Improving Simulations of Task-Based Applications on Complex NUMA Architectures

Idriss Daoudi, Thierry Gautier, Samuel Thibault, Swann Perarnau

Improving simulations of task-based applications on complex NUMA architectures

Idriss Daoudi[1][0000−0003−2425−8359], Thierry Gautier[2], Samuel Thibault[3][0000−0001−6411−809X], and Swann Perarnau[1][0000−0002−1029−0684]

[1] Argonne National Laboratory
[2] INRIA Grenoble - LIP - ENS Lyon
[3] INRIA Bordeaux - Université de Bordeaux

Abstract. Modeling and simulation are crucial in high-performance computing (HPC), with numerous frameworks developed for distributed computing infrastructures and their applications. Despite node-level simulation of shared-memory systems and task-based parallel applications, existing works overlook non-uniform memory access (NUMA) effects, a critical characteristic of current HPC platforms.
In this work, we introduce a modeling for complex NUMA architectures and enhance a simulator for dependency-based task-parallel applications. This facilitates experiments with varied data locality models: we refine a communication-oriented model leveraging topology information for data transfers, and devise a more intricate model incorporating a cache mechanism for last-level cache data storage. Dense linear algebra test cases are used to validate both models, demonstrating that our simulator reliably predicts execution time with minimal relative error.

Keywords: NUMA architectures · Modeling · OpenMP tasks · Simulation

1 Introduction

Task-based runtimes, originating with Cilk in 1998 [4], have evolved to accommodate shared-memory machines [16], heterogeneous architectures with accelerators [2, 3, 18], and distributed-memory systems [2, 6, 11, 17]. The 2008 OpenMP standard version 3.0 integrated the independent task model, expanding in 2013 to incorporate dependent task model and accelerator targeting.

To achieve optimal performance on shared-memory machines, schedulers need to tackle non-uniform memory access (NUMA) effects [26, 32]. Despite these advancements, technical constraints often hinder reproducibility of results on such platforms. Some studies [30] addressed these issues through simulation, enabling realistic and reproducible scheduling research [1] and facilitating quick prototyping before implementing on real systems.

For (OpenMP) task-based applications, a high-quality simulation of shared-memory could lead to efficient runtimes, benefiting from a robust, reproducible methodology. Prior studies [30] emphasized the need for NUMA-aware simulators that consider cache effects for accurate performance prediction.

Our earlier work [13] introduced a preliminary simulator, named sOMP, using SimGrid [9] to predict the performance of task-based applications on shared-memory architectures by modeling a simple NUMA structure and data locality impacts. This tool allowed the non-cycle-accurate simulation of task-based applications using a trace of their sequential execution obtained with OMPT. Nevertheless, it neglected cache effects and failed to model complex NUMA architectures, diminishing the relevance of some application predictions. This paper expands on that work in several ways:

- We model **complex and more intricate** NUMA and cache architectures;
- We refine the task execution simulation to take into account **overlapping between *communications* and computations**.
- We introduce **L3 caching in the simulation**, which 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.

Following a review of existing literature, we delve into the principles of simulation and our modeling of NUMA architectures. We revisit a prior model ignoring data locality, extend another model considering NUMA locality, and introduce an enhanced model factoring in cache locality. Simulation accuracy across different application algorithms (Cholesky, QR and LU factorization algorithms), matrix sizes, and Intel and AMD platforms is then demonstrated. Lastly, we discuss the simulator's cost and its applications 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. Simulators like BigSim [34], xSim [15], Dimemas [19], MERPSYS [12], CloudSim [8], and GreenCloud [23] predict performance across various applications and contexts. Some focus on specific architectures, like hybrid MPI/OpenMP applications and task-based simulations on multicore processors [21, 27–29, 31]. While these offer precision, no particular memory model is implemented, as in the case of Simany [22].

Efforts have also been made to study task-based applications performance, involving modeling NUMA access on large compute nodes [14, 20] and accelerators [30]. Some studies, like SimGrid [25] and simNUMA [24], align with our work technically or in modeling, but none currently predict performance of task-based applications with data dependencies on NUMA architectures considering both NUMA and cache locality effects.

Our previous work [13] presented the sOMP simulator, which leverages the SimGrid framework to simulate task-based application execution with NUMA effects. Despite good predictions for Cholesky factorization, the project had several limitations:

- NUMA architectures modeling was very simplistic;

– data transfers cost was trivially added the computations cost;
– cache effects were ignored.

Therefore, several trade-offs were made. For example, large tile sizes were needed to compensate for unsimulated cache effects. For the more complex QR factorization and on AMD platforms, the simulations were unreliable. This paper extends sOMP to incorporate cache effects and refines platform modeling and data transfers, greatly improving prediction accuracy and exploring potential scheduling research opportunities.

3 Context and principles

This study addresses scenarios where scheduling researchers aim to optimize task-based runtime system scheduling heuristics for specific applications and platforms. However, real execution experiments are subject to system noise, non-reproducibility due to software or firmware upgrades, and limited platform access due to high energy costs.

To overcome these challenges, it's desirable to experiment with heuristics in a simulated environment offering **perfect reproducibility** and flexibility to run on any platform. This simulation needs to accurately model the platform behavior to align the scheduling heuristics with the platform's actual performance. In multicore systems, NUMA and L3 cache effects are crucial for scheduling heuristics and need to be accurately simulated. This paper focuses on meticulously modeling the NUMA architecture, incorporating L3 cache simulation, and verifying the performance aligns with native execution. Other performance influencers like thermal constraints, dynamic voltage and frequency scaling, and OS noise, though relevant, are beyond this paper's scope.

To implement these principles, we relied on our previously developed tool sOMP, which will be improved in this work. sOMP was built using SimGrid, a powerful non-cycle-accurate simulator.

SimGrid Initially designed for simulating distributed-memory platforms to study heterogeneous platforms scheduling algorithms [9], is employed here for shared-memory architectures. The latter's L3 caches and NUMA coherency mechanisms essentially render them distributed systems, which will be discussed in Section 4. SimGrid isn't a cycle-accurate simulator. Computations are interpreted as overall calculation quantities, consuming time relative to machine performance (GFlop/s), and communications as data quantities transferred based on bandwidth (GB/s) and latency (ns). Instead of a costly cycle-by-cycle simulation, our aim is a less expensive overall behavior simulation, while still accurately observing phenomena like NUMA and cache effects, contention, and concurrency.

sOMP The simulator, presented in our previous work [13] and tailored for task-based applications with data dependencies, predicts their performance on architectures modeled in the SimGrid XML format using native execution-generated trace files. After parsing these files, sOMP submits tasks to a queue managed by

a scheduler, akin to a standard OpenMP runtime. We enhance sOMP by introducing L3 cache simulation support and refining platform profiling for increased accuracy.

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

1. Platform specifics such as L3 caches, NUMA nodes, and architectural link bandwidths are discerned through manufacturer documentation and benchmarking. The platform modelization is expressed in an XML file. This step in presented in Section 4;
2. Unmodified OpenMP applications are executed sequentially (on a single core) on the platform with varied parameters (e.g., the tile size) to observe behavior under different conditions, recording the overall application execution makespan as a reference;
3. During these executions, an execution trace is generated using OpenMP's OMPT support, from which we extract the task graph and task execution duration;
4. Using the collected information, a parallel simulation of the execution is performed, substituting tasks with virtual time accounting for cost-effective simulation, and combining it with models that take into account NUMA and cache effects. This step is presented in Section 5.

With these simulations, scheduling researchers can reliably experiment with their heuristics, modifying task schedulers, data placement, or platform specifics to investigate the effects on their scheduling heuristic. In the following sections, we describe some of these steps in more detail for our experiments.

4 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 (Intel) or Infinity Fabric (AMD) links.

An overview of NUMA architectures was presented in our previous work [13] for an Intel platform: specifically, an Intel Xeon Gold 6240 with 36 cores (Cascade-Lake microarchitecture) and two NUMA nodes, 18 cores each. The latter is represented with SimGrid components: the cores and the memory controllers are modeled as SimGrid hosts, the intrasocket interconnect is modeled as a SimGrid *backbone*, and the intersocket UPI link is modeled as a SimGrid link between routers.

While this model proves adequate for the relatively straightforward architecture of this particular Intel processor, it is not applicable to more intricate

processors that exhibit a hierarchically structured and interconnected arrangement of components. Such complexities cannot be overlooked due to their direct bearing on the accuracy of the simulation.

4.1 Modeling complex NUMA architectures

In the first contribution of this work, we consider a second more intricate processor: a dual-socket AMD EPYC 7452 with 64 cores (AMD Infinity microarchitecture) and 16 NUMA nodes, four cores each.

This architecture, based on the *zen 2* microarchitecture, is more complex than the Intel platform, featuring two sockets with four dies connected by an Infinity Fabric network. Each die contains two NUMA nodes, therefore, two caches and eight cores.

Hence, our previous Intel platform model [13] was insufficient, prompting extension for this study. Figure 1 shows our proposed SimGrid model for the AMD platform. The Infinity Fabric network is modeled as a router network based on AMD documentation. Each die, as shown on the top left, comprises a SimGrid backbone symbolizing the in-die Infinity Fabric interconnect linking the die-to-die network, RAM, and two L3 cache + 4 CPU core sets. Each L3 cache + 4 core set, known as a CCX, embeds a faster backbone than the in-die Infinity Fabric interconnect. This closely follows the actual *zen 2* architecture and is necessary to accurately reflect the varying access speeds of cores to different L3 caches and account for bandwidth contention in the die-to-die interconnect.

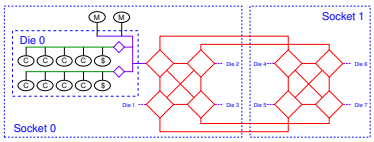

Fig. 1. AMD platform model using SimGrid components (the details of only die 0 are shown; other dies are modeled identically).

4.2 Bandwidth measurements

The critical aspect of our modeling approach hinges upon the precise measurement of machine parameters, specifically the link bandwidths interconnecting the components under study. Our preceding work [13] primarily relied on bandwidth values as stipulated by the manufacturers. However, these prescribed values exhibited limitations due to their failure to incorporate the overhead associated with coherency protocols. Furthermore, they were not wholly indicative of the attainable bandwidth, thus underscoring the need for rigorous benchmarking. To this end, we employed the Intel Memory Latency Checker v3.8 for the task of quantifying the bandwidths associated with the memory controller and inter-socket links. By manipulating the buffer size of this tool, we were able to retain data in the L3 cache. This adjustment, in turn, facilitated the measurement of

the available bandwidth connecting the cores and the shared L3 cache, thereby enabling us to determine the intra-CCX interconnect bandwidth.

However, this tool doesn't measure other topology link bandwidths, leading us to create a simple reader-writer microbenchmark for measuring bandwidth between L3 caches. This allowed us to directly measure parameters used by SimGrid to characterize a communication link: overall link bandwidth (*shared*), unilateral link bandwidth (*splitduplex*), and per-flow link bandwidth ($fatpipe$). We obtained $fatpipe$ bandwidth through a single writer+reader pair, while aggregated bandwidth from multiple pairs provided *splitduplex* bandwidth. Similarly, with writers and readers spread across two L3 caches, we determined the *shared* bandwidth.

5 NUMA and cache effects modeling

This section presents three simulation models to provide three levels of refinement: a *TASK* model presented in Section 5.1, a NUMA model in Section 5.2, and a NUMA and cache model in Section 5.3. We present the results for the three models in Section 6.

5.1 Recap: the TASK model

The initial sOMP approach presented in our previous work [13] simulates only task durations, not data transfers, using SimGrid for virtual clock accounting and task dependency adherence. It assumes non-preemptible tasks tied to CPU cores, typical in task-based runtime systems. This model, referred to as the *TASK* model, records average task durations from single-core application executions. These durations are then used to simulate parallel executions.

Discussion Parallel execution modeling is inherently complex as task durations lengthen with more cores due to data locality. Data exchanges between sockets increase, eventually hitting bandwidth limitations. Thus, the previous sOMP version proposed a model simulating NUMA data locality effects. This model separates task execution into computation and communication components. The computation part replays the average task durations from single-core executions (as the *TASK* model does), thereby excluding data locality effects. The communication simulation, added subsequently, enables sOMP to account for varying locality effects across different scheduling strategies.

5.2 NUMA effects modeling: the COMM model

The previous version of sOMP [13] extended the *TASK* model to account for NUMA data communications induced by task dependencies using SimGrid transfers over the platform model, thus accounting for NUMA effects. In this subsection, we outline the second contribution of this work: this approach is improved to include communications overlap with computations.

Data transfers modeling In the prior sOMP version [13], data communication cost was simply added to computation cost. Yet, a task on a CPU core often executes arithmetic and memory instructions (like load/store) interleaved. Depending on the implementation and CPU behavior, memory instruction latency may be overlapped by arithmetic instructions. Hence, over the duration $T(t_i)$ of task t_i, the time for memory instructions $T_M(t_i)$ is more or less overlapped with the arithmetic part of the task $T_C(t_i)$. Formally, we have:

$$max(T_C(t_i), T_M(t_i)) \leq T(t_i) \leq T_C(t_i) + T_M(t_i). \tag{1}$$

Our dense linear algebra tasks consist of single calls to BLAS operations. Overlap between computation and communication is considerable for some tasks, like *dgemm*, but less so for others, like the QR factorization tasks. We establish overlap ratios through experimentation, selecting values that minimize precision error. As such, we set overlap ratios of 60% for Cholesky, 4% for QR, and 10% for LU. If communication time is less than the overlap ratio of computation time, it is entirely overlapped; if greater, the excess is added to computation time. Memory instructions considered in $T_M(t_i)$ are those handling task input and output operands or scratch buffers. To align with SimGrid's distributed-memory platform orientation, we model task memory accesses as data transfers for task operands, meaning matrix tiles or tiled scratch buffers, grouped for tractable simulation times. Task memory accesses are modeled as concurrent by access mode, with all read or write operations 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

$$T_M(t_i) = \max_{j=1}^{n} T_{commR}(a_{i,j}) + \max_{j=1}^{n} T_{commW}(a_{i,j}), \tag{2}$$

where n is the number of memory accesses, $a_{i,j}$ is the jth operand of task t_i, and $T_{commR}(a_{i,j}$ (resp. $T_{commW}(a_{i,j})$) is 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 induces 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 [10] and thus account for contention on the simulated links.

Discussion The *COMM* model enhances simulations by considering data NUMA locality. Data flows between architecture components, affected by architecture parameters and traversed links, impact execution time. Accounting for NUMA data locality and modeling transfers as communications creates contention and concurrency effects.

However, data locality isn't static during real execution. Cache effects come into play alongside NUMA effects. When a task on a CPU core uses matrix tiles, they remain in the corresponding L3 cache. If another task on the same core needs those tiles, they can be fetched from the cache instead of the NUMA

node RAM, saving time and bandwidth. The previous *COMM* model, not considering these effects, always simulates data fetching from the NUMA nodes' RAM, leading to a pessimistic simulation.

This issue is addressed by enhancing the *COMM* model with a caching mechanism to consider data reuse between tasks.

5.3 Cache effects modeling: the COMM+CACHE model

The third contribution of this work extends the *COMM* model into a new *COMM+CACHE* model, by tracking in which L3 caches one can fetch copies of matrix tiles efficiently and modeling the communications between the RAM, the L3 caches, and the CPU cores.

Implementation of L3 caches The tile size in dense algebra is often chosen to fit tasks' datasets into the L3 cache but not the L2 cache, maximizing cache utilization. We model only the L3 caches, as including L2 would significantly increase simulation times without improving precision, as they aren't shared between cores and don't exhibit notable locality behavior.

In our L3 cache implementation, we consider the actual cache size and tile size. The cache comprises slots, calculated by $\frac{CacheSize}{TileSize}$. Data insertion in the cache follows a least recently used (LRU) behavior for evictions, approximating actual cache associativity while ignoring aspects like conflict misses.

During task execution, the data associated with the task is locked in the cache. Despite its simplicity and low simulation cost, this model provides satisfactory accuracy for dense linear algebra kernels.

Cache transfers In the improved version of sOMP, we record not only the matrix tiles' NUMA node RAM locality but also their presence in the L3 cache. This complex notion of locality acknowledges that the required tiles can be fetched from various locations - local L3 cache, remote L3 cache, or local or remote NUMA node.

We model data transfers from remote cache or RAM to the local cache, and then from the local cache to the core, as communications. If the same tile is needed for a subsequent task executed on a neighboring CPU core, only a transfer from the local cache to the core is triggered, assuming the tile hasn't been evicted. This effectively models the decongestion of intersocket links, reflecting actual platform behavior.

If a task alters a matrix tile, we remove the tile from all other L3 caches. Subsequent tasks requiring the updated tile will have to reload it. Evicted modified tiles must be transferred back to their corresponding NUMA node RAM.

In the QR factorization, we model the reuse of per-core scratch workspaces with one matrix tile per CPU core that tasks only write to, thus accurately representing the L3 caches' storage of the corresponding CPU cores' workspaces.

Discussion The *COMM+CACHE* model refines data locality over the model from Section 5.2, which assumes all data to be remote. By simulating the occupancy of L3 caches and data transfers between L3 caches and RAM, we enhance the accuracy of predicting application behavior. As tile sizes fit in the L3 cache but not L2, we model only the former, striking a balance between simulation precision and cost. This modeling is suitable for tiled dense linear algebra, while sparse linear algebra would require a more intricate task behavior modeling.

6 Results

Our experiments were conducted on the processors mentioned in Section 4. The Intel system's limited locality effects contrast with AMD's complex interconnect of numerous NUMA nodes and caches.

OpenBLAS 0.3.10 and LLVM OpenMP runtime were used, with thread binding on the core places and frequency governors set to powersave mode, to avoid uneven behavior of the software and hardware governors. Experiments in this paper, unless specified otherwise, utilized a matrix size of 16384 x 16384 (double precision), namely, 1 GB of data, and approximately 6000 tasks for the Cholesky case. The matrix size, larger than the L3 caches but allowing for substantial data reuse, will therefore require accurate simulation of cache-to-cache transfers.

A task tile size of 512 x 512 was chosen, optimal for working sets fitting in L2+L3 caches but not L2 alone. Performance was evaluated against the number of cores used, selected in proximity order (the hwloc [5] logical order) to observe topological effects, first within individual sockets and subsequently across multiple sockets.

6.1 Application case

The KASTORS [33] benchmark suite, designed to evaluate the OpenMP dependent task paradigm from OpenMP 4.0 specifications, forms the basis for our experiments. We assess three dense matrix factorization algorithms from the suite's PLASMA [7] subset: Cholesky, QR, and LU factorizations.

The Cholesky factorization incorporates four task types:$\theta(n)$ dpotrf, $\theta(n^2)$ dtrsm, $\theta(n^2)$ dsyrk, and $\theta(n^3)$ dgemm. Predominantly comprising efficient dgemm tasks involving three matrix tiles, it exhibits substantial data reuse between tasks.

QR factorization involves $\theta(n)$ dgeqrt, $\theta(n^2)$ dormqr, $\theta(n^2)$ dtsqrt, and $\theta(n^3)$ dtsmqr tasks. Dominated by less efficient dtsmqr tasks involving four matrix tiles and one scratch tile, it presents less data reuse, leading to increased cache evictions.

Lastly, LU factorization (with pivoting) includes $\theta(n)$ dgetrf, $\theta(n^2)$ dswptr, $\theta(n^3)$ dgemm, and $\theta(n^2)$ dlaswp tasks. Predominantly composed of dgemm tasks, it exhibits less data reuse, with pivoting introducing variation in behavior.

6.2 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 one that defines the relative precision error of sOMP compared with native executions:

$$PrecisionError(\%) = \frac{(T_{native} - T_{sim})}{T_{native}}. \quad (3)$$

The following graphs depict the precision error of simulated versus native execution times for varying core counts, with polynomial regression curves (5th order) indicating trends. Positive precision errors denote optimistic "undersimulation" and negative errors, pessimistic "oversimulation", hence, curves nearer to 0 signify greater precision. We initially present simulation precision results, then demonstrate simulation's time efficiency compared to real application execution, and finally, illustrate the simulator's importance in assessing scheduling policies using a cache-aware scheduling instance.

6.3 Precision results

Figure 2 presents the Cholesky case results on the Intel platform. The *TASK* model only considers task computation time and its precision decreases beyond 18 cores, equivalent to the first NUMA domain (or socket). Beyond this, the *TASK* model exhibits approximately +3% precision error due to its disregard for data locality and transfers.

The *COMM* model improves simulations beyond 18 cores by accounting for memory latencies due to platform contention, but becomes too pessimistic with many cores, as it overlooks data reuse in caches. The *COMM+CACHE* model, considering L3 cache data movements, consistently achieves under 1% average precision error across all core counts.

The *TASK* model performs well on the Intel platform due to its single NUMA domain per socket, yielding a 0.8% average precision error, implying minimal NUMA-related effects. However, on the AMD EPYC 7452 as shown in Figure 3, with 16 NUMA nodes and 16 L3 caches, the *TASK* model's precision error increases to around +3% on the first socket and up to +10% when utilizing all cores. The increased data transfers due to multiple NUMA nodes on the AMD machine necessitate more accurate modeling.

The *COMM* model lacks accuracy due to underestimation of intersocket communications, leading to up to -10% error. The *COMM+CACHE* model consistently provides reliable results, with less than 2% error, highlighting the importance of accurately modeling data reuse and L3 cache interconnections. A simplified version of this model, named "*COMM+CACHE* simple platform", ignoring the machine's hierarchical topology, becomes overly optimistic due to neglecting die-to-die network contention.

The QR factorization results in Figure 4 show that the *TASK* model is overly optimistic, and the *COMM* model is pessimistic. The *COMM+CACHE* model

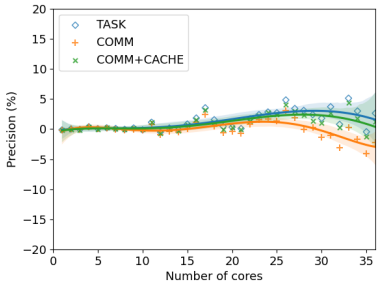

Fig. 2. Precision error of Cholesky simulations on the Intel platform.

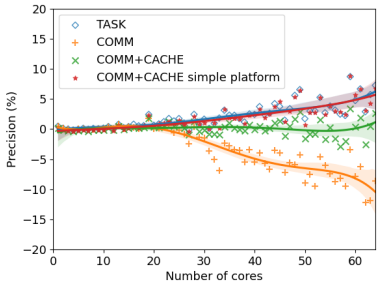

Fig. 3. Precision error of Cholesky simulations on the AMD platform.

accounts for L3 caching effects, but fails to consider bandwidth variation effects on the application kernels when many cores are used, making it optimistic in these scenarios. A more refined model **within SimGrid**, intertwining execution time and memory transfers, could address this but is beyond this paper's scope.

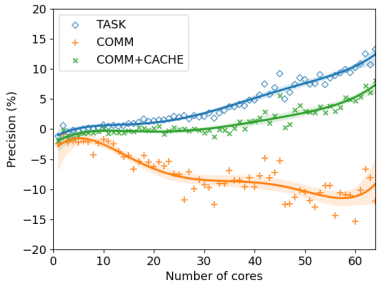

Fig. 4. Precision error of QR simulations on the AMD platform.

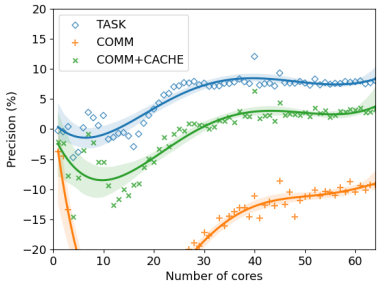

Fig. 5. Precision error of LU simulations on the AMD platform.

Figure 5 presents LU factorization results, where precision errors of models significantly vary for few-core executions due to native measurements' unpredictable performance, primarily due to pivoting in LU factorization. As core count increases, these inconsistencies diminish, and the *COMM+CACHE* model accurately simulates the computation/communication behavior. The *TASK* model remains overly optimistic, and the *COMM* model, overly pessimistic.

Our models have demonstrated consistent accuracy across various linear algebra algorithms and different matrix sizes. For matrix sizes of 12288 x 12288, 16384 x 16384, 20480 x 20480 and 24576 x 24576, which means an increase in task and data transfer volumes due to larger matrices, the *COMM+CACHE* model remains reliable with an average precision error of 1.4% for Cholesky, 4.5% for QR, and 1.4% for LU. These results validate our experimentally deter-

mined overlapping factors, confirming simulation accuracy regardless of problem size changes.

To summarize, we get good results on the Intel platform, which is a simple architecture not showing ample NUMA effects, but we also get good results on the AMD platform despite its complex architecture.

6.4 Simulation time

The simulator typically requires 1 second on a laptop core to simulate an execution for a 16384 x 16384 matrix with a 512 x 512 tile size (around 6,000 tasks) Cholesky factorization, while the actual execution on the AMD platform takes around 75 seconds on one core or 1.6 seconds on 64 cores. This is due to our use of coarse, not cycle-accurate, simulation, where all actual computations and read/write operations are replaced by single simulation steps.

Simulation time grows linearly with task number and with core count for the *COMM* and *COMM+CACHE* models due to increased concurrent communications. However, this increase is independent from the computations of the real execution and remains reasonable, and the reduced precision error is usually worth the simulation time increase.

6.5 Use case: experimenting with cache-aware schedulers

The preceding analysis confirms that the *COMM+CACHE* model offers precise simulated execution times accounting for both NUMA and cache effects, enabling realistic and reproducible scheduling research for optimizing cache affinity, similar to prior work on GPU-based platforms [30, 1]. We introduced a proof-of-concept cache-aware OpenMP task scheduler that prioritizes tasks with data operands already in the CPU core's L3 cache. This reduces L3 cache misses and overall data transfers. Performance results, displayed in Figure 6, reveal that the COMM+CACHE model closely simulates native executions. Utilizing the refined scheduler in the simulator exhibits a performance improvement that escalates with the number of cores used, demonstrating the heuristic's scalability benefits on a large multicore system. The *COMM+CACHE* model uniquely exhibits this effect in simulation due to the performance improvement resulting from cache effects. This experiment highlights the advantage of simulation. It allowed us to swiftly prototype a proof-of-concept scheduler, observe gains without the scheduler's cost impacting performance, and balance these costs and gains. This simulation-led prototyping approach enables refinement of heuristics, faster performance results, and reproducible investigation of scheduling bugs.

7 Conclusion and future work

In this study, we presented significant enhancements to simulate parallel task-based applications on shared-memory architectures using three models. We improved our previous work with an enhanced *COMM* model to better account for

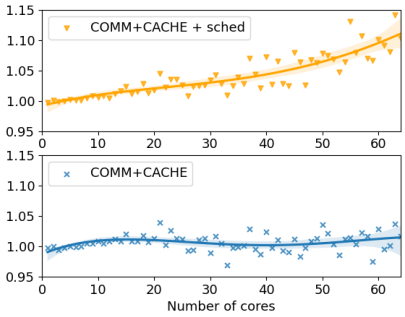

Fig. 6. Simulated performance with a cache-aware scheduler on AMD platform.

computation / communication overlap and NUMA effects. Then, they newly introduced *COMM+CACHE* model further improves this by incorporating a cache mechanism to handle data movement and cached data access. Coupled with precise modeling of the target architecture, this model achieves superior precision, underlining the importance of accurate modeling of machine components and hierarchy.

Rather than relying on error-prone manual determination of socket topology, we collected bandwidth and latency measurements from target platforms. We plan to design an automatic tool for determining platform topology and bandwidths, similar to SimGrid's automatic network discovery.

To characterize the overlap between arithmetic and memory instructions, we intend to use performance counters to observe kernel behavior and refine overlap ratios, requiring a rework of SimGrid's task execution model.

We aim to extend our evaluation to more applications and platforms, particularly those that are memory-bound. With our simulator accurately reproducing target platform behavior, it provides a reproducible experimentation platform for task-based runtime systems and schedulers, alleviating challenges associated with technical conditions on CPU-based platforms.

8 Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative, and the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computer Research, under Contract DE-AC02-06CH11357.

References

1. Agullo, E., Beaumont, O., Eyraud-Dubois, L., Kumar, S.: Are static schedules so bad? a case study on Cholesky factorization. In: IPDPS (2016)
2. Augonnet, C., Thibault, S., Namyst, R., 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**, 187–198 (Feb 2011)
3. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., 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)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1) (1996)
5. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: International Conference on Parallel, Distributed and Network-Based Processing (PDP2010). pp. 180–186. Pisa, Italia (Feb 2010)
6. Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Productive cluster programming with OmpSs. In: Proceedings of the 17th international conference on Parallel processing - Volume Part I. Euro-Par '11 (2011)
7. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: Lapack working note 191: A class of parallel tiled linear algebra algorithms for multicore architectures (2007)
8. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., 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)
9. Casanova, H.: Simgrid: a toolkit for the simulation of application scheduling. In: CC Grid. pp. 430–437 (2001)
10. 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. pp. 170– (April 2004)
11. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. SIGPLAN Notices **40**(10), 519–538 (Oct 2005)
12. Czarnul, P., Kuchta, J., Matuszek, M., Proficz, J., Rościszewski, P., Wójcik, M., Szymański, J.: MERPSYS: An environment for simulation of parallel application execution on large scale HPC systems. Simulation Modelling Practice and Theory **77**, 124–140 (2017)
13. Daoudi, I., Virouleau, P., Gautier, T., Thibault, S., Aumage, O.: sOMP: Simulating OpenMP task-based Applications with NUMA Effects. In: IWOMP 2020 (Sep 2020)
14. Denoyelle, N., Goglin, B., Ilic, A., Jeannot, E., 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)
15. 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**, 59–65 (2014)

16. Galilee, F., Cavalheiro, G., Roch, J.L., Doreille, M.: Athapascan-1: On-line building data flow graph in a parallel language. In: PACT (oct 1998)
17. Gautier, T., Besseron, X., 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. PASCO '07 (2007)
18. Gautier, T., Lima, J.V., Maillard, N., Raffin, B.: Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: IPDPS. IEEE (2013)
19. Girona, S., Labarta, J.: Sensitivity of performance prediction of message passing programs. The Journal of Supercomputing **17** (2000)
20. Haugen, B.: Performance analysis and modeling of task-based runtimes. Ph.D. thesis (2016)
21. Haugen, B., Kurzak, J., YarKhan, A., Luszczek, P., Dongarra, J.: Parallel simulation of superscalar scheduling. In: ICPP. pp. 121–130 (2014)
22. Heinrich, F.: Modeling, Prediction and Optimization of Energy Consumption of MPI applications using SimGrid. Theses, Université Grenoble Alpes (May 2019)
23. Kliazovich Dzmitry, Bouvry Pascal, K.S.U.: Greencloud: a packet-level simulator of energy-aware cloud computing data centers. The Journal of Supercomputing (2012)
24. Liu, Y., Zhu, Y., Li, X., Ni, Z., Liu, T., Chen, Y., Wu, J.: SimNUMA: Simulating NUMA-architecture multiprocessor systems efficiently. In: ICPDS (Dec 2013)
25. Mohammed, A., Eleliemy, A., Ciorba, F.M., Kasielke, F., Banicescu, I.: Experimental verification and analysis of dynamic loop scheduling in scientific applications. In: ISPDC. IEEE (2018)
26. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. Int. J. High Perform. Comput. Appl. (2) (May 2012)
27. Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramirez, A., Valero, M.: Trace-driven simulation of multithreaded applications. In: International Symposium on Performance Analysis of Systems and Software (2011)
28. Shudler, S., Calotoiu, A., Hoefler, T., Wolf, F.: Isoefficiency in practice: Configuring and understanding the performance of task-based applications. SIGPLAN Notices **52**(8) (2017)
29. Stanisic, L., Agullo, E., Buttari, A., Guermouche, A., Legrand, A., Lopez, F., Videau, B.: Fast and accurate simulation of multithreaded sparse linear algebra solvers. In: ICPDS. Melbourne, Australia (Dec 2015)
30. Stanisic, L., Thibault, S., Legrand, A., Videau, B., 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), 4075–4090 (2015)
31. Tao, J., Schulz, M., Karl, W.: Simulation as a tool for optimizing memory accesses on NUMA machines. Performance Evaluation **60**(1) (2005)
32. Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: ECPP (2016)
33. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: International Workshop on OpenMP. Springer (2014)
34. Zheng, G., Kakulapati, G., Kalé, L.V.: Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In: IPDPS. p. 78. IEEE (2004)