Enabling Partial Pivoting in Task Flow LU Factorization

Omar ZENATI

ENSEIRB-MATMECA
Computer Science Department
Internship Report

Tutors : Pierre RAMET & George BOSILCA

Bordeaux September 7, 2012

Thanks

I would like to thank first, my tutors: Pierre Ramet and George Bosilca who helped me to make this internship possible.
I would like to thank also Mathieu Faverge and Emmanuel Agullo who supervised me, advise me and teach me a lot of interesting things.
I would like to thank also Stephanie Moreaud, Aurelien Bouteiller, Thomas Herault and all the french community of the ICL and Knoxville.
I would thank again all those mentioned for their good mood. Speaking of good mood, I would like to thank Anthony Danalis for all his jokes.
I would like to thank all other persons who helped me from near and far during this internship.

I wish you will enjoy reading this report.

Contents

1 Introduction **5**

2 Context **7**

3 Background **9**
3.1 Runtimes 9
3.2 LU Decomposition Algorithm 10
3.3 Task Flow LU Decomposition over Runtimes 14

4 Panel Factorization with Partial Pivoting **19**

5 Update engine for dynamic pivoting **23**

6 Experimental **27**

Bibliography **30**

Chapter 1

Introduction

Numerical simulation has become so pervasive that it is often considered as the fourth pillar of science. Indeed, this insatiable need of computing power has fuelled the production of ever more powerful, larger High Performance Computing (HPC) systems. In recent years, power consumption and thermal issues have stalled the decades old trend of performance boost from clock frequency increase. Consequently, modern designs, such as the Titan supercomputer currently being built at the Oak Ridge National Laboratory, have to resort to packaging heterogeneous core types (including GPU accelerators), in larger number and with increasingly intricate Non Uniform Memory Access (NUMA) hierarchies. These disruptive technological trends take a toll on productivity: heterogeneity and essentially unpredictable time for memory accesses concur to jitter and asynchrony, which are challenging established programming models designed around SPMD, homogeneous and regular computation.

This *jungle*[1] of resources has called forth the emergence of novel programming languages and tools that enable a more adaptive reaction to unexpected contentions and delays, predominantly by expressing parallelism as a task flow [1, 2, 3, 4, 5]. In this execution model, the program is broken into elementary tasks and a Direct Acyclic Graph (DAG) represents the dependency flow imposed by data sharing between tasks. At runtime, this DAG can be scheduled with a dynamic task distribution according to on-the-spot resource introspection and work stealing in an architecture-aware fashion, so as to overlap communications, tolerate jitter, and benefit from performance portability by hiding the complexities inherent to heterogeneous hardware. Among task flow approaches, the DAGuE tool kit stands out by using a Parametrized Task Graph (PTG) representation [6] of the DAG. This compact and symbolic representation is cornerstone to the scalability enjoyed by the DAGuE version of dense linear algebra routines [4], as it enables an independent, distributed runtime evaluation of the successor and predecessors of

[1] Herb Sutter, "Welcome to the Jungle", 12-29-2011, `http://herbsutter.com/2011/12/29/welcome-to-the-jungle/`

any task, at any moment, without the need for unfolding the entire DAG.

Many applications use the LU factorization algorithm to solve linear systems of equations. This popularity roots in two facets: the computational complexity of LU is half that of QR, yet its numerical accuracy is similarly excellent, thanks to a mechanism called partial pivoting. With partial pivoting, the best, most stable pivot for the LU reduction is selected in the entire column and lines are swapped accordingly to promote this pivot to the diagonal. The pivot selection cannot be pre-computed, therefore the DAG representing LU with partial pivoting changes depending on the updated content of the matrix. Yet, the PTG representation used in DAGuE can be generated from a compile time analysis, and imposes a static structure to the DAG itself. The data-dependent nature of the DAG representing LU with partial pivoting is thereby difficult to capture in a PTG representation without increasing tremendously the number of tasks and edges to be considered at runtime, a significant source of overhead and memory waste. The main contribution of this internship is to introduce a variation of the LU algorithm with partial pivoting that can be expressed in a PTG representation efficiently.

The rest of this report is organized as follows. Chapter 2 presents the context of the internship and the hosting laboratories. Chapter 3 presents related work in the field of task flow scheduling and implementations of the LU algorithm. Then, Chapter 4 and 5 further discuss the issues raised by partial pivoting in PTG task flow systems, and the solutions we propose to tackle these challenges. Chapter 6 presents an experimental evaluation of the resulting algorithm and compares it with both a legacy SPMD implementation of LU with partial pivoting and the less stable but less challenging LU with incremental pivoting in a similar task flow implementation, before we conclude.

Chapter 2

Context

The work presented in this report was carried out under an internship. This was performed during three months at the Innovative Computer Laboratory[1] (ICL) and three months at the Inria of Bordeaux. It is part of the project *Matrices Over Runtime Systems @ Exascale*[2] (MORSE) which aims to design dense and sparse linear algebra methods that achieve the fastest possible time to an accurate solution on large-scale multi-core systems with GPU accelerators.

Innovative Computer Laboratory

Attached to the university of Tennessee, the Innovative Computer Laboratory (ICL) is one of the world leader laboratory in the field of high performance computing (HPC). ICL was established by the Pr Jack Dongarra from 1989. Since its inception, ICL produced and participated to several applications of high value to the HPC community including: ATLAS, BLAS, LAPACK, MPI, Netlib, PAPI, ScaLAPACK, Top 500 ...

Today, ICL continues in its desire to contribute to science. In this dynamic, I worked with the Distributed Computing Group where my mission was first to apprehend the DAGuE runtime system and then to study the feasibility of task flow LU decomposition over PTG using DAGuE as practical tool.

Inria

Inria is a public science and technology institution. It was created in 1967. It includes 8 research centers in France and has over 4000 employees. INRIA depends on the French ministries of research and industry.

The institute has strong relationships internationally. It has various partnerships in Africa, Middle East, America and Asia. Thus, it promotes exchanges

[1] `http://icl.cs.utk.edu/`
[2] `http://icl.eecs.utk.edu/morse/`

between scientists around the world.

I was hosted by the BACCHUS team and worked with the HIEPACS and Runtime teams. Firstly, my goal was to continue the work I started at the ICL. Then, my mission was to take in hand the StarPU runtime system and implement its task flow LU decomposition in order to compare performances of PTG and sequential task flow.

Chapter 3

Background

Twenty years ago, a computer with only one single processor may be considered a computing platform and be included in the Top 500[1] ranking of super computers. Nowadays, computing platforms have greatly evolved. Their architectures became more complex and varied. For this report, we propose here to distinguish them into four models. First, the **shared memory multi-cores** architectures, they are computing platforms including many cores which all have access to the same virtual memory. The **distributed memory** architectures are composed of at least two nodes. Each node has its own cores and its physical memory. Cores of different nodes cannot access to the virtual memory of other nodes, and thus, must communicate to share data. Even if in distributed memory architectures, a node can have many cores, we will use - in this report - the denomination *distributed memory* architectures only for nodes with one single core. In contrast to the **hierarchical** architectures which are distributed memory architectures where at least one node is a shared memory multi-cores architecture. More recently, accelerator as GPU can be integrated to platform. The distributed memory architectures where at least one node includes an accelerator will be called **heterogeneous** architectures.

3.1 Runtimes

The most popular method to implement parallel distributed memory programs consists of using messaging systems. There are several different types of these systems such as message-passing like Message Passing Interface (MPI)[7]. To achieve good performance with MPI, the knowledge of the whole architecture used is often advised. Moreover, the portability of performance is not always ensured. There is another type of messaging systems which tries to resolve these problems: the active-message, Charm++ is based on [8].

[1] `http://www.top500.org/`

In order to ease efficient use of supercomputers and to introduce an abstraction to computer's architectures, the task flow model was used to implement other runtimes. In fact, task flows can be represented as a Direct Acyclic Graph (DAG) where vertices are tasks and edges are the dependencies between them. These runtimes schedule tasks on different nodes and move data according to tasks dependencies. We propose here to distinguish runtimes based on task flow model into two families: those that run codes with implicit data dependencies and those that run codes with explicit data dependencies.

In order to ease the use of parallelism, while keeping the traditional way of programming, some runtimes run on codes with implicit data dependencies. These runtimes need only a set of tasks and their data access modes (read, write or read-write). With this information, the runtime extract the data dependencies and then build the corresponding DAG of task flow. Implementing parallel application over these runtimes is very similar to sequential codes. Quark implement this model for shared memory machines and is used in the Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) library [9]. StarSs is a collection of runtimes which can run on different types of architectures: CellSs for the Cell BE[10], SMPSs is for SMP architectures [11], and GPUSs for GPU [12]. In order to gather all types of architectures, StarPU is a unified runtime system that offers support for heterogeneous multicore architectures (GPGPUs, IBM Cell, ...). StarPU manage tasks execution through different architectures thanks to its data coherency protocol [13, 14].

On the other hand, there are some runtimes based on explicit data dependencies by using Parametrized Task Graph (PTG). Thanks to explicit data dependencies, these runtimes may benefit from efficient graph traversals. Moreover, PTG allows for compact representations of algorithms and induce a low overhead. Intel CnC is one of these runtimes dedicated - for the moment - to shared memory computers. DAGuE (Direct Acyclic Graph scheduler Engine) is also a runtime based on explicit data dependencies which can run on computers with shared and/or distributed memory.

In the rest of the report, we focus on runtimes using explicit data dependencies. To illustrate the benefits on hierarchical computers, we will use the DAGuE runtime system.

3.2 LU Decomposition Algorithm

Presentation of the algorithm

In order to solve square systems of linear equations $Ax = b$, computers can use the LU decomposition algorithm. It consists of factorizing a square $n * n$ matrix

A into a matrix product $A = LU$, where L is a lower triangular matrix with the identity diagonal and U is an upper triangular matrix. For that, we rely on the fact that at each step of the Gauss elimination method, there is a matrix $L^{(k)}$ such as:

$$A^{(k+1)} = L^{(k)} A^{(k)}$$

The matrix $U = A^{(n)}$ is upper triangular as:

$$\begin{gathered} U = L^{(n)} L^{(n-1)} \dots L^{(2)} L^{(1)} A \text{ , and so} \\ A = L^{(1)-1} L^{(2)-1} \dots L^{(n-1)-1} L^{(n)-1} U = LU \end{gathered}$$

The LU decomposition algorithm has the advantage that the elements of L and U may replace elements of A in memory. Note that the unity diagonal of L is not stored.

At each step $k \in [\![1, n]\!]$ of the decomposition, the algorithm execute the following operations:

$$\begin{array}{lll} l_{i,k} = 0 & i = 1, \dots, k-1 & (*) \\ l_{k,k} = 1 & & (*) \\ l_{i,k} = a_{i,k}^{(k)} / a_{k,k}^{(k)} & i = k+1, \dots, n & (1) \\ a_{i,j}^{(k+1)} = a_{i,j}^{(k)} & i = 1, \dots, k \ \& \ j = 1, \dots, n & (*) \\ a_{i,j}^{(k+1)} = 0 & i = k+1, \dots, n \ \& \ j = 1, \dots, k & (*) \\ a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - l_{i,k} a_{k,j}^{(k)} & i = k+1, \dots, n \ \& \ j = k+1, \dots, n & (2) \end{array}$$

In practice, the operations followed by an (*) are not actually made because of the identity between the storage of $A^{(k)}$ and L. The operation (1) is a *scal*ar product. In the rest of the report, we will call it a *scal* operation according to the BLAS reference. The operation (2) is an outer product, as in the BLAS reference, we will call it *ge*neral *r*ank operation (*ger*).

After the factorization, the solve of the system of linear equation $Ax = b$ is equivalent to to solve the two triangular systems $Ly = b$ and then $Ux = y$ with *forward and backward substitution*. These operations can be made with a *tr*iangular *s*olve step applied to a *m*atrix (*trsm*)with the BLAS routines.

The algorithm presented is known as an LU decomposition *without pivoting*. It is the simplest algorithm to perform an LU decomposition. However, this LU decomposition only enables one to solve linear systems for certain matrices such as diagonally dominant matrix. For general matrices, the obtained solution is not always accurate. This is due firstly to the fact that it is possible to find an element $a_{k,k}^{(k)} = 0$ and so the *scal* operation becomes impossible. Secondly, if $a_{k,k}^{(k)}$ is just enough close to zero, the scalar division will introduce a small error in the numerical values of the matrix due to the fixed precision used by computers. In

such cases, the obtained solution may be refined by applying iterative methods (such as iterative refinement, Krylov subspace methods ...).

In order to obtain a good accuracy - without requiring to post-process the solution - pivoting techniques may be employed. The solve of the system of linear equation $Ax = b$ amounts to solve $PAx = Pb$ where P is a permutation matrix and it is computed at the factorization.

There is a lot of heuristic to perform the pivoting, the most used by the scientific community is the *partial pivoting* for its practical stability and accuracy [15], it is also used in the LINPACK benchmark that is used to rank the TOP 500 supercomputers.

The characteristic of the LU decomposition with partial pivoting is that, at each step k of the matrix factorization, a research for the maximal absolute value is performed on elements of the k^{th} column from the k^{th} element. The element which is selected will be called in the rest of the report: the better pivot or the maximum. Then, the row of the selected pivot is swapped with the k^{th} row.

An other strategies of pivoting is - for example - the *full pivoting* which amounts to research the better pivot not only in the column, but in the row also. Thus, the solution computed is more accurate. However, this algorithm generate more synchronisation and double the complexity of the pivoting operation. The partial pivoting rarely reach its critical case where the best pivot of the column is not large enough to avoid mathematical issue. The partial pivoting is then more commonly used.

Blocked panel version

In order to benefit from efficient cache effects, state of the art dense linear algebra libraries implement a block version of the factorization. The matrix is split in n_p block columns - so called panels - of n_b columns ($n = n_p * n_b$). The factorization of the matrix consists in a sequence of n_p twofold operations. Indeed, at each step p ($0 \leq p < n_p$), panel p is factorized and the trailing sub-matrix is then updated. Figure 3.1 shows an example of matrix in the second step ($p = 1$) of the LU decomposition, the second panel is still being factorized, and after, the trailing sub-matrix will be updated.

At each step p, the panel factorization is performed on the p^{th} panel. Such a panel factorization consists in a loop of n_b iterations. At each iteration i ($p*n_b \leq i < (p+1)*n_b$), a search for the maximum of the i^{th} column is performed, then its row is swapped with the i^{th} row. Then, a BLAS *scal* routine is applied on the column i. It consists to divide all the element of the column i by the selected pivot (Figure 3.2 show the different parts of the panel). After that, the trailing sub-panel is updated with an outer product (*ger*). The panel factorization produces an array of size n_b containing the pivots selected, we note *ipiv* this array.

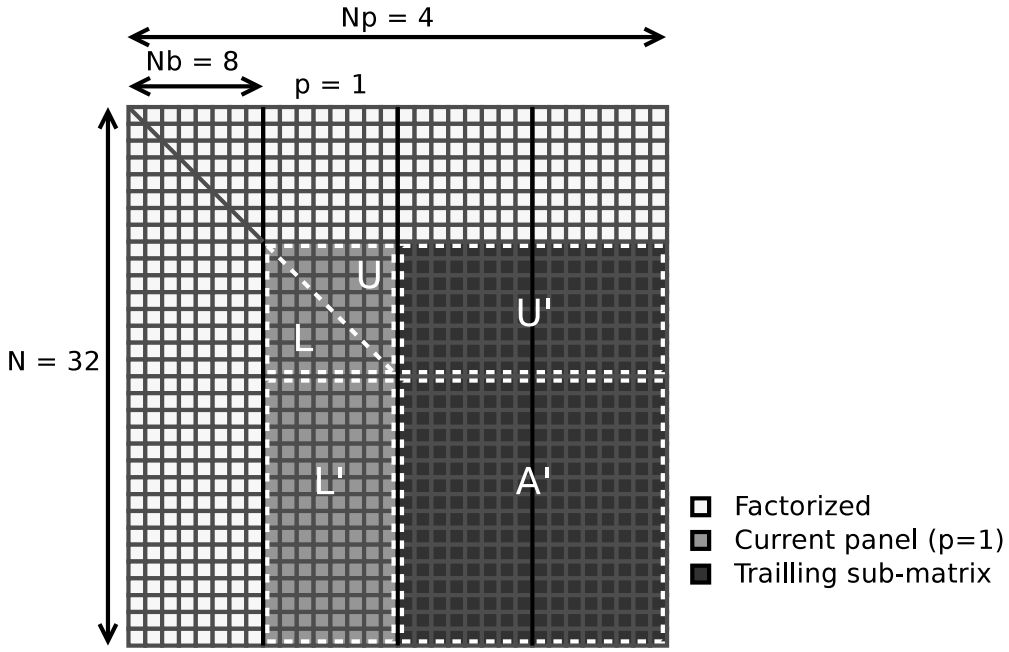


Figure 3.1: LU decomposition at step p on panel-blocked matrix

For each index x of the array $ipiv$, the row $(p * n_b + x)$ will be swapped with the row $(ipiv[x])$. Mathematically, the panel factorization can be expressed as follows:

For k from 1 to nb
    Search for a pivot, do pivoting and store index
    For i from $k+1$ to n /*scal operation*/
        $a_{i,k} = a_{i,k}/a_{k,k}$
    For i from $k+1$ to nb /*ger operation*/
        For j from $k+1$ to nb
            $a_{i,j} = a_{i,j} - a_{i,k} * k,j$

Once the p^{th} panel is factorized, the subsequent trailing sub-matrix is updated. This update depends on the result of the corresponding panel factorization. It takes as entry the pivot information and then applies the permutations on the whole trailing sub-matrix. Once the swaps have been performed on the trailing sub-matrix, the n_b block row corresponding to the eliminated rows (block U' in Figure 3.1) is updated. For that, the following equation must be solved:

$$U'^{(k)} = L * U'^{(k+1)}$$

Which is equivalent to:

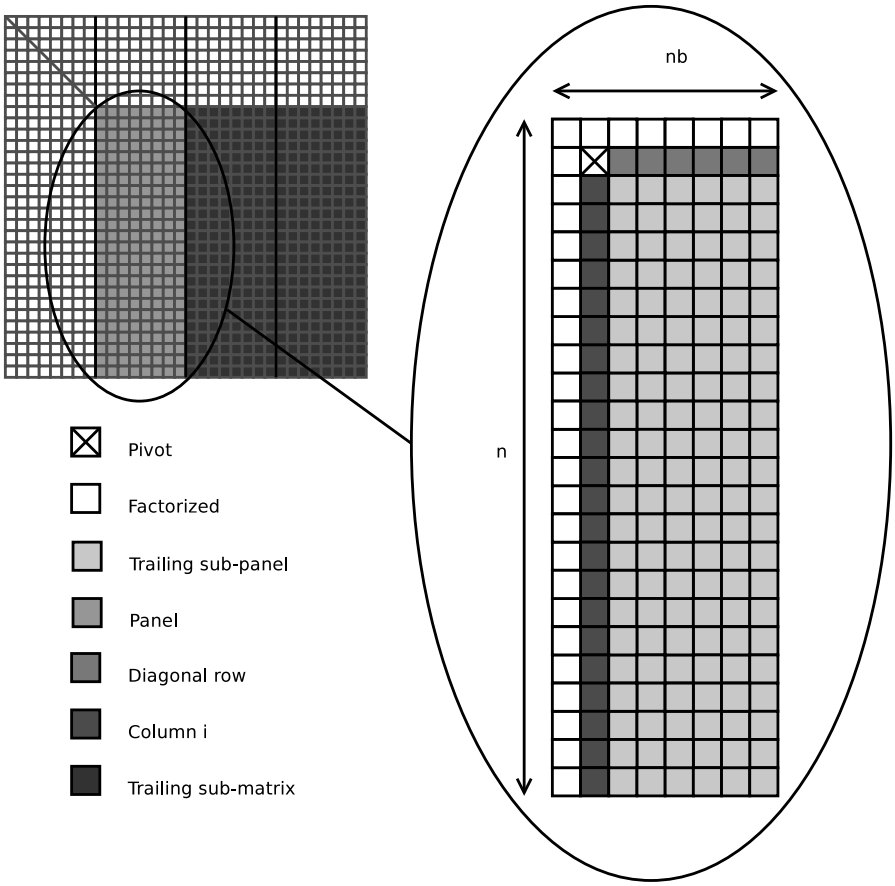


Figure 3.2: Second iteration of panel factorization (after swap)

$$U'^{(k+1)} = L^{-1} * U'^{(k)}$$

To solve this equation, we can use the BLAS routine *trsm*. The block U' obtained is then multiplied with the block L' and the resulting matrix is subtracted from the matrix A':

$$A'^{(k+1)} = A'^{(k)} - L'^{(k+1)} * U'^{(k+1)}$$

This operation is performed with the BLAS routine for matrix multiplication *gemm* (*ge*neral *m*atrix *m*atrix operation).

3.3 Task Flow LU Decomposition over Runtimes

We consider the problem of writing the LU decomposition algorithm as a task flow model in order to be executed over runtime systems (particularly on DAGuE). In

the case of dense linear algebra, the algorithms have been redesigned to cope with this model. They are expressed in terms of tasks operating on fine grain squares sub-matrices, also called tiles [16, 17]. Figure 3.3 shows a matrix partitioned into tiles.

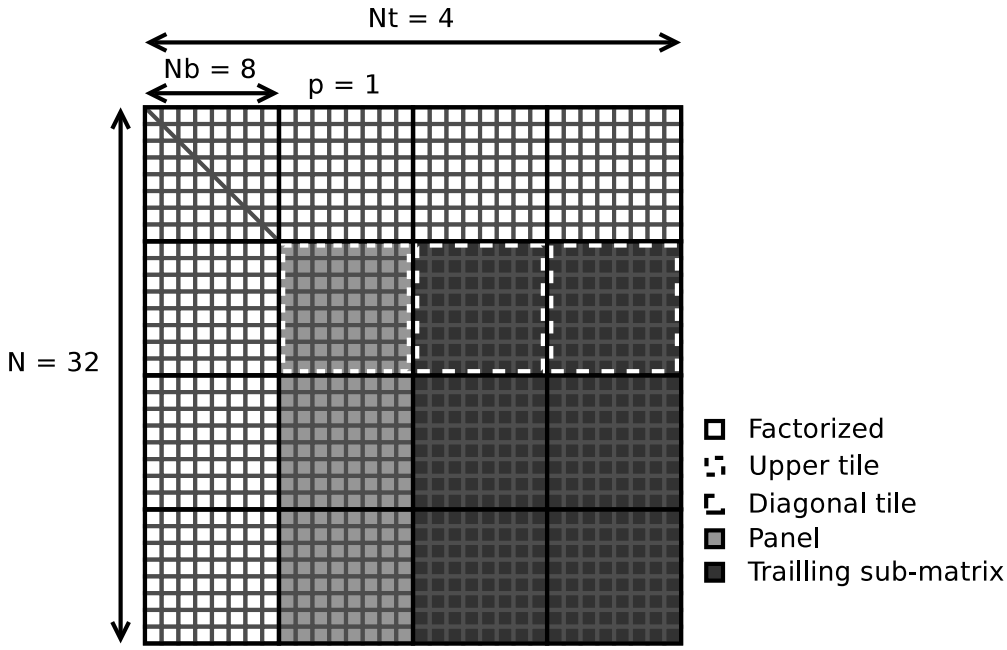


Figure 3.3: LU decomposition at step p on tiled matrix

In case of LU decomposition without pivoting, the tile algorithm is very similar to the blocked panel algorithm of LU decomposition without pivoting. In fact, the panel factorization is split into two tasks: one operating on the diagonal tile (see Figure 3.3) and the second performing on the other tiles of the panel. The first one will execute the natural LU decomposition without pivoting algorithm (presented in first part of 3.2), we will call it **GETRF** according to *getrf* routine (*ge*neral *tr*iangular *f*actorization) of Lapack which is a mathematical library build ont the BLAS routines to solve - for example - linear algebra or eigenvalues problems. The second operation has to solve the equation:

$$L'^{(k)} = L'^{(k+1)} * U$$
$$\Updownarrow$$
$$L'^{(k+1)} = L'^{(k)} * U^{-1}$$

This can be solved with *trsm* operation. Thus, the task operating this solve will be called **TRSM_L**.

As for the blocked panel algorithm, the update is a twofold operation: *trsm* for tiles owning the eliminated rows and *gemm* for the other tiles of the trailing sub-matrix. We will call them respectively **TRSM_U** and **GEMM**.

All these tasks interact by exchanging data. These interactions can be represented by an automate. Figure 3.4 shows the automate of tasks interactions of the tile LU algorithm, we can see that the task **GETRF** takes the diagonal tile as entry to factorize it. If it is the first iteration $k = 0$, it takes it directly from the memory, otherwise it takes it from the output A' of the **GEMM** task of the previous iteration. The resulting tile of the task **GETRF** - which is two matrix L and U - is respectivly sent to the tasks **TRSM_L** and **TRSM_U**.

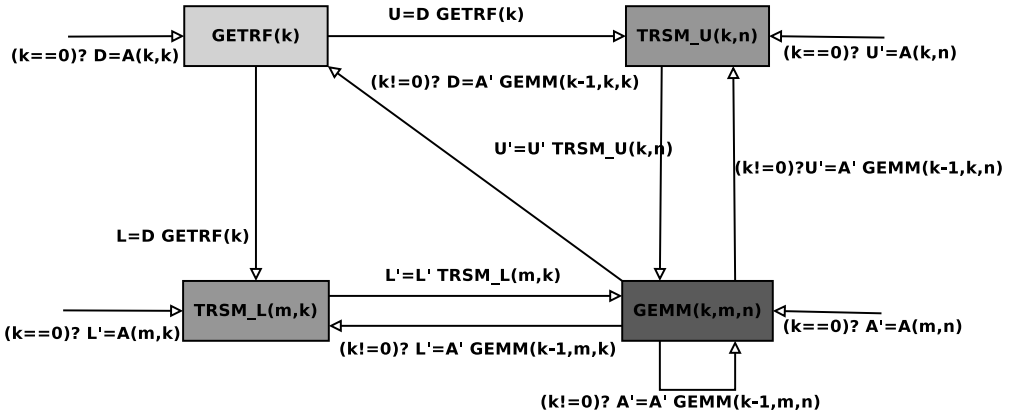


Figure 3.4: Automate of tasks interactions of tile LU decomposition algorithm

The execution of the automate of interactions provides a DAG of task flow. Figure 3.5 shows the DAG obtained after unrolling the automate of Figure 3.4 on a matrix of three tiles.

As for the blocked panel algorithms, the LU decomposition without pivoting is numerically unstable. In [18], the authors proposed a new pivoting strategy called *incremental pivoting* based on [19] more suitable to tile algorithms and achieved high performance by limiting the number of synchronizations and enabling more parallelism. The algorithm applies a partial pivoting only on the diagonal tile, then it factorizes sequentially every other tiles of the panel and if a better pivot is found during this decomposition, the upper triangular matrix of the diagonal tile is accordingly updated. The update of the trailing sub-matrix is also done sequentially by using its own routines which are less efficient than *gemm* operations.

However, this algorithm has been proven numerically unstable [20]. In order to achieve better stability, we choose to adapt the partial pivoting algorithm to the task flow model.

To illustrate the complexity of implementing the algorithm, we consider the pivot research. This research occurs at each column factorization and lies on the

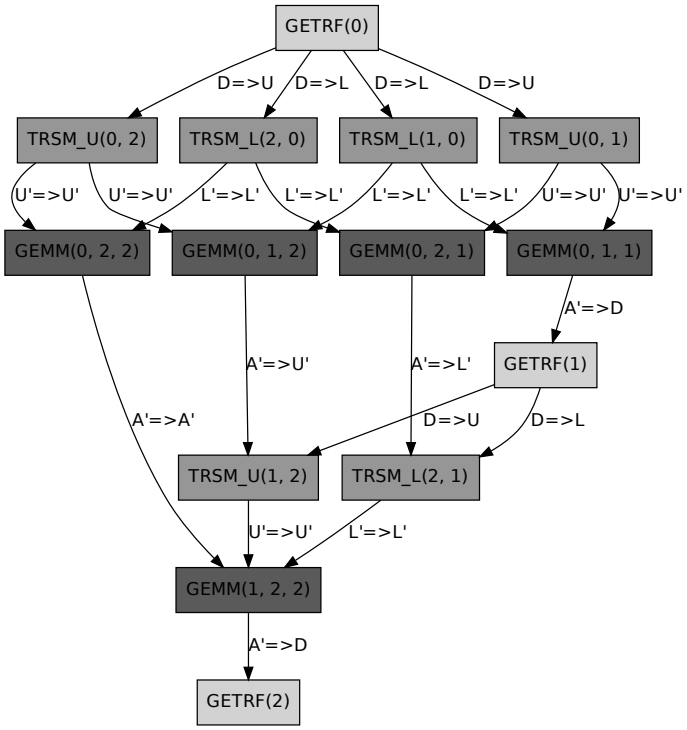


Figure 3.5: DAG of the task flow LU decomposition algorithm on a matrix of 3 tiles

critical path of the decomposition. Partial pivoting requires to select the maximum of the elements below the diagonal in the column. Then, elements lie on $(n-k)/n_b$ different tiles which may be potentially mapped on a huge number of cores in order to ensure state of the art load balancing techniques. The research induces a synchronization at each column which may overwhelm all potential benefits of tile algorithms and deliver too many tasks to be processed by the runtime.

Another issue with the implementation of the partial pivoting algorithm over the task flow model, is the swapping operation of the update. In fact, after receiving the pivots array from the panel factorization, the upper tile has to send the selected rows to other tiles and receive back the substitute ones. If the swap is done rows by rows, the upper tile may exchange a row with another tile of the panel depending on the numerical values of the pivots. The task flow model can thus no longer be statically build in advance but has then to be dynamically composed. Figure 3.6 represent a simplified task flow of the sending operation with an automaton. Each time the execution of an algorithm will depend on a value of a

data, we will call this phenomenon a *data dependency*. The algorithm which contains data dependencies will be called *dynamic algorithm*. These algorithm may be represented by automatons or others conditional mathematical objects. Unfortunately, almost runtimes supports only static representation of task flow (DAG). Thus, the challenge is to represent a dynamic algorithm with a static representation covering the collection of possibilities. A solution is to create a DAG with a path where all concurrent tasks are sequential and move conditions of transitions to the kernel of the tasks. Figure 3.7 represent the same dynamic algorithm of Figure 3.6 with a DAG, we can see that all the tasks are represented sequentially and that the conditions of the dependencies are integrated to the kernels. This transformation may increase tremendously the number of tasks and communications required to execute the algorithm. In the next sections we will present how we reduce the size of this static representation to perform the panel factorization and then update the trailing sub-matrix.

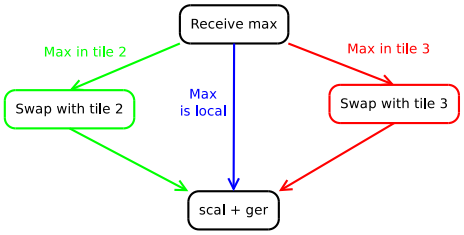


Figure 3.6: Dynamic representation of task flow

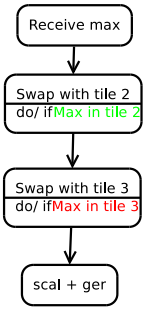
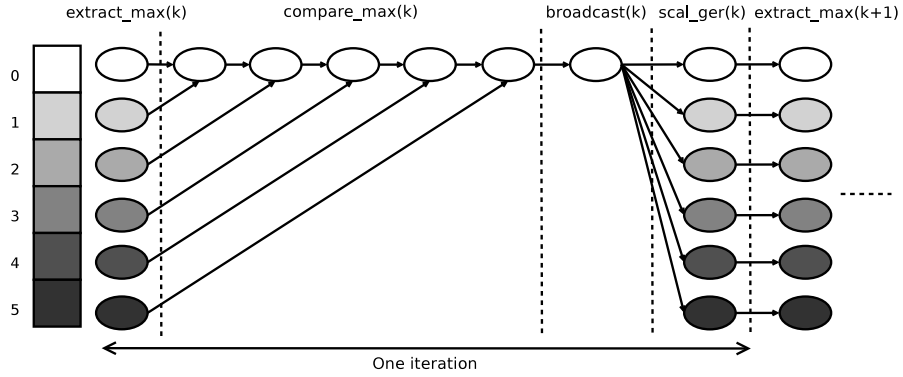


Figure 3.7: Static representation of task flow

Chapter 4

Panel Factorization with Partial Pivoting

In order to reduce the important number of tasks and communications, optimisations have to be applied at all levels of the algorithm. For that, we present in this section the evolution of the algorithm from the first natural version to the most optimized. As said in the section 3.2, the panel factorization is a loop of n_b iterations. At each iteration k, several operations are executed on the panel. The first natural version is that the node which contains the diagonal tile compares progressively its maximum with others tiles. Each time it finds a better pivot, it saves it and uses it for next comparisons. After this operation, the task operating on the diagonal tile broadcasts the initial diagonal row and the selected row owning the column maximum to all others tiles in order that the concerned tile apply the swap, and every tile apply the *scal* and *ger* operations (Level 1 BLAS). Task flow 1 shows one iteration of the panel factorization on a panel of 6 tiles. Each node has its own color, here all tiles are distributed on different nodes. We will use the same coding color for following Task flows of the report.

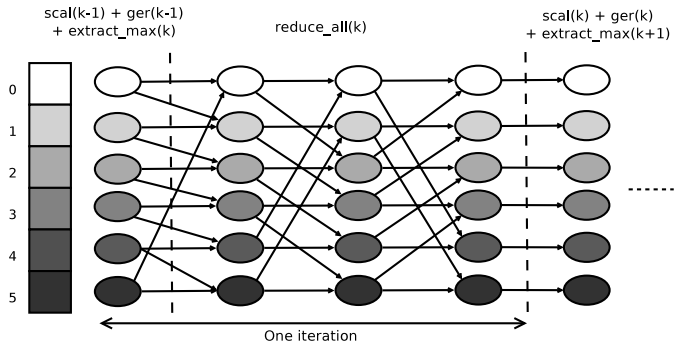


Task flow 1: One iteration of panel factorization on distributed architecture

The first remark is that there are some communications which can be optimized.

In fact, instead of using natural sequential comparison, it is possible to use a *reduce* operation. It will allow to perform a faster election of the row owning the best pivot of the column. Therefore, because the broadcast is following the operation of comparison, we can merge the two operations into one single *all_reduce* operation. In order to reduce the number of fine-grained tasks, and so, reduce the cost of scheduling, we can also merge tasks of extraction and tasks of panel update. For that, the *extract_max* of step k and *scal_ger* of step $k-1$ tasks can be done in one single task.

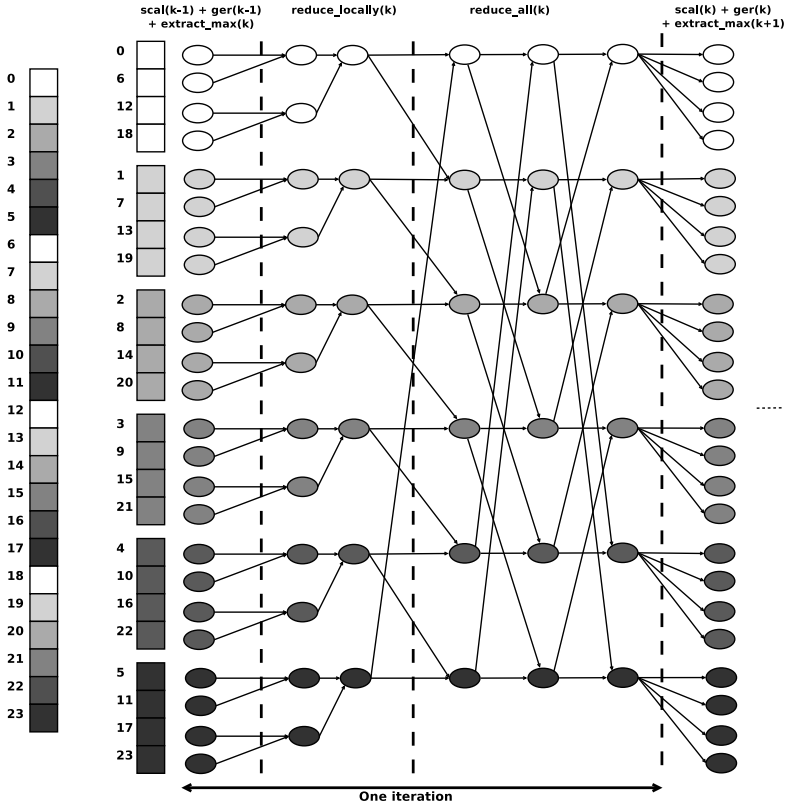
Concerning communications, DAGuE and other runtimes handle point-to-point communications and can manage some collective communications as broadcast. But, to perform more complex communications operations (reduce, gather...), we have to express them as a task flow. This is due to the fact that reduce and/or gather operations are more complex to express with PTG and as of today are not available in the language used by DAGuE. For the *all_reduce* operation, the task flow needed is based on the task flow of the Bruck's algorithm [21]. In the natural version, we broadcast two rows (the initial diagonal row and the selected row owning the maximum value of the column). Thus, for the *all_reduce* operation, an array of two rows per node is needed, we will call it a workspace. The first one is filled at beginning by the diagonal tile with the initial diagonal row, and the second row is filled by tasks operating on all panel tiles with the row holding the maximum value in the local column. At each step of the *all_reduce* operation, task copy first row from the received workspace if it is not empty, and reduce the seconds rows of the two workspaces according to the maximum value of their pivots. Thus, at the end of the *all_reduce* operation, all workspaces will be filled with the same values. The resulting task flow is presented in Task flow 2. Each tile has its own color which mean that each tile belong to a different node.



Task flow 2: One iteration of panel factorization on distributed architecture (combining reduce and broadcast communications)

Nowadays, most distributed computers contain many cores at each node. To

balance well the computations among the processors, a two-dimensional block-cyclic distribution is used to spread the data over the nodes[22]. If we consider the fact that each node can be multi-core, the task flow 2 can still be optimized. For that, the idea is to reduce first locally the maximum on the node and then, apply the *all_reduce* operation between the nodes. The *reduce* operation is performed with a binary tree. Task flow 3 shows one iteration of panel factorization for hierarchical architecture. We can see that tiles are distributed over nodes in a cyclic way. Each node run tasks that extract from their tiles the best pivots, then the node reduce locally the best pivot to participate with it in the Bruck's algorithm.



Task flow 3: One iteration of panel factorization on hierarchical architecture

Whether in distributed, shared or hierarchical, the task flow showed in Task flows 1, 2 and 3 represent only one iteration of panel factorization. For the last iteration, an additional task is needed to finalize the panel factorization. In fact, this task perform the last *scal* and *ger* operations then release the dependencies to update the trailing sub-matrix.

Beside optimizations in task flow, we apply some optimizations on kernel ex-

ecuted by tasks. In case of the task *scal+ger+search_max*, we use the notion of internal blocking. This optimization is the same applied from LINPACK to LAPACK libraries [23]. Thanks to the internal blocking, we can use more Level 3 BLAS which increase the performance obtained. In fact, at each step of the panel factorization, instead applying *ger* on the whole trailing sub-panel, we apply it only on a block. After each *ib* iterations, *ib* being the internal blocking parameter, the rest of the trailing sub-panel is updated by the level 3 BLAS operations *trsm* and *gemm*. Figure 4.1 shows a matrix - with tiles of $n_b == 9$ and an internal blocking of $ib = 3$ - which is at the second step of the panel factorization. We can see that the *scal* is applied on the whole column, but the *ger* is applied only on the current internal block, the trailing sub-panel will be updated with a *trsm* and *gemm* in the next step.

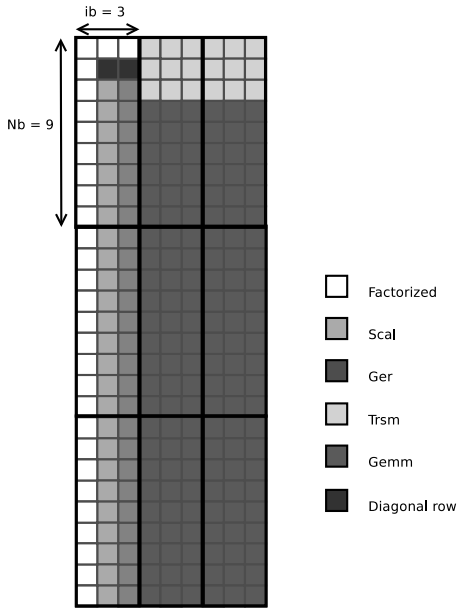


Figure 4.1: Panel factorization with internal blocking

Chapter 5

Update engine for dynamic pivoting

After the panel factorization, the operation performed on the trailing sub-matrix is the update. The update takes the array of pivots produced by the panel factorization, swaps the rows and then apply *trsm* and *gemm* operations.

To perform one swap, the algorithm depends on the value of the pivots, this is what we call data dependency and so the algorithm is a dynamic algorithm (section 3.3). The solution to represent a dynamic algorithm with a static DAG of task flow is to make a path with all tasks. In practice, it means that all tiles must participate in swapping even if they are not concerned. For that, each tile will have a workspace of two rows: the first to store one row coming from the upper tile and the second to store one row going to the upper tile. Nodes will share between them the workspaces and fill them with the appropriate rows. The *all_reduce* seems to be the right operation to use. At the step k, $n_t - k$ tiles participate to the election of the best pivot. The cost of an *all_reduce* operation is log_2. Thus, a swap operation costs:

$$log_2(n_t - k)$$

There is n_b swaps per panel. Thus, the cost of all swaps of one panel is:

$$n_b * log_2(n_t - k)$$

. The cost of swaps of all panel of one step k is:

$$n_b * log_2(n_t - k) * (n_t - k)$$

. Finally, the cost of all swaps of the LU decomposition is:

$$\sum_{k=0}^{nt-1} n_b * log_2(n_t - k) * (n_t - k)$$

.

In SPMD model, a single swap costs $2 * n_b$. Thus, with the same reasoning, the cost of all swaps of the SPMD LU decomposition obtained is:

$$2 * n_b * (n_t^2 - n_t)/2$$

.

The natural task flow algorithm will be very expensive relatively to the cost of SPMD algorithm. Moreover, at the end of each *all_reduce* operation, only two nodes will really use the rows collected in their workspace (the upper tile and the tile which exchanges with it).

In order to reduce the cost of swaps, a good idea is to perform all swaps at the same time. However, this is not possible with pivots. In fact, because the same row can own several times the row including the maximum of the column, it is necessary to execute the pivots in the right order (from the first to the last). Figure 5.1 shows an example of a row which contains successively two times the maximum value of the column, we can see that the row 1 goes down to the row 13 then comes back to the row 2. Thus, it forces us to perform the first pivots before the second.

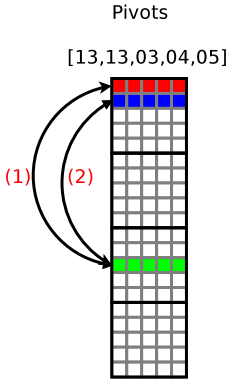


Figure 5.1: Row movements with pivots

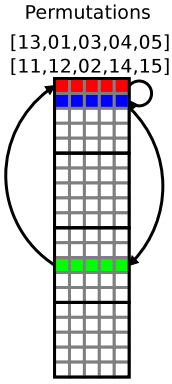


Figure 5.2: Row movements with permutations

To reduce high cost of all swaps, the key is to use another structure instead of pivots. For that, the permutations are the right solution. In fact, permutations can be represented by an array of size n (we see after that it can be reduced). For each index x of the array $perm$, the row $perm(x)$ will be moved in place of the row x. Thus, with permutations, we know from the beginning the final place of each row. Figure 5.2 shows the use of permutations instead of pivots which is showed in Figure 5.1, we can see that the row 1 goes directly to the row 2 and does not move again. Thanks to this structure, all the rows of one panel can be swapped in one single step. Thus, the cost of all swaps of one panel will be just:

$$log_2(n_t - k)$$

And then, the cost of all swaps of the LU decomposition algorithm is :

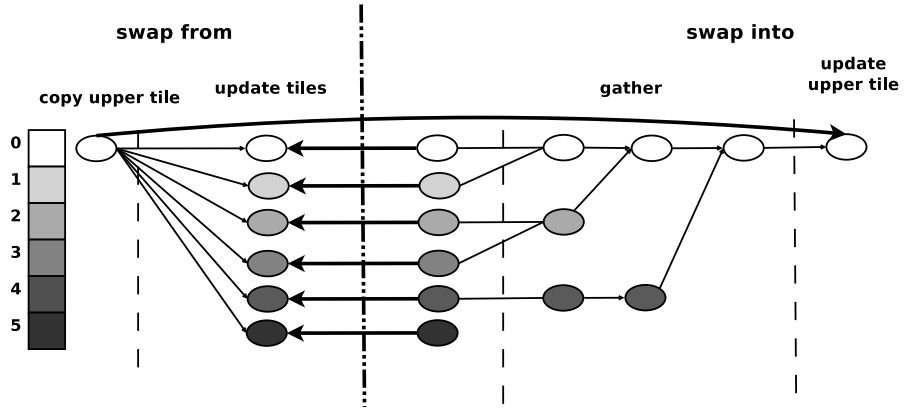
$$\sum_{k=0}^{nt-1} log_2(n_t - k) * (n_t - k)$$

We remark that at most n_b rows go into and from the diagonal tile. Thus, the array of permutations may be limited to a size of $2 * n_b$ elements, the first n_b elements will be used to store permutations and the second n_b elements will be used to store the inverse of permutations. Therefore, instead of using a workspace of two arrays, it is necessary to use two buffers - with the size of one tile - for communications: the first is a copy of the upper tile, it is broadcasted to all nodes operating on the panel. Each node will extract the rows needed from it. We will call this operation *swap from*. The second buffer is used to gather rows moving to the upper tile. Each node creates its own buffer, fills it with rows intended to be stored in the upper tile and then participates with it in a *gather* operation. This operation will be called *swap into*. This solution allows us to perform the *swap from* and the *swap into* in parallel due to the fact that the broadcast and the gather are completely independent.

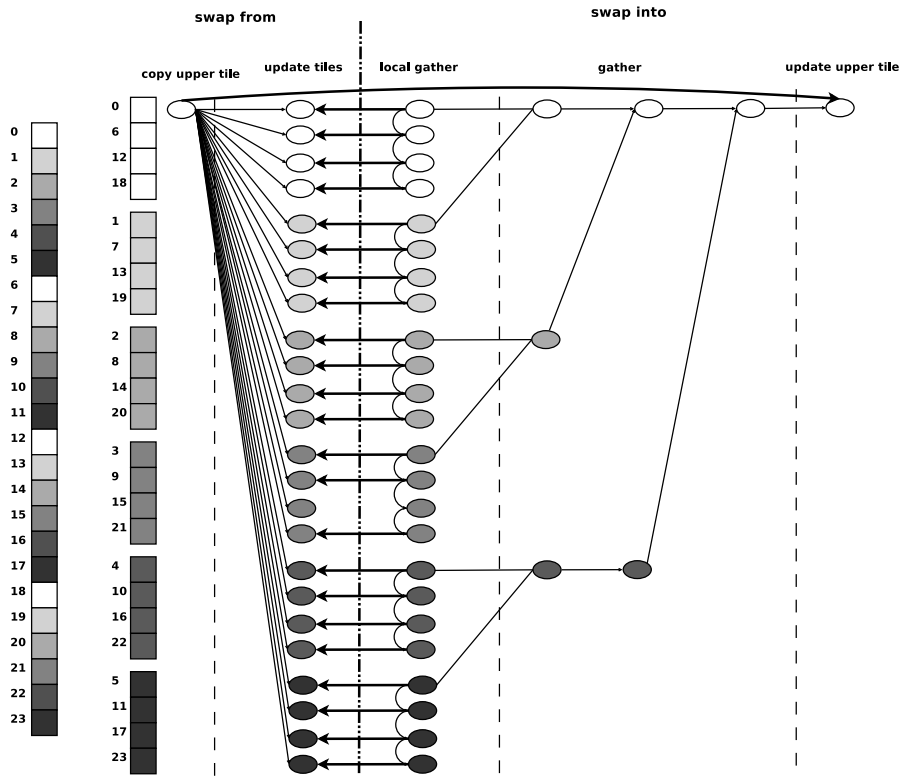
Task flow 4 represents the swapping operation of update for distributed architecture. We observe on the left of the Figure the broadcast of the upper tile copy, and on the right, the gather of rows moving to the upper tile. Because that the gather must be done before the update, some synchronizations have been added to prevent from the *read after write* effects. The bold arrows show these synchronizations. We can see also that the copy of the upper tile must be applied before its update.

As for the panel factorization, we consider that nodes can be multi-core. In order to reduce global communications, each node shares its buffers over its local tiles before to send them to other nodes. Task flow 5 shows the update operation for hierarchical architecture. We can see that each node gather first locally the rows moving to the upper tile, before participating to global gather. We also observe that the broadcast can produce a tremendously number of communications, but in fact, the runtime can manage broadcasts operations in order to minimize communications. For that, it can change the DAG of communications to a three, ring or other models of broadcast.

Moreover, this update algorithm implemented is a generic solution that can execute update operation after any panel factorization which provides an array of pivots (incremental pivoting, CALU ...).



Task flow 4: Swapping operation of update on distributed architecture



Task flow 5: Swapping operation of update on hierarchical architecture

Chapter 6

Experimental

For experiences, the architecture used is a hierarchical platform. It has 16 nodes interconnected with an Infini Band network. Each node includes 2 Intel(R) Xeon(R) E5220 quad core. Thus, the total number of cores is 128.

The software used are Linux as operating system, DAGuE as runtime system and the Intel Mkl library version 11.1 for LAPACK and Blas routines. We used the sequential version of BLAS.

For the experience, we provide a *gemm* peak performance of the patform. It is measured as the best performance obtained by a single core to compute a matrix multiplication using the *gemm* BLAS routine. For our platform, one core *gemm* peak is around 9.3 Gflop/s. This measurement is then multiplied by the total number of cores used. Thus, for 128 cores, the *gemm* peak obtained is 1190 Gflop/s. It is considered as the practical peak performance of the platform.

However, the *gemm* peak will never be reached by the LU decomposition algorithm. In order to have a better upper bound for the LU decomposition with partial pivoting, we also implemented an LU decomposition without pivoting. It is similar to a partial pivoting but without swapping operation. It may be considered a Cholesky's algorithm applied to the two sides of the matrix.

The partial pivoting implemented was run with only 7 cores by node for computation. The eighth core was completely dedicated to the communication. In fact, experiments show that is more efficient to allow one core to manage the huge number of small communications on the panel. Thus, in the Figure 6.1, it is showed two *gemm* peaks, one using 8 cores per node and the second using only 7 cores per node. Thus, the dashed curves of the Figure represent the measurements done with 7 cores per node and the solid curves shows mesurements executed with 8 cores per node.

DAGuE includes already an implementation of LU decomposition with incremental pivoting. Thus, we use also to compare with the performances obtained of partial pivoting.

To complete the experiences, we also compare results with ScaLAPACK which

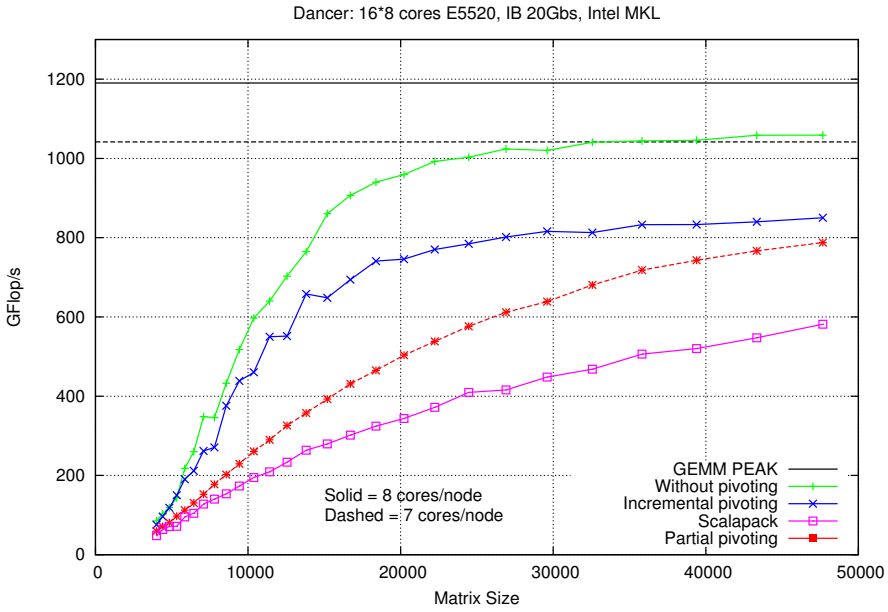


Figure 6.1: Performance of LU decomposition with PTG

is a mathematical library for parallel distributed memory machines using the LAPACK library. For that, we used the Netlib[1] version. ScaLAPACK was run by assigning one process MPI per core.

We implemented Task flow 3 for panel factorization and Task flow 5 for swapping operations in update. We used a 2D block cyclic distribution for matrix. The size of tile used was 200 for DAGuE and 120 for ScaLAPACK and has been chosen to get the best asymptotic performance. The matrices where generated randomly with the same side for all measurements.

For each measurement, we do five iterations of the execution, we do not take in count the maximal and minimal values obtained. We display the average of the three other values. In most of the tests, we obtain a low standard deviation (less than 1 Gflop/s).

The results obtained are encouraging because our implementation outperform the ScaLAPACK implementation and reached 75% of the GEMM peak. However, the incremental pivoting algorithm which enables more parallelism and less synchronization in the panel factorization still provide better performance.

[1] `http://www.netlib.org/`

Conclusion

Solving large systems of linear equations is one of the most important operations in matrix computation and represents a time-consuming step, arising in many scientific and engineering applications. The LU decomposition algorithm is used in most state of the art libraries such as in Matlab, Lapack and HPL libraries. Its expression is much more complex because part of the operations to be performed have to be scheduled dynamically based on numerical criteria. Despite this difficulty, we managed to implement a task flow LU decomposition with PTG and obtained promising performances.

We try now to optimize the DAGuE runtime system to better manage the small communications messages, in order to take full advantage of all cores of nodes. Thanks to this representation of the algorithm, we hope to create an algorithm that will fully exploit heterogeneous architectures by offloading GEMM to accelerators as GPUs or Intel MIC on distributed architectures. Nowadays, GPU implementations of the LU partial pivoting are either relying on the ScaLAPACK sequential execution of the algorithm or HPL implementation and basically offload the GEMM on the GPU, or are able to fully exploit a single heterogeneous node by using block-column distribution. The first solution prevents enabling dynamic scheduling and automatic look-ahead specific to tile algorithm, while the second solution would be a disaster in the load balance on distributed architectures where a 2D block-cyclic distribution is required to decrease the volume of communication and average out the load over the multiple nodes. Thus, our implementation of LU decomposition over DAGuE will be the right solution to keep an automatic scheduling and good load balance on heterogeneous architectures to achieve better performances.

In parallel, we implemented another task flow LU decomposition on StarPU. In fact, StarPU runtime allow to implement dynamic algorithm thanks to its task insertion system. Moreover, it allows to make *reduce* operation which enable to select the best pivot without using tree implementation. For now, the results obtained are not efficient. We are still investigating to understand the lack of performance. Thereafter, we will be able to compare results to PTG in general and DAGuE particularly.

Bibliography

[1] Sterling, T.: HPC in phase change: Towards a new execution model. In: HPCCS – VECPAR 2010. Volume 6449 of LNCS. Springer (2011) 31–31

[2] Perez, J., Badia, R., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: Cluster Computing, 2008 IEEE International Conference on. (2008) 142 –151

[3] Song, F., YarKhan, A., Dongarra, J.J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM (2009) 1–11

[4] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing. Parallel Computing **38**(1–2) (2012) 37 – 51

[5] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2) (2011) 187–198

[6] Cosnard, M., Jeannot, E., Yang, T.: Compact dag representation and its symbolic scheduling. Journal of Parallel and Distributed Computing **64**(8) (2004) 921–935

[7] Message Passing Interface Forum: MPI: A message-passing interface standard. To appear in the International Journal of Supercomputing Applications **8**(3/4) (1994) Computer Science Department Technical Report CS-94-230 may'94.

[8] Kale, L.V., Krishnan, S.: CHARM++ : A portable concurrent object-oriented system based on C++. In Paepcke, A., ed.: Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA), ACM Press (September 1993) 91–108

[9] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The plasma and magma projects. Journal of Physics: Conference Series **180**(1) (2009) 012037

[10] Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: A programming model for the cell BE architecture. In: SC'2006 Conference CD, Tampa, FL, IEEE/ACM SIGARCH (November 2006)

[11] Badia, R.M., Herrero, J.R., Labarta, J., Pérez, J.M., Quintana-Ortí, E.S., Quintana-Ortí, G.: Parallelizing dense and banded linear algebra libraries using SMPSs. Concurrency and Computation: Practice and Experience **21**(18) (2009) 2438–2456

[12] Ayguade, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Orti, E.S.: An extension of the starss programming model for platforms with multiple GPUs. In Sips, H.J., Epema, D.H.J., Lin, H.X., eds.: Euro-Par 2009 Parallel Processing (15th Euro-Par'09). Volume 5704 of Lecture Notes in Computer Science (LNCS). Springer-Verlag (New York), Delft, The Netherlands (August 2009) 851–862

[13] Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment. In: Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'2007), Boston, Massachusetts, USA (October 2007) 1–5

[14] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2) (2011) 187–198

[15] Higham, N.: Accuracy and Stability of Numerical Algorithms, Second Edition. SIAM (2002)

[16] Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P., Tomov, S.: The impact of multicore on math software. In Kågström, B., Elmroth, E., Dongarra, J., Wasniewski, J., eds.: PARA. Volume 4699 of Lecture Notes in Computer Science., Springer (2006) 1–10

[17] Chan, Quintana-Orti, Quintana-Orti, van de Geijn: Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In: SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures. (2007)

[18] Buttari, A., Langou, J., Kurzk, J., Dongarra, J.J.: A class of parallel tiled linear algebra for multicore architectures. Parallel Computer Systems Appl. **35** (2009) 38–53

[19] Quintana-Ortí, E.S., Van De Geijn, R.A.: Updating an LU factorization with pivoting. ACM Transactions on Mathematical Software **35**(2) (July 2009) 11:1–11:16

[20] Grigori, L., Demmel, J., Xiang, H.: CALU: A communication optimal LU factorization algorithm. SIAM J. Matrix Analysis Applications **32**(4) (2011) 1317–1350

[21] Bruck, Ho, Kipnis, Upfal, Weathersby: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEETPDS: IEEE Transactions on Parallel and Distributed Systems **8** (1997)

[22] Dongarra, J., van de Geijn, R., Walker, D.: A look at scalable dense linear algebra libraries. In: Proceedings of the 1992 Scalable High Performance Computing Conference, Williamsburg, VA (April 1992) 372–379

[23] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK: A portable linear algebra library for high-performance computers. LAPACK Working Note 20, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, USA (May 1990) UT-CS-90-105, May 1990.