Performance Analysis and Optimization of the Tiled Cholesky Factorization on NUMA Machines

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

***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 placement and data placement in order to achieve performance gain up to 50% compared to state-of-the-art libraries such as Plasma or MKL.**

I. 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 (cache, node, memory bank), the access time of a memory page by a thread depends on 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. It is now common to see NUMA machines featuring cache coherency with more than 100 cores. 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 big impact on the whole performance of the application.

In this paper we study the tiled version of the Cholesky Factorization on such NUMA machine, we show that a simple data flow analysis of the code can provide a good 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 show that grouping threads by memory node have a huge impact on the performance especially for large matrix. Last, we study the conversion of the data storage from the standard LAPACK format to the tiled format. We show that the way the 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 II. Then we show how to statically analyze the code and automatically determine the thread placement in Section III. Then, in Section IV, 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 V before concluding in Section VI.

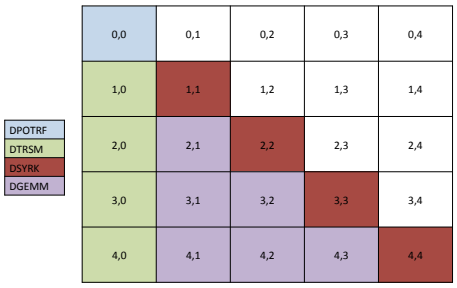


Figure 1. Kernels modifying each tile for $k = 1$ and $T = 5$

II. The Cholesky Factorization

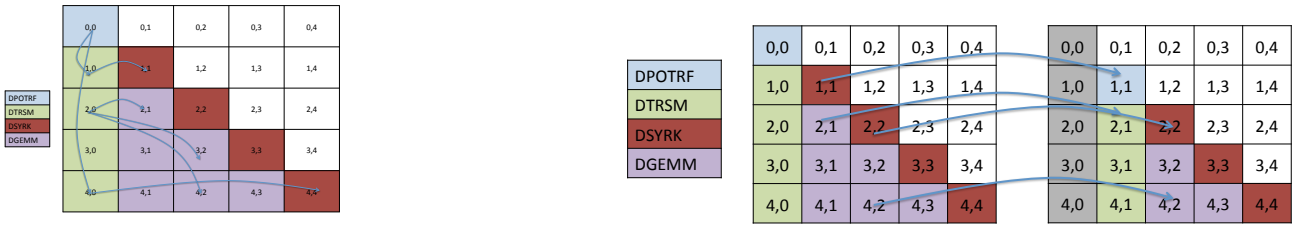
Algorithm 1: Tiled Version of the Cholesky Factorization

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

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) 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.

[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

The advantage of the tiled version is that it has a lot of parallelism. For instance, when $T = 6$ we have the task graph depicted in Figure 3 where each node is a kernel and each directed edge describes the data dependencies between kernels. More generally, for a given T this tiled version rapidly expresses as much parallelism as $O(T^2)$ which is usually much greater than the number of cores.

III. Static Analysis of the Cholesky Factorization

A. Parameterized Task Graph of the Cholesky Factorization

The task graph displayed in Fig. 3 seems to express a lot of different dependencies. However, a careful look at the way kernels depend on each-other exhibits only 8 different kind 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 has first been 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 mainly composed of *communication rules*. *Communication rules* are symbolic descriptions of task graph edges. There are two types of communication rules. The first type is emission rules that describe data sent by tasks. The second type is reception rules that describe input data of tasks. We can automatically transform a reception rule into an emission rule. Hence, we give only the formal definition of an emission rule. Let: T_a and T_b two generic tasks with iteration vector $\vec{u}$ and $\vec{v}$; D a data exchanged between T_a and T_b and $\vec{y}$ a vector of same dimension than D; P a parameterized polyhedron. An emission rule as the following form:

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

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

For the Cholesky factorization, we have 4 generic tasks that correspond to the 4 kernels (DPOTRF, DSYRK, DTRSM and DGEMM) and 8 rules. 4 rules come from dependencies that happen 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) : $A[n][k]|\{0 \leq k \leq T-1; k+1 \leq n \leq T-1; k+1 \leq j \leq n-1\}$

For instance, rule R_1 describes the dependencies between, DPOTRF of iteration k and all the DTRSM of the same iteration.

We have also 4 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$(k+1, m, n)$: $A[n][m]|\{0 \leq k \leq T-2; k+2 \leq n \leq T-1; k+2 \leq m \leq n-1\}$

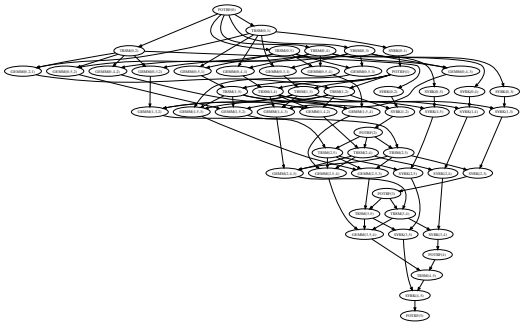


Figure 3. Task graph describing the dependencies between the kernels of the Cholesky Factorization for $T = 6$

For example, rule R_5 describes the dependency between the first DSYRK of the diagonal at iteration k with the DPOTRF of iteration $k+1$

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, teh code must have a static control (see [4]). Many compute-intensive kernels found in the literature have a static control. This is the case of the QR factorization and the LU factorization. See [5] for other examples.

All the above rules have T as parameter, when T is instantiated, a task graph can easily be expanded by applying all the rules starting from DPOTRF(0). For $T = 6$ we obtain the task graph depicted in Fig. 3. However, a PTG is independent of the problem size and we can analyze it in order to provide a mapping of the task onto the resources and the corresponding data allocation for any values of this parameter and hence any size of the problem.

B. Static Data Allocation Mapping of the Kernels

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

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.

The goal of SLC was to built a linear clustering of any instantiated task graphs that can be built from a given parameterized task graph. However, it was designed in the context of coarse-grained task graph which does not apply to the Cholesky PTG as proposed here. Therefore, the proposed clustering is not guaranteed to be linear (discussion about linear clustering and its property can be found in [7]). However, to better take into account the tiled structure of the Cholesky factorization, we have enhanced SLC with data mapping in order to force the owner-compute rule.

Our new algorithm (i.e. SMA) has the following very desirable properties. SMA finds, given a PTG, an allocation function for each generic task. 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 parameter value and hence 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,
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 clustering with no join or fork,
3) *computing the symbolic allocation*. The previous step does not always have 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.

2) Bijection Rules: Bijection rules are rules that describe point-to-point communication. We have a point-to-point communication in a rule $T_a(\vec{u}) \rightarrow T_b(\vec{v}) : D(\vec{y})|P$ if for any $\vec{u}$ we have only one $\vec{v}$. In the Cholesky case we have 3 broadcasts (rule R_1, R_3 and R_4) and 5 bijection rules R_2 and R_5 to R_8. Finding bijection rules automatically is done by a simple analysis of the rule as explained in [6].

3) Conflicting Rules: To obtain a clustering we have to suppress conflicting rules. There can be two types of conflicts:

1) fork conflict: if two rules describe the same sending task instance for different receiving tasks instances,
2) join conflict: if two rules describe the same receiving task instance for different sending tasks instances.

Let us consider two rules: $R_1 : T_a(\vec{u}) \rightarrow T_b(\vec{v})|P_1$ and $R_2 : T_c(\vec{w}) \rightarrow T_d(\vec{z})|P_2$

In order to determine automatically if these rules are in join conflict (the fork conflict case is symmetric) we proceed as follow. We compute the set I which is the intersection between $P_1 \backslash \vec{u}$ (P_1 restraint to $\vec{u}$) and $P_2 \backslash \vec{w}$. If I is empty there cannot be any fork conflict. Otherwise if $T_b = T_d$ and $R_1 \backslash I$ (R_1 restricted to the I set) is different from $R_2 \backslash I$ there is a fork conflict. I, $R_1 \backslash I$ and $R_2 \backslash I$ can be symbolically computed with the omega calculator from the university of Maryland [8].

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. Therefore, most DSYRK have two predecessors if we put these three tasks in the same cluster, the cluster has a join and we lose parallelism. To ensure more parallelism (at the cost of more communication), we have to remove one of the two rules from the set of rules that we use to compute the symbolic allocation. To choose, which rule to remove, we will test the possible cases in conjunction with the computation of the data mapping.

4) Symbolic Task Allocation and Data mapping: In order the data-mapping problem to be tractable, we impose that it

is affine [9]. Let μ the mapping function. We impose that $\mu(A[i][j]) = \alpha_{A,1}i + \alpha_{A,2}j + \beta_A + \gamma_{A,1}T$, where T, the number of tiles, is the only parameter of the problem. More generally, we have $\mu(D, \vec{y}) = \vec{\alpha_D}\vec{y} + \beta_D + \vec{\gamma_D}\vec{p}$, where $\vec{p}$ is the vector of parameters of the program. At the same time, we want to compute the symbolic allocation of the tasks. Here again, we impose an affine mapping: we need to find a function $\kappa(T_a, \vec{u}) = \vec{\alpha_a}\vec{u} + \vec{\beta_a} + \gamma_a\vec{p}$ that gives the cluster number of task $T_a(\vec{u})$ for any valid value of $\vec{u}$ and $\vec{p}$.

The symbolic allocation works as follows:

1) Let $\mathcal{Z}$ a set of non conflicting rules and $\vec{p}$ the vector of parameters.
2) Each rule $R \in \mathcal{Z}$ is of the form: : $T_a(\vec{u}) \longrightarrow T_b(\vec{v})$: $D(\vec{y})|P$
3) We are going to build a clustering function $\kappa(T_a, \vec{u})$ and a data mapping function $\mu(D, \vec{y})$.
4) 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. We denote the sending by $T_a(\vec{u}) \triangleright D(\vec{y})$ and consequently, we have: $\kappa(T_a, \vec{u}) = \mu(D, \vec{y})$. Therefore:
$$\vec{\alpha_a}\vec{u} + \beta_a + \vec{\gamma_a}\vec{p} = \vec{\alpha_D}\vec{y} + \beta_D + \vec{\gamma_D}\vec{p} \quad (1)$$
5) Second, we built equations for the task allocation. If a rule is selected, this means that the sending task is going to be put on the same cluster than the receiving task: $\kappa(T_a, \vec{u}) = \kappa(T_b, \vec{v})$. Therefore:
$$\vec{\alpha_a}\vec{u} + \beta_a + \vec{\gamma_a}\vec{p} = \vec{\alpha_b}\vec{v} + \beta_b + \vec{\gamma_b}\vec{p} \quad (2)$$
6) For all rule in $\mathcal{Z}$, we obtain a system
$$\mathcal{S} : Cb_G = 0$$
where C is obtained by algebraic manipulation of eq. like (1) and (2) and b_G is a vector of α's, β's and γ's.
7) If $\mathcal{S}$ admits a non-trivial solution $b_G \neq \emptyset$, we display this solution.
8) If the only solution is $b_G = \emptyset$, we discard this solution and try the next possible set.

C. Example on the Cholesky factorization PTG

1) Mapping constraints: We have two sets of non-conflicting rules $\{R_2, R_5, R_7, R_8\}$ and $\{R_5, R_6, R_7, R_8\}$, however they both concern the same mapping:

- DPOTRF$(k) \triangleright A[k][k]$: $\alpha_{p,1}k + \beta_p + \gamma_{p,1}T = \alpha_{A,1}k + \alpha_{A,2}k + \beta_A + \gamma_{A,1}T$. This leads to $\alpha_{p,1} = \alpha_{A,1} + \alpha_{A,2}$, $\beta_p = \beta_A$ and $\gamma_{p,1} = \gamma_{A,1}$.
- DSYRK$(k, n) \triangleright A[n][n]$: $\alpha_{s,1}k + \alpha_{s,2}n + \beta_s + \gamma_{s,1}T = \alpha_{A,1}n + \alpha_{A,2}n + \beta_A + \gamma_{A,1}T$. This leads to $\alpha_{s,1} = 0$, $\alpha_{s,2} = \alpha_{A,1} + \alpha_{A,2}$, $\beta_s = \beta_A$ and $\gamma_{s,1} = \gamma_{A,1}$.
- DTRSM$(k, n) \triangleright A[n][k]$: $\alpha_{t,1}k + \alpha_{t,2}n + \beta_t + \gamma_{t,1}T = \alpha_{A,1}n + \alpha_{A,2}k + \beta_A + \gamma_{A,1}T$. This leads to $\alpha s, 1 = \alpha_{A,2}$, $\alpha_{s,2} = \alpha_{A,1}$, $\beta_s = \beta_A$ and $\gamma_{s,1} = \gamma_{A,1}$.
- DGEMM$(k, m, n) \triangleright A[m][n]$:$\alpha_{g,1}k + \alpha_{g,2}m + \alpha_{g,3}n + \beta_g + \gamma_{g,1}T = \alpha_{A,1}m + \alpha_{A,2}n + \beta_A + \gamma_{A,1}T$. This leads to $\alpha_{g,1} = 0$ $\alpha_{g,2} = \alpha_{A,1}$, $\alpha_{g,3} = \alpha_{A,2}$, $\beta_s = \beta_A$ and $\gamma_{g,1} = \gamma_{A,1}$.

Based on the above equations we have:

- $\mu(A[i][j]) = \alpha_{A,1}i + \alpha_{A,2}j + \beta_A + \gamma_A T$
- $\kappa(\text{DPOTRF}(k)) = (\alpha_{A,1} + \alpha_{A,2})k + \beta_A + \gamma_A T$
- $\kappa(\text{DSYRK}(k, n)) = (\alpha_{A,1} + \alpha_{A,2})n + \beta_A + \gamma_A T$
- $\kappa(\text{DTRSM}(k, n)) = \alpha_{A,1}n + \alpha_{A,2}k + \beta_A + \gamma_A T$
- $\kappa(\text{DGEMM}(k, m, n)) = \alpha_{A,1}m + \alpha_{A,2}n + \beta_A + \gamma_A T$

Note that after this step, all the clustering functions are expressed in function of the mapping parameters $\alpha_{A,*}$, β_A and γ_A.

2) Task allocation: We can now add the constraints related to the task allocation. As we have two possible sets of rule, let us first consider rule R_5, R_7 and R_8 that are in common. For rule R_5, this leads to:

$$\begin{aligned} & \kappa(\text{DSYRK}(k, n)) = \kappa(\text{DPOTRF}(k+1)) \text{ and } n = k+1 \\ \Leftrightarrow \quad & (\alpha_{A,1} + \alpha_{A,2})n = (\alpha_{A,1} + \alpha_{A,2})n \end{aligned}$$

Which is always true. Therefore, no constraints are added. Concerning rule R_7, we have:

$$\begin{aligned} & \kappa(\text{DGEMM}(k, m, n)) = \kappa(\text{DTRSM}(k+1, n)) \\ & \text{and } m = k+1 \\ \Leftrightarrow \quad & \alpha_{A,1}m + \alpha_{A,2}n = \alpha_{A,1}n + \alpha_{A,2}m \\ \Leftrightarrow \quad & \alpha_{A,1} = \alpha_{A,2} \end{aligned}$$

Last, for rule R_8: we have $\kappa(\text{DGEMM}(k, m, n)) = \kappa(\text{DGEMM}(k+1, , m, n))$ which is also always true.

We have to decide if we use rule R_2 or rule R_6. If we write the constraints concerning rule R_2 (DTRSM$(k, n) \rightarrow$ DSYRK(k, n) : $A[n][k]|\{0 \leq k \leq T-1; k+1 \leq n \leq T-1\}$) we have:

$$\begin{aligned} & \kappa(\text{DTRSM}(k, n)) = \kappa(\text{DSYRK}(k, n)) \\ \Leftrightarrow \quad & \alpha_{A,1}n + \alpha_{A,2}k = (\alpha_{A,1} + \alpha_{A,2})n \\ \Leftrightarrow \quad & \alpha_{A,2} = 0 \end{aligned}$$

Due to constraints of rule R_7 this imposes also that $\alpha_{A,1} = 0$. In this case, all the data are mapped on the same location and we have no parallelism. Therefore, this solution is not feasible.

If we look at rule R_6 we have:

$$\begin{aligned} & \kappa(\text{DSYRK}(k, n)) = \kappa(\text{DSYRK}(k+1, n)) \\ \Leftrightarrow \quad & (\alpha_{A,1} + \alpha_{A,2})n + \beta_A + \gamma_A T \\ & = (\alpha_{A,1} + \alpha_{A,2})n + \beta_A + \gamma_A T \end{aligned}$$

Which does not add any constraints. Therefore, the simplest solution fulfilling all our constraints is: $\alpha_{A,1} = \alpha_{A,2} = 1$, $\beta_A = 0$ and $\gamma_A = 0$, which leads to the following mapping:

- $\mu(A[i][j]) = i + j$

- $\kappa(\text{DPOTRF}(k)) = 2 \times k$
- $\kappa(\text{DSYRK}(k,n)) = 2 \times n$
- $\kappa(\text{DTRSM}(k,n)) = k + n$
- $\kappa(\text{DGEMM}(k,m,n)) = m + n$

D. Discussion

The SMA algorithm finds a mapping that enforces a diagonal mapping of the tiles onto the resources. Note that only the mapping of DPOTRF and DTRSM depends on the step number. This is correct as at each step k DPOTRF is perform on the k^{th} tile on the diagonal and for DTRSM the panel advances from one column when we go from step k to step $k+1$.

The mapping is the one that reduces the most point-to-point communication while giving a lot of parallelism. Actually, the number of generated clusters is $2T - 1$. Therefore, there can be much more clusters that the number of compute units (For a matrix of size 20480 and a tile size 256 we have $T = 80$). However this might not be sufficient. This is one reason why we will group clusters to node as shown in the next Section.

IV. Efficient Mapping of the Kernels Taking into Account the Machine Topology

A. Grouping Threads, Clusters and Tiles

To execute the application, we have developed a simple runtime system, that 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. Therefore, we have an equivalence between the used cores and threads. Cores/threads are grouped in G groups. Each group is logically responsible of executing a set of clusters. This is done by having a pool of tasks ready to be executed for each group. 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 mod G. Each group has its own logical memory and hence tiles are also mapped to group cyclically (i.e. tile $A[i][j]$ is bound to group $\mu(A[i][j])$ mod G. Based on that, the threads of a given group only execute the tasks belonging to the clusters bound to that specific group.

The question that remains 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.

At the machine level, we have only one group ($G = 1$). The machine is viewed as having only one big memory and tiles are mapped regardless of the memory hierarchy. Each cluster and each task can be executed by any threads.

A node is a component of the machine having its own memory bank(s). When a thread, executed on a given node, accesses a memory page it is faster to access a page on the memory bank of this node than on an other node. Thanks to the mapping and allocation proposed in this paper, building groups at the node level implies that the threads will execute tasks that will write data on the node of the core of this thread. Moreover, as we have many threads per node, we need, as for the machine level, to manage concurrent access to a pool of tasks ready to be executed.

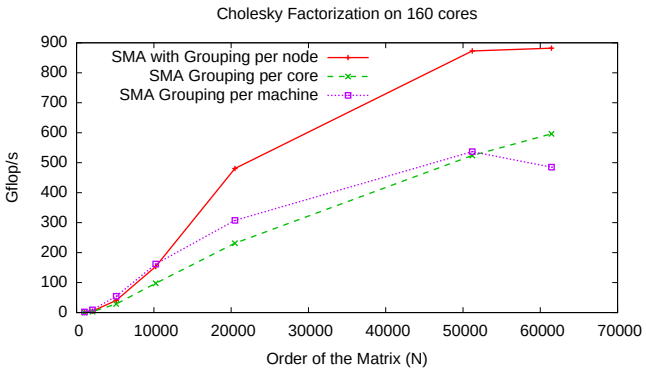


Figure 4. Comparison of different grouping strategies for the Cholesky Factorization on a 160 cores, 20 nodes machine (NT=512, double precision)

At the core level we have one group per core $G = N_{\text{core}}$. Each thread is responsible of its own pool of task and therefore there is no concurrency for accessing it. On the other hand, when there is few parallelism (e.g. at the end of the algorithm), some group might have no task to execute and therefore some threads may be idle due to lack of parallelism.

In summary, the larger the number of groups the lower the concurrency in accessing structures attached to these groups but the higher the risk that some groups become idle due lack of parallelism.

In order to bind threads to cores and to group threads onto NUMA nodes we use the HWLOC library [10] which offers a portable way to model NUMA machine and map threads. It provides logical and physical numbering of each element of the hierarchy (core, cache, node, etc.) as well as there interaction and interconnection.

B. 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. 4. This is done one a 160 cores NUMA machine composed of 20 nodes of one 8 cores socket Intel Nehalem Eagleton (E7–8837) at 2.67GHz.

In Fig. 4 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

executes 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 performances are similar for small size matrix 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 compared to the node grouping increases as the generated parallelism increases: when a few tasks/clusters are available, all cores cannot execute something and some stay idle.

C. Comparison with MKL and Plasma

The Intel Math Kernel Library (MKL) [11] is a multi-threaded library that provides many linear algebra kernels (BLAS, LAPACK, etc.). It is a highly tuned and optimize library for Intel processors such as the ones used in this experiments. In this work, we use the MKL version shipped with the Intel C compiler version 11.1-075.

Plasma [12] is a multithreaded library developed at the University of Tennessee and based on task parallelism such as the work presented here. The Cholesky kernel we use in this work is directly inspired from the tiled version proposed in Plasma. 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 [13], which is far more developed than the simple one presented above. In these experiments, we use Plasma version 1.4.5.

Nervertheless, Plasma and our simple runtime both relies on the same BLAS kernels of the MKL 11.1-075 except that we use the sequential version of them.

In Fig. 5, 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 size of each case. For MKL, the blocking is handled automatically.

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 (but still not sufficient to outperform Plasma) as no conversion is required in this case: MKL uses LAPACK format natively. Moreover, one can argue that once data are converted in tiled format, it can be used for many kernels and therefore the cost can be amortized. Comparison without format conversion will be done in the next section.

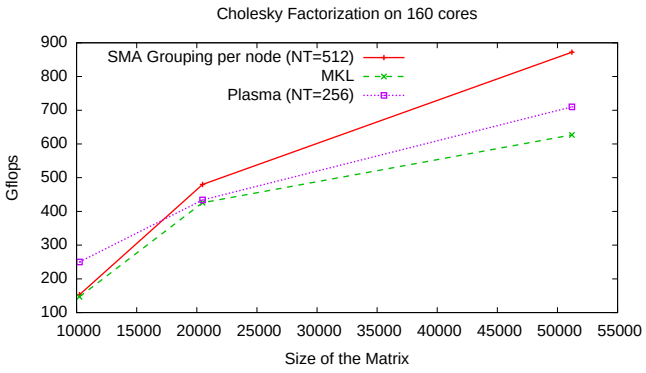


Figure 5. Comparison of SMA versus Plasma and MKL for the Cholesky Factorization on a 160 cores, 20 nodes machine (double precision, $N \leq 51200$). Format conversion included

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. Plasma is faster for small matrix size due to its optimize runtime system that is able to manage a lot of parallelism more efficiently that the simple system implemented for this work.

One can wonder what happen for $N > 51200$. It appears that the SMA performance degrades after this size. What is happening and how we handled that problem, is explained in the next section.

V. 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.

We see here a big performance drop for SMA-1 near $N = 64000$ 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 kernel computation without format conversion) we see that the performances are far better as shown in Fig. 6(b).

When we compare Fig. 6(a) with Fig. 6(b), we see several differences. First, the SMA-1 and Plasma performance are better when we exclude the conversion between the LAPACK format and the tiled format. This is due to the fact that such conversion can take up to 25% of the overall execution

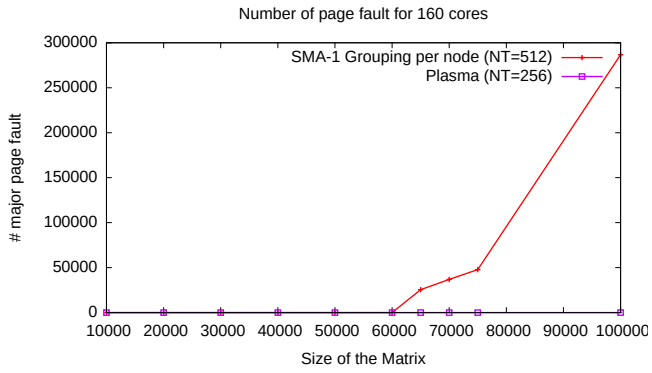


Figure 7. Number of system page-fault when increasing the matrix size. SMA-1 vs. PLASMA for the Cholesky Factorization (double precision)

time. This is not the case for MKL as it only uses the LAPACK format (there is no conversion). We also see that MKL is outperformed by the tiled algorithms (PLASMA and SMA-1) because there is much more parallelism in the tiled version of Cholesky and because the format conversion is very helpful in getting all the performance of the machine. 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.

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. We have a major page fault when the system needs to swap a memory page between main memory and secondary storage (e.g. disk). Results are shown in Fig. 7.

We see that, for SMA-1, the number of page faults suddenly increases at the same point as the performance of SMA-1 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 and start using the disk as a secondary storage.

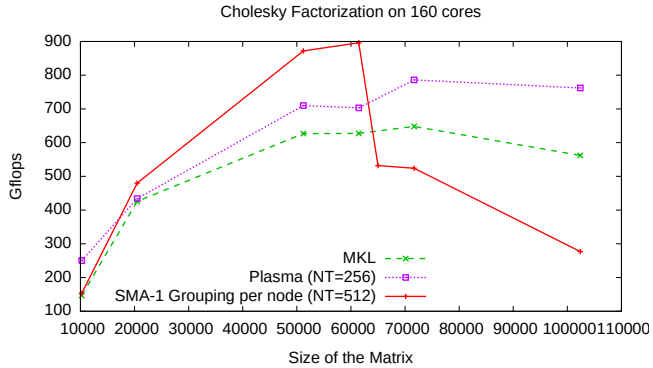
The reason is the following: despite the fact that the whole machine has 600 Gb of memory each of the 20 nodes has 30 Gb. When $N \geq 64000$ the size of the matrix requires more than $64000^2 \times 8 = 3.2710^{10}$ bytes or 30.5 Gb. In our code, the matrix is allocated and filled by a single thread 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 larger the matrix the higher the number of swaps and the slower the performance of the program as shown in Fig. 6 and 7.

To solve this problem, there exist several solutions. First, as done in Plasma, the filling of the matrix can be multithreaded as the rest of the program. As the system allocates pages on the memory node of the writing thread, if the threads are evenly scattered on the different nodes the pages will also be distributed on the different memory node. 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 strong performance even for large matrix size as shown in Fig. 8.

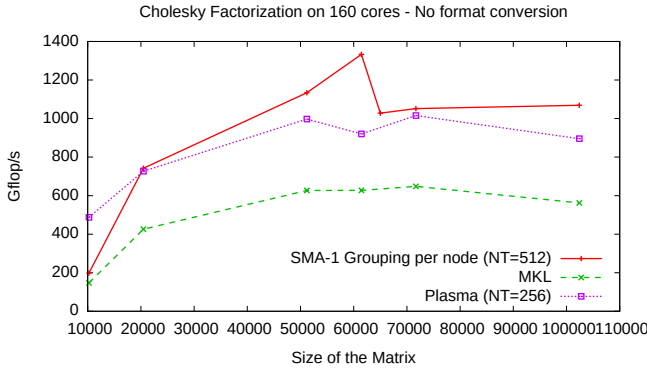
From Fig. 8, we see that the page-fault aware version of SMA is able to continue to increase performance for large matrix size contrary to the first version. Moreover, the gain against the other reference libraries (MKL and Plasma) is very large up to 74.7% for MKL and 49.1% for Plasma.

VI. Conclusion

NUMA parallel machines are fairly simple to program as they offer a flat view of the memory. However, data



(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).

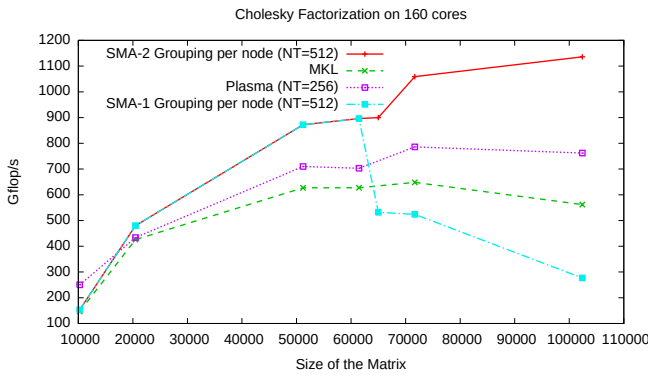


Figure 8. Final result. SMA version 1 (no data interleaving) and SMA version 2 (page-fault aware) vs. Plasma and MKL for the Cholesky Factorization (double precision). Format conversion included

placement, data movement and thread placement have a huge impact on the performance has 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. SMA provides a clustering function and a data allocation function that reduces unnecessary communications while keeping a good amount of parallelism. Moreover, the way data is allocated and kernels are executed is tightly coupled thanks to the use of the owner-compute rule, which greatly improves cache reuse.

In the second part, we have then implemented a simple runtime system as a proof-of-concept. We have discussed the issues of grouping threads, cores and cluster according to the mapping found by SMA. 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 size.

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.

VII. Acknowledgments

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

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/).

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] A. Gerasoulis and T. Yang, "On the Granularity and Clustering of Direct Acyclic Task Graphs," *IEEE TPDS*, vol. 4, no. 6, pp. 686–701, Jun. 1993.

[8] W. Pugh, "The Omega Test a fast and practical integer programming algorithm for dependence analysis," *Communication of the ACM*, Aug. 1992.

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

[10] 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 PDP2010,Pisa, Italia.

[11] 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

[12] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan., "Plasma users guide," Technical Report ICL-UT, 2010.

[13] 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.