone, we average around +3% precision error, and we reach +10% when all the machine cores are used. The amount of data transfers (not simulated with the task model) is indeed more important here since we have multiple NUMA nodes, and not taking them into account costs us precision.

The communication model is not providing accurate results either. Since we have more data transfers between NUMA nodes on this machine, the communication model is more pessimistic even before reaching 32 cores (the number of cores on the first socket), and inter-socket communications further accentuate the pessimism of the simulation, down to -10%.

Undoubtedly, the communication+cache model is the most reliable. Once again, this model remains consistent regardless of the number of used cores: modeling the reuse of data provides better accuracy (less than 2% accuracy error on average).

Figure 1 at the beginning of this paper additionally provides absolute numbers in GFlop/s for the comparison of these results.

Figure 4 also shows the performance of the communication+cache model when using a simple platform model. For this case, we modeled the AMD platform the same trivial way that was used by the previous sOMP version. This means assuming that for each socket, all memory controllers, L3 caches, and cores of the four dies are directly connected through an Infinity Fabric, whose speed was set to the speed provided by the Intel Memory Latency Checker.

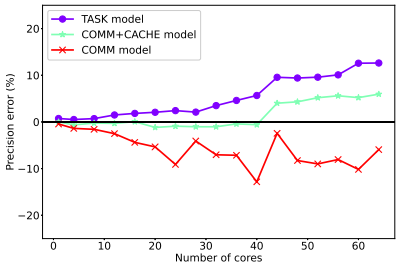


Figure 5: Precision error of QR simulations on the AMD platform.

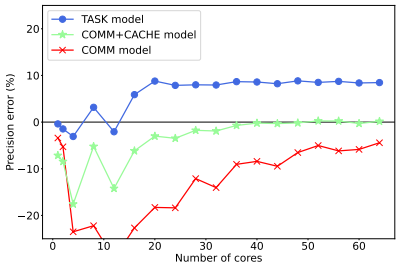


Figure 6: Precision error of LU simulations on the AMD platform.

In other terms, we are in that case ignoring the hierarchical topology of the machine and instead just merely modeling its components. The result is the "COMM+CACHE simple platform" curve of Figure 4. As expected, it is much more optimistic than the communication model since it does not consider the contention of the die-to-die network. This shows that it is critical for accuracy to properly model the network of the L3 cache interconnection.

Figure 5 shows the results obtained for the QR factorization. The task model is optimistic as it regularly diverges, starting from the execution on 9 cores until the execution on 64 cores, and the communication model is again very pessimistic. On the other hand, the communication+cache model seems to avoid getting too optimistic, and catches the L3 caching effects. However, it does not seem to avoid the divergence that the task model is affected by for the executions on the second socket, starting from the executions on 40 cores. This loss of precision can be explained by the effects of bandwidth variation on the applications kernels: as we are using more cores, contention on the machine links reduces the available bandwidth, therefore changing the amount of computational time that can overlap the slowed data transfers, and resulting on a slower application execution time. This makes our communication+cache model optimistic when many machine cores are used, as observed in Figure 5, which is the scenario where those effects have the most influence on the application's execution time. Considering these effects would require refining the execution model of a task inside SimGrid itself, to subtly entangle execution time and memory transfers, which is beyond the scope of this paper.

Model \ Matrix size	12288	16384	20480	24576
TASK	9.2%	7.5%	6.5%	5.1%
COMM	-8.4%	-12.4%	-6.9%	-7.2%
COMM+CACHE	**3.1%**	**0.1%**	**1.1%**	**1.4%**

Table 1: Precision error of Cholesky simulations on the AMD platform for various matrix sizes - Number of cores = 64

Model \ Matrix size	12288	16384	20480	24576
TASK	11.8%	10.7%	9.4%	9%
COMM	-7%	-15.9%	-17.5%	-21.8%
COMM+CACHE	**5.4%**	**5.2%**	**3.6%**	**3.9%**

Table 2: Precision error of QR simulations on the AMD platform for various matrix sizes - Number of cores = 64

The results for the LU factorization are shown in figure 6. The precision error of the different models varies significantly in the case of executions with few cores. This is actually the native measurements which have slightly chaotic performance behavior. Indeed, the LU factorization uses pivoting, which brings erratic behavior depending on the content of the matrix data, which this paper does not aim to simulate. With more cores, such misbehavior in native execution flattens out, and the communication+cache model quickly properly reproduces the overall computation/communication behavior of the factorization, while the task model remains too optimistic and the communication model remains too pessimistic.

Overall, we have shown that our models are able to achieve good precision for various linear algebra algorithms. The results have so far been presented for matrix size 12288×12288, but even with exactly the same simulation parameters (platform model, task average duration, overlap factor) other matrix sizes exhibit the same kind of results. We have summarized results for larger matrix sizes in Tables 1, 2, and 3, which show the corresponding precision errors when simulating applications using all the machine cores. We observe that our

Model \ Matrix size	12288	16384	20480	24576
TASK	8.53%	8.22%	8.51%	8.9%
COMM	-4.31%	-9.11%	-16.1%	-22.34%
COMM+CACHE	**0.27%**	**2.04%**	**1.91%**	**1.21%**

Table 3: Precision error of LU simulations on the AMD platform for various matrix sizes - Number of cores = 64

simulator remains reliable despite the increase in matrix size, i.e. despite an increase in the number of tasks and data transfers. The communication+cache model stays accurate, averaging around 1,4% precision error across the presented matrix sizes for the Cholesky factorization, 4.5% for QR, and 1.4% in the LU case.

To summarize, we are getting good results on the Intel platform which is a simple architecture not showing ample NUMA effects, but also good results on the AMD platform for the Cholesky, QR and LU cases, despite its very complex architecture.

7.2 Simulation time

In terms of performance, the simulator typically takes 1 s on one core of a commodity laptop to simulate one execution of the Cholesky factorization for matrix side size 16 384, and tile size 512, i.e. around 6 000 tasks. Simulations can additionally be trivially run in parallel for the different points of the figures shown in the paper. The real execution on the AMD platform, on the other hand, requires around 75 s to complete the same factorization on one core, and around 1.6 s on 64 cores. Therefore, the time to run a simulation is way smaller than the execution time of the real application. This is because SimGrid uses *coarse* simulation and not cycle-accurate simulation: all the actual computations of a task are replaced by a single simulation step, and all the actual read/write operations for a given task operand are replaced by a single simulated communication. Furthermore, the simulation time grows only linearly with the number of tasks, and grows only linearly with the number of cores for the communication and communication+cache models (due to the increased number of communications that SimGrid has to manage concurrently). This remains reasonable, the reduced precision error is usually worth the simulation time increase.

7.3 Use case: experimenting with cache-aware schedulers

The previous subsections have shown that the communication+cache model provides accurate simulated execution times that consider both NUMA and cache effects. Previous simulation work on simulating GPU-based platforms [34] had opened the door for scheduling research on such platforms that is both realistic and reproducible [1]. The simulation results shown in this paper now similarly open up realistic and reproducible scheduling research that aims at optimizing cache affinity.

We have implemented an initial proof-of-concept cache-aware OpenMP task scheduler. When a CPU core terminates a task, most default task schedulers (and notably the LLVM scheduler used here) pick up the next task without real consideration for locality [28]. Our cache-aware scheduler, however, privileges picking a task whose data operands are already available in the L3 cache of the CPU core, thus reducing L3 cache misses and thus reducing overall data transfers over the platform.

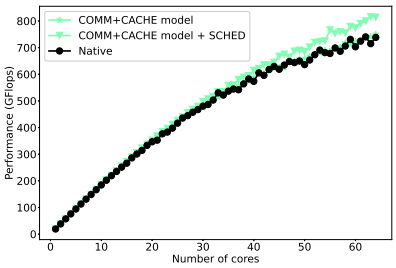


Figure 7: Simulated performance with a cache-aware scheduler on AMD platform.

The results are shown in Figure 7, in terms of GFlop/s according to the number of cores used for execution. The communication+cache model accurately matches the native measurements, as was initially presented in Figure 1. When we make the simulator use the refined scheduler, however, we notice a performance improvement that increases with the number of cores used for execution (up to 9.6%). This shows that the heuristic does help with scalability over a large multicore system. Only the communication+cache model is able to show this effect in simulation, since it is the cache effects which bring this performance improvement.

It should be noted that the current implementation of this scheduler is very simplistic: at each step, it scans all tasks that are ready for execution, to find one that minimizes L3 cache misses. This typically yields to an $O(ntasks^2)$ complexity. This is not a concern with our simulation since the application simulated performance is not affected by the time taken by the scheduler. However, the implementation cannot be integrated into an actual OpenMP environment yet because that cost is prohibitive; more algorithmic work is needed to design an implementation with a lower complexity. This is actually where simulation benefits lie: we were able to quickly prototype a proof-of-concept scheduler and experiment with it and observe the obtained gains in the simulation without being affected by the cost of the scheduler itself. Since the latter can be measured separately, this allows to clearly get the balance between the scheduler cost and the provided gains. We can refine heuristics and try to improve the performance gains, without caring about implementation complexity, as a first prototyping step. Simulation also provides the performance results much faster than real executions, and without all the complexity of reserving the target platform and suffering from real-life platform concerns which hinder result reproducibility. It is even possible to reproducibly investigate scheduling bugs. Thanks to a trace of the simulated execution one can set breakpoints at the exact time when apparently a wrong decision is made, and inspect the state of the scheduler. Once a heuristic that provides solid gains is devised, we will spend time on producing an implementation with an acceptable complexity, for actual use in a real OpenMP runtime. Similarly, data NUMA placement heuristics could be experimented with thanks to our reproducible environment, before spending time on a

real-life implementation. Simulation also permits trying to artificially alter the simulated hardware platform and observe the obtained effects on the application performance.

8 Conclusion and future work

In this work, significant enhancements were presented to simulate the execution of parallel task-based applications on shared-memory architectures. We discussed 3 different models to tackle this objective. We have show the limitations of the task model which simulates execution time without considering data transfers. We have improved the communication model to make it better take into account computation/communication overlap and NUMA effects. Finally, the communication+cache model enhances the previous model with a cache mechanism to take into account the movement of data and access to cached data. We have shown that, coupled with a more precise modeling of the target architecture, this last model entails better precision. We also showed that accurate modeling of the components of a machine and its hierarchy is essential to achieve more reliable precision.

Diving in the documentation to determine the socket topology is in general quite error-prone and provides largely optimistic values, so we made measurements to collect bandwidth and latency values for the targeted platforms. We plan to design a tool that performs combinations of data transfers to automatically determine the topology of the platform and its bandwidths, similarly to the automatic network discovery that was proposed in the context of the SimGrid framework [17].

The amount of overlapping between arithmetic instructions and memory instructions would also need to be characterized, we plan to use performance counters to observe in-situ the behavior of kernels to refine the overlapping ratios. This will require to rework SimGrid's task execution model to be able to tune the interactions between computation and communication.

We will then try to generalize our evaluation work to more applications, and notably more memory-bound applications, and on more platforms.

Since in the current state of our simulator, the real behavior of the target platform is reproduced accurately enough, this opens the path for reproducible experimentation with task-based runtime systems and schedulers. CPU-based platforms are indeed susceptible to various technical conditions such as software stack, platform availability, etc., making them painful to experiment with daily. Our work allows us to perform daily experiments in simulation with an acceptable level of realism confidence and then confirm results in reality, similarly to previous work with GPUs [1]. It even allows performing experiments on non-existing platforms to observe the behavior of heuristics in extreme situations.

In this work, we have so far only aimed for schedulers that optimize for cache affinity. With more and more cores getting assembled tightly in CPU sockets, thermal effects however have an impact on performance through frequency throttling. Schedulers then have to manage a "thermal budget" to optimize ex-

ecution. Experimenting with them on real platforms is however subject to a lot of variability of the frequency governors. We thus plan to model these thermal effects in the simulator and their effects on the application's execution time, so as to provide a reproducible experimentation testbed for thermal effects.

References

[1] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Are Static Schedules so Bad ? A Case Study on Cholesky Factorization. In *International Parallel & Distributed Processing Symposium, IPDPS'16*, 2016.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[3] Rocco Aversa, Beniamino Di Martino, Massimiliano Rak, Salvatore Venticinque, and Umberto Villano. Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Computing*, 31(10), 2005. OpenMP.

[4] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th Euro-Par Conference*, Delft, The Netherlands, August 2009.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[6] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia, February 2010.

[7] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Productive cluster programming with OmpSs. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par '11, 2011.

[8] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Lapack working note 191: A class of parallel tiled linear algebra algorithms for multicore architectures, 2007.

[9] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1), 2011.

[10] H. Casanova. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437, 2001.

[11] H. Casanova. Modeling large-scale platforms for the analysis and the simulation of scheduling strategies. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 170–, April 2004.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Notices*, 40(10):519–538, October 2005.

[13] Paweł Czarnul, Jarosław Kuchta, Mariusz Matuszek, Jerzy Proficz, Paweł Rościszewski, Michał Wójcik, and Julian Szymański. Merpsys: An environment for simulation of parallel application execution on large scale hpc systems. *Simulation Modelling Practice and Theory*, 77:124 – 140, 2017.

[14] Idriss Daoudi, Philippe Virouleau, Thierry Gautier, Samuel Thibault, and Olivier Aumage. sOMP: Simulating OpenMP Task-Based Applications with NUMA Effects. In *IWOMP 2020 - 16th International Workshop on OpenMP*, September 2020.

[15] N. Denoyelle, B. Goglin, A. Ilic, E. Jeannot, and L. Sousa. Modeling non-uniform memory access on large compute nodes with the cache-aware roofline model. *IEEE Transactions on Parallel and Distributed Systems*, 30(6), 2019.

[16] Christian Engelmann. Scaling to a million cores and beyond: Using lightweight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems*, 30:59 – 65, 2014.

[17] Lionel Eyraud-Dubois, Arnaud Legrand, Martin Quinson, and Frédéric Vivien. A First Step Towards Automatically Building Network Representations. In *13th International Euro-Par Conference - Euro-Par 2007*, Rennes, France, 2008.

[18] F. Galilee, G.G.H. Cavalheiro, J.-L. Roch, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Parallel Architectures and Compilation Techniques*, oct 1998.

[19] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, 2007.

[20] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS)*. IEEE, 2013.

[21] Sergi Girona and Jesús Labarta. Sensitivity of performance prediction of message passing programs. *The Journal of Supercomputing*, 17, 2000.

[22] B. Haugen. *Performance analysis and modeling of task-based runtimes*. PhD thesis, 2016.

[23] B. Haugen, J. Kurzak, A. YarKhan, P. Luszczek, and J. Dongarra. Parallel simulation of superscalar scheduling. In *2014 43rd International Conference on Parallel Processing*, pages 121–130, 2014.

[24] Franz Heinrich. *Modeling, Prediction and Optimization of Energy Consumption of MPI Applications using SimGrid*. Theses, Université Grenoble Alpes, May 2019.

[25] Khan Samee Ullah Kliazovich Dzmitry, Bouvry Pascal. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 2012.

[26] Y. Liu, Y. Zhu, X. Li, Z. Ni, T. Liu, Y. Chen, and J. Wu. SimNUMA: Simulating NUMA-Architecture Multiprocessor Systems Efficiently. In *2013 International Conference on Parallel and Distributed Systems*, Dec 2013.

[27] Ali Mohammed, Ahmed Eleliemy, Florina M Ciorba, Franziska Kasielke, and Ioana Banicescu. Experimental verification and analysis of dynamic loop scheduling in scientific applications. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2018.

[28] Ananya Muddukrishna, Peter A Jonsson, and Mats Brorsson. Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors. *Scientific Programming*, 2015, 2015.

[29] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl.*, 26(2), May 2012.

[30] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *International Symposium on Performance Analysis of Systems and Software*, 2011.

[31] Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, and Felix Wolf. Isoefficiency in practice: Configuring and understanding the performance of task-based applications. *SIGPLAN Notices*, 52(8), 2017.

[32] Mohammed Slimane and Larbi Sekhri. HLSMN: High Level Multicore NUMA Simulator. *Electrotehnica, Electronica, Automatica*, 65(3), 2017.

[33] Luka Stanisic, Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Arnaud Legrand, Florent Lopez, and Brice Videau. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. In *The 21st IEEE International Conference on Parallel and Distributed Systems*, Melbourne, Australia, December 2015.

[34] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090, 2015.

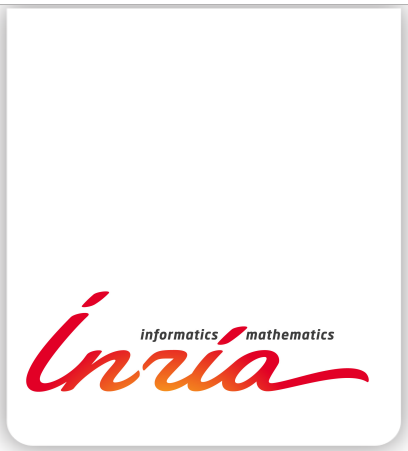
[35] Jie Tao, Martin Schulz, and Wolfgang Karl. Simulation as a tool for optimizing memory accesses on NUMA machines. *Performance Evaluation*, 60(1), 2005.

[36] Jan Treibig, Georg Hager, and Gerhard Wellein. likwid-bench: An extensible microbenchmarking platform for x86 multicore compute nodes. In *Tools for High Performance Computing 2011*, pages 27–36, 2012.

[37] Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello. Using Data Dependencies to Improve Task-Based Scheduling Strategies on NUMA Architectures. In *European Conference on Parallel Processing*, 2016.

[38] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In *International Workshop on OpenMP*. Springer, 2014.

[39] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 78. IEEE, 2004.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399