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

Idriss Daoudi, Samuel Thibault, Thierry Gautier

▶ To cite this version:

Idriss Daoudi, Samuel Thibault, Thierry Gautier. Draft: sOMP: NUMA and cache-aware simulations for task-based applications. [Research Report] RR-9400, Inria. 2021, pp.23. hal-03177026v1

HAL Id: hal-03177026
https://inria.hal.science/hal-03177026v1

Submitted on 22 Mar 2021 (v1), last revised 29 Apr 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

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

Idriss Daoudi, Samuel Thibault, Thierry Gautier

Project-Teams STORM - AVALON

Research Report n° 9400 — March 2021 — 23 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 propose a simulator for dependency-based task-parallel applications, that enables experimenting with multiple data locality models. We also introduce two locality-aware performance models: a lightweight communication-oriented model that uses topology information to weight data transfers, and a more complex communications and cache model that takes into account data storage in the LLC. We validate both models on 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 la capacité à prendre en compte les effets NUMA (Non-Uniform Memory Access), même si pratiquement toutes les plates-formes HPC présentent aujourd'hui de tels effets. Nous proposons ainsi 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 prenant en compte la localité: un modèle léger orienté communications qui utilise les informations de topologie pour pondérer les transferts de données, et 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 et montrons qu'en moyenne, notre simulateur prédit de manière reproductible le temps d'exécution avec une petite erreur relative.

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

1 Introduction

Simulation tools are of significant interest in the field of application and runtime system development. They allow, among other things, to understand if an 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.

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 [19], to run on heterogeneous architectures with accelerators [4, 2, 21] or distributed memory systems [20, 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 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 of task-based OpenMP programs have been proposed so as to deal with NUMA effects [30, 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 complex, with many NUMA nodes inside a socket such as within AMD EPYC processors, it is necessary to understand the impact of architectural decisions on the performance of applications. With the capacity of simulating (OpenMP) task-based applications on shared-memory architectures, it will be possible to design and implement the right runtime for the right hardware, thanks to a robust methodology based on reliable simulations.

Some works [34] simulated dependent tasks on shared-memory architectures with GPUs, but also showed that accurately predicting the performance of the current complex platforms will strongly require taking NUMA effects into account, and none of the available tools meets this growing need for NUMA awareness of dependent tasks simulation.

In our previous work [14], we proposed a preliminary simulator, based on SimGrid [10], to predict the performance of a task-based application on a shared-memory architecture. We modeled the NUMA structure of the platform and studied the impact of data locality on execution times. However, our initial architecture model was too simple to capture complex NUMA architectures, thus making the prediction on some applications less relevant. In this paper, we extend our previous work:

- We model multiple NUMA architectures including a complex AMD EPYC memory and cache hierarchy;
- We refine the task execution simulation to take into account overlapping between communication and computation;
- We extend our simulator with a new model using an L3 cache mechanism, that strongly improves simulation accuracy;
- We study the cost of the simulation;
- We show that we can easily experiment a proof-of-concept cache-aware scheduling algorithm with sOMP using the refined models.

After presenting the state of the art, we recap the simulation principles. We then describe a model that does not take data locality into account, and a model that takes into account NUMA communications. We then introduce a refined communication+cache model that improves the simulation accuracy. We eventually present the 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 [40], xSim [17], the trace-driven Dimemas tool [22], or MERPSYS [13] for performance and energy consumption simulations. Some others are oriented towards cloud simulation like CloudSim [9] or GreenCloud [26].

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 [31, 33, 24, 35]. All these studies present approaches with reliable precision, but, as with Simany [25], 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 [16, 23], 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 [28], or in the modeling sense, such as simNUMA [27] on multicore machines (achieving around 30% precision error on LU algorithm) 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 data locality effects.

In our previous work [14], we therefore tackled this goal. We succeeded in obtaining good predictions for the Cholesky algorithm while using relatively large tile sizes to mitigate cache effects. However, for the more complex QR algorithm, the simulations were much less reliable, which required more investigation into the additional effects impacting execution times, leading us to here extend this work and take into account the cache effects, but also refine our modeling of the platforms.

3 Context, principles and implementation

This work targets the situation where, for instance, a scheduling researcher wants 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. The access to the platform may also itself be limited by CPU.hour quotas.

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 reflected in the simulation.

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 through benchmarking [6, 36]: hierarchy of cores, 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 reference time to be reproduced.
- During the application executions, an execution trace is recorded with, e.g., OpenMP's OMPT support.
- As explained in our previous work, we extract the task graph and tasks execution durations from the trace.
- 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 runtime task scheduler can be changed, or the platform details can be tuned (to e.g., check for the impact of the socket-socket bandwidth) and simulations can be re-run again, etc., in a reproducible way.

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:

- 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;
- 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

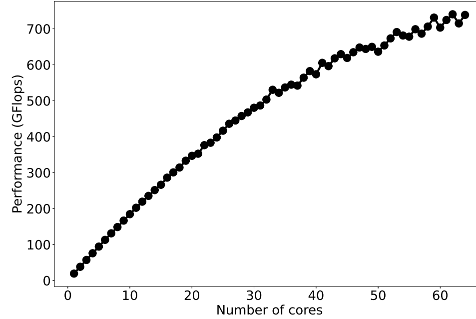

Figure 1: Cholesky performance on the AMD platform according to the number of cores used on the platform.

set of the L3 caches, and thus allows for significant data reuse and cache-to-cache transfers that improve performance.

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 on 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.

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 matrix factorization algorithms extracted from the PLASMA library [8], in double precision.

The Cholesky algorithm 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 algorithm, 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 algorithm (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 TiKKi

The libKOMP [39] OpenMP runtime embeds a trace and monitoring tool, based on the work of de Kergommeaux et al. [15]. The tool, called TiKKi, was developed using the initial OMPT API from OpenMP, allowing developers to instrument tools with standard trace-based methodologies.

As described in our previous work, we improved TiKKi to capture all events required to construct the program's task graph, and record them to a file. It also enriches the recording with performance information. For instance, task attributes may contain locality information [39], and hardware performance counters may be registered, in addition to time, within specific events (task creation, task termination...). Hence, TiKKi can generate several output forms of execution traces: the task graph as a `.dot` file, a Gantt chart as an R script, or a specific file format for the simulations performed by sOMP.

3.5 SimGrid

The SimGrid[10] framework allows the study of scheduling algorithms on heterogeneous platforms. It is a tool that provides basic functionalities for the simulation of heterogeneous distributed applications in distributed environments. This framework's specific objective is to facilitate research in programming parallel applications on distributed computing platforms, from a simple network interconnecting workstations, to computing grids.

The operating principle is as follows: an *actor*, i.e., an independent stream of execution in a distributed application, can perform several *activities*, such as computations or communications, on a *host*, representing some physical resource with computing and networking capabilities. Several *actors* can communicate by sending a message on *mailboxes* linked to *hosts*, representing a meeting point for network communications.

From a platform description point of view, SimGrid provides essential elements for a detailed description of each of the elements of a distributed system, such as computing *hosts*, *clusters*, *disks*, *links*, *routes*, etc..., but also the routing of the platform, i.e., the path taken between two hosts. These elements have arguments that allow configuration and tuning of the platform in order to simulate different scenarios, which will be discussed in section 3.8. The SimGrid network model used here is the LV08 default model tuned with TCP gamma set to -1, the bandwidth factor set to 1, and the latency factor and weight-S set to 0. These disable all the TCP-specific behaviors of SimGrid.

SimGrid itself is not parallelized and thus cannot benefit by itself from running on a multicore system. One can however run in parallel a series of separate simulations.

It is also 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 (GFlops), and communications are overall quantities of data to be transferred according to the bandwidth (GBps) / 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 effects, contention, concurrency, ...).

Although SimGrid is originally intended for the simulation of distributed memory 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 3.8.

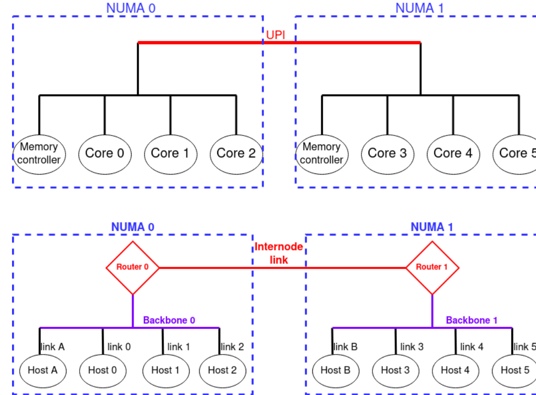

Figure 2: Intel platform NUMA outline (top) and model using SimGrid components (bottom).

3.6 sOMP

Our simulator named sOMP is geared towards parallel task-based applications with data dependencies. As presented in our previous work, this tool is used to predict the performance of these applications on architectures modeled in the SimGrid XML format, using the trace files generated by TiKKi (other tools can be used, the trace format is independent of the simulator).

After parsing the trace file, we proceed by inserting tasks in a submission queue (FIFO) handled by a scheduler. We use a centralized task queue for now, which is similar to the one performed by a typical OpenMP runtime [29]. In section 7.3, we show how other scheduling policies can be tested to improve the application performances relating to that field. We do not use the SimDAG (deprecated) and disk support of Simgrid since they do not allow us to finely control data transfers and interactions on the memory bus.

In this paper, we improved the previous version of sOMP by taking into account OpenMP process binding policies (close, spread), but also by adding a cache mechanism and refining the platform modeling to improve the simulation accuracy, which will be discussed in the next sections.

3.7 Methodology

To measure the reliability 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.

3.8 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. This overview of the platform is represented at the top of Figure 2 for the Intel

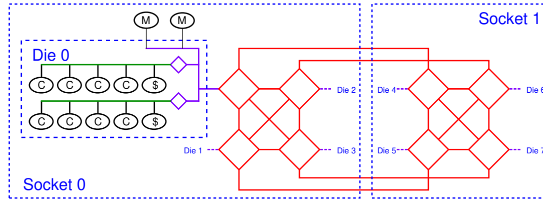

Figure 3: AMD platform model using SimGrid components (the details of only die 0 are shown, other dies are modeled identically)

machine. We express this platform in SimGrid as represented at the bottom of Figure 2. The cores and the memory controllers are modeled as SimGrid hosts. The intra-socket interconnect is modeled as a SimGrid backbone, while the inter-socket UPI link is modeled as a SimGrid link between routers.

Based on the zen2 microarchitecture, the AMD platform is much more complex than the Intel platform: it comprises 2 sockets of 4 dies connected through an Infinity Fabric network. Each die contains two NUMA nodes, 2 caches, and 8 cores.

The modeling used for the Intel platform (shown on Figure 2), that we used in our previous work, thus had to be extended for this paper. Figure 3 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 Infinity Fabric 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 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 controllers 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. We thus wrote a simple reader-writer microbenchmark that measures the bandwidth between L3 caches, so as to confirm the preciously-obtained values, and measure more precisely the concurrency on the interconnect links (shown in purple and red). This allowed us to directly measure the different parameters used by SimGrid to characterize a communication link: 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*).

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 communica-

tion+cache model in section 6. We present the results for the three models in section 7.

4 Task model

4.1 Principle

As a first approach proposed in our previous work [14], we simulate only the task durations and not data transfers, meaning that we only use SimGrid to replace the task computations with virtual clock accounting and implement the tasks synchronizations according to task dependencies.

For this simple model, we record the average durations of the different types of tasks 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. We then use 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. For example, on the Intel platform, for executions up to 18 cores, data remains on the NUMA node and the L3 cache of the first socket. However, executions using more than 18 cores require data exchanges between the two sockets, which hit at some point the limitation of the socket-socket bandwidth. Tasks then get slowed down depending on whether their data is available in the local NUMA node or the local L3 cache.

In the next sections, we simulate the data locality effects because they will open up highly interesting experimentation. Previous work on simulating GPU-based platforms [34] had indeed opened the door for scheduling research on such platforms that is both realistic and reproducible [1]. Simulating data locality effects will similarly enable reproducible realistic scheduling experimentation with locality-aware schedulers. To achieve this, we however need 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 will indeed allow us to take into account the locality effects of the varying scheduling decisions, as will be illustrated in section 7.3.

Simulating data locality effects also enables tuning the simulated hardware platform and observing the obtained effects on the application performance. For instance, this can be used for hardware provisioning to determine which platform bandwidth parameters are high enough for the targeted application instead of remaining tied to the measured platform's parameters. It is even possible to invent yet-nonexistent platforms by injecting in the simulator several interconnected sockets and trying different bandwidth parameters.

That is why, in the next sections, for the computation part we will keep measuring average tasks durations for executions on a single core only. That will indeed catch only the cost of computations without any data locality effects. The communication simulation will then be added on top of this, as described in the next section.

5 Communication model

As an extension of the *TASK model*, we here model the NUMA data communications induced by data dependencies between tasks. With the platform architecture modeling in SimGrid as

a distributed platform, we can simulate the data transfers. This part was partly described previously [14], we here very briefly recap the principle, and extend it to take into account communication overlap with computation.

5.1 Data transfers modeling

A task executing on a CPU core is 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. 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 algorithm, 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 2D 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 2D scratch buffer. This grouping allows matching with SimGrid's programming model, which is oriented towards distributed memory platforms: we model the task memory accesses with data transfers for the task operands, i.e., as SimGrid *communications* between the CPU core and the RAM, one per operand.

Since 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.

We then let SimGrid account for contention of the various communications on the simulated links. According to the principles described above, the set of tasks executing at the same time

on the different cores induce a set of communications that progress concurrently on the platform. SimGrid can then determine, all along the simulation, the bandwidth sharing between the communications [11].

5.2 Implementation of data locality

To simulate these task memory operations, we also need to know the location of the data. Therefore, when performing a run with sOMP, the simulator records the NUMA location of the data allocation tasks (thus using the first-touch memory policy, as commonly used by systems). The simulator thus records, for each matrix tile, on which NUMA node RAM it was allocated. During the execution of the other tasks, R/W-type *communications* are initiated between the NUMA node RAM where the data of the task is located (recorded according to the allocation tasks) and the task execution core, in order to model the data transfers between the different components of the real architecture. This impacts the simulated times according to the bandwidth and the latency of the crossed links, thus modeling the NUMA effects.

5.3 Discussion

With the communication model, the quality of the simulations is improved by taking into account data 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 3.8). By therefore taking into account the 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 is not static. 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 described in this section does not take these effects into account, the simulation will be pessimistic. Indeed, data which is actually local will here always be considered remote, thus simulating a higher cost in terms of transfer times.

To remedy this problem, we improve our 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 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. 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.

6.2 Cache transfers

In section 5, we 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, then 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 meanwhile). 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. 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., considering 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 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 used 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.

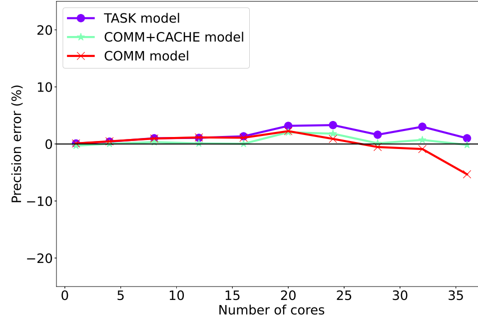

Figure 4: Precision error of Cholesky simulations on the Intel platform.

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 are shown in Figure 4. 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 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.

Figure 5 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.

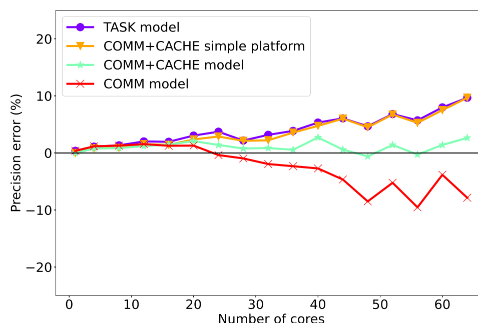

Figure 5: Precision error of Cholesky simulations on the AMD platform.

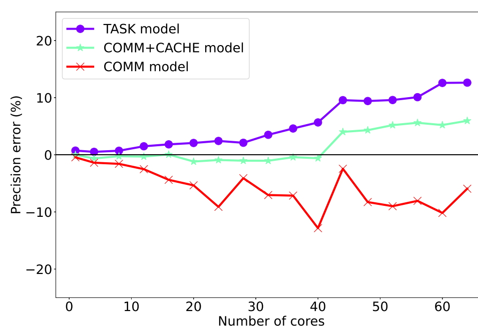

Figure 6: Precision error of QR simulations on the AMD platform.

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 5 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 we did for Intel. 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. 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 5. 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 6 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

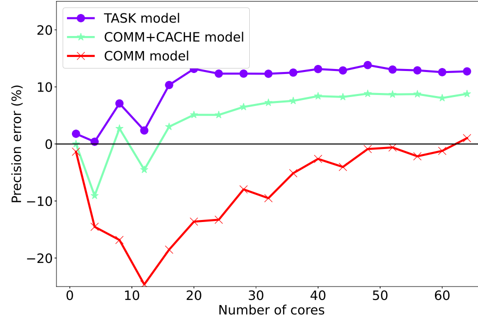

Figure 7: Precision error of LU simulations on the AMD platform.

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

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 6, 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.

The results for the LU algorithm are shown in figure 7. This time, the task model's precision error is increasing from the start of the first socket. While the communication+cache model follows the same trend and corrects its precision error with data transfers and the cache mechanism, the communication model starts very pessimistically. This behavior can be explained by the bandwidth variations discussed in the previous paragraph and the algorithm's nature. Here, we are using the LU algorithm with pivoting, which impacts execution time as kernels will go faster or slower depending on the complexity the pivoting, with more or fewer pivot elements memory transfers. Even if these effects are not taken into account in our simulator, we manage to achieve good overall precision with the communication+cache model (less than 10%). The communication model happens to coincidentally end up being accurate at a full machine core number as its pessimistic data locality transfers compensate for the other unmodeled effects.

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

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

Model \ Matrix size	12288	16384	20480	24576
TASK	12.7%	15.8%	16.5%	17.6%
COMM	1.1%	-3.1%	-7.5%	-11.1%
COMM+CACHE	**9.3%**	**10.9%**	**11.3%**	**12.4%**

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

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 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 algorithm, 4.5% for QR, and 10.9% in the LU case. These values confirm the reliability of the simulator and its ability to predict fine-grain applications' performance.

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 and QR cases despite its very complex architecture. The LU case will probably require better modeling of the kernels' behavior.

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 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 a Cholesky factorization of such a matrix 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, even if the simulator has not been optimized yet. This is because our approach 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.

Figure 8 shows that the simulation time grows linearly with the number of tasks. Simulation thus remains reasonable to use even for large cases.

Figure 9 shows that the simulation time also grows according to the number of cores for the communication and communication+cache models. This is due to the increased number of communications that SimGrid has to manage concurrently. The increase however remains reasonable, the reduced precision error is usually worth the simulation time increase.

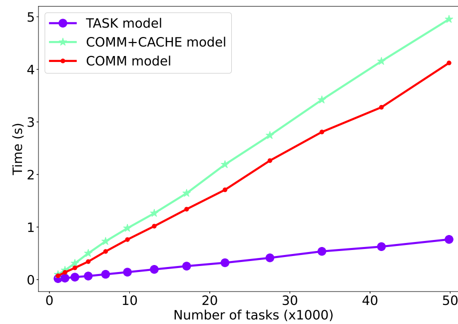

Figure 8: Simulation time for different matrix sizes (represented in terms of number of tasks).

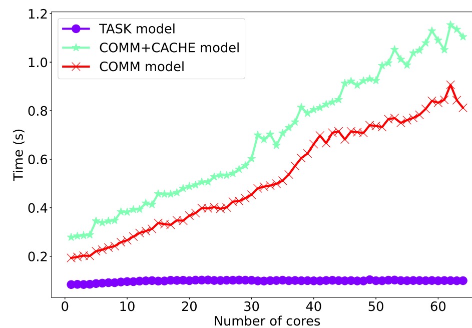

Figure 9: Simulation time for a single matrix size (16384×16384) on various numbers of cores

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 [29]. 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.

The results are shown in Figure 10, in terms of GFlop/s according to the number of cores used for execution. The communication+cache model accurately matches the native measurements that were 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

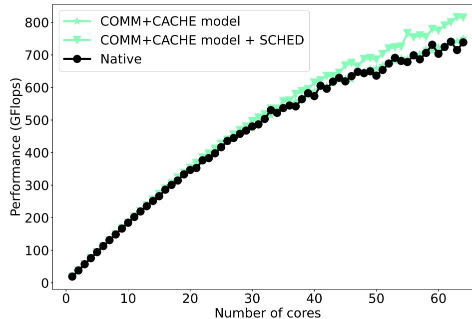

Figure 10: Simulated performance with a cache-aware scheduler on AMD platform.

multicore system.

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 where the application simulated performance is not affected by the time taken by the scheduler. However, the implementation can not be integrated into an actual OpenMP environment as it is now because that cost is prohibitive; more algorithmic work is needed to design an implementation with a lower complexity. This is indeed where our simulator benefits lie: we were able to quickly prototype a proof-of-concept scheduler and experiment with it quickly with our simulator and observe the obtained gains in the simulation. We can then refine heuristics and try to improve the performance gains, without caring about implementation complexity, as a first prototyping step. We also benefit from the fact that simulation 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. Once a heuristic that provides interesting gains is devised, we will spend time on producing an implementation with an acceptable complexity, for actual use in a real OpenMP runtime.

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 presented 3 different models to tackle this objective. The task model, which allows execution times to be simulated without considering data transfers, has shown its limitations. We then incremented the task model with the communication model, which makes it possible to take into account the effects of data locality -induced by the dependencies between tasks- by considering that all our accesses are towards the main memory. This approach is very pessimistic.

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 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 [18].

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 conditions such as temperature, turbo boost strategies, software stack, 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 execution. 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.

9 Acknowledgments

This work was funded by the Inria Project Lab HAC-SPECIS (see `http://hacspecis.gforge.inria.fr/`).

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed (`https://www.plafrim.fr/`), supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine.

References

[1] Agullo, E., Beaumont, O., Eyraud-Dubois, L., and Kumar, S. Are Static Schedules so Bad ? A Case Study on Cholesky Factorization. In *International Parallel & Distributed Processing Symposium, IPDPS'16.*

[2] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23* (Feb. 2011), 187–198.

[3] Aversa, R., Di Martino, B., Rak, M., Venticinque, S., and Villano, U. Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Computing 31*, 10 (2005). OpenMP.

[4] Ayguadé, E., Badia, R. M., Igual, F. D., Labarta, J., Mayo, R., and Quintana-Ortí, E. S. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th Euro-Par Conference* (Delft, The Netherlands, Aug. 2009).

[5] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput. 37*, 1 (1996), 55–69.

[6] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)* (Pisa, Italia, Feb. 2010), pp. 180–186.

[7] Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R. M., Ayguadé, E., and Labarta, J. Productive cluster programming with OmpSs. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I* (Berlin, Heidelberg, 2011), Euro-Par '11.

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

[9] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. 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] Casanova, H. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid* (2001), pp. 430–437.

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

[12] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not. 40*, 10 (Oct. 2005), 519–538.

[13] Czarnul, P., Kuchta, J., Matuszek, M., Proficz, J., Rościszewski, P., Wójcik, M., and Szymański, J. Merpsys: An environment for simulation of parallel application execution on large scale hpc systems. *Simulation Modelling Practice and Theory 77* (2017), 124 – 140.

[14] Daoudi, I., Virouleau, P., Gautier, T., Thibault, S., and Aumage, O. sOMP: Simulating OpenMP Task-Based Applications with NUMA Effects. In *IWOMP 2020 - 16th International Workshop on OpenMP* (Austin / Virtual, United States, Sept. 2020), vol. 12295 of *LNCS*, Springer.

[15] de Kergommeaux, J. C., Guilloud, C., and de Oliveira Stein, B. Flexible performance debugging of parallel and distributed applications. In *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference,* (2003), vol. 2790 of *Lecture Notes in Computer Science*, Springer.

[16] Denoyelle, N., Goglin, B., Ilic, A., Jeannot, E., and Sousa, L. 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), 1374–1389.

[17] Engelmann, C. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems 30* (2014), 59 – 65. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers.

[18] Eyraud-Dubois, L., Legrand, A., Quinson, M., and Vivien, F. A First Step Towards Automatically Building Network Representations. In *13th International Euro-Par Conference - Euro-Par 2007* (Rennes, France).

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

[20] Gautier, T., Besseron, X., and Pigeon, L. 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* (2007), PASCO '07.

[21] Gautier, T., Lima, J. V., Maillard, N., and Raffin, B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS)* (2013), IEEE.

[22] Girona, S., and Labarta, J. Sensitivity of performance prediction of message passing programs. *The Journal of Supercomputing 17* (2000).

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

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

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

[26] Kliazovich Dzmitry, Bouvry Pascal, K. S. U. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing* (2012).

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

[28] Mohammed, A., Eleliemy, A., Ciorba, F. M., Kasielke, F., and Banicescu, I. Experimental verification and analysis of dynamic loop scheduling in scientific applications. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)* (2018), IEEE.

[29] Muddukrishna, A., Jonsson, P. A., and Brorsson, M. Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors. *Scientific Programming 2015* (2015).

[30] Olivier, S. L., Porterfield, A. K., Wheeler, K. B., Spiegel, M., and Prins, J. F. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl. 26*, 2 (May 2012).

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

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

[33] Stanisic, L., Agullo, E., Buttari, A., Guermouche, A., Legrand, A., Lopez, F., and Videau, B. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. In *The 21st IEEE International Conference on Parallel and Distributed Systems* (Melbourne, Australia, Dec. 2015).

[34] Stanisic, L., Thibault, S., Legrand, A., Videau, B., and Méhaut, J.-F. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurrency and Computation: Practice and Experience 27*, 16 (2015), 4075–4090.

[35] Tao, J., Schulz, M., and Karl, W. Simulation as a tool for optimizing memory accesses on NUMA machines. *Performance Evaluation 60*, 1-4 (2005), 31–50.

[36] Treibig, J., Hager, G., and Wellein, G. likwid-bench: An extensible microbenchmarking platform for x86 multicore compute nodes. In *Tools for High Performance Computing 2011* (2012), pp. 27–36.

[37] Virouleau, P., Broquedis, F., Gautier, T., and Rastello, F. Using Data Dependencies to Improve Task-Based Scheduling Strategies on NUMA Architectures. In *European Conference on Parallel Processing* (2016).

[38] Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., and Gautier, T. Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In *International Workshop on OpenMP* (2014), Springer, pp. 16–29.

[39] Virouleau, P., Roussel, A., Broquedis, F., Gautier, T., Rastello, F., and Gratien, J.-M. Description, implementation and evaluation of an affinity clause for task directives. In *International Workshop on OpenMP* (2016), Springer, pp. 61–73.

[40] Zheng, G., Kakulapati, G., and Kalé, L. V. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* (2004), IEEE, p. 78.

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