Enabling Partial Pivoting in Task Flow LU Factorization

Omar ZENATI

ENSEIRB-MATMECA
Computer Science Department
Internship Report

Tutors : Pierre RAMET & George BOSILCA

Bordeaux September 1, 2012

Abstract

Thanks to its high efficiency and numerical accuracy, LU with partial pivoting is a cornerstone algorithm in modern science. Yet, its data dependent nature, due to pivoting and swapping, is difficult to capture for emerging task flow programming models and runtimes. In this paper we design and evaluate a version of the LU algorithm with partial pivoting that can be expressed as a static graph suitable for efficient dataflow scheduling.

Contents

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 paper 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 paper is organized as follows. Section 2 presents related work in the field of task flow scheduling and implementations of the LU algorithm. Then, sections 3 and 4 further discuss the issues raised by partial pivoting in PTG task flow systems, and the solutions we propose to tackle these challenges. Section 5 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

Background

Twenty years ago, a computer with only one single processor may be considered as computing platform and be included in the Top 500 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** architecture, it is a computing platform consisting many cores and all have access to the same virtual memory. The **distributed memory** architecture is composed of at least two node. Each node has its own cores and its physical memory. Cores of different nodes cannot access to the virtual memory of each other, and thus, must communicate to share data. Even if in a distributed memory architecture, a node can consist many cores, we will use - in this paper - the denomination *distributed memory architecture* only for nodes with one single core. In contrast to the **hierarchical architecture** which is a distributed memory architecture where at least one node is a shared memory multi-cores architecture. Moreover, the **heterogeneous architecture** is a distributed memory architecture where at least one node is a GPU.

2.1 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].

In order to ease efficient use of supercomputers and to introduce an abstraction to computers 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 which run codes with implicit data dependencies and those which 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 architecture [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 allow 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 paper, we focus on runtimes using explicit data dependencies. To illustrate the benefits on hierarchical computers (**??**), we will use the DAGuE runtime system.

2.2 LU Decomposition Algorithm

The LU decomposition algorithm is based on the Gauss elimination method and consists of factorizing a matrix into a product of a lower and an upper triangular matrix. In order to obtain a good accuracy, the swaps are - most of the time - needed. There is a lot of heuristic to perform the swapping, the most used by the scientist community is the partial pivoting for its practical stability and accuracy [15], it is also used in the LINPACK benchmark which is used to rank the TOP 500 super-computers.

To illustrate an example of LU decomposition, we consider a $n*n$ square matrix. 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 (right looking) 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 2.1 shows an example of matrix in the second step of the LU decomposition, the second panel is still being factorized, and after, the trailing sub-matrix will be updated.

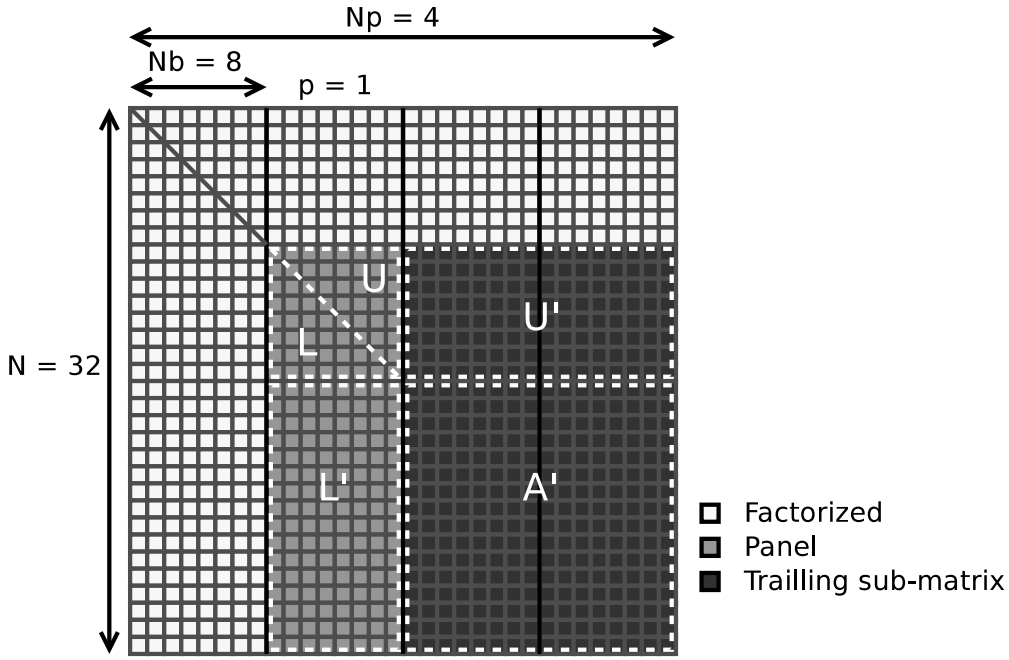


Figure 2.1: LU decomposition at step p on panel-blocked matrix

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 consist to divide all the element of the column i by the selected pivot (Figure 2.2 show the different parts of the panel). After that, the trailing sub-panel is updated with an outer product - so called *ger* according to the BLAS reference -. The panel factorization produces an array of size n_b containing the pivots selected, we note *ipiv* this array. 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 then 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
            $a_{i,j} = a_{i,j} - a_{i,k} * _{k,j}$

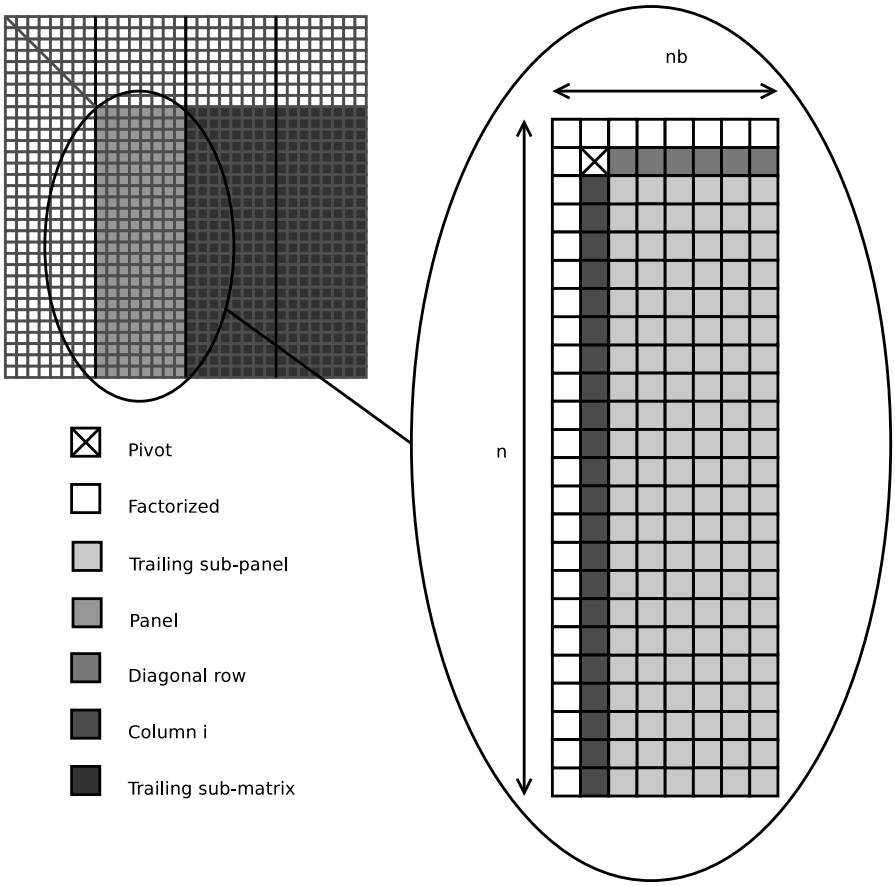


Figure 2.2: Second iteration of panel factorization (after swap)

Once the p_{th} panel is factorized, the subsequent trailing sub-matrix is updated. This update depend on the panel factorization. It takes as entry the pivot information and then applies the permutations on the whole trailing sub-matrix. Once

the swap have been performed on the trailing sub-matrix, the n_b block row corresponding to the eliminated rows (block U' in Figure 2.1) is updated. For that, the following equation must be solved:

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

Which is equivalent to:

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

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*.

2.3 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 submatrices, also called tiles [16, 17]. Figure 2.3 shows a matrix partitioned into tiles. In [18], the authors proposed a new pivoting strategy called *incremental pivoting* based on [19] more suitable tile algorithms and achieved high performance by limiting the number of synchronizations. 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 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

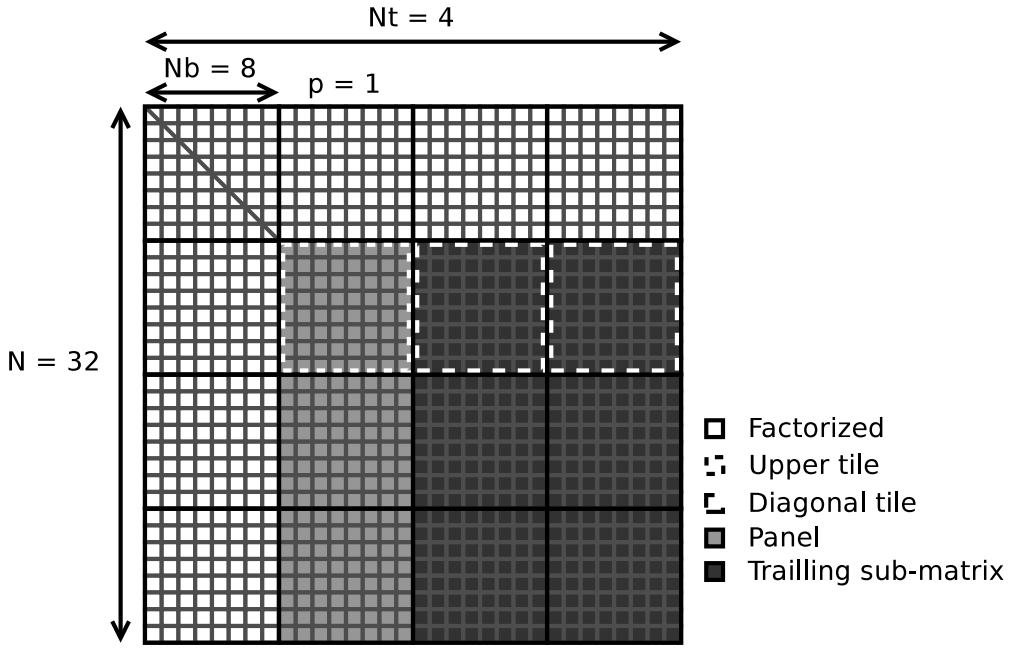


Figure 2.3: LU decomposition at step p on tiled matrix

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 2.4 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 contain 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 2.5 represent the same dynamic algorithm of Figure 2.4 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.

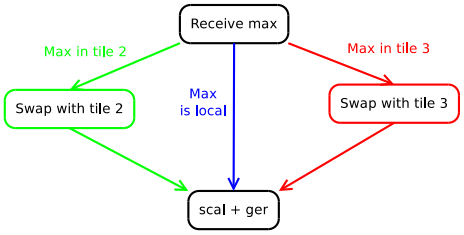


Figure 2.4: Dynamic representation of task flow

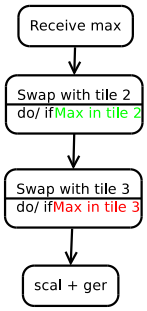
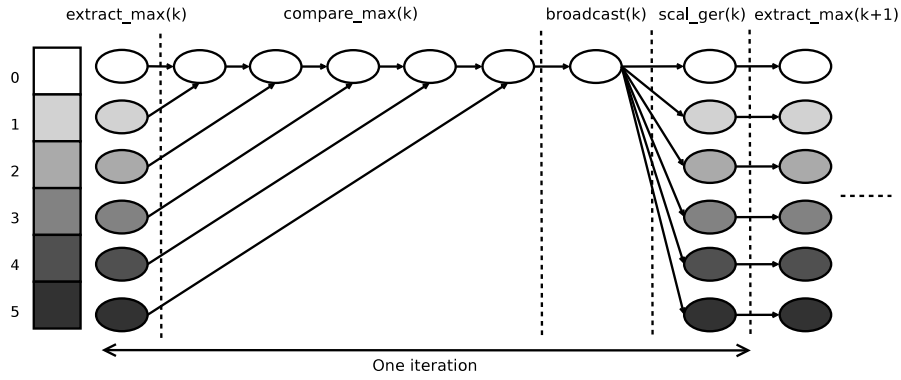


Figure 2.5: Static representation of task flow

Chapter 3

Panel Factorization with Partial Pivoting

In order to reduce the important number of tasks and communications, optimisations had 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 2.2, the panel factorisation 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 contain the diagonal node compare progressively its maximum with others tiles. Each time it find a better pivot, it saves it and uses it for next comparisons. After this operation, the diagonal tile broadcasts the initial diagonal row and the selected row owning the column maximum to all others tile 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 for distributed architecture.

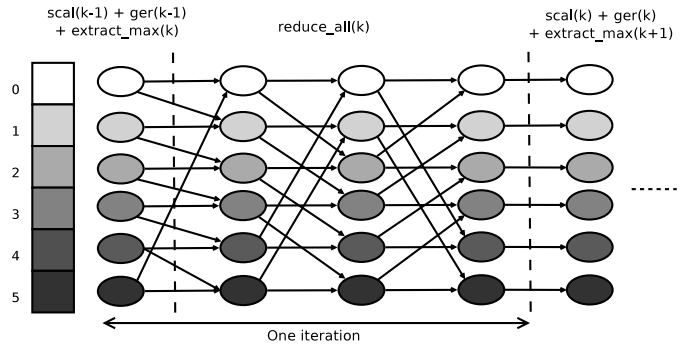


Task flow 1: One iteration of panel factorization on distributed architecture

The first remark is that there is 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 consisting the

maximum 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. 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.

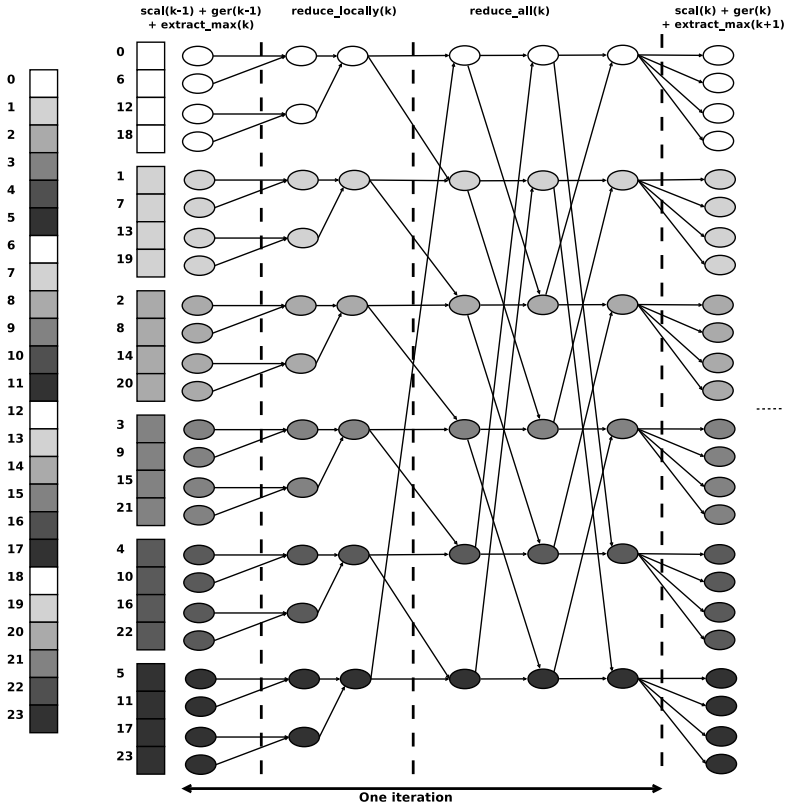
Concerning communications, DAGuE and other runtimes handle point-to-point communications and can manage some collective communications as broadcast or group communications. But, to perform more complex communications operations (reduce, gather...), we have to express them as a task flow. This is due to the fact that a complex communications are not expressible with PTG. 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 resulting maximum row). 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 all tiles with their maximum row. At each step of the *all_reduce* operation, task copy first row from the received workspace if it is not empty, and reduce the maximum row in the second row of their workspace. 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.



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 hier-

archical architecture. As for task flow 2, a similar workspace is used for *all_reduce* communications with one difference. The workspace contains four rows: two for the diagonal row and two for the maximum row. In fact, at each step of the panel factorization, only one of the two row is used depending on the step parity. This configuration is necessary to protect from the *read after write* effect on the workspace which is shared between all tiles of the same node.



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 optimization on kernel executed 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 library [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 factoriza-

tion, 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 3.1).

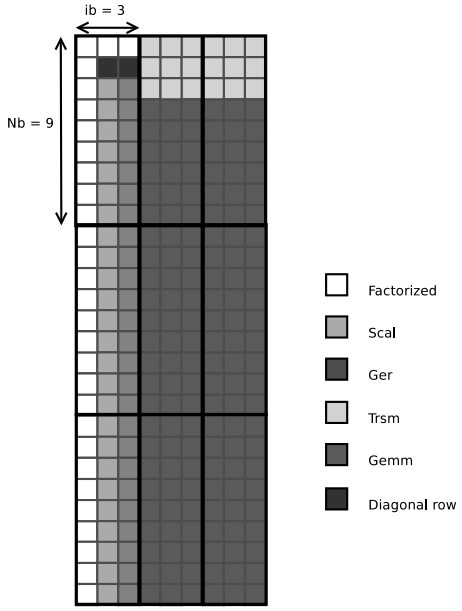


Figure 3.1: Panel factorization with internal blocking

Chapter 4

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 depend on the value of the pivots, this is what we call data dependency and so the algorithm is a dynamic algorithm (section 2.3). 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 arrays: 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. The cost of a *all_reduce* operation is $log_2(n_t)$. Thus, the cost of all swaps of one panel is $n_b * log_2(n_t)$. This is very expensive relatively to the cost of SPMD model which is at most $2 * n_b$. Moreover, a 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 exchange with it).

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 be several time the maximum row, it is necessary to execute the pivots in the right order (from the first to the last). Figure 4.1 shows an example of a row which contains successively two time the maximum row, 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.

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 4.2 shows the use of permutations instead of pivots which is showed in Figure 4.1, we can see that the row 1 goes directly to the row 2 and does not move

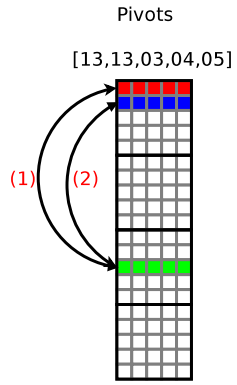


Figure 4.1: Row movements with pivots

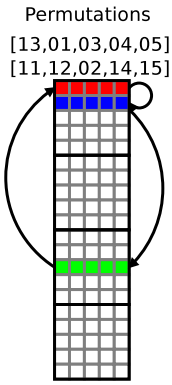


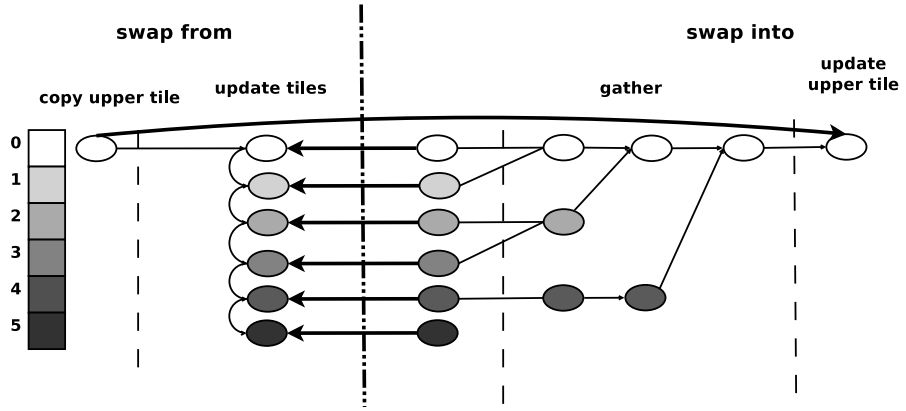
Figure 4.2: Row movements with permutations

again. Thanks to this structure, all the rows can be swapped in one single step and the cost will be just $log_2(n_t)$. 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 nb elements will be used to store permutations and the second nb 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 buffer - with size of tile - for communications: the first is a copy of the upper tile, it is shared from one node to the others. Each node will extract from the copy rows that it needs. We will call this operation *swap from*. The second buffer is used to gather rows required by 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 allow us to perform the *swap from* and the *swap into* in parallel.

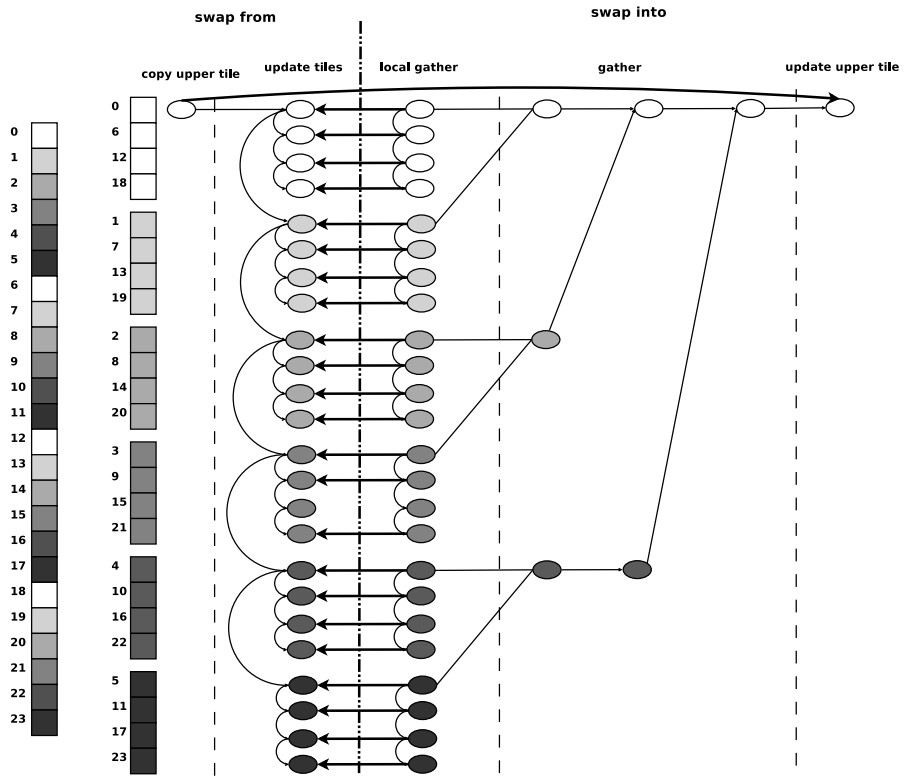
Task flow 4 represents the swapping operation of update operation for distributed architecture. The bold arrows show some dependencies which add synchronizations to avoid *read after write* effects. For example, the copy of the upper tile must be applied before its update.

As for the panel factorization, we consider that nodes can be multi-core. In order to reduce global communication, each node shares its buffers over its local tiles before to send them to others nodes. Task flow 5 shows the update operation for hierarchical architecture.

Moreover, this update algorithm implemented is a generic solution that it can execute update operation after any panel factorization which provide 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 5

Experimental

In order to have an upper bound for the LU decomposition with partial pivoting, we also implemented an LU decomposition with static pivoting. It is similar to a partial pivoting but without swapping operation. It may be considered as a Cholesky's algorithm applied to the two sides of the matrix.

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

To complete the experiences, we also compare results with Scalapack performances.

Exploiting hierarchical platform with PTG

For this experience, we used DAGuE as PTG runner. The architecture used is and hierarchical platform. It consists of 128 cores - Intel(R) Xeon(R) E5220 - distributed over 16 nodes and interconnected with an Infini Band. Each core runs with a frequency of 2.27GHz.

We implemented Task flow 3 for panel factorization and Task flow 5 for swapping operations in update. The softwares used are Linux as operating system, DAGuE as runtime system and the Intel Mkl library version 11.1 for Blas routines. We used the sequential version of BLAS in DAGuE but the Scalapack tests ran with parallel BLAS routines (PBLAS).

The partial pivoting implemented was ran 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. In the figure 5.1, it is showed two *gemm* peak, one using 8 core and the other only 7. The results obtained are satisfying. In fact, the red curve - which is our implementation of LU decomposition over DAGuE- reaches 75% of the *gemm* peak. Moreover, the performance of

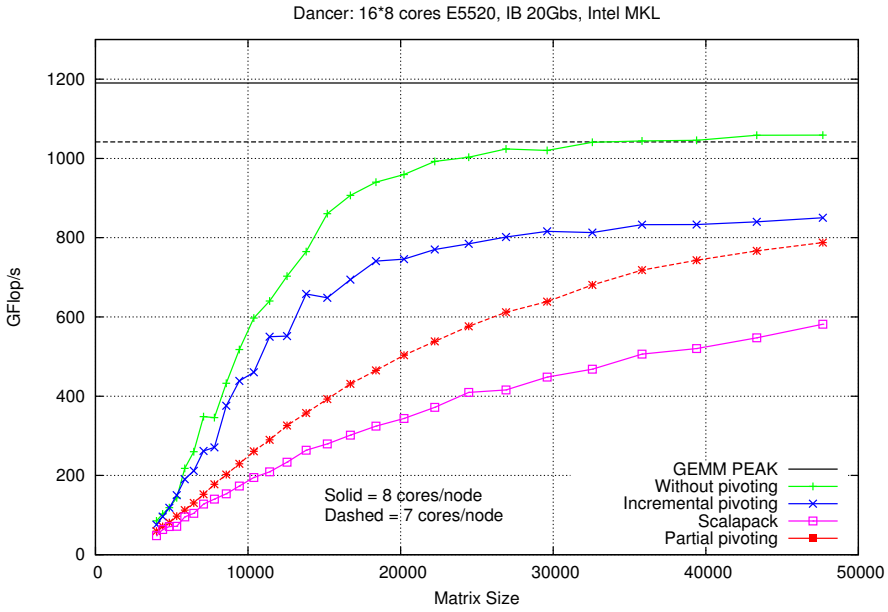


Figure 5.1: Performance of LU decomposition with PTG

our LU decomposition is far better than Scalpack which is one of the most used mathematical library by the scientist community. The curves of the LU decomposition with incremental pivoting and without pivoting have better performances because their algorithms have less synchronisations, but their numerical values of their results are not accurate.

Conclusion

LU decomposition with partial pivoting is a dynamic algorithm with its data dependencies in the swapping operation for the update. Despite this difficulty, we managed to implement a task flow LU decomposition with PTG and obtained hopeful 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. We will be able to have one of the most efficient LU decomposition implementation. Beside its use for resolving system of linear equations, it will be possible to create a new benchmark to analyse computers 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. 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.