DOI: 10.14529/jsfi150103

Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing

Rabab Al-Omairy[1]***, Guillermo Miranda***[2]***, Hatem Ltaief***[1]***, Rosa M. Badia***[2,3]***, Xavier Martorell***[2,4]***, Jesus Labarta***[2,4]***, David Keyes***[1]

We employ the dynamic runtime system `OmpSs` to decrease the overhead of data motion in the now ubiquitous non-uniform memory access (NUMA) high concurrency environment of multicore processors. The dense numerical linear algebra algorithms of Cholesky factorization and symmetric matrix inversion are employed as representative benchmarks. Work stealing occurs within an innovative NUMA-aware scheduling policy to reduce data movement between NUMA nodes. The overall approach achieves separation of concerns by abstracting the complexity of the hardware from the end users so that high productivity can be achieved. Performance results on a large NUMA system outperform the state-of-the-art existing implementations up to a twofold speedup for the Cholesky factorization, as well as the symmetric matrix inversion, while the `OmpSs`-enabled code maintains strong similarity to its original sequential version.

Keywords: Dense Matrix Computations, Dynamic Runtime Systems, Software Productivity, Non-Uniform Memory Access, Data Locality, Work Stealing, High Performance Computing.

Introduction

Multicore Non-Uniform Memory Access (NUMA) machines are increasingly common in high performance computing, and a primary challenge of extreme computing today (see, e.g., the international exascale campaign [13]) is in expanding the number of cores per node in strong scaling. In contrast, expanding the number of nodes, which has already reached 10^5 for the BlueGene/Q system ranked #3 in the November 2014 TOP500 list, in weak scaling is well understood, at least for typical scientific SPMD codes based on load-balanced domain decomposition run on performance-reliable nodes [21]. In the last decade, dense linear algebra software underwent drastic algorithm and software stack redesigns, to maintain pace with the hardware evolution towards high concurrency. The Standard dense numerical algorithms and their state-of-the-art implementations in `LAPACK` [5] and `ScaLAPACK` [7] rely on the bulk synchronous parallel model for performance purposes. This model will display increasing vulnerability to synchronization going forward. Parallel programming models based on fine-granularity computations have shown promising results to weaken global synchronizations and to reduce data motion. In particular, task-based programming models have been successfully developed and integrated in several high performance dense linear algebra libraries (i.e., `PLASMA` [1] and `Libflame` [30]). Along with "taskifying" existing dense numerical algorithms, one of the main challenges is to deal with the actual scheduling of these tasks and the ever-changing hardware with its NUMA complexity.

While the potential of multicore NUMA architectures can be exploited with considerable ease for algorithms with good load balance at arbitrary concurrency, such as matrix multiplication, there are others in which load balance and data locality cannot be maintained simultaneously,

[1]Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia, email: Rabab.Alomary, David.Keyes, Hatem.Ltaief@kaust.edu.sa

[2]Barcelona Supercomputing Center, Centro Nacional de Supercomputación, Barcelona, Spain, email: Guillermo.Miranda, Rosa.M.Badia, Xavier.Martorell, Jesus.Labarta@bsc.es

[3]Artificial Intelligence Research Institute (IIIA) - Spanish National Research Council (CSIC)

[4]Universitat Politècnica de Catalunya

and the cost of accessing remote NUMA nodes comes with performance penalties. This is the case of the Cholesky factorization and the symmetric matrix inversion algorithms studied in this paper, which are representative benchmarks of dense factorizations and more advanced dense matrix operations, respectively. These compute-intensive operations are the basic blocks for many scientific applications (e.g., in statistics and machine learning), which require the explicit calculation of the inverse of large covariance matrices [2]. The trade-off between load balance and data locality is the key not only to the future of algorithms with time-varying work per unit memory, such as the Cholesky factorization and even more for the symmetric matrix inversion, but also to the future of hardware that is less performance-reliable due to compromises required to reduce the energy of computation. As future hardware is operated closer to unit signal to noise ratio in voltage level and as core clock rates are varied to maintain safe thermal dissipation levels, work stealing will be required to maintain load balance even for algorithms with regular work per unit memory ratios throughout their execution. Work stealing that does not cost more in performance than the imbalance it is designed to rectify is challenging to arrange.

The authors herein consider a limited type of "distance-aware" work stealing that respects the critical path of execution and the realities of NUMA. We have extended some of the functionalities of the `OmpSs` task-based programming model [6], [15] to efficiently handle NUMA architectures through an innovative distance-aware scheduling policy to reduce data movement between NUMA nodes, for instance, while performing work stealing. We have chosen `OmpSs` as the programming model because it provides a simple and non-intrusive interface (OpenMP-like) to program challenging hardware systems by abstracting the hardware complexity from end users, while keeping high productivity in mind. This new scheduling policy is aware of the NUMA architecture on which it is running, spreads work over the available cores, and implements stealing to prevent starvation.

On shared-memory systems composed of tens of NUMA nodes and for asymptotic matrix sizes, the Cholesky factorization and the symmetric matrix inversion achieve up to a twofold improvement in flop rate relative to less discriminating policies, while improving performance by a twofold speedup over the best existing implementations for both dense matrix operations on the same hardware overall.

This paper is structured as follows. We continue with the related work in Section 1. Section 2 describes the `OmpSs` framework and presents its different components. In Section 3, we briefly recall the Cholesky factorization and the symmetric matrix inversion. The implementation details of the scheduling policy are given in Section 4. In Section 5, we present the performance impact of the scheduling policy on various systems and compare against the state-of-the-art commercial and open-source high performance dense numerical libraries. Section 6 shows performance traces of both algorithms to support our performance results, and we conclude in Section 6.

1. Related Work

The volume of literature on NUMA-aware work stealing indicates the importance of adapting many classes of algorithms, linear algebra among them, to strong scaling within shared memory. Our review cannot be complete within page scope, but focuses on contributions that provide the context for ours.

Runtime frameworks for scheduling dense linear algebra algorithms have been well studied in the last few years [10, 20, 23, 24, 26]. The key idea is the redesign of the numerical algorithms so that more parallelism can be exposed and a runtime system is then employed to concurrently

schedule the computational tasks. This is the approach adopted by the PLASMA and FLAME high performance dense linear algebra libraries. The PLASMA library [1] provides a collection of dense linear algebra operations and is intended eventually to supersede LAPACK [5]. It can use internally a static (originally introduced in [22]) or a dynamic runtime system [29]. It has shown significant improvement compared to the existing approaches [4]. The FLAME library [30] provides similar functionalities and relies on the dynamic runtime system SuperMatrix [11] to execute the algorithms-by-blocks. These are high-productivity runtime systems in the sense that the user does not need to adapt to the architecture at the source-code level. However, although the scheduling frameworks of both libraries aforementioned provide features for data locality, work stealing and task priority on shared-memory systems, they do not offer scheduling policies to cope with challenging NUMA architecture. Moreover, Jeannot [19] has proposed a symbolic mapping with a static data allocation on NUMA machines, specifically for the Cholesky factorization. The main idea is to group threads by NUMA nodes to exploit the memory hierarchy. This static methodology precludes work stealing and may further impede performance for load imbalanced applications.

Recently, LAWS [12] proposes a runtime library for Divide and Conquer applications in NUMA systems. It features a work stealing algorithm designed for NUMA systems, very focused in reducing remote memory accesses and last-level cache pollution. However, it does not take into account that distances between nodes might differ across the whole system; the applications targeted are recursive, unlike Cholesky; their tasks use the same amount of data, which allows the auto tuning of the cut-off threshold.

Drebes [14] presents a scheduling and allocation algorithm for the OpenStream language. While similar to this paper in topic and features (detection of the node with the most data for a given task, locality aware stealing that takes into account distances between NUMA nodes), there are considerable differences: a task is assigned to a thread based on its input location only, experimental results are validated only up to 64 threads, a slab allocator is used that may give incorrect information about the node where a memory chunk is allocated which in turn, according to the paper, results in a speedup of 0.99 over their base line (random stealing).

There are also standalone dynamic runtime system libraries for general purpose. HPT [28] presents an abstraction for task parallelism and data movement. The memory hierarchy is represented as a tree where workers belong to leaf nodes. In this approach, a task assigned to a memory place (cache, NUMA node, etc.) will be executed only by workers below its assigned place. For instance, a task assigned to an L3 cache can run in any core below that cache, but not in the cores that share a different L3 cache. In the scheduling policy described in this paper, we further leverage this principle to tackle NUMA node locality. The Wool library [16] presents an efficient work stealing approach, but it does not take into account data locality. Wool requires the programmer to modify the source code, whereas we use OmpSs to annotate the source code, with very few modifications. Furthermore, Wool tasks must be independent and therefore, the scope of applications susceptible to adhere to this restriction is rather narrowed. Recently, Muddukrishna [25] proposes some ideas to preserve locality in an OpenMP runtime/compiler infrastructure, but the evaluation was performed in a 48-core machine, and using benchmarks (such as the matrix multiplication) that do not have dependencies between tasks as complex as the dense matrix operations experimented in this paper. Last but not least, providing locality hints has been proposed and studied in Broquedis et al. [8], where scheduling hints are used to choose the best thread and data distribution. Their approach was evaluated using a 16-core

machine, while our experimental platforms are additionally composed by a 8-node, 48-core and a 128-node, 1024-cores NUMA systems. Similarly, `ForestGOMP` runtime maps nested thread teams to the underlying hardware resources.

2. The `OmpSs` Framework

`OmpSs` is a high-level, task-based, parallel programming model supporting SMPs, heterogeneous systems (like GPGPU systems) and clusters. `OmpSs` consists of a language specification, a source-to-source compiler for C, C++ and Fortran; and a runtime. The `OmpSs` programming model is powered by the Nanos++ runtime, which provides services to support task parallelism using synchronizations based on data-dependencies. Data parallelism is also supported by means of services mapped on top of its task support.

2.1. The Nanos++ Dynamic Runtime System

Figure 1 depicts the infrastructure of the Nanos++ dynamic runtime system. Nanos++ interfaces the application with the underlying hardware architecture. A core component of the runtime is in charge of handling data movement and ensuring coherency, so that the programmer does not need to deal with memory allocation and movements. Another core component is the module of scheduling policies, which will ultimately integrate the distance-aware work stealing policy introduced in this paper. `OmpSs` also supports heterogeneous programming and that is reflected in the modular support for different architectures in Nanos++ (SMP, CUDA, OpenCL, simulators such as tasksim, etc.). Nanos++ provides also instrumentation tools to help identifying performance issues.

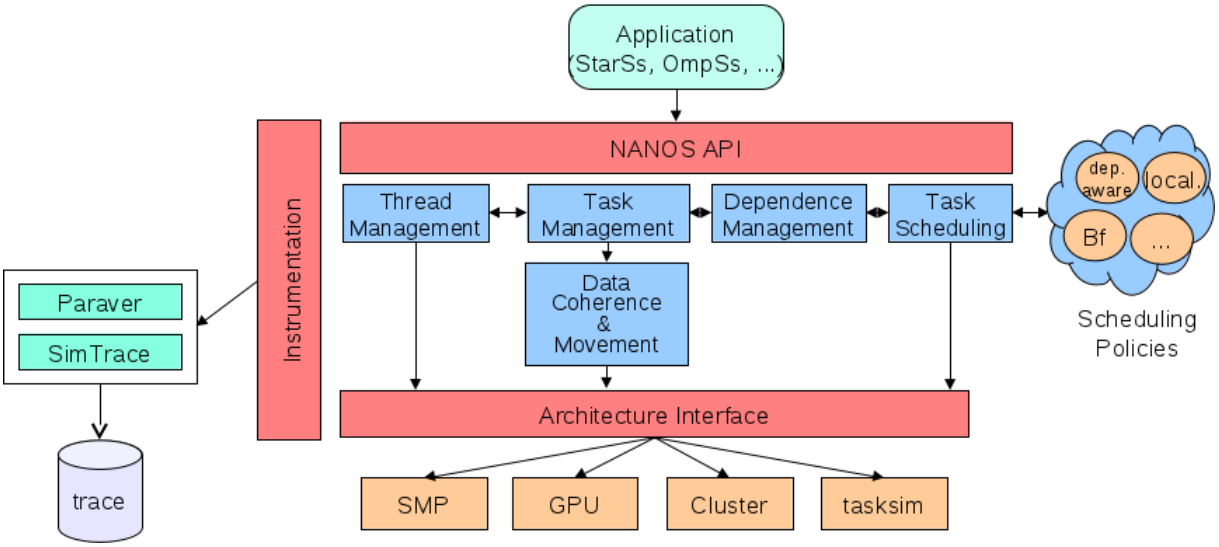


Figure 1. The Nanos++ infrastructure

2.2. Task-Based Programming Model in `OmpSs`

In `OmpSs`, data dependencies are contained in a directed acyclic graph (DAG). Tasks are blocked until all their dependencies are satisfied. Once a task is released, the scheduler is then in charge of taking the proper runtime decisions. Each scheduling policy defines a certain behavior. For instance, a simple policy might implement a global first-in, first-out (FIFO) queue, where

tasks are executed in the order of dependency release; or we could define a policy that holds one queue per worker thread, where tasks are sorted based on a priority value selected by the programmer. The thread management consists in a pool of worker threads that start to execute work once it becomes available. When a worker thread is aware of new available work, it will query the scheduling policy, which will provide a new task to the worker if it so decides. For instance, in a NUMA aware scheduling policy that ensures locality on top of every other aspect, if a worker thread is requesting new work and the scheduling policy only has work for threads of other NUMA nodes, it will not give a new task to the demanding worker thread, for the sake of enforcing locality. Regarding data dependencies, Nanos++ delegates that component to plugins, in a similar way to scheduling policies. As this runtime is designed to work with a wide range of applications, some may be simple (when it comes to specify dependencies) and do not need the added complexity inherent to the handling of more complex dependencies (such as non-contiguous memory regions) that other applications rely on.

2.3. `OmpSs` Tools for Performance Analysis

Nanos++ has an instrumentation plugin system used to obtain traces of the executions. In this paper, we have chosen the plugin that works with `Extrae`, the core instrumentation package developed by the Performance Tools group at Barcelona Supercomputing Center, and `Paraver`, a very flexible data browser developed by the same group. Together, they enable programmers to analyze the behavior of the applications, identify potential problems and understand how they can be solved. `Extrae` uses different interposition mechanisms to inject probes into target applications so as to gather information regarding the application performance. Nanos++ features an instrumentation module with support for `Extrae`, thus, obtaining traces for `OmpSs` applications is just a matter of running them with the instrumentation plugin enabled. As for `Paraver`, Nanos++ comes with a vast set of `Paraver` configuration files that convert tracing events to human-readable information that can be displayed as timelines (such as user functions duration), histograms (e.g., thread state to know how much the application was running, idling and the overhead introduced by the runtime) and three-dimensional histograms.

3. The Cholesky Factorization and the Symmetric Matrix Inversion Algorithm

This Section briefly recalls the standard block and the new tile variants of the Cholesky factorization and the symmetric matrix inversion. Further algorithmic details can be found in [3, 10, 11].

3.1. Block Algorithm Variant

Computing the Cholesky factorization is the first step toward solving dense systems of linear equations for symmetric positive-definite matrices, which arise in many scientific applications [17]. Based on the Cholesky factorization, the symmetric matrix inversion is also important for the computation of the variance-covariance matrix in statistics [18]. The state-of-the-art dense linear algebra library `LAPACK` [5] uses block algorithms. The computation is basically split into successive sequences composed by two phases: (1) the panel computation phase, mainly based on level 2 BLAS, in which the transformations are accumulated within a panel of the matrix

and (2) the update of the trailing submatrix, in which the transformations from the panel phase are applied at once to the trailing submatrix in terms of level 3 BLAS operations. One of the bottlenecks with such approach is the creation of unnecessary synchronization points between the phases. Moreover, `LAPACK` extracts its performance for the most part from the parallel multithreaded BLAS. The parallel paradigm behind it is very similar to the `OpenMP` fork-join model, which further exacerbates the issue related to artifactual synchronization points. The design of the block algorithms for calculating the Cholesky factorization and the symmetric matrix inversion also fall into this category. The original matrix is reduced using the Cholesky factorization $A = LL^T$ (using the *DPOTRF* routine), where L is a lower triangular square matrix with positive diagonal elements. Following the factorization, there are two additional dense operations for the symmetric matrix inversion: (1) triangular matrix inversion $A = L^{-1}$ (using the *DTRTRI* routine) and (2) triangular matrix product $A = L^{-T}L^{-1}$ (using the *DLAUUM* routine). The block variant of the Cholesky factorization and the symmetric matrix inversion are therefore very limited in terms of parallelism and cannot fully benefit from now commonly available highly-parallel processing units.

#pragma omp task inout([NB][NB]A) priority(HIGHEST)
void DPOTRF(double *A);
#pragma omp task inout([NB][NB]A)
void DTRTRI(double *A);
#pragma omp task inout([NB][NB]A)
void DLAUUM(double *A);
#pragma omp task input([NB][NB]A) inout([NB][NB]C) priority(NT - k)
void DSYRK(double *A, double *C);
#pragma omp task input([NB][NB]A) inout([NB][NB]C) priority(NT - k)
void DTRSM(double *A, double *C);
#pragma omp task input([NB][NB]A) inout([NB][NB]C) priority(NT - k)
void DTRMM(double *A, double *C);
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void DGEMM(double *A, double *B, double *C);
for k = 0 to NT−1 **do**
  // Stage 1: Cholesky factorization $A = LL^T$
  DPOTRF($A_{k,k}$);
  for m = k+1 to NT−1 **do**
    DTRSM($A_{k,k}$, $A^T_{m,k}$);
  end for
  for m = k+1 to NT−1 **do**
    DSYRK($A_{m,k}$, $A_{k,k}$);
    for n = k+1 to m−1 **do**
      DGEMM($A_{m,k}$, $A^T_{n,k}$, $A_{m,n}$);
    end for
  end for
  // Stage 2: Calculate $A = L^{-1}$
  for m = k+1 to NT−1 **do**
    DTRSM($A_{k,k}$, $A_{m,k}$);
    for n = 0 to k−1 **do**

```
      DGEMM(A_{m,k}, A_{k,n}, A_{m,n});
    end for
  end for
  for m = k+1 to k−1 do
    DTRSM(A_{k,k}, A_{k,m});
  end for
  DTRTRI(A_{k,k});
  //Stage 3: Compute A^{-1} = L^{-T} × L^{-1}
  for n = 0 to k−1 do
    DSYRK(A^T_{k,n}, A_{n,n});
    for m = n+1 to k−1 do
      DGEMM(A_{k,m}, A^T_{k,n}, A_{m,n});
    end for
  end for
  for n = 0 to k−1 do
    DTRMM(A^T_{k,k}, A_{k,n});
  end for
  DLAUUM(A_{k,k});
end for
```

Algorithm 1: Tile `OmpSs`-enabled distance-aware Cholesky factorization and symmetric matrix inversion algorithms.

3.2. Tile Algorithm Variant

The idea behind tile algorithms is to transform the original matrix data with column-major data layout into tile data layout. The parallelism becomes then exposed to the user, thanks to the task fine granularity. Indeed, the matrix tiles can be seen as the fundamental unit of computations of the numerical algorithms. The rigid panel-update sequence, previously described in Section 3.1, is now replaced by an out-of-order task execution flow, where computational tasks operating on tiles from different loop iterations can concurrently run. The algorithmic complexity of the block variant of both dense matrix computation algorithms does not change with the tile variant and is equal to $1/3n^3$ and n^3 for the Cholesky factorization and the symmetric matrix inversion algorithm, respectively.

The sequential program can now be represented as a DAG, where nodes stand for tasks and edges correspond to data dependencies. The strong and artifactual synchronization points, seen in block algorithm variant, are considerably alleviated using tile algorithms. For instance, the next panel factorization can proceed while the updates of the previous panel have not finished yet, as long as data dependencies on the corresponding tiles are satisfied. Now, it is up to a runtime system to schedule all generated tasks and to enforce their inter-task data dependencies. In particular, the `OmpSs` programming model has the advantage of being non-intrusive and relies on simple pragmas, similar to OpenMP programming model syntax. In fact, OpenMP 3.0 onwards, the tasking concept has been integrated to further help supporting mainstream applications. Algorithm 1 shows the parallel `OmpSs`-enabled version of the tile sequential Cholesky factorization (stage 1 only) and the symmetric matrix inversion (all stages) for an $NT \times NT$ tile symmetric positive-definite matrix A with a tile size NB. It is still the user's duty to describe the data

directions (`input`, `output` and `inout`) for each computational task, through compiler directives (i.e., pragmas).

4. Distance-Aware Work Stealing Scheduling Policy

This section provides implementation details of the new distance-aware work stealing scheduling policy and its incorporation into Nanos++ runtime.

4.1. Task queues

In order to maintain data locality, we have a task queue per NUMA node. This queue is a linked list sorted by task priority: the programmer is able to specify the priority of each task as a way to outline the critical path. Sorting is performed on insertion, thus the queue is sorted at any given time (ensuring the execution of tasks based on priorities as much as possible).

Priority-sorted linked lists are not a burden, given the granularity of the tasks of the experiments we carried. To prove that point, we developed a synthetic benchmark creating 10000 tasks lasting one microsecond each. There was no statistically measurable difference between a vector-based queue and a linked list based queue in total execution time or in runtime overhead when analyzing execution traces.

Threads belonging to a NUMA node are only able to retrieve tasks from their node's queue. This is accomplished by obtaining the number of NUMA nodes and the node corresponding to each worker thread from the Portable Hardware Locality (*hwloc*) library [9], and assigning each thread the number of the queue it should query.

4.2. Data distribution and locality detection

The runtime is able to track the location of the data and schedule tasks in the node with the highest number of bytes. To track the locality, it assumes a first touch policy and looks for initialization tasks. The criteria to detect such tasks is plain and simple: tasks with an output dependency (at least one) where it is the first time that data will be written. In our Cholesky implementation those tasks are the ones that initialize a block of the matrix.

Initialization tasks are scheduled in round robin across the available NUMA nodes, enabling us to use a similar data distribution. When a task of that type is executed, the data it initializes will be marked as located in the NUMA node of the running thread.

Otherwise, when a non initialization task is submitted, the number of bytes accessed by each node will be computed, based on the dependency information provided by the programmer, and the task will be scheduled in the node with the largest amount of data.

Note that once work stealing is introduced, the locality information becomes a hint that the runtime will always follow unless there is starvation in the local node.

Kurzak [23] described an implementation of the Cholesky factorization using static scheduling where threads work only on a one-dimensional cyclic distribution in order to keep locality.

4.3. Distance-aware work stealing

We choose to steal from neighbor nodes following a round-robin approach, with each node having an independent node index for stealing: steal from nodes in a cyclic way. In a NUMA architecture where some nodes are further than others, the distance-aware work stealing schedul-

ing policy ensures a thread will only steal tasks that are as close as possible. Thankfully, this information is provided by the Linux kernel. The distance between nodes provided by the operating system is not an accurate measure of the practical latency. The OS first tries to get this information by reading the System Locality Information (SLIT) table in the BIOS and if it fails, it may generate its own table, even if the vendor has explicitly provided a SLIT table. For instance, assume a system with 8 nodes, with the distance between node 1 and node 3 is 21 and between node 1 and node 4 is 42. In fact, these distances are default values from the BIOS SLIT table and they do not necessarily mean that the latency of accessing memory of node 3 from node 1 is half of accessing node 4. We can only be sure that node 4 is further from 1 than 3, and this is the reason why we only consider valid steal targets adjacent nodes. If in the previous example there was a node 5 with distance 22, it could still have a latency higher enough than node 3 that would hamper performance and eliminate any benefit of work stealing. We observed when comparing executions with and without work stealing, where the execution time of the tasks was noticeably worse with work stealing. In the execution without work stealing we discovered small gaps between a finalized task and the next one that were not present in the work stealing trace. The reason for such gaps is that the coming tasks are not yet available in the corresponding ready queue. As worker threads are constantly looking for work, with work stealing is enabled we observed that delaying for a small number of iterations (of the `OmpSs` runtime loop that looks for work for a given thread before giving up) would prevent those situations, thus increasing locality at the cost of a lower balance factor. This is controlled via a user defined variable, which can be set to zero to disable waiting. Please note that if that value is set to a large number it could effectively disable work stealing: worker threads will have to wait so long that they might find work available in their local queues before allowed to steal. To sum up, stealing does not come for free: while it reduces load imbalance, it obviously reduces data locality for stolen tasks. In other words, some tasks will take longer to complete and that is the reason why work stealing cannot be allowed to be performed indiscriminately.

These mechanisms we have described are able to improve load balance, while minimizing an increase in task execution time, as highlighted in the next Section.

5. Experimental Results

This Section highlights the impact of the distance-aware with work stealing scheduling policy in terms of performance (Gflop/s) and compares the new `OmpSs`-enabled Cholesky factorization and symmetric matrix inversion against existing commercial and open-source high performance dense linear algebra libraries.

5.1. Environment Description

Experiments were performed on three NUMA systems. System (A) is a quad processor AMD Magny-Cours 6172 with four sockets, twelve cores each, running at 2.1GHz with *two* NUMA nodes per socket (there are two dies in each physical package), with four HyperTransport links by socket resulting in a maximum distance between NUMA nodes of two hops. System (B) is an AMD Istanbul 8439 SE Processor with eight sockets, six cores each, running at 2.8GHz with *one* NUMA node and three HyperTransport links per socket. Both systems have 128GB of memory. It is noteworthy to mention that the NUMA nodes of system (B) are unequally distant and further away from each other compared to the NUMA nodes of system (A), as this system (B)

is composed of two 4P boards connected via two HyperTransport connectors, for a maximum distance of three hops between nodes. The third system (C) is an SGI Altix 1000 UltraViolet shared-memory machine based on Intel Nehalem EX processors featuring 128 sockets, eight cores each, running at 2.0GHz with *one* NUMA node per socket. This system has 4 TB of global shared memory in a single system image. We compared the Cholesky factorization and the symmetric matrix inversion using `OmpSs` against `PLASMA` v2.4.5, `Libflame` v5.0, `LAPACK` v3.4.2, and Intel compiler/`MKL` v10.1.015 on systems (A) and (B) and v11.1.038 on system (C). The `PLASMA` (providing `QUARK` as well as a static runtime system) and `Libflame` (using `SuperMatrix`) libraries are compiled with the sequential `MKL` BLAS, while `LAPACK` uses the multithreaded `MKL` BLAS. `PLASMA` and `Libflame` have been tuned for the underlying hardware by selecting an optimal block size. The command `numactl --interleave=all` has been executed to ensure a fair comparison against `LAPACK` and `MKL` implementations, which uses static data distribution. Moreover, in order to prevent false sharing, memory is aligned to the page size using *posix_memalign*. Otherwise, the distance-aware scheduling policy would be working with invalid locality information. Last but not least, all performance graphs in Gflop/s report the theoretical peak performance of the different system. The idea is to provide a good (but not realistic) upper-bound on all performance curves.

5.2. Distance-Aware Scheduling Policy Optimization Analysis

One of the main critical tasks of the Cholesky factorization and the symmetric matrix inversion is the matrix multiplication kernel *DGEMM*. Based on its number of tasks (called in the inner loop of Algorithm 1) and its execution time, we observed that it was the utmost task to focus on, when it comes to increasing the overall performance.

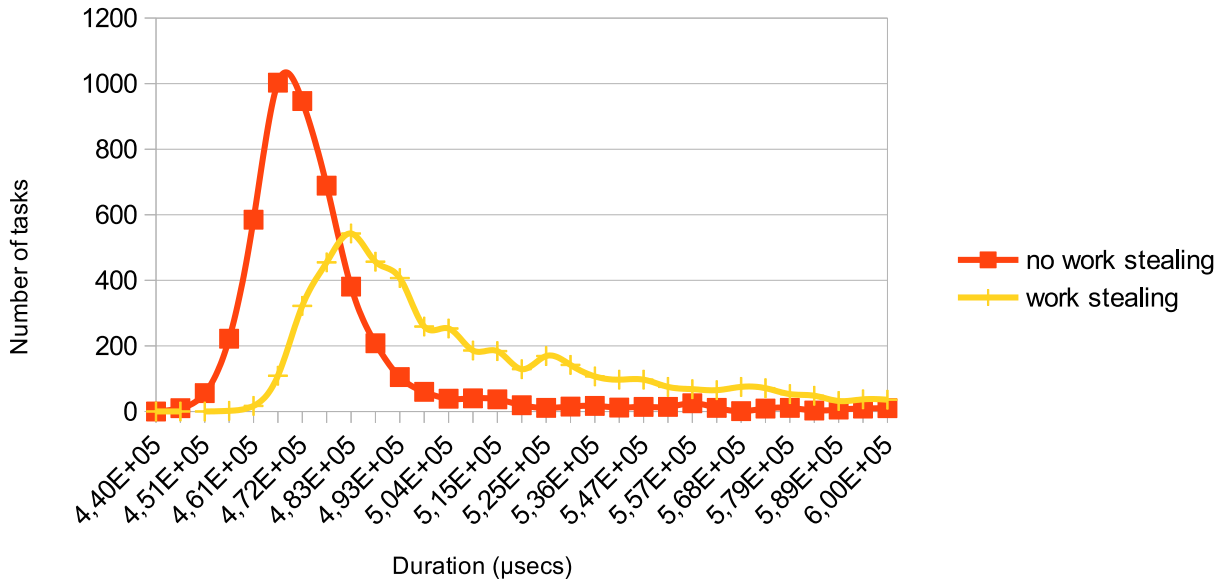


Figure 2. Task execution time histogram of *DGEMM*'s tasks on System (A) with the distance-aware scheduling policy

Figure 2 shows the task execution time histogram (horizontal axis) for *DGEMM*'s tasks, when distance-aware scheduling policy is turned on (work stealing mode) or not (no work stealing mode). A high value in the vertical axis represents a high concentration of *DGEMM*'s tasks for that particular time interval, and a low one means a small concentration. The other kernels involved in the dense matrix computation algorithms have been removed from the timing trace diagram for simplicity of presentation. Ideally, we should have an almost vertical line. This directly translates to very little variation (no scattered points across the horizontal and therefore, high concentration of tasks). In the case with no work stealing we have a situation very similar to the ideal one, where a very high data locality results in the system taking more or less the same time to execute tasks.

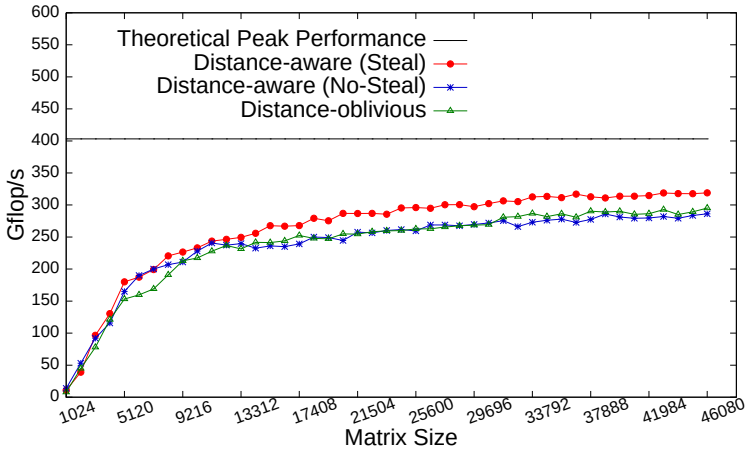
On the contrary, when work stealing takes effect, Figure 2 shows that most of *DGEMM*'s tasks have slightly shifted to the right of the histogram, because they take longer to terminate. There is also higher variability compared to the distance-aware without stealing policy. This indicates that threads have accessed data from remote NUMA nodes. Indeed, transferring data between farther NUMA nodes through the various hops creates a huge performance penalty because of the higher memory latency required when accessing distant nodes compared to local memory or adjacent NUMA nodes. Note that a perfect data distribution on large NUMA system is challenging due to the nature of the dense matrix computation algorithms and the complexity of the system studied here. Numerical algorithms using hierarchical data representation (e.g., fast multipole method [27]) or divide-and-conquer mechanism can better leverage such hardware architecture, hence the challenge.

These observations validate our initial concern that work stealing introduces a performance penalty and that we should only steal from adjacent nodes to mitigate its negative impact. The next Section demonstrates whether the `OmpSs` framework is still able to compensate the work stealing overhead by diminishing load imbalance and ultimately increasing the overall performance (Gflop/s).

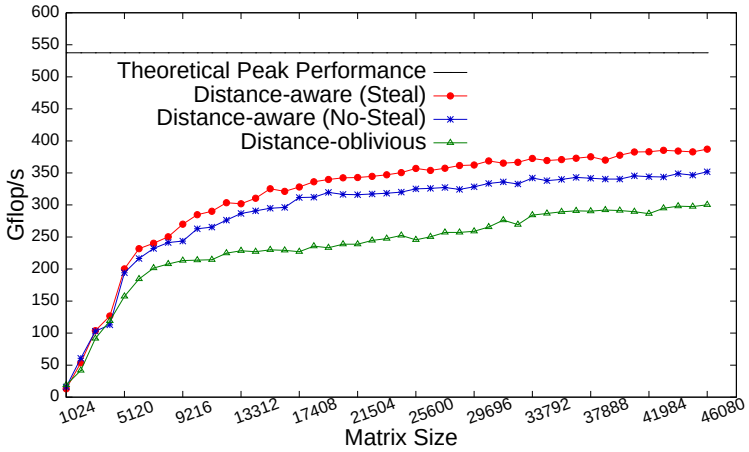
5.3. Performance Impact of Various Scheduling Policies

Figure 3 and 4 show the performance impact of various scheduling policies on the Cholesky factorization and the symmetric matrix inversion using systems (A) and (B), respectively, where the matrix size is increased in 4K increments along the horizontal axis until an asymptotic performance is reached. Due to the proximities of the NUMA nodes on system (A), Figure 3a does not show any difference whether we are running with or without the distance-aware policy. There is still a slight performance improvement, when work stealing is turned on (320 Gflop/s i.e., 80% of peak), thanks to a better data locality management. However, when NUMA nodes are farther, as in system (B), we can clearly distinguish the performance impact of scheduling policies. Figure 3b captures this discrepancy. The distance-aware with work stealing scheduling policy scores a 30% and 15% improvement in Gflop/s compared to the distance-aware without stealing and the distance-oblivious scheduling policies, respectively and reaches 390 Gflop/s i.e., 72% of peak. Although the symmetric matrix inversion presents more complex memory accesses due to two additional computational stages besides the Cholesky factorization, the distance-aware with work stealing scheduling policy is able to maintain as similar performance impact as the Cholesky factorization, on system (A) (Figure 4a) and system (B) (Figure 4b).

The performance impact is even further amplified with large number of NUMA nodes from system (C), as shown in Figures 5 and 6 for the Cholesky factorization and the symmetric matrix



a) System (A)



b) System (B)

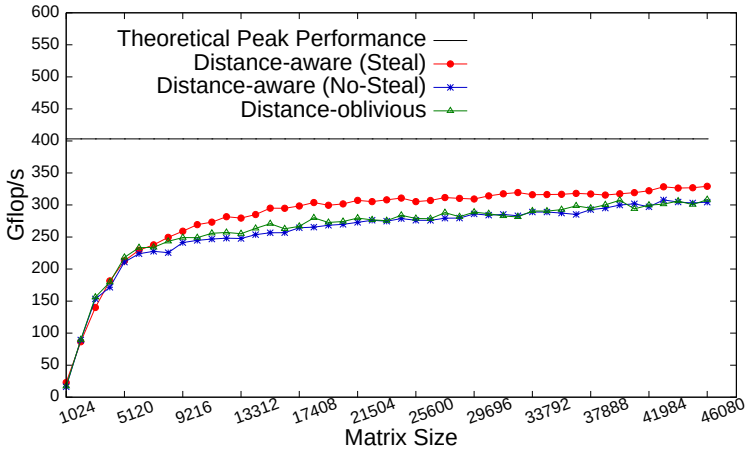
Figure 3. Performance impact of various scheduling policies on the Cholesky factorization

inversion, respectively. The distance-aware with work stealing scheduling policy become critical to sustain performance, as the number of NUMA nodes increases. On 32 sockets (256 threads), the distance-aware with work stealing scheduling policy achieves roughly fourfold and twofold performance improvement against the distance-aware without stealing and the distance-oblivious scheduling policies, respectively.

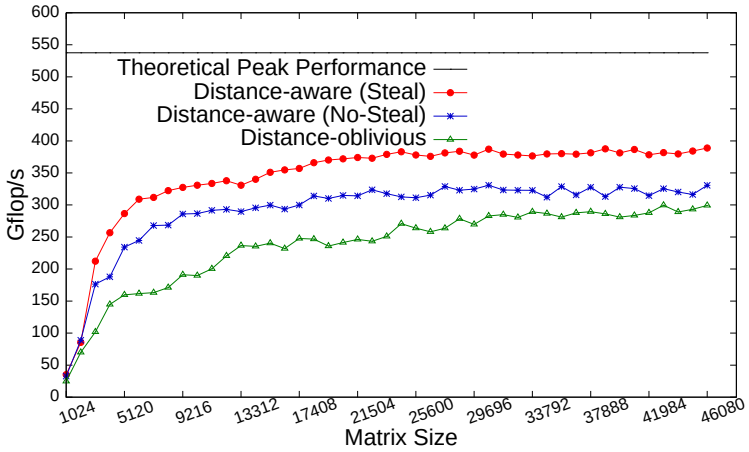
5.4. Performance Comparisons Against State-of-the-Art High Performance Dense Linear Algebra Libraries

Figures 7 and 8 show performance comparisons of the `OmpSs`-enabled Cholesky factorization and symmetric matrix inversion, respectively, with distance-aware with work stealing scheduling policy against existing high performance dense linear algebra implementations: `PLASMA` (providing two scheduler types: static and `QUARK`), `Libflame` (`SuperMatrix`), the commercial Intel `MKL` and the open-source `LAPACK` library. For the `PLASMA` library, only the scheduler type achieving the best performance is reported.

On system (A) (Figure 7a), the `LAPACK` implementation of the Cholesky factorization performs the worst due to the inefficient panel-update sequences, which generated lots of synchronizations, as previously mentioned in Section 3.1. The performance of the Intel `MKL` variant



a) System (A)



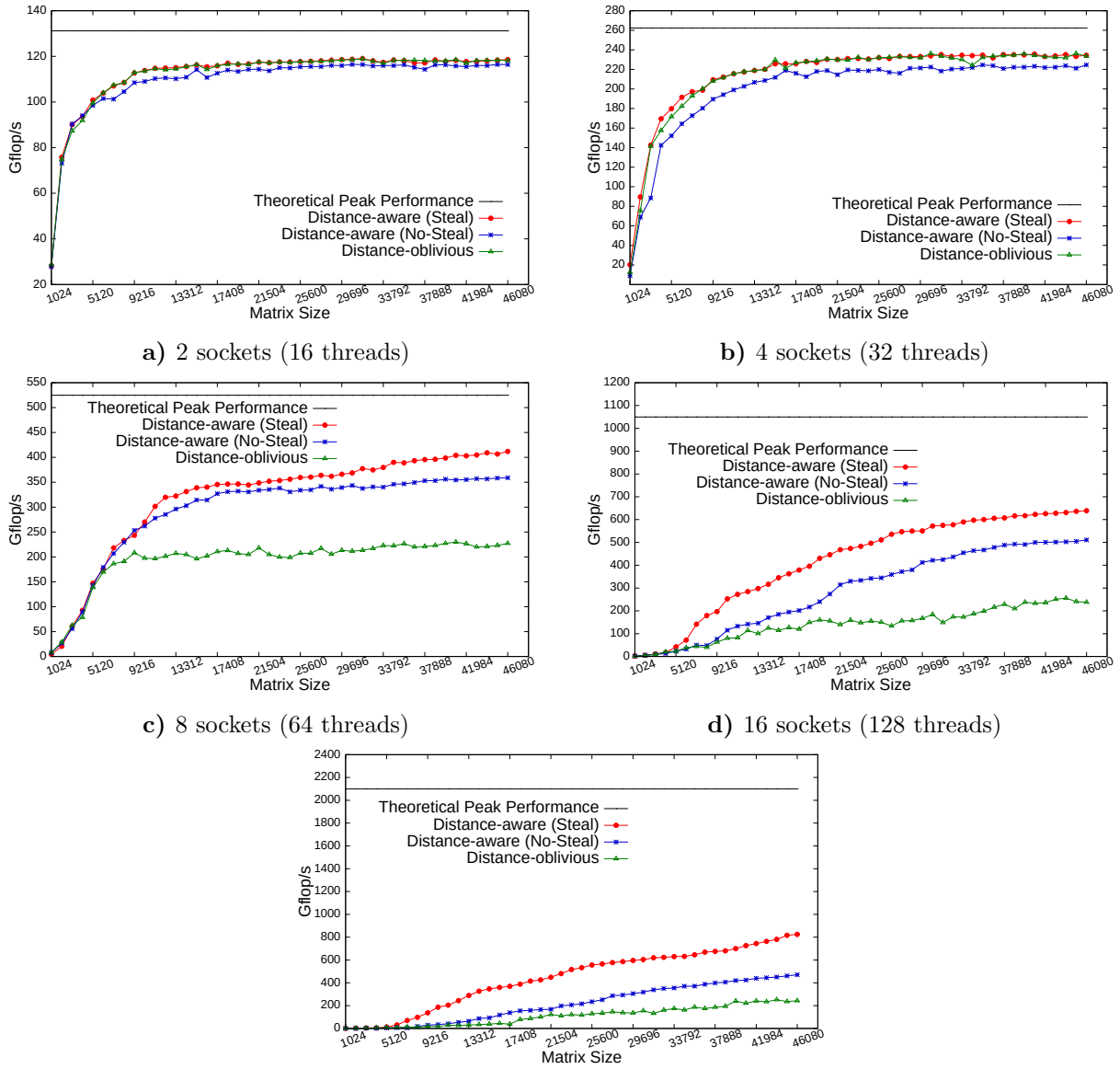
b) System (B)

Figure 4. Performance impact of various scheduling policies on the symmetric matrix inversion algorithm

increases substantially but still seems to suffer from memory accesses, as indicated by the curve's dips. The Cholesky implementations of `OmpSs`, `PLASMA` and `Libflame` have similar performance behavior on this system (A) with clustered NUMA nodes. On system (B) (Figure 7b), the OmpSs implementation scores up to 30% improvement against `PLASMA` with the dynamic scheduler `QUARK` (open-source package) for asymptotic sizes and more than twofold speedup against Intel `MKL` (commercial package).

Figures 8a and 8b show the performance of the symmetric matrix inversion on systems (A) and (B), respectively. The performance of the `LAPACK` symmetric matrix inversion is extremely low on both systems, again due to the lack of parallelism and data locality as well as artifactual barriers. The Cholesky factorization using the distance-aware with work stealing scheduling policy from `OmpSs` outperforms `Libflame` up to 50% across all matrix sizes and `PLASMA` by up to 50% and 25% for small and large matrix sizes, respectively.

The Cholesky factorization using static scheduler from `PLASMA` gives lower performance than `QUARK` and is not reported in Figures 7 and 8. The dynamic scheduler `QUARK` has thread binding, affinity mechanisms and is able to deal with load imbalance coming from thread starvation on these systems with rather small number of NUMA nodes. However, the overhead of the work stealing strategy as implemented in `QUARK` becomes significant in presence of large number



a) 2 sockets (16 threads)

b) 4 sockets (32 threads)

c) 8 sockets (64 threads)

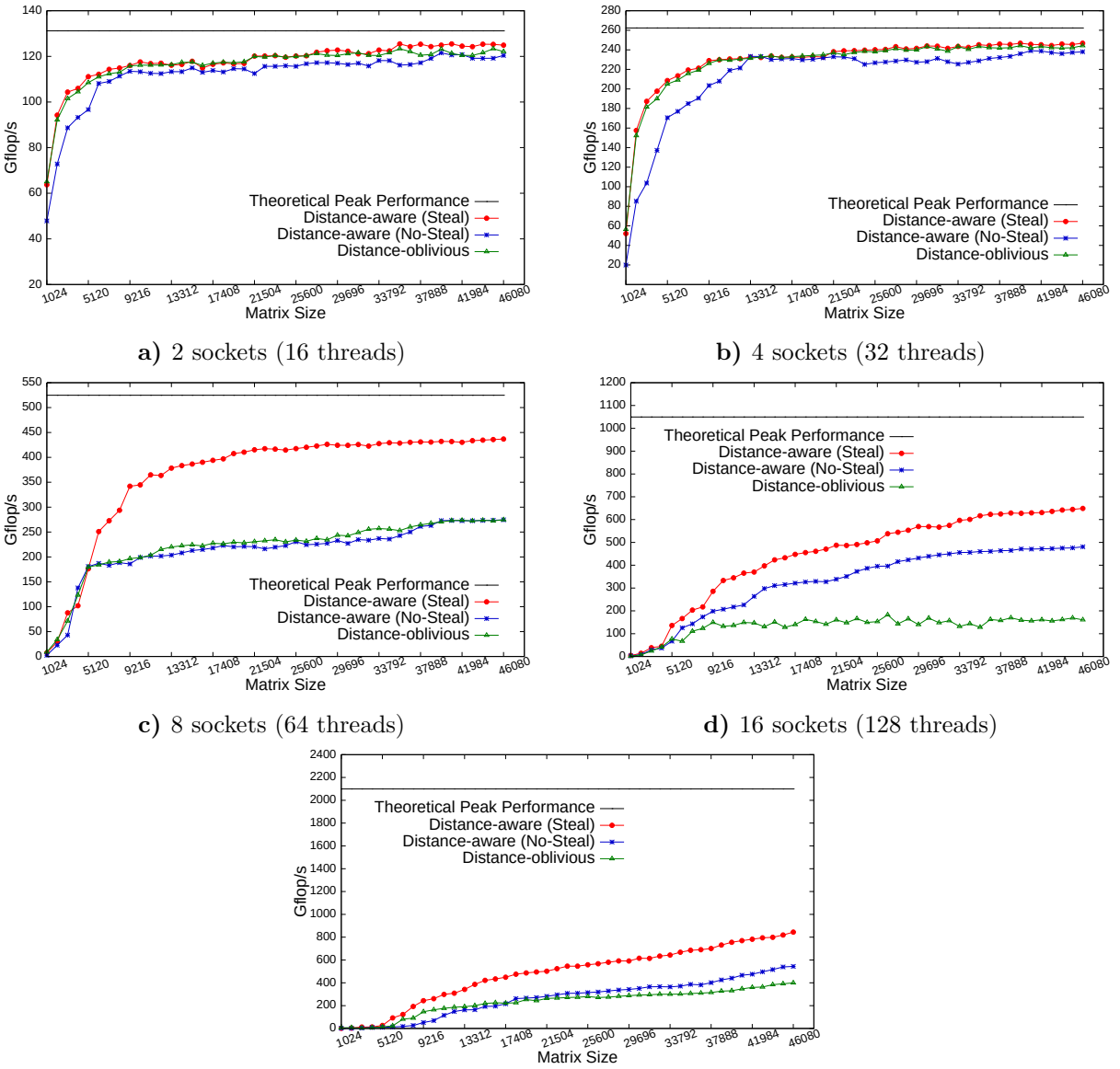
d) 16 sockets (128 threads)

e) 32 sockets (256 threads)

Figure 5. Performance impact of various scheduling policies on the Cholesky factorization on system (C)

of distant NUMA nodes and exceeds the overhead of load imbalance generated by the static scheduler due to idling of many worker threads. For the following subsequent graphs, we will therefore only refer to `PLASMA` static scheduler.

Figures 9 and 10 highlight the performance comparisons against existing Cholesky factorization and symmetric matrix inversion implementations on system (C). The `OmpSs`-enabled Cholesky factorization using the distance-aware with work stealing scheduling policy outperforms `PLASMA` (static scheduler) and `Libflame` implementations up to 65% on eight NUMA nodes and up to 200% on 32 NUMA nodes. `MKL` and `LAPACK` Cholesky (similarly for the symmetric matrix inversion) have not been run beyond eight sockets because performance would have been extremely low anyway. By the same token, the `OmpSs`-enabled symmetric matrix inversion using the distance-aware with work stealing scheduling policy is capable of sustaining the Cholesky factorization performance against the same other implementations i.e., twofold



a) 2 sockets (16 threads)

b) 4 sockets (32 threads)

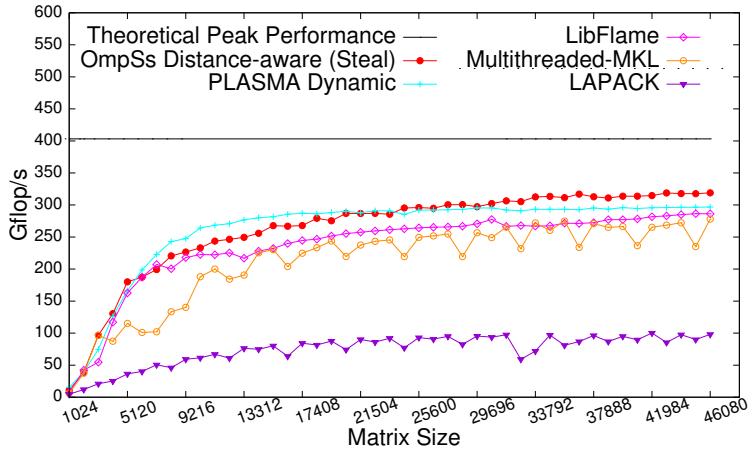
c) 8 sockets (64 threads)

d) 16 sockets (128 threads)

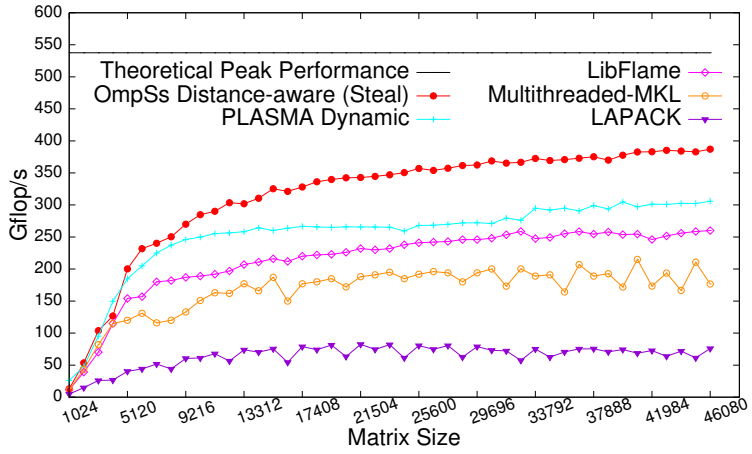
e) 32 sockets (256 threads)

Figure 6. Performance impact of various scheduling policies on symmetric matrix inversion using system (C)

performance improvement. Although the `OmpSs` dynamic runtime system associated with the new scheduling policy seems to decently exploit the underlying NUMA architecture, it is noteworthy to mention that the best performance achieved by the distance-aware with work stealing scheduling policy represents only 40% of the theoretical peak of system (C) on 32 sockets. It is well-known that a system's theoretical peak performance is a loose upper bound. We can still identify possible reasons for this low sustained peak performance. The shared-memory system is a shared resource and other users' applications running at the same time may engender memory bandwidth saturation causing further overheads. The overhead of the OS for ensuring cache coherency may also explain it, which is typical for such a the large memory system. There is also, of course, room for improvement at the runtime level and also further tuning of the tile size for large matrix sizes may further pay off.



a) System (A)



b) System (B)

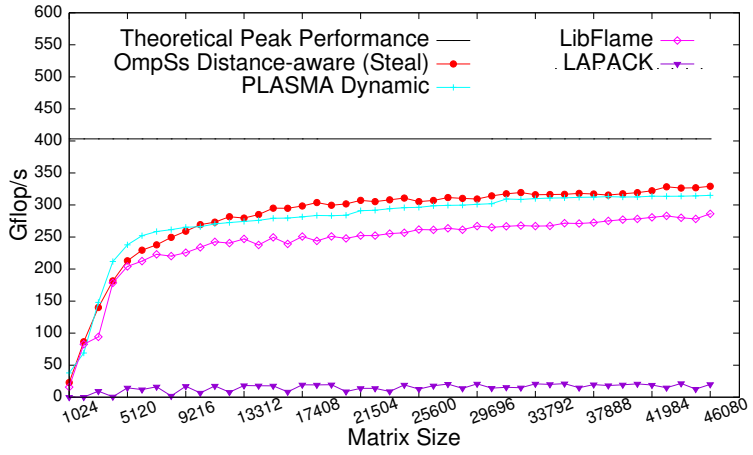
Figure 7. Performance comparisons against existing Cholesky factorization implementations

6. Performance Traces

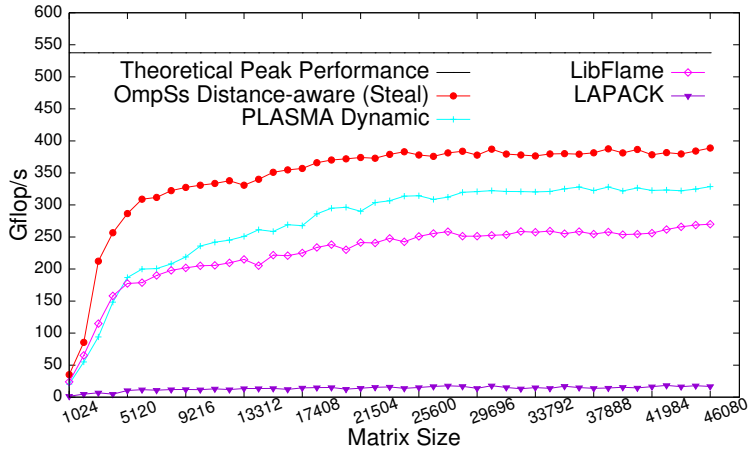
To further analyze the performance results shown in Section 5, the `OmpSs`-enabled Cholesky factorization and symmetric matrix inversion have been instrumented in order to generate traces using the `Extrae` tracing and the `Paraver` libraries.

Figures 11 and 12 show the execution traces of distance-oblivious and distance-aware (with and without work stealing) scheduling policies of the Cholesky factorization and the symmetric matrix inversion algorithm on system (B) using 48 cores, respectively, with a matrix size 30720×30720. The horizontal axis represents the timeline, the vertical axis the threads and the colors refer to different tasks. Figures 11a and 12a, representing traces for both dense matrix computation algorithms based on the distance-oblivious scheduling policy, show rather long but compact timelines.

Figures 11b and 12b represent traces for both dense matrix computation algorithms based on the distance-aware without stealing scheduling policy. These figures show shorter timelines but reveal severe idle time. This is mainly due to thread starvation, which engenders significant load imbalance between NUMA nodes. Targeting only data locality should not exclusively be the main concern for high performance applications. In fact, performance can be hindered by excessively hinting for data locality.



a) System (A)



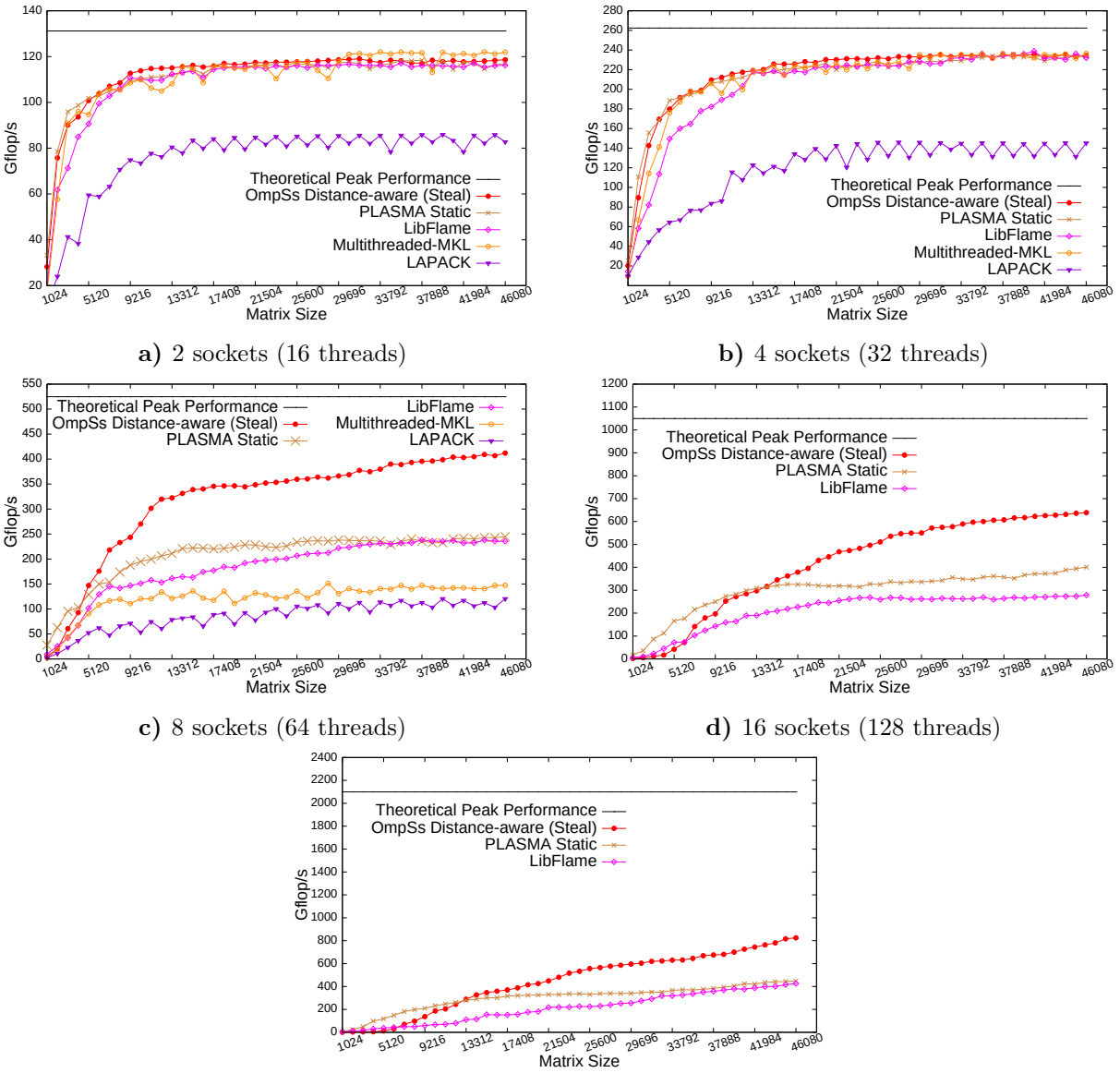
b) System (B)

Figure 8. Performance comparisons against existing symmetric matrix inversion implementations

Figures 11c and 12c highlight traces for both dense matrix computation algorithms based on the distance-aware with stealing scheduling policy. The timelines are now even shorter than distance-aware without stealing scheduling policy, despite a slightly longer elapsed time for *DGEMM* kernel, as detailed previously in Figure 2. The distance-aware with stealing scheduling policy is able to compensate the overhead of increased *DGEMM*'s elapsed time by removing most of stalls in the execution trace. One can now visualize the benefit of stealing from adjacent nodes, where tasks are continuously scheduled without gaps until the end of execution.

Conclusions and Future Work

We have demonstrated the role for the distance-aware scheduling policy, which increases data locality, to significantly improve the performance of two important dense linear algebra algorithms with time-varying work per unit memory at relatively high programmer productivity. We have also highlighted that work stealing in addition to distance-aware scheduling policy is paramount to attenuate load imbalance and to be ultimately effective on NUMA systems. Performance results on a large NUMA system outperform the best state-of-the-art existing implementations up to a twofold speedup for the Cholesky factorization as well as the symmetric



a) 2 sockets (16 threads)

b) 4 sockets (32 threads)

c) 8 sockets (64 threads)

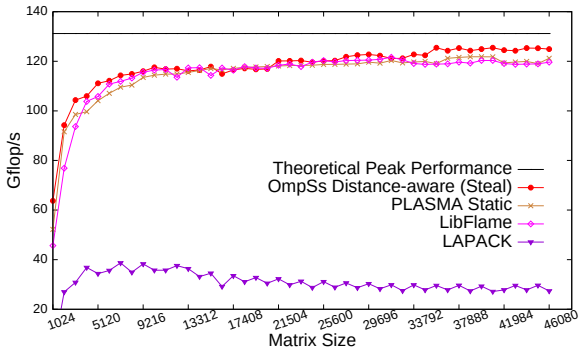
d) 16 sockets (128 threads)

e) 32 sockets (256 threads)

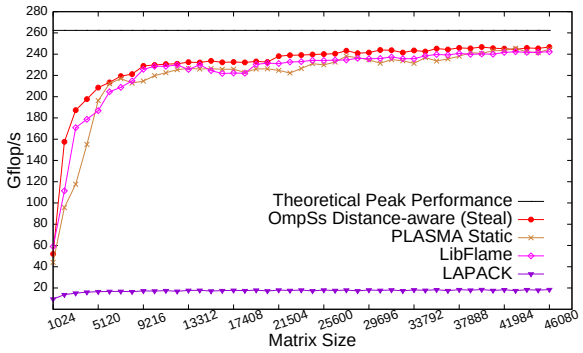
Figure 9. Performance comparisons against existing Cholesky factorization implementations on system (C)

matrix inversion algorithm. Developed in the context of the `OmpSs` framework, this new systematic scheduling policy approach allows the `OmpSs`-enabled code to maintain strong similarity to its original sequential version.

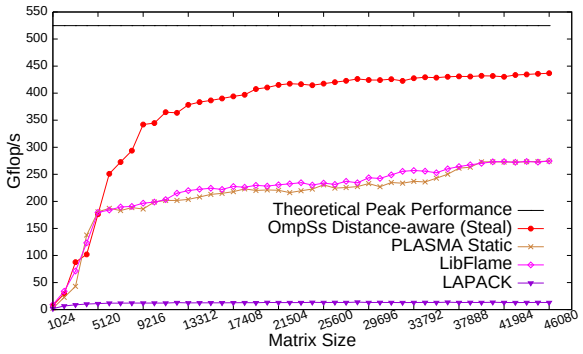
One of the challenges we faced is that the distance between NUMA nodes provided by the operating system does not proportionally translate into access times. A future refinement would be to compute an accurate distance matrix offline and provide that information to the runtime. The Portable Hardware Locality library has facilities to export the system topology information in an XML file, modify the topology information (in our case, the node distance), and use the modified XML file as the topology information. With this, we could relax the conditions for work stealing and allow stealing from non-adjacent nodes if the access time is feasible. Another way would be to modify the System Locality Information Table in the BIOS, but this is usually difficult in production supercomputers.



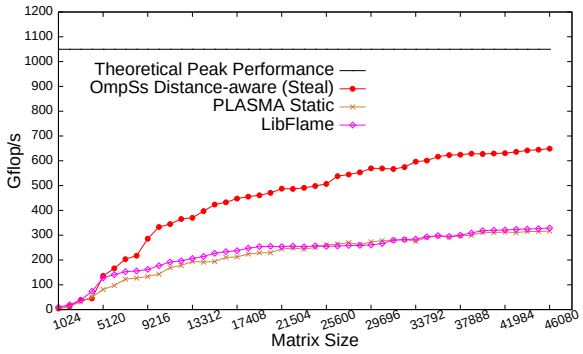
a) 2 sockets (16 threads)



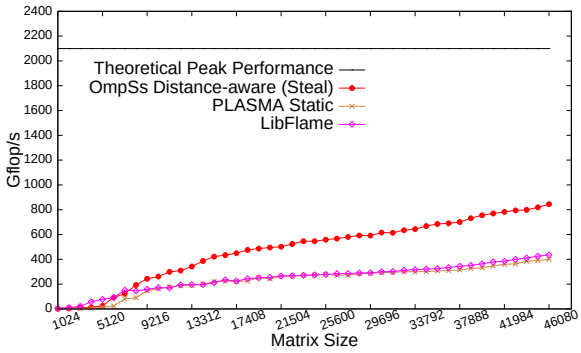
b) 4 sockets (32 threads)



c) 8 sockets (64 threads)



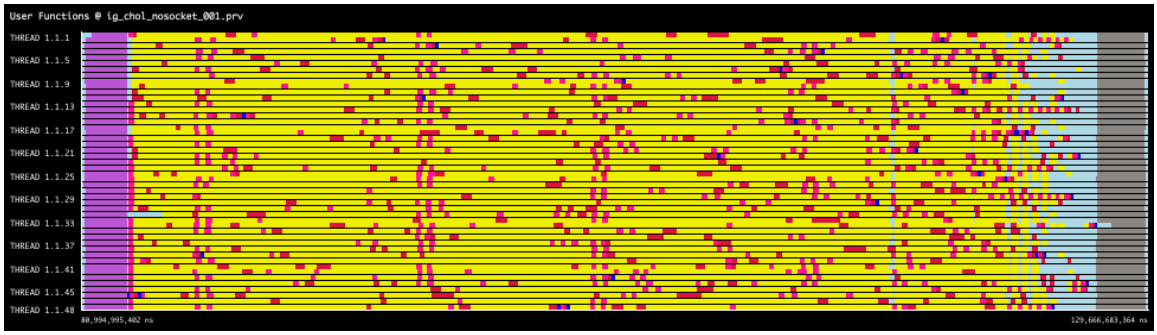
d) 16 sockets (128 threads)



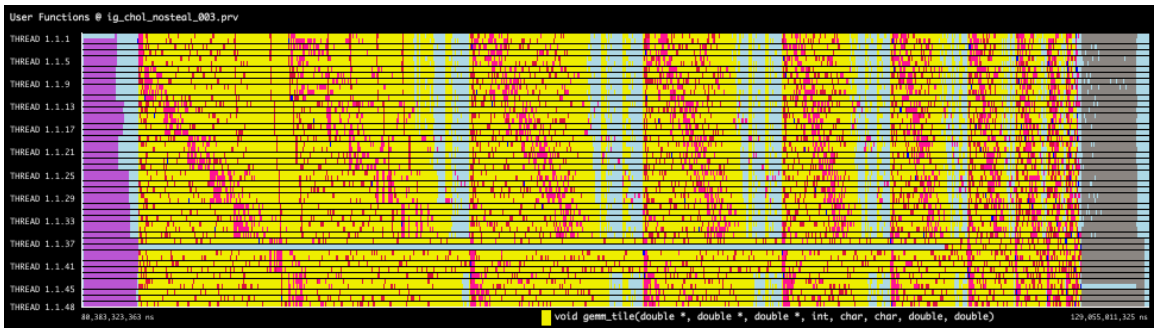
e) 32 sockets (256 threads)

Figure 10. Performance comparisons against existing symmetric matrix inversion implementations on system (C)

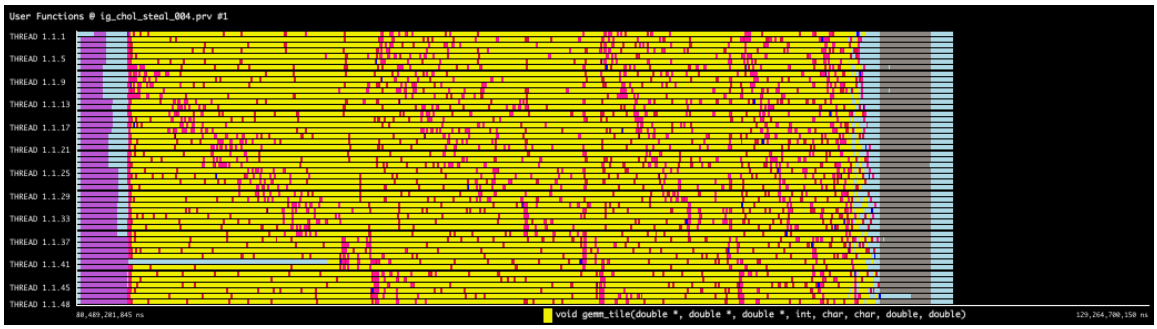
Other tasks in dense linear algebra are generally combinations of routines of similar time-varying load character and easier-to-handle regular workloads. The policy demonstrated herein should be applicable to such general tasks on multicore NUMA architectures, with benefits proportional to the fraction of dynamically varying workload. Many high profile computational tasks beyond dense linear algebra, such as sparse linear algebra, adaptive algorithms for partial differential equations, and complex physics/multiphysics simulations should also be amenable to performance improvements through the same philosophy, if not identical heuristics. Beyond multicore NUMA, multi-GPU systems and hybrid architectures introduce trade-offs between load balance and data locality that will require locality-aware work stealing. We believe that the mechanisms of `OmpSs` and similar programming models possess great potential in practically extending the performance portability of scientific simulation.



a) Distance-oblivious



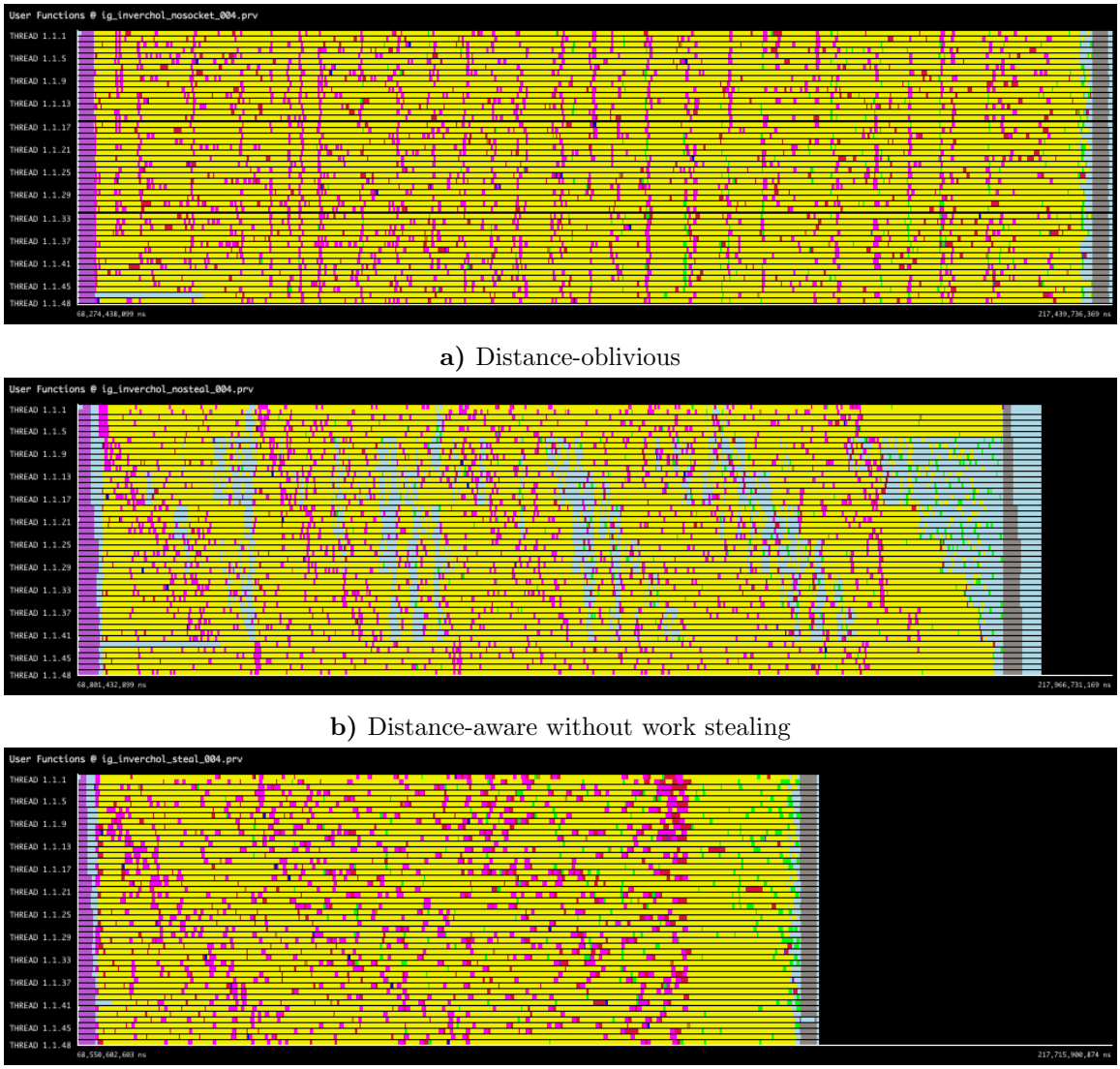
b) Distance-aware without work stealing



c) Distance-aware with work stealing

Figure 11. Traces of the Cholesky factorization using various scheduling policies on system (B)

The authors would like to thank the National Institute for Computational Sciences for granting us access on the Nautilus system. The KAUST authors acknowledge support of the Extreme Computing Research Center. The BSC-affiliated authors thankfully acknowledges the support of the European Commission through the HiPEAC-3 Network of Excellence (FP7-ICT 287759), Intel-BSC Exascale Lab and IBM/BSC Exascale Initiative collaboration, Spanish Ministry of Education (FPU), Computación de Altas Prestaciones VI (TIN2012-34557), Generalitat de Catalunya (2014-SGR-1051) and the grant SEV-2011-00067 of the Severo Ochoa Program.



a) Distance-oblivious

b) Distance-aware without work stealing

c) Distance-aware with work stealing

Figure 12. Traces of the symmetric matrix inversion algorithm using various scheduling policies on system (B)

References

1. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.4.5*, November 2012.

2. A. Charara, H. Ltaief, D. Gratadour, D. Keyes, A. Sevin, A. Abdelfattah, E. Gendron, C. Morel and F. Vidal. Pipelining Computational Stages of the Tomographic Reconstructor for Multi-object Adaptive Optics on a Multi-GPU System. *SC '14: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 262–273, 2014.

3. E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg. Towards an Efficient Tile Matrix Inversion of Symmetric Positive Definite Matrices on Multicore Architectures. *High Performance Computing for Computational Science VECPAR 2010*, 6449:129–138, 2011.

4. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

5. E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 3rd edition, 1999.

6. E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Ortí. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 154–167, Berlin, Heidelberg, 2009. Springer-Verlag. DOI: 10.1007/978-3-642-02303-3_13.

7. L. Suzan Blackford, J. Choi, A. Cleary, E. F. D'Azevedo, J. W. Demmel, I. S. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

8. F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P. Wacrenier, and R. Namyst. Structuring the Execution of OpenMP Applications for Multicore Architectures. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2010. DOI: 10.1109/IPDPS.2010.5470442.

9. F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italie, February 2010. DOI: 10.1109/PDP.2010.67.

10. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009.

11. E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix Out-of-order Scheduling of Matrix Operations for SMP and Multi-core Architectures. In *SPAA '07: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 116–125, New York, NY, USA, 2007. ACM. DOI: 10.1145/1248377.1248397.

12. Q. Chen, M. Guo, and H. Guan. LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-core Architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 3–12, New York, NY, USA, 2014. ACM. DOI: 10.1145/2597652.2597665.

13. J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, C. Xuebin, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, J. Zhong, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka,

P. Messina, P. Michielse, B. Mohr, M. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The International Exascale Software Project Roadmap. *International Journal of High Performance Computer Applications*, 25(1):3–60, February 2011. DOI: 10.1177/1094342010391989.

14. A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Transactions on Architecture and Code Optimization*, 11(3):1–25, August 2014. DOI: 10.1145/2641764.

15. A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009. DOI: 10.1007/s10766-009-0101-1.

16. K. Faxén. Wool – A Work Stealing Library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, June 2009. DOI: 10.1007/10.1145/1556444.1556457.

17. G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

18. N. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM, 2002.

19. E. Jeannot. Symbolic Mapping and Allocation for the Cholesky Factorization on NUMA Machines: Results and Optimizations. *IJHPCA*, 27(3):283–290, 2013. DOI: 10.1177/0734904113492410.

20. L. Karlsson and B. Kågström. Parallel Two-Stage Reduction to Hessenberg Form Using Dynamic Scheduling on Shared-Memory Architectures. *Parallel Computing*, 2011. DOI: 10.1016/j.parco.2011.05.001.

21. D. E. Keyes. Exaflop/s: The Why and The How. *Comptes Rendus Mcanique*, 339(23):70 – 77, 2011. Le Calcul Intensif.

22. J. Kurzak, A. Buttari, and J. J. Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1–11, September 2008.

23. J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurrency and Computation: Practice and Experience*, 21(1):15–44, 2009.

24. H. Ltaief, J. Kurzak, and J. Dongarra. Parallel Band Two-Sided Matrix Bidiagonalization for Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 21(4), April 2010.

25. A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson. Locality-Aware Task Scheduling and Data Distribution on NUMA Systems. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 156–170. Springer, 2013.

26. G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee. Scheduling of QR Factorization Algorithms on SMP and Multi-Core Architectures. In *PDP*, pages 301–310. IEEE Computer Society, 2008. DOI: 10.1109/PDP.2008.37.

27. R. Yokota, G. Turkiyyah and D. Keyes. Communication Complexity of the Fast Multipole Method and its Algebraic Variants. *International Journal of Supercomputing frontiers and innovations*, 1(1), 2014.

28. Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 172–187, Berlin, Heidelberg, 2010. Springer-Verlag. DOI: 10.1007/978-3-642-13374-9_12.

29. A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.

30. F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti. The `libflame` Library for Dense Matrix Computations. *Computing in Science and Engineering*, 11(6):56–63, November/December 2009. DOI: 10.1109/MCSE.2009.207.

Received March 16, 2015.