Symbolic Mapping and Allocation for the Cholesky Factorization on NUMA machines: Results and Optimizations

Emmanuel Jeannot

Symbolic Mapping and Allocation for the Cholesky Factorization on NUMA machines: Results and Optimizations

Emmanuel Jeannot
Inria Bordeaux Sud-Ouest, LaBRI, France
Email: emmanuel.jeannot@inria.fr

April 8, 2013

Abstract

We discuss some performance issues of the tiled Cholesky factorization on non-uniform memory access-time (NUMA) shared memory machines. We show how to optimize thread and data placement in order to achieve performance gains up to 50% compared to state-of-the-art libraries such as PLASMA or MKL.

1 Introduction

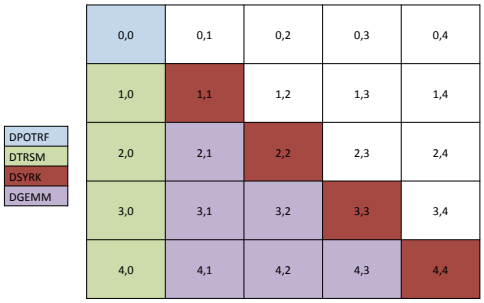
Nowadays, parallel shared memory machines provide a unified view of the memory. A multi-threaded program can be executed on all the cores of the machine using all the memory available. However, due to the memory hierarchy (node, memory banks, cache), the access time to a memory page by a thread depends on both the location of this thread and the page. Therefore, these machines are often called non-uniform memory access-time (NUMA) to account for these effects. Hence, despite the fact that a process (and its own threads) has the illusion of a flat address space, thread placement, data placement and data movement may have a huge impact on the overall performance of the application.

In this paper, we study the tiled version of the Cholesky factorization on such NUMA machines. We show that a simple data flow analysis of the code can provide a relevant placement of the threads and the tiles. Then, we study how threads need to be grouped according to the topology of the machine and we demonstrate that grouping threads by memory nodes has a huge impact on the performance especially for large matrices. Last, we study the conversion of the data storage from the standard LAPACK format to the tiled format. We show that the way a matrix is loaded into memory has a huge impact when converting the format. At the end, the proposed optimizations result in a gain of up to 50% for some matrices compared to PLASMA a state-of-the-art implementation of the Cholesky factorization.

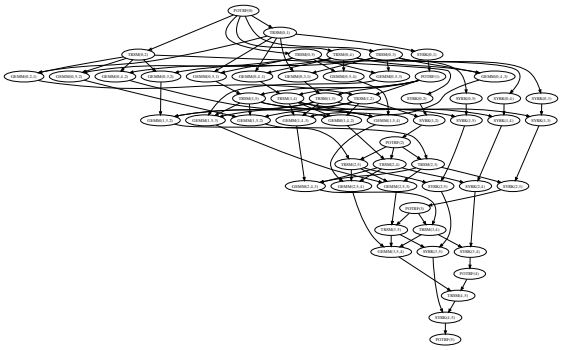
The paper is organized as follows: first, we shortly describe the Cholesky factorization in Section 2. Then we show how to statically analyze the code and automatically determine the thread placement in Section 3. Then, in Section 4, we discuss the execution of the mapping according to the topology of the machine. Finally, we examine the conversion of the LAPACK format to the tiled format in Section 5 before concluding in Section 6.

2 The Cholesky Factorization

The Cholesky factorization takes a symmetric positive definite matrix A as input and finds a lower triangular matrix L such that $A = LL^T$.



(a) Kernels modifying each tile for $k = 1$ and $T = 5$



(b) Task graph describing the dependencies between the kernels of the Cholesky factorization for $T = 6$

Figure 1: Cholesky factorization Kernels and Task Gaph

In Algorithm 1, we depict the tiled version of the algorithm. The matrix is decomposed in $T \times T$ square tiles where $A[i][j]$ is the tile of row i and column j. At each step k (see Fig. 1(a)), we perform a Cholesky factorization of the tile on the diagonal of panel k (DPOTRF kernel[1]). Then, we update the remaining of the tiles of the panel using triangular solve (DTRSM kernel). Then, we update the trailing sub-matrix using the DSYRK kernel for tiles on the diagonal and matrix multiply (DGEMM kernel) for the remaining tiles.

Algorithm 1: Tiled Version of the Cholesky Factorization

```
1 for k = 0...T − 1 do
2     A[k][k] ← DPOTRF(A[k][k])
3     for m = k + 1...T − 1 do
4         A[m][k] ← DTRSM(A[k][k], A[m][k])
5     for n = k + 1...T − 1 do
6         A[n][n] ← DSYRK(A[n][k], A[n][n])
7         for m = n + 1...T − 1 do
8             A[m][n] ← DGEMM(A[m][k], A[n][k], A[m][n])
```

The advantage of the tiled version is that it features a lot of parallelism. For instance, when $T = 6$ we have the task graph depicted in Figure 1(b) where each node is a kernel and each directed edge describes the data dependencies between kernels.

3 Static Analysis of the Cholesky Factorization

3.1 Parameterized Task Graph of the Cholesky Factorization

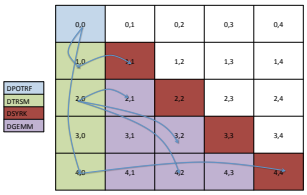
The task graph displayed in Fig. 1(b) seems to express a lot of different dependencies. However, a careful look at the way kernels depend on each-other exhibits only 8 different kinds of dependencies. We therefore can find a *parameterized task graph* (PTG) that is a compact and symbolic representation of the task graph. The parameterized task graph model was first proposed by Cosnard and Loi in [1] for automatically building task graphs. It uses parameters that can be instantiated for building the corresponding task graph. In our case, we have only one parameter: T.

A PTG is composed of rules formally defined as follows. Let: T_a and T_b be two generic tasks with iteration vectors $\vec{u}$ and $\vec{v}$; Let D be a data exchanged between T_a and T_b and $\vec{y}$ be a vector of same dimension than D; P a parameterized polyhedron. A rule as the following form:

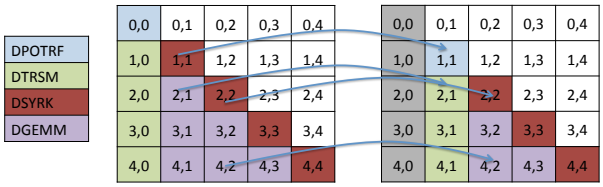
$$T_a(\vec{u}) \rightarrow T_b(\vec{v}) : D(\vec{y})|P$$

This rule reads: " for all $\vec{u}$ $\vec{v}$ and $\vec{y}$ in polyhedron P task $T_a(\vec{u})$ sends data $D(\vec{y})$ to task $T_b(\vec{v})$.

[1] In this paper, kernel names are prefix by D to account for double precision computation. However, this work applies to any other precision: simple, complex, etc.



(a) Some dependencies between kernels for the same iteration of the Cholesky factorization



(b) Example of the four types of dependencies between kernels between 2 consecutive iterations of the Cholesky factorization

Figure 2: Cholesky Factorization Kernel Dependencies

For the Cholesky factorization, we have four generic tasks that correspond to the 4 kernels (DPOTRF, DSYRK, DTRSM and DGEMM) and eight rules. The four rules come from dependencies that occur during the same iteration. Figure 2(a), describes some of such dependencies. This corresponds to the following rules (with $T \geq 1$):

R_1: DPOTRF$(k) \rightarrow$ DTRSM$(k, j) : A[k][k]|\{0 \leq k \leq T-1; k+1 \leq j \leq T-1\}$

R_2: DTRSM$(k, n) \rightarrow$ DSYRK$(k, n) : A[n][k]|\{0 \leq k \leq T-1; k+1 \leq n \leq T-1\}$

R_3: DTRSM$(k, n) \rightarrow$ DGEMM$(k, j, n) : A[n][k]|\{0 \leq k \leq T-1; k+1 \leq n \leq T-1; n+1 \leq j \leq T-1\}$

R_4: DTRSM$(k, n) \rightarrow$ DGEMM$(k, n, j) : A[n][k]|\{0 \leq k \leq T-1; k+1 \leq n \leq T-1; k+1 \leq j \leq n-1\}$

We have also four rules between iteration k and iteration $k+1$ as described in Fig. 2(b). This corresponds to the following rules (with $T \geq 1$):

R_5: DSYRK$(k, m) \rightarrow$ DPOTRF$(k+1) : A[m][m]|\{0 \leq k \leq T-1; m = k+1\}$

R_6: DSYRK$(k, m) \rightarrow$ DSYRK$(k+1, m) : A[m][m]|\{0 \leq k \leq T-1; k+2 \leq m \leq T-1\}$

R_7: DGEMM$(k, m, n) \rightarrow$ DTRSM$(k+1, n) : A[n][m]|\{0 \leq k \leq T-2; k+2 \leq n \leq T-1; m = k+1\}$

R_8: DGEMM$(k, m, n) \rightarrow$ DGEMM$(k+1, m, n) : A[n][m]|\{0 \leq k \leq T-2; k+2 \leq n \leq T-1; k+2 \leq m \leq n-1\}$

Such rules can be automatically found by statically analyzing the sequential code with compiler tools such as PlusPyr of Cosnard and Loi [2] or DAGuE from Bosilca et al. [3].

To obtain such rules from a sequential program, the code must have static control (see [4]). Many compute-intensive kernels found in the literature have a static control. This is the case for the QR factorization and the LU factorization. See [5] for other examples.

3.2 Static Data Allocation and Kernel Mapping

In this section, we propose an algorithm called *SMA* (Symbolic Mapping and Allocation). SMA takes a PTG as input and outputs a mapping of the data and an allocation of the tasks in order to reduce communication costs while keeping parallelism.

3.2.1 Overview of SMA

In our previous work [6], we have proposed an algorithm, called SLC (Symbolic Linear Clustering), for scheduling statically PTG, SMA is directly inspired from SLC.

SMA finds, given a PTG, a mapping function (a clustering) for each generic task: a *cluster id* such that each task with the same id will be mapped on the same processor. This function depends only on the parameters of the program, the iteration vector of the generic task and the number of available processors for execution. The result is independent of the parameter value and therefore of the instantiated task graph. The memory gain is double: no full task graph is required and the schedule has a size proportional to the number of generic tasks.

Staring from a PTG, SMA is decomposed in the following steps (the first one being directly inspired from SLC):

1. *Extracting bijection rules.* We analyze the communication rules in order to extract *bijection rules*. Bijection rules describe point-to-point communications and are those that will be

part of the clustering. In the Cholesky case, we have three broadcasts (rule R_1, R_3 and R_4) and five bijection rules R_2 and R_5 to R_8,

2. *Selecting Non-Conflicting Rules.* Given a set of bijection rules this step consists in selecting some of them in order to guaranty that, for all parameter values, the selected rules will always form a cluster with no join or fork operations. In our case, rule R_2 and rule R_6 are in join conflict. Indeed, the same DSYRK kernel receives a tile from a DTRSM and from a DSYRK.

3. *Computing the symbolic allocation.* The previous step does not always yield to a unique solution. In order to reduce the solution space, we can enforce the owner-compute rule. The other advantage of the owner-compute rule is a better cache reuse in the context of shared memory machine. Based on the clustering and the owner-compute rule, we can construct a function that, once parameter values are known, computes the cluster number of a given task obeying these constraints.

3.2.2 Symbolic Mapping and Allocation

SMA works as follows:

1. Let $\mathcal{Z}$ a set of non conflicting rules and $\vec{p}$ the vector of parameters. Each rule is of the form: $: T_a(\vec{u}) \longrightarrow T_b(\vec{v}) : D(\vec{y})|P$

2. We are going to build a clustering function $\kappa(T_a, \vec{u})$ and a data mapping function $\mu(D, \vec{y})$. In order the problem to be tractable, we impose that both functions are affine [7]: $\mu(D, \vec{y}) = \vec{\alpha_D}\vec{y} + \beta_D + \vec{\gamma_D}\vec{p}$ and $\kappa(T_a, \vec{u}) = \vec{\alpha_a}\vec{u} + \vec{\beta_a} + \gamma_a\vec{p}$. where $\vec{p}$ is the vector of parameters ($\vec{p} = (T)$ for Choleky) .

3. We first build equations for data mapping. When a rule sends a data it is because it has updated it. Therefore, the mapping of the updated data must match the clustering of the task. Therefore: $\vec{\alpha_a}\vec{u} + \beta_a + \vec{\gamma_a}\vec{p} = \vec{\alpha_D}\vec{y} + \beta_D + \vec{\gamma_D}\vec{p}$

4. Second, we built equations for the task allocation. If a rule is selected, this means that the sending task is going to be placed on the same cluster than the receiving task: $\kappa(T_a, \vec{u}) = \kappa(T_b, \vec{v})$. Therefore: $\vec{\alpha_a}\vec{u} + \beta_a + \vec{\gamma_a}\vec{p} = \vec{\alpha_b}\vec{v} + \beta_b + \vec{\gamma_b}\vec{p}$

5. We build the equations for each rules leading to a linear system. The solution of the system defines the mapping of the data and the tasks.

3.3 Example on the Cholesky factorization PTG

We have two sets of non-conflicting rules $\{R_2, R_5, R_7, R_8\}$ and $\{R_5, R_6, R_7, R_8\}$. The first set leads to a trivial solution whith no parallelism. The second set of rules leads to the diagonal solution that will be used in the remaining of the paper.

Mapping	Diagonal	block-cyclic	priority
$\mu(A[i][j])$	$i + j$	$((j \bmod Q) \times P + i \bmod P)$	high
$\kappa(\text{DPOTRF}(k))$	$2 \times k$	$((k \bmod Q) \times P + k \bmod P)$	high
$\kappa(\text{DSYRK}(k, n))$	$2 \times n$	$((n \bmod Q) \times P + n \bmod P)$	high
$\kappa(\text{DTRSM}(k, n))$	$k + n$	$((n \bmod Q) \times P + k \bmod P)$	high
$\kappa(\text{DGEMM}(k, m, n))$	$m + n$	$((n \bmod Q) \times P + m \bmod P)$	low

The above table also presents the block-cyclic mapping on $P \times Q$ processor array. DGEMM are tasks with low priority and therefore are executed by the runtime system only when no high priority tasks are ready.

4 Efficient Mapping of the Kernels Taking into Account the Machine Topology

4.1 Grouping Threads, Clusters and Tiles

To execute the application, we have developed a simple runtime system, which works as follows. We have N_{core} cores. Tasks (that execute the Cholesky kernels) are executed by threads. The number of threads (N_{thread}) is specified by the user ($N_{\text{thread}} \leq N_{\text{core}}$). This number is given just before the execution. A given thread is bound to a given core. Task $T_a(\vec{u})$ is mapped to cluster $\kappa(T_a(\vec{u}))$, as computed by the SMA algorithm. Cluster i is cyclically put in group $i \bmod G$. Each group has its own logical memory and hence tiles are also mapped cyclically (i.e. tile $A[i][j]$ is bound to group $\mu(A[i][j]) \bmod G$. Based on that, threads of a given group only execute the tasks belonging to the clusters bound to that specific group.

The remaining question is how to group clusters, cores and threads and what should be the size of G? On a modern NUMA shared-memory machine, we can target three levels of the memory hierarchy. The *machine* level, the *node* level and the *core* level.

4.2 Experimental Evaluation

In order to compare the different ways of grouping threads, cores and clusters we have tested the three levels experimentally on different settings. A representative result is depicted in Fig. 3. This is done one a 160 cores NUMA machine composed of 20 nodes of one eight cores socket Intel Nehalem Eagleton (E7–8837) at 2.67GHz[2].

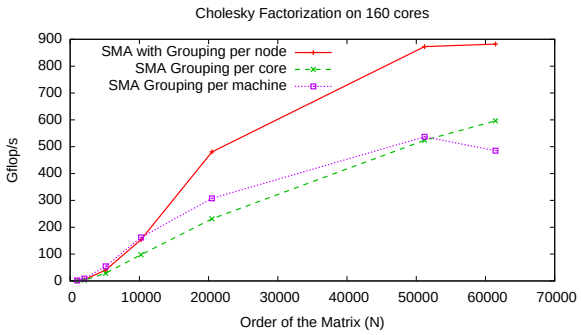


Figure 3: Comparison of different grouping strategies for the Cholesky factorization on a 160 cores, 20 nodes machine. Grouping per machine, node and core respectively corresponding to G=1, 20 and 160.

In Fig. 3 we show the performance in Gflop/s versus the matrix size (N). Results show that grouping threads and clusters by NUMA memory node is the most efficient strategy. This helps to take into account the memory hierarchy: we have one pool of 8 threads per node that execute the clusters mapped to this node (group). Thanks to this, we have two levels of parallelism that efficiently take into account the memory hierarchy of the machine. With one group for the whole machine, the performance is similar for small size matrices up to 10240. But, when the size of the input increases, all the parallelism generated by the application cannot be efficiently handled by such a flat view of the architecture and the performance degrades compared to the node grouping. The core grouping shows an opposite behavior. The efficiency, when compared to the node grouping, increases as the generated parallelism increases: when a few tasks/clusters are available, not all cores can be kept busy.

4.3 Comparison with MKL and PLASMA

The Intel Math Kernel Library (MKL) [8] is a multithreaded library that provides many linear algebra kernels (BLAS, LAPACK, etc.). In this work, we use the MKL version shipped with the Intel C compiler version 11.1-075.

[2]Experiments presented in this paper were carried out using the PLAFRIM experimental testbed being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Rgional d'Aquitaine, FeDER, Universit de Bordeaux and CNRS (see`https://plafrim.bordeaux.inria.fr/`)

PLASMA [9] is a multithreaded library based on task parallelism such as the work presented here. PLASMA offers two versions of the Cholesky factorization, one using the *LAPACK format* and that does not require data format conversion and one using the *tiled format* which is more efficient but requires data format conversion. In this paper, we use only the later version. PLASMA features its own runtime system called QUARK [10], which is far more developed than the simple one presented above. In these experiments, we use PLASMA version 2.4.5 [3]. Nevertheless, PLASMA and our simple runtime both relies on the same sequential BLAS kernels of the MKL 11.1-075.

In Fig. 4, we present the performance in Gflop/s versus the matrix size for the MKL version of Cholesky, the PLASMA version and the SMA version. The PLASMA version being more efficient for tile size of NT=256 and the SMA version being more efficient for the tile size of NT=512, we present results for different tile sizes of each case (Actually, we tested several tile sizes (64, 128, 256 512), and we present only the best cases). For MKL, the blocking is handled automatically.

Last, we also present results with the static scheduling strategy of PLASMA as the dynamic strategy is constantly outperformed by the static one.

We also include in the timing, the format conversion between the LAPACK format (matrix store in row major) to the tiled format (each tile is stored consecutively in memory). We think it is important to measure the format conversion as most existing programs use the LAPACK format. However, this is in favor of MKL as it uses LAPACK format natively.

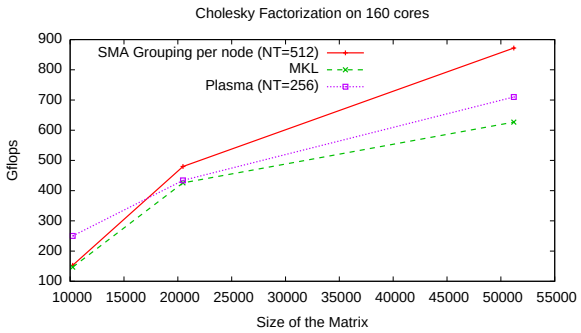


Figure 4: Comparison of SMA versus PLASMA and MKL for the Cholesky factorization on a 160 cores, 20 nodes machine (double precision, $N \leq 51200$). Format conversion included

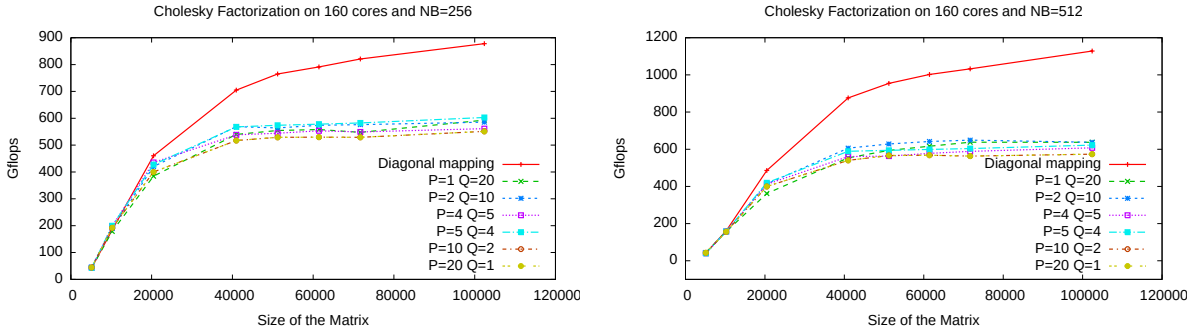
Results show that our proposed version is far more efficient than the two reference libraries. For $N = 51200$, the SMA version reaches 872 Gflop/s while MKL and PLASMA respectively reach 637 and 710 Gflop/s. The main difference between SMA and these two libraries is the NUMA awareness, none of these libraries take into account the memory hierarchy for allocating threads and managing the memory.

4.4 Comparison with block-cyclic mapping

Block-cyclic mapping with process layout $P \times Q$ is presented in section 3.3. Here, we present experiments comparing the SMA result which provides a *diagonal* mapping against the *block-cyclic* mapping with different values of P and Q. As we have 20 nodes, we need to have $P \times Q = 20$. Special cases are when $P = 1$ (cyclic mapping by column) and $Q = 1$ (cyclic mapping by row). results are depicted in Fig. 5.

We see that the diagonal mapping outperforms all combinations of P and Q. This is due to the fact that with SMA, almost all point-to-point communications are suppressed which is not the case with block-cyclic mapping.

[3] Early experiments with version 2.5.0 do not show significant difference

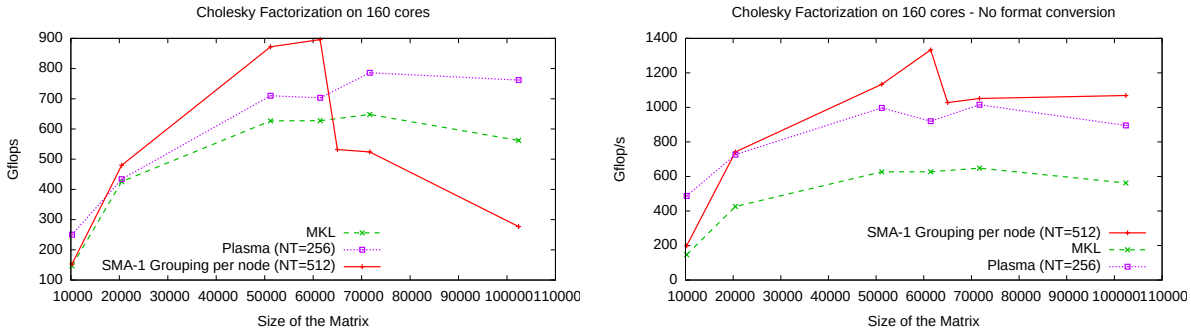


(a) NT=256 (b) NT=512

Figure 5: Comparison of the SMA/ diagonal mapping with block-cyclic mapping for different processor layout and different tile size.

5 LAPACK to Tiled Format Conversion

In Fig. 6(a), we show the performance in Gflop/s of the original version of SMA (called SMA-1 in this section) versus PLASMA and MKL when we include the format conversion timing and for matrix size up to 102400 (previous section was for $N \leq 51200$).



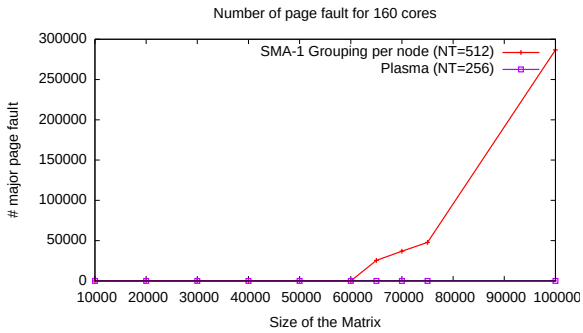
(a) Format conversion included (b) factorization time only: format conversion excluded

Figure 6: Comparison of SMA versus PLASMA and MKL for the Cholesky factorization on a 160 cores, 20 nodes machine (double precision).

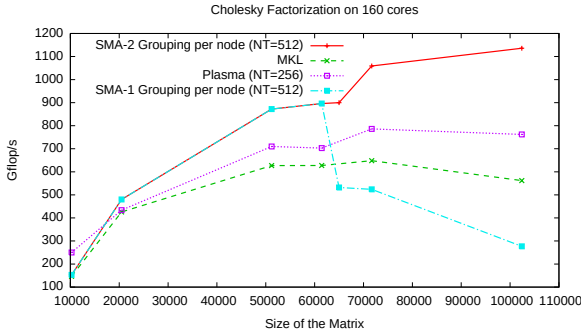
We see here a big performance drop for SMA-1 around $N = 64000$ and beyond while PLASMA and MKL continue to have roughly the same performance. When we analyze the timing, we see that a lot of time is lost by SMA-1 for format conversion. Indeed, if we plot the raw performance of SMA-1 and PLASMA (Cholesky factorization without format conversion) we see that the performances are far better as shown in Fig. 6(b). The increase of performance is due to the fact that such conversion can take up to 25% of the overall execution time. This is not the case for MKL as it only uses the LAPACK format (there is no conversion). We also see that at around $N = 64000$ the performance drop for SMA-1 is much smaller when we do not take into account format conversion. This means that the performance lost is mainly due to this conversion step.

To better understand what is going on at $N = 64000$, we have measured the number of major page faults made by the system for PLASMA and SMA-1 when N is increasing using the `/usr/bin/time` UNIX command. Results are shown in Fig. 7(a).

We see that, for SMA-1, the number of page faults suddenly increases at the same point as its performance decreases. Moreover, there is no page fault for the PLASMA version. This means that after $N = 64000$, the system does not have enough memory for the SMA-1 version



(a) Number of system page-fault when increasing the matrix size. SMA-1 vs. PLASMA for the Cholesky factorization (double precision)



(b) Final result. SMA version 1 (no data interleaving) and SMA version 2 (page-fault aware) vs. PLASMA and MKL.

Figure 7: Performance analysis and final result (format conversion included).

and start using the disk as a secondary storage.

The reason is the following: despite the fact that the whole machine has 600 GB of memory each of the 20 nodes has 30 GB. When $N \geq 64000$ the size of the matrix requires more than $64000^2 \times 8 = 3.2710^{10}$ bytes or 30.5 GB. In our code, the matrix is allocated and filled by a single thread using the `malloc` function of the standard C library. This means that all the pages of the matrix are put on the memory node of this thread. Starting for $N \approx 64000$, the memory is not large enough to store the whole matrix and the system begins to swap. The problem increases with the matrix size as shown in Fig. 6 and 7(a).

To solve this problem, there exist several solutions. First, as done in PLASMA, the filling of the matrix can be multithreaded: pages will be scattered in the memory in the same way than threads are. An other solution consists in forcing the allocation of the pages across memory banks. This can be done by using `numa_alloc_interleaved` function of the *NUMA policy library* available in most systems. By doing so, the new version of SMA (SMA-2) does not exhibit page faults anymore and have very good performance even for large matrix sizes as shown in Fig. 7(b). We see that the page-fault aware version of SMA is able to continue to increase its performance for large matrix sizes as opposed to the first version. Moreover, the gain against the other reference libraries (MKL and PLASMA) is very large, up to 74.7% for MKL and 49.1% for PLASMA.

6 Conclusion

NUMA parallel machines are fairly simple to program as they offer a flat view of the memory. However, data placement, data movement and thread placement have a huge impact on the performance as shown in this paper, where we have studied the tiled version of the Cholesky factorization.

The paper is decomposed in three parts. In the first part, the dependencies between the four Cholesky kernels, expressed as a parameterized task graph, are statically analyzed. We have proposed a new static algorithm to perform symbolic data allocation and kernel mapping called SMA (symbolic mapping and allocation) inspired from our previous work.

In the second part, we have implemented a simple runtime system as a proof-of-concept. We have shown that grouping them by node is more efficient than by core or on the whole machine. Moreover, despite its simple implementation but thanks to the NUMA-awareness of the grouping, this runtime system is able to outperform the MKL and PLASMA tiled versions.

However, we have seen a degradation of performance for large matrix sizes.

This performance issue is studied in the third part of the paper. We have seen that the problem comes from the way memory pages are allocated onto memory banks. A careful allocation of the memory allows to solve the problem and the final version of our runtime (SMA-2) does not suffer from performance degradation.

7 Acknowledgments

We would like to thanks Guillaume Mercier, Brice Goglin, Emmanuel Agullo and Georges Bosilca for very helpful discussions about ideas and results of this paper.

References

[1] M. Cosnard and M. Loi, "Automatic Task Graph Generation Techniques," *Parallel Processing Letters*, vol. 5, no. 4, pp. 527–538, 1995.

[2] ——, "A Simple Algorithm for the Generation of Efficient Loop Structures," *International Journal of Parallel Programming*, vol. 24, no. 3, pp. 265–289, Jun. 1996.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, ""dague: A generic distributed dag engine for high performance computing"," *Parallel Computing*, vol. 38, no. 1-2, pp. 27–51, 2012.

[4] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[5] M. Cosnard, E. Jeannot, and T. Yang, "SLC: Symbolic Scheduling for Executing Parameterized Task Graphs on Multiprocessors," in *International Conference on Parallel Processing (ICPP'99)*, Aizu Wakamatsu, Japan, Sep. 1999.

[6] ——, "Compact Dag Representation and its Symbolic Scheduling," *J. of Parallel and Dist. Comp/*, vol. 64, no. 8, pp. 921 – 935, Aug. 2004.

[7] P. Feautrier, "Toward Automatic Distribution," *Parallel Processing Letters*, vol. 4, no. 3, pp. 233–244, 1994.

[8] R. Intel, "Intel math kernel library reference manual," Tech. Rep. 630813-051US, 2012. [Online]. Available: http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf

[9] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan., "Plasma users guide," University of Tennessee, Innovative Computing Laboratory, Technical Repport, 2010.

[10] 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.