Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures

Mohamed Wissam Sid Lakhdar

▶ **To cite this version:**

Mohamed Wissam Sid Lakhdar. Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures. Other [cs.OH]. Ecole normale supérieure de lyon - ENS LYON, 2014. English. NNT : 2014ENSL0958 . tel-01111259

HAL Id: tel-01111259

https://inria.hal.science/tel-01111259

Submitted on 30 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon, délivré par l'École Normale Supérieure de Lyon

Discipline : Informatique

Laboratoire : Laboratoire de l'Informatique du Parallélisme - UMR5668 - LIP

École Doctorale : Informatique et Mathématiques

présentée et soutenue publiquement le 1^{er} Décembre 2014

par Monsieur Wissam M. SID-LAKHDAR

Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures

Directeur de thèse : Jean-Yves L'EXCELLENT

Devant la commission d'examen formée de :

Mr Patrick AMESTOY	*Invité*
Mr Iain DUFF	*Rapporteur*
Mr Jean-Yves L'EXCELLENT	*Directeur*
Mme Xiaoye Sherry LI	*Examinateur*
Mr Jean-François MÉHAUT	*Rapporteur*
Mr François PELLEGRINI	*Examinateur*
Mr Antoine PETITET	*Examinateur*

Abstract

The solution of sparse systems of linear equations is at the heart of numerous application fields. While the amount of computational resources in modern architectures increases and offers new perspectives, the size of the problems arising in today's numerical simulation applications also grows very much. Exploiting modern architectures to solve very large problems efficiently is thus a challenge, from both a theoretical and an algorithmic point of view. The aim of this thesis is to address the scalability of sparse direct solvers based on multifrontal methods in parallel asynchronous environments.

In the first part of this thesis, we focus on exploiting multi-threaded parallelism on shared-memory architectures. A variant of the Geist-Ng algorithm is introduced to handle both fine grain parallelism through the use of optimized sequential and multi-threaded BLAS libraries and coarser grain parallelism through explicit OpenMP based parallelization. Memory aspects are then considered to further improve performance on NUMA architectures: (i) on the one hand, we analyse the influence of memory locality and exploit adaptive memory allocation strategies to manage private and shared workspaces; (ii) on the other hand, resource sharing on multicore processors induces performance penalties when many cores are active (machine load effects) that we also consider. Finally, in order to avoid resources remaining idle when they have finished their share of the work, and thus, to efficiently exploit all computational resources available, we propose an algorithm which is conceptually very close to the work-stealing approach and which consists in dynamically assigning idle cores to busy threads/activities.

In the second part of this thesis, we target hybrid shared-distributed memory architectures, for which specific work to improve scalability is needed when processing large problems. We first study and optimize the dense linear algebra kernels used in distributed asynchronous multifrontal methods. Simulation, experimentation and profiling have been performed to tune parameters controlling the algorithm, in correlation with problem size and computer architecture characteristics. To do so, right-looking and left-looking variants of the LU factorization with partial pivoting in our distributed context have been revisited. Furthermore, when computations are accelerated with multiple cores, the relative weight of communication with respect to computation is higher. We explain how to design mapping algorithms minimizing the communication between nodes of the dependency tree of the multifrontal method, and show that collective asynchronous communications become critical on large numbers of processors. We explain why asynchronous broadcasts using standard tree-based communication algorithms must be used. We then show that, in a fully asynchronous multifrontal context where several such asynchronous communication trees coexist, new synchronization issues must be addressed. We analyse and characterize the possible deadlock situations and formally establish simple global properties to handle deadlocks. Such properties partially force synchronization and may limit performance. Hence, we define properties which enable us to relax synchronization and thus improve performance. Our approach is based on the observation that, in our case, as long as memory is available, deadlocks cannot occur and, consequently, we just need to keep enough memory to guarantee that a deadlock can always be avoided. Finally, we show that synchronizations can be relaxed in a state-of-the-art solver and illustrate the performance gains on large real problems in our fully asynchronous multifrontal approach.

Acknowledgment

Je tiens tout d'abord à remercier mon encadrant, Jean-Yves L'EXCELLENT. J'ai eu la chance de le connaitre et de le côtoyer, et ai pu me rendre compte de ces qualités remarquables, aussi bien humaines que professionnelles. Durant des années, il a consacré un temps incommensurable à me former, à m'encourager et à me guider, outrepassant largement son rôle d'encadrant. Pour tout cela et bien plus encore, merci Jean-Yves.

Je souhaite ensuite remercier Patrick AMESTOY, de m'avoir suivi, des premiers pas en master jusqu'à l'achèvement de la thèse, de m'avoir promulguer de très précieux conseilles, et surtout, de m'avoir permis d'acquérir cette vision globale des choses, fondamentale à la recherche scientifique.

J'aimerai tout particulièrement remercier Iain DUFF et Jean-François MÉHAUT de m'avoir fait l'honneur d'être les rapporteurs de ma thèse. Leurs remarques m'ont été d'une grande aide, et m'ont permis d'améliorer la qualité de mon manuscrit.

Je voudrai aussi remercier Sherry LI, François PELLEGRINI et Antoine PETITET de bien avoir accepté d'être membres de mon jury, et d'avoir fait l'effort de voyager de très loin a cette occasion.

Aussi, je tiens à remercier les membres de l'équipe MUMPS et de l'équipe ROMA (ainsi que de membres d'autres équipes :-), d'avoir été des amies et même une seconde famille, et pour tout ces moments forts agréables et enrichissants que l'on a passé ensemble.

Enfin, j'aimerai remercier les personnes sans qui rien n'aurai été possible, a savoir, ma famille, et tout particulièrement, mon père et ma mère. Ils ce sont sacrifiés toute leur vie pour nous permettre a mon frère et a moi même de nous épanouir et d'atteindre nos objectifs.

La dernière personne que j'aimerai remercier, non des moindres, est mon oncle et mentor: Daddy. Aussi longtemps que je me souvienne, il a toujours joué et continue à jouer un rôle clef dans ma vie. Je lui dédie cette thèse.

Contents

General Introduction

At the time when scientists started using computers for solving linear systems of equations, they were still able to solve them by hand. Since that time, each increase in the problem size has induced the need for more powerful computers. In turn, the fulfilment of this need has opened the door to ever increasing problem sizes and has brought an increasing interest in the subject. The solution of sparse systems of linear equations is nowadays at the heart of numerous application fields. Such systems now contain billions of unknowns, making them hard to solve without involving advanced algorithms and techniques. Modern computers, which are being used to solve them, have been also increasing in size. They now contain hundreds of thousands of computing units, at the forefront of technology, organized into structures of increasing complexity, with distributed-memory machines composed of nodes interconnected with hierarchies of networks, each shared-memory node containing, in turn, hierarchies of processors and memories, called NUMA[1] architectures. Exploiting modern architectures to solve very large problems efficiently has become a challenge, from both a theoretical and an algorithmic point of view. For instance, as the cost of synchronizations increases along with the number of computing resources, it has become worthwhile to handle parallelism in an asynchronous way.

The aim of this thesis is to address the scalability of the solution of such systems in parallel asynchronous environments. We consider here the case of the LU factorization of a sparse matrix using the so-called multifrontal method, where the factorization of a sparse matrix boils down to a succession of computations on smaller dense matrices. We provide the necessary background in Chapter 1. The rest of the thesis is then organized in the following two parts.

In the first part, we focus on improving the exploitation of multithreaded parallelism on shared-memory architectures. An algorithm is introduced to handle both fine and coarser grain parallelisms through the use of optimized sequential and multithreaded linear algebra libraries and through the use of explicit parallelization directives, respectively. In order to take advantage of modern NUMA architectures, it is necessary to consider memory aspects. On the one hand, we analyse the influence of memory locality and exploit adaptive memory allocation strategies to manage data over threads. On the other hand, we also consider resource sharing and machine load effects on multicore processors, which induces performance penalties when many cores are active simultaneously. Finally, to further improve performance, we try to exploit efficiently all computational resources available by avoiding resources remaining idle when they finish their share of the work. We propose an algorithm which is conceptually very close to the work-stealing approach and that consists in dynamically assigning idle cores to busy threads/activities.

In the second part of the thesis, we aim at improving the scalability of processing large problems on hybrid shared-distributed memory architectures.

[1] Non-Uniform Memory Access.

As they lie at the heart of distributed asynchronous multifrontal methods, we will first study the specific dense linear algebra kernels used in such methods, through simulations, experimentations and profiling. We then optimize them and tune the parameters controlling them, that depend on problem sizes and computer architecture characteristics. Because of their nice numerical properties, we are concerned with distributions of dense frontal matrices in a single dimension (so called 1D), and we revisit right-looking and left-looking variants of the 1D LU factorization with partial numerical pivoting within a distributed-memory context. Additionally, when computations are accelerated with multithreaded shared-memory parallelism, the relative weight of communication with respect to computation increases.

In particular, when the sets of processes which are used in two related tasks are distinct, a type of communications that can be costly may consist in migration operations aiming at ensuring a good load balance. We therefore explain how to design efficient mapping and remapping algorithms that minimize the amount of communication associated with such operations.

Furthermore, on large numbers of processors, one-to-all asynchronous communications become a bottleneck for performance. We thus explain how asynchronous broadcast algorithms can be used effectively in a fully-asynchronous environment. When several such broadcasts coexist, deadlock issues can arise because of limited memory for communication. Managing them often comes at the price of synchronizations which hinder performance. We will therefore discuss deadlock prevention and deadlock avoidance approaches that relax synchronizations (and maximize performance) while controlling memory for communication.

Throughout this thesis, we will use a state-of-the-art asynchronous sparse direct solver to validate our algorithms. This allows us to illustrate performance gains with experiments on large real-life problems on modern computer architectures.

This work was granted access to the HPC resources of CALMIP under the allocation 2014-P0989 and to GENCI resources under allocation x2014065063.

Chapter 1

Background

1.1 Introduction

In this chapter, we give an overview of the existing methods to solve linear systems of the form

$$Ax = b \ , \tag{1.1}$$

where A is a **large sparse** matrix of order n, and where x and b are vectors (or matrices) of size n (or $n \times m$). Given the matrix A and the right-hand side b, the goal is to find the unknown x.

In this chapter, we give a short overview of existing methods to solve this problem and provide some background that will be useful for the remainder of the thesis. We will guide the reader through the successive steps, from the (possible) origin of the problem to its solution, based on an illustrative example from a typical physical application.

Illustrative example Let us suppose that a physicist needs to simulate a phenomenon, such as, for example, the thermodynamic evolution of a simple metal plate. The physical law governing the evolution of temperatures in this system is given by the heat equation, described by the following partial differential equation

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0 \ , \tag{1.2}$$

where $u \equiv u(x, y, t)$ is the temperature at a given position in space and time, and α is the thermal diffusivity of the plate. This equation comes along with additional initial and boundary conditions appropriate to the problem at hand.

In practice, exact analytical solutions $u(x, y, t)$ that give the values of u at every point and any time are simply out of reach. Instead, we usually try to find approximate solutions restricted to a discrete subset of points and at discrete time intervals. The widely used methods to find such approximate solutions are the finite-difference method (FDM) and the finite-element method (FEM). The main steps of these approaches are the following.

The first step consists in *discretizing* the system by choosing a relevant subset of points (or polygons forming a *mesh*) that will represent it. For the sake of simplicity, we will consider in our example a finite-difference discretization of the plate using only 9 points (See Figure 1.1a).

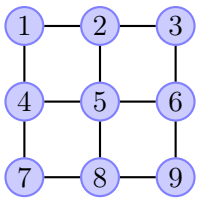
The second step then consists in *applying the PDEs* on the discretized mesh, taking into account the dependencies between neighbours. This step typically relies on an approximation of the derivative of a function f at a given point a in space or time, which can take the form:

$$f'(a) \approx \frac{f(a+h) - f(a)}{h} \ ,$$

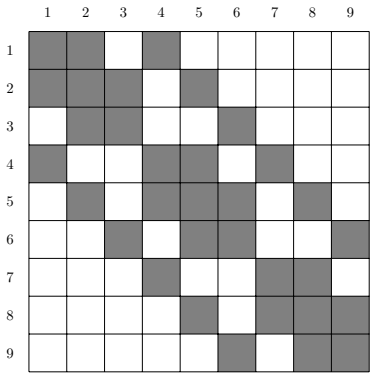
where the discretization step h should be sufficiently small for the approximation to be accurate.

Considering that $a - h$, a, and $a + h$ are part of the discretization, second order derivatives can also be approximated:

$$f''(a) \approx \frac{f(a+h) - 2f(a) + f(a-h)}{h^2} \quad . \tag{1.3}$$



(a) Mesh



(b) Sparse matrix. Nonzero (gray). Zero (white).

Figure 1.1: Discretization mesh and corresponding sparse matrix

The so-called *explicit methods* can then be applied to directly compute the approximate solution at any point and at any time $t + \Delta t$, given the state of its neighbours (and itself) at time t. However, due to stability issues, such methods tend to be inapplicable in practice. Indeed, the stability condition generally comes in the form $\Delta t < h^2 * constant$. Therefore, choosing a small space step h (to get a reasonably good spatial resolution) will require a prohibitively small time step Δt. The so-called *implicit methods* may be applied instead. They are not subject to any stability condition but require the solution of a linear system at each time step. They rely on a formulation of the equations yielding a linear system of equations, where the old state of the system is represented by the vector b, while the new state of the system, which is given by x, must be found through the solution of the linear system. As an example, let us consider the case of equation (1.2) applied to the point (numbered 5) at the middle of the mesh in Figure 1.1a. We note $u_i(t)$ the approximate value of u at point i of the mesh at time t. Approximating the time derivative by $\frac{u_5(t+\Delta t) - u_5(t)}{\Delta t}$, and generalizing the second derivative approximation of equation (1.3) to the Laplacian of the heat equation in two dimensions, we obtain one of the equations of the aforementioned linear system:

$$u_5(t+\Delta t) - \frac{\alpha \Delta t}{h^2}(u_2(t+\Delta t) + u_4(t+\Delta t) - 4u_5(t+\Delta t) + u_6(t+\Delta t) + u_8(t+\Delta t)) = u_5(t) \quad .$$

After forming the linear system, the next step is the one that concerns us, i.e. finding x (in Equation (1.1)). The key point to notice here is that the matrix A generated by an implicit method is usually **sparse**, i.e. with many **zero** entries (See Figure 1.1b).

Indeed, as each row or column of the matrix represents a link between the corresponding vertex in the mesh and a neighbour, and as the number of neighbour vertices of each vertex is usually limited, the number of nonzero entries (denoted by NNZ) of the matrix is usually $O(n)$. This sparsity of the matrix makes the exploitation of the zero entries worthwhile for reducing computations and resources, as will be shown in Section 1.3.

1.2 Solution methods

The mathematical solution of the problem defined by equation (1.1) is simply

$$x = A^{-1}b \ . \tag{1.4}$$

However, using A^{-1} explicitly is computationally tough and often numerically unstable. There are two main families of approaches for the solution of linear systems: **direct methods** and **iterative methods**.

1.2.1 Direct methods

In general, it is hard to find the solution of a system immediately. However, methods do exist for matrices with special characteristics.

For example, when the matrix is diagonal ($A \equiv D$), the solution is simply given by

$$x_i = \frac{b_i}{D_{ii}} \ . \tag{1.5}$$

Moreover, when the matrix is orthogonal ($A \equiv Q$), as the inverse of an orthogonal matrix coincides with its transpose, the solution is given by

$$x = Q^T b \ . \tag{1.6}$$

Furthermore, when the matrix is triangular, say lower triangular, ($A \equiv L$), the solution is obtained by successively computing each x_i as:

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} L_{i,j}x_j}{L_{i,i}}, \ i = 1, \cdots, n \ . \tag{1.7}$$

The idea behind direct methods is to **decompose** or **factorize** the (general) matrix A into a product of (special) matrices, for which it is easier to find a solution

$$A = A_1 \times A_2 \times \cdots \times A_k \ . \tag{1.8}$$

The solution of the system is then obtained by solving each subsystem successively, using

$$\left\{ \begin{array}{rl} y_1 & = A_1^{-1}b \ , \\ y_{i+1} & = A_{i+1}^{-1}y_i, \ i = 1, \cdots, k-1 \ , \\ x & = y_k \ . \end{array} \right. \tag{1.9}$$

Various direct methods exist, with various computational costs and various advantages. For general square unsymmetric matrices, the LU **decomposition** may be used. It consists in factorizing A as

$$A = LU \ , \tag{1.10}$$

where L is a lower triangular matrix with 1's on the diagonal and U is an upper triangular matrix. The solution x of the system is then obtained by applying successively two triangular solution schemes: a **forward elimination** on L followed by a **backward substitution** on U.

A possible implementation of the LU decomposition is depicted in Algorithm 1.1, where the matrix A is overwritten by its factors L and U.

for $p = 1$ **to** $n - 1$ **do**
$A(p+1:n, p) = \frac{A(p+1:n)}{a_{pp}}$;
$A(p+1:n, p+1:n) = A(p+1:n, p+1:n) - A(p+1:n, p) \times A(p, p+1:n)$

Algorithm 1.1: LU decomposition algorithm (algorithm of Doolittle). A is overwritten by its LU factors such that the upper triangular part of A (including the diagonal) is the U factor, and the strictly lower triangular part of A is that of the L factor (as the diagonal of L is composed of 1's and is not stored).

Some variants exist that can help increase the performance in the case where the matrix A exhibits some special properties. For example, the LDL^T decomposition can be applied on symmetric matrices. Moreover, the LL^T decomposition, also known as the **Cholesky** decomposition, can be applied on a symmetric positive definite matrix. In addition, other very important methods exist, which can be applied to the general case, such as the QR and the SVD decompositions.

In this thesis, we will focus on the LU decomposition, although most algorithms and ideas can also be applied to LDL^T and LL^T decompositions.

1.2.2 Iterative methods

The main principle of an iterative method is to generate a sequence of iterates $x^{(k)}$, which converge to the solution of the linear system $Ax = b$. These essentially involve matrix-vector products, but are often combined with preconditioning techniques, where one considers the linear system $(MA)x = Mb$ where MA has better convergence properties than A (ideally $M = A^{-1}$) and where the systems $My = z$ are easier to solve than $Ax = b$.

Basic iterative methods (stationary methods) compute the next iterate $x^{(k+1)}$ as a function of $x^{(k)}$ only (Jacobi method) or as a function of $x^{(k)}$ and the already updated elements of $x^{(k+1)}$ (Gauss-Seidel method). More advanced methods like Krylov methods (e.g., Conjugate gradients, GMRES) build a subspace of increasing size at each iteration, in which the next iterate is found. It must be noted that iterative methods can be combined with direct methods, either to build a preconditioner (e.g. incomplete LU factorizations), or to design hybrid methods (like block Cimmino [23] or domain decomposition methods where a direct solver is used within each domain [55]).

Therefore any progress in sparse direct methods will also impact those hybrid direct-iterative methods that, depending on the problem properties and size, can be well adapted (e.g. relatively well conditioned three-dimensional problems of very large sizes).

1.3 Sparse direct methods

When dealing with sparse matrices, Algorithm 1.1 performs many operations on zeros, which can be avoided to limit the amount of work compared to a dense factorization.

Although they differ in some aspects, direct methods, when applied to sparse matrices (sparse direct methods), usually rely on the same three main steps to find the solution of a linear system of equations. The first step, the **analysis phase**, consists in analysing the large sparse matrix A and in defining a graph representing the dependencies between computations on smaller dense blocks. The purpose is: (i) to estimate and limit the amount of computations and resources required for the factorization; and (ii) to apply a pre-processing to the matrix in order to improve

its numerical characteristics. The second step, the **factorization phase**, consists in decomposing the matrix into its factors. This is done by applying the corresponding computations to the graph that was produced in the first step. The third step, the **solve phase**, consists in applying the forward elimination and backward substitution to get the solution vector x of the linear system.

In the following sections, we will detail the first two steps, which are the most relevant for our work. For the sake of clarity, we illustrate our explanations by running on the example in Figure 1.1b.

1.3.1 Analysis phase: Sparse matrices and graphs

During the analysis phase, the original sparse matrix is analysed in order to prepare the factorization. Permutations are sought to limit the work during the factorization.

1.3.1.1 Adjacency graph

The structure of any sparse matrix A may be represented by its associated **adjacency graph** $\mathcal{G}(A)$.

Definition 1.1 (Adjacency graph)**.** *The adjacency graph of a matrix A is a graph $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$ such that:*

- *there are n vertices, each vertex $v_i \in \mathcal{V}$ represents the variable i (row or column) of A,*
- *there is an edge $< v_i, v_j > \in \mathcal{E}$ iff $a_{ij} \neq 0 \wedge i \neq j$.*

This formalism is the basis of all the further steps. When the matrices are symmetric, or just structurally symmetric, the associated graph can be considered to be undirected, while it must be directed in the case of matrices with an unsymmetric structure. In this section, we only consider the structurally symmetric case and thus undirected graphs.

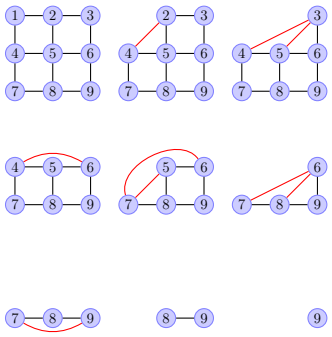
In our example, the adjacency graph of the matrix in Figure 1.1b simply corresponds to the mesh we presented in Figure 1.1a, with edges between i and j when a_{ij} is nonzero.

1.3.1.2 Symbolic factorization: filled graph

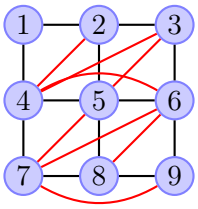
When applying an LU decomposition to a matrix A with a symmetric structure, by applying a succession of pivot eliminations, the sparsity pattern of A changes, together with $\mathcal{G}(A)$. Indeed, when eliminating a pivot in Algorithm 1.1, the right-looking update of the trailing matrix ($A(p+1:n, p+1:n)$ in the algorithm) changes some zeros from that submatrix into nonzeros, thus increasing the computational cost for the rest of the factorization. Indeed, eliminating a pivot p in A modifies entries a_{ij}, $i, j > p$, for which both a_{ip} and a_{pj} are nonzero. Performing this elimination on $\mathcal{G}(A)$ creates a **clique**, i.e. a fully connected graph, between all its direct neighbours $\{v \in \mathcal{V} | < v, v_p > \in \mathcal{E} \vee < v_p, v > \in \mathcal{E}\}$. Considering that we must then factorize the trailing matrix $A(p+1:n, p+1:n)$, we then remove v_p from $\mathcal{V}$ and all its adjacent edges from $\mathcal{G}(()A)$. This creation of edges in $\mathcal{G}(A)$ corresponds to the introduction of nonzeros in places previously occupied by zeros in A. This phenomenon is known as **fill-in**. After the LU decomposition is completed, we define the result of the successive transformations on the adjacency graph $\mathcal{G}(A)$ of A as the **filled graph** of A, i.e, the adjacency graph $\mathcal{G}(F)$ of $F = L+U$, the **filled matrix** of A.

Figure 1.2 illustrates these concepts on our example. The fill-in is represented in (dark) red in all subfigures. Figure 1.2a illustrates the successive graph transformations induced by each step of pivot eliminations on A. Figure 1.2b represents the resulting filled graph. Finally, Figure 1.2c

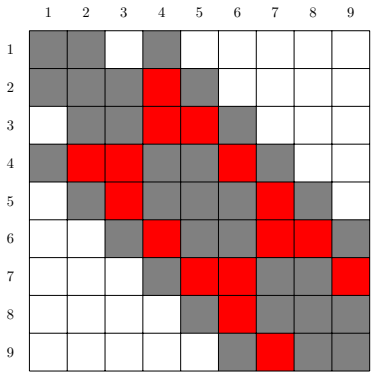
shows the sparsity pattern of the resulting L and U matrices given the original pattern of A. Taking the example of pivot 1, the update operation in Algorithm 1.1 is $A(2:9,2:9) = A(2:9,2:9) - A(2:9,1) \times A(1,2:9)$ which, considering that there are only a few zeros in column 1 and row 1, modifies entries $a_{2,2}$, $a_{2,4}$, $a_{4,2}$, $a_{4,4}$. Because $a_{2,4}$ and $a_{4,2}$ were originally zero, fill-in occurs, which corresponds to an edge between vertices 2 and 4 in the graph.



(a) Elimination of variables.



(b) Filled graph.



(c) Sparsity pattern of the LU factors. Original nonzero entries in gray, fill-in in red, zero entries in white.

Figure 1.2: Construction of the filled-graph (fill-in in red).

1.3.1.3 Reordering and permutations

Unfortunately, the increase in the number of nonzeros, due to the fill-in phenomenon, increases the amount of computations and resources. Fortunately, **reordering** (or **permuting**) the rows and columns of the matrix may greatly reduce the fill-in.

The mathematical meaning of permuting the rows and columns of a matrix A corresponds to multiplying it on the left and on the right by permutation matrices, i.e. square binary matrices that have exactly one entry 1 in each row and each column, and '0's elsewhere. This results in the modified matrix

$$A' = PAQ \ , \tag{1.11}$$

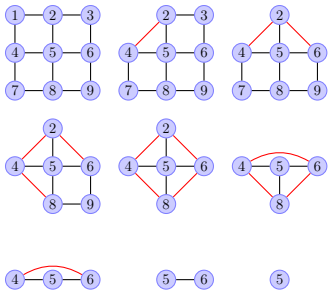
where P permutes the rows of the matrix, and Q permutes the columns. As we are considering matrices with a symmetric structure, we impose $Q = P^T$ in order to maintain symmetry, which amounts to renumbering the variables of the matrix. The problem of finding an optimum permutation matrix P (or an ordering of the variables) that minimizes the fill-in (or the amount of computations) is NP-complete [114]. There are, however, two main classes of ordering heuristics, which apply to the case of symmetric matrices.

- **Bottom-up** approaches rely on **local** criteria to decide, at each iteration, which variable to eliminate. Several algorithms exist that consist in eliminating, at each iteration, the variable in the graph which has the lowest degree of connectivity or which induces the minimum fill. Among these algorithms, the most common are:
  - The *Minimum Degree* algorithm (`MD`) with its variants
    - *Approximate Minimum Degree* [10] (`AMD`)

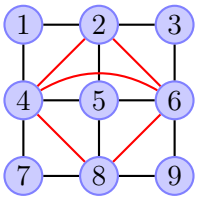
    - *Multiple Minimum Degree* [83] (`MMD`)
  - The *Minimum Fill* algorithm (`MF`) with its variants
    - *Approximate Minimum Fill* [4, 90] (`AMF`)
    - *Multiple Minimum Fill* [97] (`MMF`)
    - A recent `MF` implementation by [89]

- **Top-down** approaches are based on **global** considerations. They recursively partition the graph associated with the matrix, isolating the variables of the partitions at each level of recursion. *Nested-Dissection* techniques [53, 54] belong to this class of heuristics.

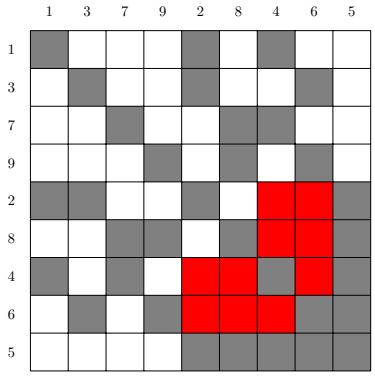
  In practice, partitioning libraries such as `PORD` [102], `METIS` [73] and `SCOTCH` [32, 93] implement hybrid schemes. Top-down approaches are first used to obtain partitions of a certain granularity, on which bottom-up approaches are then applied to finish the work.



(a) Elimination of variables.

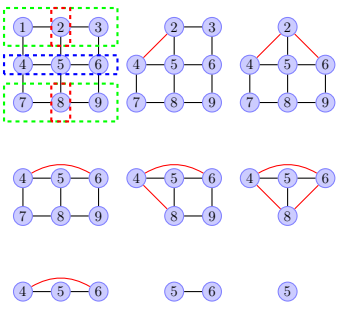


(b) Filled graph.

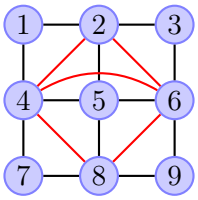


(c) Sparsity pattern of the *LU* factors.

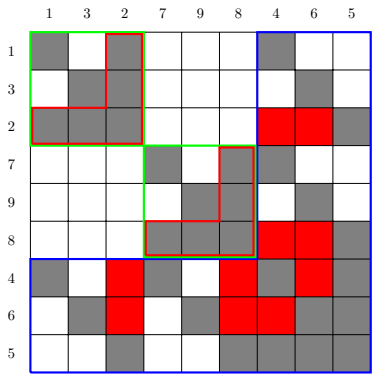
Figure 1.3: Minimum Degree ordering (fill-in in red).



(a) Elimination of variables.



(b) Filled graph.



(c) Sparsity pattern of the *LU* factors.

Figure 1.4: Nested-dissection ordering (fill-in in red).

In the case of our example, we illustrate the different steps and the effect of the application of the *Minimum Degree* and *Nested Dissection* algorithms in Figures 1.3 and 1.4, respectively.

At each step of the minimum degree algorithm, the vertex with smallest degree in the filled subgraph at the corresponding step is chosen, leading, in our example, to the elimination order $1-3-7-9-2-8-4-6-5$ (ties are broken arbitrarily). Concerning nested dissection, we can see in Figure 1.4 the first level of dissection with the blue set of variables (4,5,6) separating the two green domains consisting of variables (1,2,3) and (7,8,9). By ordering the separator last, large zero blocks appear in the matrix at the intersection of the two domains, in which no fill-in will appear. This idea is applied recursively, leading (for example) to the elimination order $1-3-2-7-9-8-4-6-5$.

1.3.1.4 Dependencies between variables and elimination tree

As we have seen, the elimination of a pivot during an LU decomposition generally impacts the remaining non-eliminated variables and the structure of the submatrix that is not yet factored. However, when the matrix is sparse, the pivot variable y may be **independent** from another variable z. y will then have *no* impact on z. This is then a source of parallelism: y and z could then be processed independently. More formally, we rely on the following definitions.

Definition 1.2 (Direct dependency)**.** *Given two variables y and z such that y is eliminated before z, we say that z directly depends on y ($y \rightarrow z$) iff $l_{zy} \neq 0$ (or $u_{yz} \neq 0$, since we consider matrices with a symmetric structure). Equivalently, $y \rightarrow z$ iff there is an edge between y and z in the filled graph of A, or if column y has a direct impact on column z.*

Definition 1.3 (Filled directed graph)**.** *We define the filled directed graph $\lceil\mathcal{G}(F)$ of a matrix A as the filled graph of A in which undirected edges between a pair of nodes y and z have been replaced by directed edges that respect the elimination order ($y \rightarrow z$ if y is eliminated before z).*

We show in Figure 1.5a the directed filled graph $\lceil\mathcal{G}(F)$ for our example, in the case of nested dissection. It is the same graph as the undirected filled graph $\mathcal{G}(F)$ of Figure 1.4b except that edges are now oriented in a way compatible with the ordering.

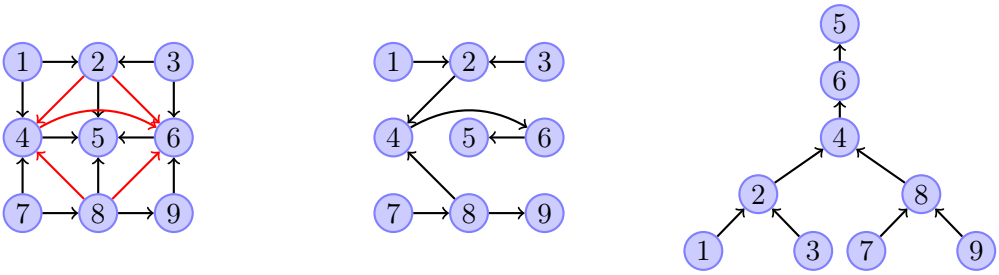
Definition 1.4 (Independent variables)**.** *Two variables y and z are independent iff there is no edge between y and z in the transitive closure of the filled directed graph $\lceil\mathcal{G}(F)$ of A. Said differently, there is no path between y and z in $\lceil\mathcal{G}(F)$.*

The independence between variables is a key characteristic upon which sparse direct methods rely. It may be exploited to express the potential **parallelism** of the computations.

Although the dependencies may be observed in $\lceil\mathcal{G}(F)$, it is difficult to visualize them there, as it contains many paths. We may then build the **transitive reduction** of the filled graph of A, which exhibits the dependencies in the most compact way. It is a minimum edge graph that has the same reachability relation (or transitive closure) as $\lceil\mathcal{G}(F)$.

As the dependencies always follow the order defined by the ordering, there are no cycles in $\lceil\mathcal{G}(F)$. It is thus a **directed acyclic graph** (**DAG**). In this case, its transitive reduction is unique. Furthermore, when the matrix A is structurally symmetric, the DAG is actually a tree, known as the **elimination tree**. We refer the reader to [87] for detailed information on elimination trees in a more general context. This elimination tree is a key structure on which sparse direct methods rely, as it denotes not only the order of the computations, but also reveals the possible parallelism to be exploited. Indeed, any computation occurring on separate branches of the tree could be executed in parallel. This kind of parallelism is called **tree parallelism**.

Figure 1.5b shows the transitive reduction of the filled-graph from Figure 1.5a (resulting from a nested-dissection ordering). Figure 1.5c then shows the elimination tree representation of the transitive reduction of our example.



(a) Directed filled graph. (b) Transitive reduction. (c) Elimination tree.

Figure 1.5: Transitive reduction and Elimination tree based on Nested-dissection ordering

1.3.1.5 Supernodes

In dense linear algebra, the operations related to pivot elimination and the update of the remaining variables are not performed on single pivots, but rather on sets of pivots, for performance issues. This allows for the use of very efficient `BLAS` routines to accomplish such operations. On sparse matrices, we could then use the same approach by grouping the variables with the same (or similar [19]) sparsity structures. The resulting sets of variables are called **supernodes**. In our example, we can group for instance variables 4, 5 and 6 into a single supervariable or supernode, leading to an amalgamated tree, sometimes called an **assembly tree**. An example of an assembly tree, where variables 4, 5, and 6 have been amalgamated to form a supernode, is given in Figure 1.6, which we will further discuss in the description of the multifrontal method. The LU factorization can then be generalized to work on supernodes instead of single variables. This allows the use of dense linear algebra kernels on matrices with higher dimension, leading to higher performance. One then talks of **supernodal methods**. One thing to notice is that the nodes and supernodes in Figure 1.5c are the separator variables in the nested-dissection ordering of Figure 1.4. At the first level of dissection, the blue set of variables dissecting the two green sets of variables represent the set of variables at the root of the tree. Then, each green subset is recursively dissected by the red set of variables, which in turn represents the child supernodes of the root of the tree.

Because large amounts of computations are done at each supernode, parallelism can in that case also be exploited within the nodes of the tree. The resulting type of parallelism is called **node parallelism**.

1.3.1.6 Symmetric and unsymmetric matrices

All the results considered so far concern (structurally) symmetric matrices. The same approach may be adapted to the case of unsymmetric matrices. The elimination tree described above will be in this case a DAG. Among the many ways of handling such matrices, one of them consists in working on the symmetrized structure of the matrix (i.e. $A^T + A$) and applying the above techniques on it. In the remainder of this thesis, even though we consider the unsymmetric LU decomposition, we will always consider the symmetric or symmetrized structure of the matrix $(A^T + A)$.

1.3.2 Factorization phase: left-looking, right-looking and supernodal approaches

We discuss in this section the factorization phase, and how the numerical factorization can be performed on sparse matrices. We delay to Section 1.3.3 the discussion of the multifrontal

method, a particular sparse direct method on which we will focus in this thesis.

We consider here two main types of operations occurring during LU decompositions, known as FACTO and UPDATE (using the notations of [40]):

- The FACTO(i) operation applied on column i divides by a_{ii} the under-diagonal part of i.
- The UPDATE(i,j) operation uses column A_i to update column A_j.

Considering that A is overwritten by the factors (L and U) so that eventually, $A = L + U - I$, we have more formally the following definitions:

- FACTO(A_j): $A_j(j+1:n) \leftarrow A_j(j+1:n)/a_{jj}$;
- UPDATE(A_i,A_j): $A_j(i+1:n) \leftarrow A_j(i+1:n) - a_{ij}.A_i(i+1:n)$.

Algorithm 1.1 can be formulated using these notations, leading to Algorithm1.2.

for $i = 1$ **to** n **do**
  FACTO(A_i);
  for $j = i$ **to** n **do**
    UPDATE(A_i,A_j);

Algorithm 1.2: Dense right-looking factorization algorithm.

There are n FACTO operations during the whole factorization, where n is the order of the matrix. In the case of sparse matrices, the elimination tree represents a **partial order** or **topological order** on these operations and some of the UPDATE(A_i,A_j) operations can be skipped when i and j are independent. Indeed, a node in the elimination tree must be processed after all its children. Formally, if j is an ancestor of i in the tree, then, FACTO(A_j) has to be performed after FACTO(A_i).

In spite of these dependency constraints, the structure of the elimination tree still provides some freedom to schedule the FACTO operations, and once this scheduling is fixed, there still remain some more flexibility to schedule the UPDATE operations. Among all the possible schedules, there are two main families of approaches: the so-called left-looking and right-looking methods. On the one hand, left-looking algorithms delay the UPDATE operations as late as possible: all the UPDATE($*$,A_j) are performed just before FACTO(A_j), looking to the **left** to nonzero entries in row j. On the other hand, right-looking algorithms perform the UPDATE operations as soon as possible: all the UPDATE(A_i,$*$) operations are performed immediately after FACTO(A_i), looking **right** to all columns that need to be updated.

for $j = 1$ **to** n **do**
  foreach i *such that* $i \rightarrow j$ *(see Definition 1.2)* **do**
    UPDATE(A_i,A_j);
  FACTO(A_j);

Algorithm 1.3: General left-looking factorization algorithm.

Algorithms 1.3 and 1.4 illustrate the left-looking and the right-looking factorizations, respectively.

The **supernodal method** refers to either the sparse left-looking or right-looking methods explained above, when generalized to supernodes instead of single variables. Supernodal methods are used in several well-known sparse direct solvers, for example the sequential version of `SuperLU`, which implements a left-looking approach [38].

for $i = 1$ **to** n **do**
  FACTO(A_i);
  foreach j *such that* $i \rightarrow j$ *(see Definition 1.2)* **do**
    UPDATE(A_i,A_j);

Algorithm 1.4: General right-looking factorization algorithm.

1.3.3 Multifrontal method

The **multifrontal method** [45, 46, 87, 101] has been introduced in the 80's and fully exploits the elimination tree [86] in the sense that the task dependency and the communication scheme of a multifrontal scheme is fully described by the node dependency of the elimination tree. It is quite different in this respect from the right-looking and left-looking approaches described in the previous section. It should be noticed that the "good" properties of memory locality of the multifrontal approach have been used over the years to efficiently address vector- and RISC-based shared memory computers as well as distributed memory computers.

In a nutshell, the order of the FACTO operations is thus defined by the elimination tree, as in the previously described methods. However, the UPDATE operations are not performed directly from the factorized columns to the updated ones anymore. Instead, the result of the update (a.k.a. contribution) of a variable i to a variable j (with $i \rightarrow j$) is carried throughout the path from i to j in the elimination tree (or assembly tree). The idea is to build a **contribution block** (or **Schur complement**) associated to each (super)node, as a condensed representation of all the contributions from the pivots in the current subtree to the ancestors depending on them in the elimination tree. This is possible because, at each step of the processing of a node, we build a dense matrix, so-called frontal matrix, capable of storing all contributions from the children in the elimination tree. In terms of data dependency, processing a node of the elimination tree will thus only depend on the processing of its children.

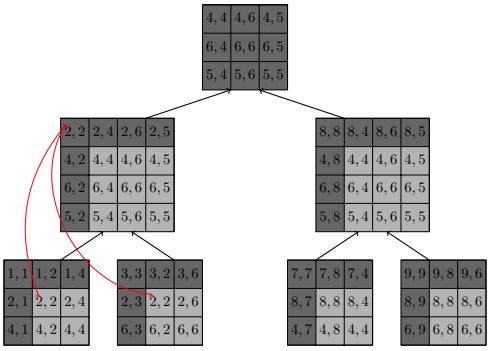


Figure 1.6: Assembly tree based on the Nested-dissection ordering. Each node of the tree is a front. The dark gray part of each front represents the fully-summed rows and columns of that front, and each light gray part represents the contribution block of that front. Curved arrows illustrate the assembly of an element in a parent front from the contribution blocks of its children fronts.

The data structure that we use at each node of the tree is thus a dense matrix [1] we call **frontal matrix**, or simply, **front**. It comprises two parts. First, we have the so-called **fully-summed** rows and columns, which correspond to the supernode variables that must be factorized. Once

[1] Usually square, in case of (structurally) symmetric matrices, and rectangular in case of unsymmetric matrices, or in other multifrontal methods, like in sparse QR decompositions.

factorized, this part is referred to as **factors**. Second, we have the contribution block of the supernode, which is the remaining square part (see Figure 1.6), and which corresponds to the Schur complement resulting from the factorization of the fully-summed rows and columns. We will denote by *nfront* the order (size) of a front, by *npiv* the number of pivots eliminated at this front, and by *ncb* the size of its contribution block. We thus have the relation $nfront = npiv+ncb$. Figure 1.6 illustrates the association of the frontal matrices with the nodes of the elimination tree on a symmetrized matrix. For unsymmetric multifrontal factorizations, we refer the reader to [17, 37, 48].

The multifrontal factorization then consists in traversing the tree while applying a series of tasks on each front. First, we allocate the front in memory and initialize all its entries to zero. Second, we **assemble** it by applying an **extend-add** operation (i.e. addition) of the entries from the contribution blocks of the children fronts together with the values of the elements of the current front in the original sparse matrix A. Third, we apply a **partial** LU decomposition to the front, i.e. only the fully-summed part is factorized, the contribution block only being updated with the factorized part. A full factorization is performed on the root of the tree as there is no contribution block there. Finally, in multifrontal implementations, we **stack** the front's contribution block, i.e. we compress it in memory to free up some of it for reuse in future computations. The process of accessing to contribution blocks using a stack mechanism depends on the fact that nodes of the tree are processed following a post-order (recursively traversing children subtrees from left-most to right-most before travesing the root). We remark that the amount of memory dedicated to contribution blocks depends on the order in which the tree is processed. In practice, a postorder defined by Liu [84], or a more general topological order [71, 85] is followed, aiming at reducing memory consumption. After stacking the contribution block, factor entries are generally also made contiguous in memory to save space, or can be written to disk in the case of so called **out-of-core** solvers [1, 103].

Early implementations of parallel multifrontal methods are discussed by Duff [43, 44]. We will further discuss in Chapters A and C some mechanisms and implementations in shared-memory and distributed-memory environments, respectively. This will be explained in the context of `MUMPS` [13, 15], the multifrontal solver that we use.

We end this section by making a remark on numerical pivoting, which has not been discussed yet. Numerical pivoting is essential to ensure a stable LU factorization. In sparse linear algebra, *threshold partial pivoting* is generally used, in which a pivot is accepted if it is larger than a threshold u multiplied by the largest entry in the column, with $0 \leq u \leq 1$ ($u = 1$ for partial pivoting). Finding acceptable pivots requires permutation of the rows/columns of the matrix, leading to dynamic tasks graphs. In the case of the multifrontal method, numerical stability may prevent a fully summed pivot from beeing eliminated. In that case, the pivot is delayed to the parent front where it will be eliminated if it is stable with respect to the threshold pivoting criterion. A pivot may be delayed several times (to an ancestor), until it becomes stable thanks to the assemblies from other siblings. In the worst case, the pivot may be delayed until the root, where it can always be chosen. In practice, the tree structure in the multifrontal method is maintained, but the size of the fronts increases, in an unpredictable way.

1.4 Parallelism on shared- and distributed-memory architectures

When desining sparse solvers, much attention has been paid to shared-memory and distributed-memory architectures.

Multithreaded sparse direct solvers aimed at addressing multicore environments have been the object of much work [7, 21, 28, 36, 39, 49, 61, 67, 70, 81, 100]. These solvers tend to use serial,

rather than multithreaded, BLAS (Basic Linear Algebra Subprograms [78]) libraries as building blocks, applied to dense matrices whose size is sometimes too small to make multithreaded BLAS efficient. Those solvers manage all the parallelism themselves, through the use of multithreading technologies like `Cilk` [24] and OpenMP [31]. We will focus on OpenMP, as it is a widely used standard.

Moreover, in the so called DAG-based approaches [28, 67] (and in the codes described much earlier in references [7], [39] and [44]), a much finer grain parallelism allows tree parallelism and node parallelism not to be separated: each individual task can be either a node in the tree or a subtask inside the node of the tree, which can start as soon as the dependencies of the task are satisfied. This is also the case of the distributed-memory approach we start from, where a processor can, for example, start working on a parent node even when some work remains to be done at the child level [14]. In this thesis, in order to use simple mechanisms to manage the threads, we study and push, as far as possible, the approach which consists in using tree parallelism up to a certain level and then switches to node parallelism, at the cost of some synchronization.

Additionally, many approaches rely on local / global task pools managed by a task dispatch engine [50, 28, 67]. In the case of NUMA architectures, the task dispatch engine aims at maintaining memory affinity, by placing tasks as close as possible to the cores that will work on them. For example, in HSL `MA87` [67], threads sharing the same cache also share a common local task pool. Similarly, work-stealing strategies also tend to be NUMA-aware (see, for example, [50]), by preferably stealing tasks from the pools associated to close cores. Whereas a local memory allocation is generally preferred, a round-robin memory allocation may also provide good performance in some cases [28].

Furthermore, some recent evolutions in linear algebra use runtime systems like ParSeq (formerly `DAGuE`)[25], `StarPU` [20] and `PFunc` [72], to cite a few. This type of approach has also been experimented in the context of sparse direct solvers on SMP (Symmetric Multi-Processor) machines [41, 44] and on modern architectures [3, 77], sometimes with the possibility to also address GPU accelerators. However, numerical pivoting leading to dynamic task graphs, specific numerical features, or application-specific approaches to scheduling, still make it hard to use generic runtime systems in all cases. All the observations and algorithmic contributions of this thesis are general and apply whether runtime systems are used or not.

In the case of distributed-memory architectures, supernodal methods can be parallelized by assigning each column (or block of columns, considering supernodes) to different processors. The two main classes of methods are called left-looking and right-looking (see Section 1.3.2). They generalize to the so-called *fan-in* and *fan-out* approaches [63]. In the fan-in approach (similar to left-looking), messages are received from descendants in the supernodal tree just before a Facto operation (all-to-one messages), whereas in the fan-out approach (similar to right-looking), messages are sent immediately after a Facto operation to ancestors in the tree holding columns on which updates need to be done (one-to-all messages). Some examples of parallel solvers for distributed-memory architectures based on supernodal methods include `SuperLU_DIST` [82, 99] and `PaStiX` [64]. The multifrontal method has also been successfully parallelized on distributed-memory architectures and has led to several parallel software packages, among which we can cite `PSPASES`, `WSMP` [59, 60], and `MUMPS`.

Although it might be interesting to consider higher level abstractions like PGAS languages and runtime systems, most solvers rely on MPI to implement their distributed-memory algorithms, for its efficiency through low-level programming close to machine architectures and for the full control it offers.

In this thesis, we use `MUMPS` as our experimental environment to validate our contributions.

Part I

Shared-memory environments

Chapter A

Introduction of part I

In this first part of the thesis, we consider the sparse multifrontal method in a pure shared-memory environment.

Our aim is to study the combined use of optimized multithreaded libraries (in particular BLAS) and of loop-based fine-grain parallelism, inside tasks, with the use of coarse-grain parallelism, between independent tasks. We propose in Chapter 2 an algorithm, based on a performance model, which chooses when to use coarse-grain parallelism (tree parallelism) and when to use fine-grain parallelism (node parallelism).

In Chapter 3, we show that, on NUMA architectures, the memory allocation policy and the resulting memory affinity strongly impact performance. We describe when to use each kind of policy depending on the mapping of the threads.

Furthermore, when treating independent tasks in a multithreaded environment, the parallel scheduling strategies may lead to situations where no ready task is available for a thread that has finished its share of the work. It will then stay idle, waiting for new tasks to become available. In an OpenMP environment, we describe how to re-use idle cores to dynamically increase the amount of parallelism. This approach, which can be viewed as an alternative to work-stealing, is described in Chapter 4.

As an introduction to this part of the thesis, we first describe the main kernels used in multifrontal factorizations, together with the test problems and the test machines used. At the end of the study (conclusion of Part I), we will further show the impact of this work on applications resulting from collaborations done during this thesis.

A.1 Shared-memory multifrontal kernels

As explained in Section 1.3.3, the multifrontal method consists of a sequence of operations on fronts (dense matrices) in a tree task-graph, that should be processed from bottom to top. Three main operations arising in the multifrontal method are applied to each of these fronts, namely: assembly, factorization and stacking, corresponding to three computational kernels of our multifrontal solver in shared-memory environments. We describe these in Algorithms A.1, A.2, and A.3.

The assembly operations consist in assembling data from the contribution blocks of the children into the parent front. They are done using Algorithm A.1. It first allocates and resets the memory zone relative to a parent front P to be assembled. Then, it merges the lists of variables from children of P and from the original matrix entries to be assembled, in order to build the necessary indirections defining where each child contribution row should be copied to in P, before

actually copying it. In case of numerical difficulties, the list of variables to be assembled from each child could be different from the initial list made at the construction of the elimination (or assembly) tree, as the variables that could not be eliminated in this child will have to be assembled into the parent, where they may be eliminated. Figure 1.6 shows an illustration of the shape of parent and children fronts during the assembly of the parent with the contribution blocks of the children. The rows of the children which are common to them are (usually) the fully-summed rows of the parent, while the other distinct rows remain in the contribution block of the parent.

1: **1. Build row and column structures of frontal matrix associated with node $\mathcal{N}$:**
2: Merge lists of variables from children and from original matrix entries to be assembled in $\mathcal{N}$
3: Build indirections (overwriting index list IND_c of child c with the relative positions in the parent)
4: **2. Numerical assembly:**
5: **for all** children c of node $\mathcal{N}$ **do**
6: **for all** contribution rows i of child c **do**
7: **for all** contribution columns j of child c **do**
8: Assemble entry (i,j) at position ($\text{IND}_c(i)$,$\text{IND}_c(j)$) of parent (*extend-add* operation)
9: **end for**
10: **end for**
11: Assemble entries from original matrix in fully-summed rows and columns of $\mathcal{N}$
12: **end for**

Algorithm A.1: Assembly of frontal matrix F_N of node $\mathcal{N}$.

Factorization operations consist in applying a dense LU decomposition on the fronts. They are done using Algorithm A.2. We consider the right-looking variant of the LU factorization, where the elimination of the fully-summed variables precedes the update of the contribution block so as to separate operations. After having eliminated a block of variables in a panel, they need to update the remaining trailing matrix before continuing the factorization. In our multifrontal context, though, as we only do partial factorization, we only update the fully-summed rows, while keeping the remaining rows unchanged. Once all the fully-summed rows are factored, we factorize the L part and update the contribution block of the remaining rows, using all the factorized variables. This approach has the advantage of using only one very large call to TRSM (line 12) and GEMM (line 13), which tends to increase the performance of the update. We define in Figure A.1 the main areas of a front involved during the factorization algorithm.

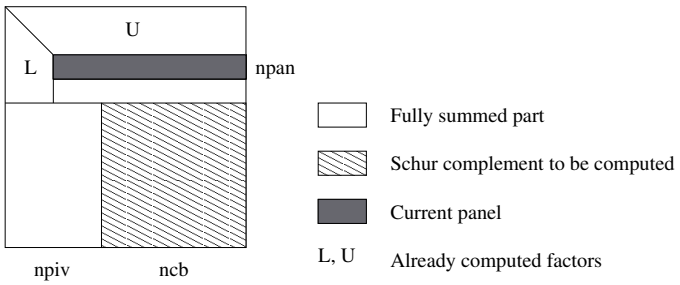


Figure A.1: Structure of a frontal matrix during its factorization (see Algorithm A.2).

1: **for all** horizontal panels $P = F(k : k + npan - 1, k : npiv + ncb)$ in fully-summed rows **do**
2: `BLAS 2` **factorization of the panel:**
3: **while** a stable pivot can be found in columns $k : npiv$ of P **do**
4: Perform the associated row and/or column exchanges
5: Scale pivot column in panel (_SCAL)
6: Update panel (_GER)
7: **end while**
8: Update fully-summed column block $F_N(k + npan : npiv, k : k + npan - 1)$ (_TRSM)
9: Right-looking update of remaining fully-summed part
$F_N(k + npan : npiv, k + npan : npiv + ncb)$ (_GEMM)
10: **end for**
11: *% All fully-summed rows have been factorized*
12: Update $F_N(npiv + 1 : npiv + ncb, 1 : npiv)$ (_TRSM)
13: Update Schur complement $F_N(npiv + 1 : npiv + ncb, npiv + 1 : npiv + ncb)$ (_GEMM)

Algorithm A.2: Right-looking factorization of a frontal matrix F_N of node $\mathcal{N}$ of order $npiv + ncb$ with $npiv$ variables to eliminate and a Schur complement of order ncb. $npan$ is the block size for panels. We assume that all pivots are eliminated.

The stacking step is described by Algorithm A.3. The aim of this step is, firstly, to move the factors of the current active front to the factors area, and secondly, to copy the contribution block of the current active front to the stacking area, in order to free space in the active area for the computation of the next fronts.

1: Reserve space in stack area
2: **for** $i = npiv + 1$ **to** $npiv + ncb$ **do**
3: Copy $F(i, npiv + 1 : npiv + ncb)$ to stack area
4: **end for**
5: Make L factors $F(npiv + 1 : npiv + ncb, 1 : npiv)$ contiguous in memory (write them to disk)

Algorithm A.3: Stacking operation for a contribution block of order ncb, part of a frontal matrix F_N of node N of order $npiv + ncb$. Frontal matrices are stored by rows.

A.2 Experimental environment

Test problems The set of matrices that we use in our experiments are from real applications. They are given in Table A.1. Although some of them are symmetric, we consider them as unsymmetric, as our study covers only unsymmetric factorizations. We use a nested dissection ordering (`METIS` [74], in our case) to reorder the matrices. Furthermore, we use only double precision arithmetic, either real or complex, depending on the type of the matrix.

The horizontal lines in the table define five areas. The first one (at the top) corresponds to matrices for which there are very large fronts (3D problems). The third one corresponds to matrices with many small fronts, sometimes even near the root (e.g., circuit simulation matrices). The second one corresponds to intermediate matrices, between these two extremes. Finally, the fourth and fifth zones correspond to 3D and 2D geophysics applications [91, 105], respectively.

[1] Tim Davis' and GridTLSE collections

Matrix	Symmetric	Arithmetic	N	NNZ	Application field
3Dspectralwave	Yes	real	680943	30290827	Materials
AUDI	Yes	real	943695	77651847	Structural
conv3D64 (*)	No	real	836550	12548250	Fluid
Serena (*)	Yes	real	1391349	64131971	Structural
sparsine	Yes	real	50000	1548988	Structural
ultrasound	No	real	531441	33076161	Magneto-Hydro-Dynamics
dielFilterV3real	Yes	real	1102824	89306020	Electromagnetism
Haltere	Yes	complex	1288825	10476775	Electromagnetism
ecl32	No	real	51993	380415	Semiconductor device
G3_circuit	Yes	real	1585478	7660826	Circuit simulation
QIMONDA07	No	real	8613291	66900289	Circuit simulation
GeoAzur_3D_32_32_32	No	complex	110592	2863288	Geo-Physics
GeoAzur_3D_48_48_48	No	complex	262144	6859000	Geo-Physics
GeoAzur_3D_64_64_64	No	complex	512000	13481272	Geo-Physics
GeoAzur_2D_512_512	No	complex	278784	2502724	Geo-Physics
GeoAzur_2D_1024_1024	No	complex	1081600	9721924	Geo-Physics
GeoAzur_2D_2048_2048	No	complex	4260096	38316100	Geo-Physics

Table A.1: Set of test problems. N is the order of the matrix and NNZ its number of nonzero entries. The matrices come from the University of Florida and the University of Toulouse[1] and from geophysics applications .For symmetric matrices, we work on the associated unsymmetric problem, although the value of NNZ reported only represents the number of nonzeros in the lower triangle. The most resource consuming matrices indicated by (*) will only be used on the largest system, `AMD4x6`.

Test machines In the first part of the thesis, we use three multicore computers:

- `Intel2x4`: a 2 × 4-Core Intel Xeon Processor E5520 2.27 GHz (Nehalem), with 16 Gigabytes of memory, from LIP-ENS Lyon;
- `AMD4x6`: a 4 × 6-Core AMD Opteron Processor 8431 2.40 GHz (Istanbul), with 72 Gigabytes of memory, from ENSEEIHT-IRIT in Toulouse;
- `vargas`: a 2 × 16-Core IBM Power6 Processor, Dual-core p575 IH, 4.7 GHz, with 128 Gigabytes of memory, from IDRIS, in Orsay.

On both `Intel2x4` and `AMD4x6`, we use the Intel compilers (icc and ifort) version 12.0.4 20110427, together with the Intel(R) Math Kernel Library (`MKL`) version 10.3 update 4. On `vargas`, we use the IBM compilers (xlc and xlf), together with the ESSL library as the BLAS library.

Chapter 2

Multithreaded node and tree parallelism

As shown in Chapter 1, there are typically two sources of parallelism in multifrontal methods. From a coarse-grain point of view, assembly trees are DAGs that define dependencies between the processing of frontal matrices. The structure of the tree thus offers a natural level of parallelism, which consists in factorizing different independent fronts at the same time: this is called *tree parallelism*. From a fine-grain point of view, the partial factorization of a frontal matrix at a given node of the assembly tree (see Algorithm A.2) can also be parallelized: this is called *node parallelism*.

In Section 2.1, we describe how to exploit better the node parallelism inside each front of the multifrontal tree, while in Section 2.2, we describe how to improve tree parallelism between different fronts of the tree. In this chapter, only the `Intel2x4` computer (see Section A.2) is used to illustrate the algorithmic issues.

2.1 Multithreaded node parallelism

We discuss two ways of exploiting node parallelism in shared memory: first, via implicit parallelization, through multithreaded libraries, as described in Section 2.1.1; second, via explicit parallelization, through OpenMP directives, as described in Section 2.1.2. The combination of explicit shared-memory parallelism with distributed-memory parallelism is discussed in Section 2.1.3, where we summarize the results and identify the limits of such an approach.

2.1.1 Use of multithreaded libraries

In multifrontal factorizations, most of the time is spent in dense linear algebra kernels, namely the BLAS library. A straightforward way to parallelize Algorithm A.2 consists in using existing optimized multithreaded BLAS libraries. Such libraries have been the object of much attention and tuning by the dense linear algebra community. Using them is transparent to the application as no change to the algorithms nor to the code is required.

As indicated in Algorithm A.2, most of the computations in the partial factorization of a front are spent in the TRSM and GEMM routines and in the updates of non fully-summed rows after the fully-summed rows are factorized (lines 12 and 13). Thus, these updates consist of a single call to TRSM and to GEMM, operating on very large matrices that can be efficiently parallelized using multithreaded BLAS libraries.

2.1.2 Directive-based loop parallelism

Loop-level parallelism can easily be done using OpenMP directives during assembly Algorithm A.1 and stack Algorithm A.3 (see preliminary work in [34]). Pivot search operations can also be multithreaded, with limited success though, due to small granularity. The main difficulties encountered consisted in choosing, for each loop, the minimum granularity above which it is worth parallelizing. We use the default OpenMP scheduling policy which, in our environment, consists in using one static chunk of maximum size for each thread.

One simple way of parallelizing assembly operations (Algorithm A.1) consists in using a parallel OpenMP loop at line 6 of algorithm A.1. All the rows of a given child front are then assembled in parallel in the corresponding parent front. We observed experimentally that such a parallelization is worth doing only when the size of the contribution block to be assembled from child to parent is large enough. Another more complicated way of parallelizing Algorithm A.1, leading to slightly larger granularity, would be to parallelize the assembly with respect to the parent rows. This would consist in processing block-wise the rows of the parent node. Each block would then assemble all contribution rows to be assembled in it from all the children. Such an approach is used in the distributed version of our solver. It has not been implemented in the case of multithreading, although it might be interesting for wide trees when the assembly costs may be large enough to compensate for the additional associated symbolic costs and the cost of irregular memory access.

Stack operations (Algorithm A.3), which are basically memory copy operations, can also be parallelized by using a parallel loop at line 2 of the algorithm, with a minimum granularity in order to avoid speed-downs on small fronts.

2.1.3 Experiments on a multicore architecture

Table 2.1 shows the effects of using threaded BLAS and OpenMP directives on factorization times on 8 cores of `Intel2x4`, compared to an MPI parallelization using version 4.10.0 of `MUMPS`. Different combinations of numbers of MPI processes and numbers of threads per MPI process are presented, with threaded BLAS and OpenMP directives used within each MPI process in such a way as to always obtain a total number of 8 threads.

Before discussing these results, we briefly comment on the choice and settings regarding the BLAS library. Commonly used optimized BLAS libraries are `ATLAS` [113], `OpenBLAS` (formerly `GotoBLAS` [56]), `MKL` (from Intel), `ACML` (from AMD), and `ESSL` (from IBM). One difficulty with `ATLAS` is that the number of threads to be used has to be defined at compile-time. The conflicting interaction of `OpenBLAS` with OpenMP regions [34] limited the scope of its use in a general context. Since `Intel2x4` has Intel processors, we use `MKL` rather than `ACML`. With the version of `MKL` we use (version 10.3, update 4), the `MKL_DYNAMIC` setting (similar to `OMP_DYNAMIC`) is activated by default. When providing too many threads on small matrices, extra threads are not used, which does not result in speed-downs. This was not the case with some former versions of `MKL`, where it was necessary to set manually the number of threads to 1 (using the OpenMP routine *omp_set_num_threads*) for fronts too small to benefit from threaded BLAS. Unless stated otherwise, the reported experiments use `MKL_DYNAMIC` set to its default ("True") value.

Coming back to Table 2.1, we first observe that, in general, OpenMP directives improve the amount of node parallelism, yet with limited gains (compare columns "Threaded BLAS" and "Threaded BLAS + OpenMP directives" in the "1 MPI $\times$ 8 threads" configuration).

On the first set of matrices (3Dspectralwave, AUDI, sparsine, ultrasound80), the ratio of large fronts over small fronts in the associated elimination trees is high. Hence, the more threads per MPI process, the better the performance, node parallelism and the underlying multithreaded BLAS routines being able to reach their full potential on many fronts.

On the second set of matrices, the ratio of large fronts over small fronts is medium. The best computation times are generally reached when mixing tree parallelism at the MPI level with node parallelism at the BLAS level.

On the third set of matrices, where the ratio of large fronts over small fronts is very small (most fronts are small), using only one core per MPI process is often the best solution. Tree parallelism is critical whereas node parallelism brings no gain. This is because parallel BLAS is inefficient on small fronts where not enough work is available for all the threads.

On the GEOAZUR series of matrices, we also observe that tree parallelism is more critical on the 2D problems than on the 3D problems. On the 2D problems, the best results are obtained with more MPI processes and less threads per MPI process.

All these remarks show the benefits and limits of node parallelism in a shared-memory environment. With the increasing number of cores per machine, this approach is scalable only when most of the work is done in very large fronts (e.g., on very large 3D problems), tree parallelism being necessary otherwise.

Because message-passing in our solver was primarily designed to tackle parallelism between computer nodes rather than inside multicore processors and because of the availability of high performance multithreaded BLAS libraries, we believe that both node and tree parallelism should be exploited at the shared-memory level. This will be expanded in the following section.

	Sequential	Threaded BLAS only	Threaded BLAS + OpenMP directives			Pure MPI
Matrix	1 MPI × 1 thread	1 MPI × 8 threads	1 MPI × 8 threads	2 MPI × 4 threads	4 MPI × 2 threads	8 MPI × 1 thread
3DSPECTRALWAVE	2061.95	372.83	**371.87**	392.98	387.57	N/A
AUDI	1270.20	251.14	**249.21**	250.87	300.43	315.85
SPARSINE	314.58	62.52	**61.87**	82.01	80.22	94.42
ULTRASOUND80	441.84	89.05	**89.16**	95.67	124.07	124.10
DIELFILTERV3REAL	271.96	60.69	59.31	52.13	**47.85**	61.92
HALTERE	691.72	121.29	120.81	**115.18**	140.34	145.55
ECL32	3.00	1.13	1.05	**0.93**	0.98	0.94
G3_CIRCUIT	16.99	8.84	8.73	6.24	4.21	**3.61**
QIMONDA07	25.78	27.42	28.49	18.21	9.63	**5.54**
GEOAZUR_3D_32_32_32	75.74	16.09	**15.84**	16.28	18.68	19.62
GEOAZUR_3D_48_48_48	410.78	73.90	72.96	**69.71**	95.02	106.86
GEOAZUR_3D_64_64_64	1563.01	254.47	**254.38**	276.98	303.15	360.96
GEOAZUR_2D_512_512	4.48	2.30	2.33	1.46	**1.40**	1.56
GEOAZUR_2D_1024_1024	30.97	11.54	11.65	8.38	6.53	**6.21**
GEOAZUR_2D_2048_2048	227.08	64.41	64.27	49.97	**43.33**	43.44

Table 2.1: Factorization times (in seconds) on `Intel2x4`, with different core/process configurations. The best time obtained for each matrix appears in bold. N/A: insufficient memory for the factorization.

2.2 Introduction of multithreaded tree parallelism

The objective of this section is to introduce tree parallelism at the threads level, allowing different frontal matrices to be treated by different threads simultaneously.

Many algorithms exist to exploit tree parallelism in sparse direct methods. Among them, *proportional mapping* [94] and the *Geist-Ng algorithm* [52] have been widely used in both shared-memory and distributed-memory environments. We propose in Sections 2.2.1 and 2.2.2 two

variants of the Geist-Ng algorithm, aiming at determining a layer in the tree (we call it $\mathcal{L}_{th}$ for *"Layer Thread"*) under which tree parallelism is used at a thread level (i.e., one thread per subtree). We use simulations in Section 2.2.3 to evaluate the potential of this approach, before explaining in Section 2.2.4 how an existing multifrontal solver can be modified to take advantage of tree parallelism without too much redesign. We finally present in Section 2.2.5 some experimental results showing the gains obtained.

2.2.1 Balancing work among threads (AlgFlops algorithm)

We first present and comment the Geist-Ng algorithm (See Algorithm 2.1) on which our AlgFlops algorithm is essentially based.

The main idea of the Geist-Ng algorithm is as follows: *"Given an arbitrary tree and P processors, to find the smallest set of branches in the tree such that this set can be partitioned into exactly P subsets, all of which require approximately the same amount of work ... "* [52].

The Geist-Ng algorithm analysis phase starts by defining an initial layer containing the root of the tree only. When the sparse matrix to be factorized is reducible, the corresponding elimination tree might be a forest of elimination trees.

$\mathcal{L}_{th} \leftarrow$ roots of the elimination tree
repeat
  Find the node $\mathcal{N}$ in $\mathcal{L}_{th}$, whose subtree has the highest estimated cost {**Subtree cost**}
  $\mathcal{L}_{th} \leftarrow \mathcal{L}_{th} \cup \{children\ of\ \mathcal{N}\} \setminus \{\mathcal{N}\}$
  Map $\mathcal{L}_{th}$ subtrees onto the processors {**Subtree mapping**}
  Estimate load balance: $\frac{load(least\text{-}loaded\ processor)}{load(most\text{-}loaded\ processor)}$
until load balance $>$ threshold {**Acceptance criterion**}

Algorithm 2.1: Geist-Ng analysis step: finding a satisfactory layer $\mathcal{L}_{th}$.

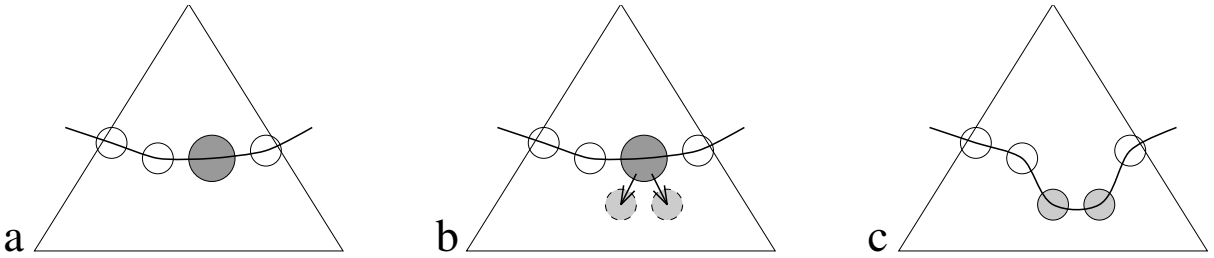


Figure 2.1: One step in the construction of the layer $\mathcal{L}_{th}$.

In such a case, the initial layer contains the roots of each tree in the forest and repeats three steps using Algorithm 2.1 until an acceptance criterion described later is reached (see Algorithm 2.1 and Figure 2.1).

Firstly, it identifies the node in the current layer that is the root of the largest subtree, and replaces it by its child nodes. It does so not only to find a good load balance but also because a node and one of its ancestors cannot both belong to $\mathcal{L}_{th}$, due to the dependency between a parent node and its children, as expressed by the elimination tree. Secondly, it tries to map the independent subtrees rooted at the current layer to the available processors. Thirdly, based on the previous mapping, it checks whether the current layer respects a certain acceptance criterion, based on load balance. In the following, we use the notation $\mathcal{L}_{th}$ to denote the final layer yielded by Algorithm 2.1.

In the Geist-Ng factorization phase, each thread selects the heaviest untreated subtree under $\mathcal{L}_{th}$ and factorizes it. Then, if no more subtrees remain, the thread starts working above the $\mathcal{L}_{th}$ by picking fronts one at a time in a round-robin manner, when they are ready to be factorized.

Our AlgFlops algorithm behaves like the Geist-Ng algorithm during the analysis phase. However, it differs during the factorization phase. Whereas the Geist-Ng algorithm only uses tree parallelism, our proposed AlgFlops algorithm uses only tree parallelism under $\mathcal{L}_{th}$ and only node parallelism above it, for the following reasons: (i) there are fewer nodes near the root of a tree, and more nodes near the leaves; (ii) fronts near the root often tend to be large, whereas fronts near the leaves tend to be small. This approach matches the pros and cons of each kind of parallelism observed in Section 2.1.3.

In the AlgFlops factorization phase, we treat all fronts under $\mathcal{L}_{th}$ before treating any front above it. When a thread is working under $\mathcal{L}_{th}$, if no more subtree remains, it goes idle and waits for the others to finish. Then, once all the subtrees under $\mathcal{L}_{th}$ are treated, all threads are used for the factorization of each front above $\mathcal{L}_{th}$, one at a time, following a postorder.

The following three points characterise AlgFlops:

- **Subtree cost:** The cost of a subtree is defined as the sum of the floating-point operations required to process the fronts constituting the subtree.
- **Subtree mapping:** Mapping subtrees over processors is known as the multiprocessor scheduling problem, or bin-packing problem, and is an NP-complete optimization problem, the solution of which could be approximated by the **LPT** (**L**ongest **P**rocessing **T**ime first) algorithm, whose solution has a maximal runtime ratio with respect to the optimal solution of $\frac{4}{3} - \frac{1}{3*nbproc}$ [42].
- **Acceptance criterion:** The acceptance criterion is a user-defined minimal tolerated imbalance ratio of the processors under $\mathcal{L}_{th}$.

However, the approach followed by AlgFlops has some limitations. Firstly, the load balance threshold for the threads under a given $\mathcal{L}_{th}$ may need to be tuned manually, and may depend on the test problem and target computer. We observed that a tight threshold (such as 90%) is adequate for most problems, at least when reordered with nested dissection-based techniques such as `METIS` [74] or `SCOTCH` [92]. Unfortunately, for some matrices, it is not always possible to reach so high a threshold. In such cases, unless another arbitrary stopping criterion is used, the algorithm will show poor performance, as it may not stop before the bottom of the tree. Secondly, we observed that when the 90% load balance criterion on flops is reached, the effective load balance on execution time is significantly worse. This is because the GFlops rate is in general much higher for large frontal matrices than for small ones. This limitation is amplified on unbalanced trees because the ratio of large vs. small frontal matrices may then be unbalanced over the subtrees. Thirdly, and more fundamentally, a good load balance under $\mathcal{L}_{th}$ may not necessarily lead to an optimal total runtime, which is the sum of the running times under and above $\mathcal{L}_{th}$.

2.2.2 Minimizing the global runtime (AlgTime algorithm)

2.2.2.1 Algorithm principle

To overcome the aforementioned limitations, we propose a modification of the AlgFlops algorithm, which we refer to as AlgTime and present as Algorithm 2.2. This new version modifies the computation of $\mathcal{L}_{th}$, while the factorization step remains unchanged, with tree parallelism only under $\mathcal{L}_{th}$ and node parallelism only above $\mathcal{L}_{th}$. This means that we use one thread in

kernels under $\mathcal{L}_{th}$ and all available threads in kernels above $\mathcal{L}_{th}$, with a strong synchronization to switch from one type of parallelism to the other.

$\mathcal{L}_{th} \leftarrow$ roots of the assembly tree/forest
$\mathcal{L}_{th}\_best \leftarrow \mathcal{L}_{th}$
$best\_total\_time \leftarrow \infty$
$new\_total\_time \leftarrow \infty$
$cpt \leftarrow extra\_iterations$
repeat
  Find the node $\mathcal{N}$ in $\mathcal{L}_{th}$, whose subtree has the highest estimated serial time
  $\mathcal{L}_{th} \leftarrow \mathcal{L}_{th} \cup \{children\ of\ \mathcal{N}\} \setminus \{\mathcal{N}\}$
  Map $\mathcal{L}_{th}$ subtrees onto the processors
  Simulate $time\_under\_\mathcal{L}_{th}$
  $time\_above\_\mathcal{L}_{th} \leftarrow time\_above\_\mathcal{L}_{th} + cost(\mathcal{N}, nb_{threads})$
  $new\_total\_time \leftarrow time\_under\_\mathcal{L}_{th} + time\_above\_\mathcal{L}_{th}$
  if $new\_total\_time < best\_total\_time$ **then**
    $\mathcal{L}_{th}\_best \leftarrow \mathcal{L}_{th}$
    $best\_total\_time \leftarrow new\_total\_time$
    $cpt \leftarrow extra\_iterations$
  else
    $cpt \leftarrow cpt - 1$
  end if
until $cpt = 0$ or $\mathcal{L}_{th}$ is empty
$\mathcal{L}_{th} \leftarrow \mathcal{L}_{th}\_best$

Algorithm 2.2: AlgTime algorithm.

Instead of considering the number of floating-point operations, AlgTime focuses on the runtime. Its goal is not to achieve a good load balance but rather to minimize the total factorization time. It relies on a performance model of the single-threaded (under $\mathcal{L}_{th}$) and multithreaded (above $\mathcal{L}_{th}$) processing times. These are estimated on dense frontal matrices (see Section 2.2.2.2): serial and multithreaded benchmarks of the GFlop/s rates of Algorithm A.2 are performed with different sample values of *npiv* (number of fully summed variables) and *ncb* (size of the Schur complement), and a bilinear interpolation is used to estimate the cost associated with any given *npiv* and *ncb*. Because of this model, it becomes possible to get an estimate of the time associated with a node both under and above $\mathcal{L}_{th}$, where single-threaded and multithreaded dense kernels are applied, respectively. We note that an approach based on performance models has already been used in the context of the sparse supernodal solver `PaStiX` [64], where `BLAS` routines are modelled using a polynomial interpolation.

The AlgTime algorithm computes $\mathcal{L}_{th}$ using the same main loop as the Geist-Ng algorithm. At each step of the loop, though, it keeps track of the total factorization time induced by the current $\mathcal{L}_{th}$ layer as the sum of the estimated time that will be spent under and above it. As long as the estimated total time decreases, we consider that the acceptance criterion has not been reached yet and replace the node of $\mathcal{L}_{th}$ with the largest estimated running time by its children in the corresponding subtree.

In order to further improve the solution, the algorithm proceeds as follows. Once a (possibly local) minimum is found, the current $\mathcal{L}_{th}$ layer is temporarily saved as the best obtained so far but the algorithm continues for a few extra iterations. The algorithm stops if no further decrease in time is observed within the authorized extra iterations. Otherwise, the algorithm continues after resetting the counter of additional iterations each time a layer better than all previous ones

is reached. We observed that a value of 50 extra iterations is normally enough to reach a stable solution on all problems tested, without inducing any significant extra cost. By its design, this algorithm is meant to be robust for any tree shape.

2.2.2.2 Performance model

Let α be the GFlop/s rate of the dense factorizations described in Algorithm A.2, which is responsible for the largest part of the execution time of our solver. α depends on the number *npiv* of eliminated variables and on the size *ncb* of the computed Schur complement. We may think of modelling the performance of dense factorization kernels by representing α in the form of a simple analytic formula parametrized experimentally. However, due to the great unpredictability of both hardware and software, it is difficult to find an accurate enough formula. For this reason, we have run some benchmark campaigns on dense factorization kernels on a large sample of well-chosen dense matrices for different numbers of cores. Then, using interpolation, we have obtained an empirical grid model of performance.

Given a two-dimensional field associated with *npiv* and *ncb*, we define a grid whose intersections represent the samples of the dense factorization kernel's performance benchmark. This grid should not be uniform since α tends to vary greatly for small values of *npiv* and *ncb*, and tends to have a more constant behaviour for large values. This is directly linked to the BLAS effects. Consequently, many samples must be chosen on the region with small *npiv* and *ncb*, whereas fewer and fewer samples are needed for large values of these variables. An exponential grid might be appropriate. However, not enough samples would be kept for large values of *npiv* and *ncb*. That is why we have adopted the following linear-exponential grid, using linear samples on some regions, whose step grows exponentially between the regions:

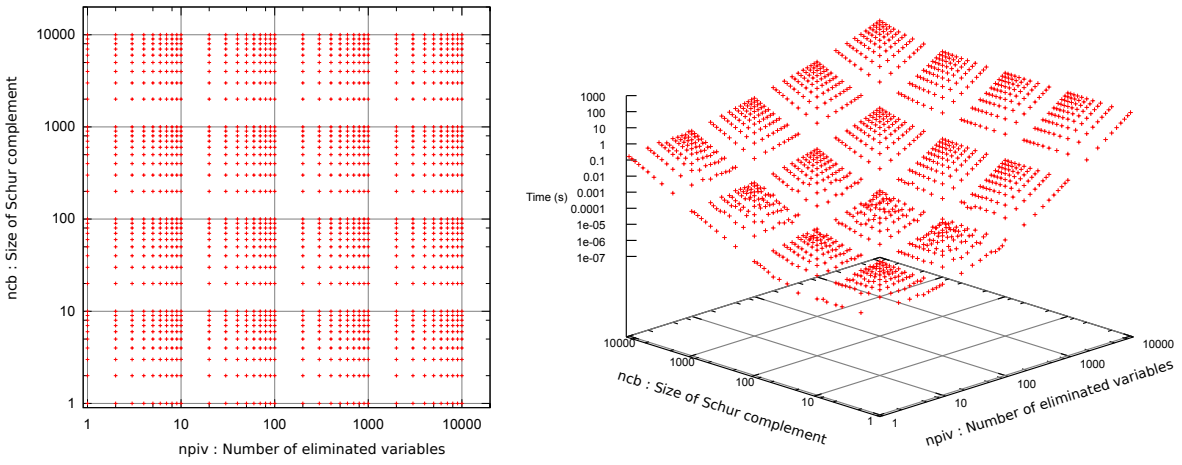
$$\begin{cases} \text{npiv or ncb} \in [1, 10] & \text{step} = 1 \\ \text{npiv or ncb} \in [10, 100] & \text{step} = 10 \\ \text{npiv or ncb} \in [100, 1000] & \text{step} = 100 \\ \text{npiv or ncb} \in [1000, 10000] & \text{step} = 1000 \end{cases}$$

Figure 2.2a shows this grid in log-scale and Figure 2.2b shows the benchmark on one core on `Intel2x4`. In order to give an idea of the performance of the dense factorization kernels based on Algorithm A.2, the GFlop/s rate of the partial factorization of a 4000×4000 matrix, with 1000 eliminated pivots, is $9.42 GFlops/s$ on one core, and is $56.00 GFlops/s$ on eight cores (a speed-up of 5.95). We note that working on the optimization of dense kernels is outside the scope of this thesis.

Once the benchmark is completed at each grid point, we can later estimate the performance for any arbitrary desired point (*npiv*,*ncb*) using the following simple bilinear interpolation:

$$\begin{aligned} \alpha(npiv, ncb, NbCore) \approx\ & \alpha(npiv_1, ncb_1, NbCore)\frac{(npiv_2 - npiv_1)(ncb_2 - ncb_1)}{(npiv_2 - npiv)(ncb_2 - ncb)} \\ & + \alpha(npiv_2, ncb_1, NbCore)\frac{(npiv_2 - npiv_1)(ncb_2 - ncb_1)}{(npiv - npiv_1)(ncb_2 - ncb)} \\ & + \alpha(npiv_1, ncb_2, NbCore)\frac{(npiv_2 - npiv_1)(ncb_2 - ncb_1)}{(npiv_2 - npiv)(ncb - ncb_1)} \\ & + \alpha(npiv_2, ncb_2, NbCore)\frac{(npiv_2 - npiv_1)(ncb_2 - ncb_1)}{(npiv - npiv_1)(ncb - ncb_1)}. \end{aligned}$$

where $(npiv_1, ncb_1)$ and $(npiv_2, ncb_2)$ define the limits of the rectangle surrounding the desired value of (*npiv*,*ncb*). In cases where (*npiv*, *ncb*) is outside the limits of the benchmark grid, the corresponding performance is chosen by default to be that of the limit of the grid (asymptote).



(a) Grid

(b) Benchmark

Figure 2.2: Grid and benchmark on one core of `Intel2x4`.

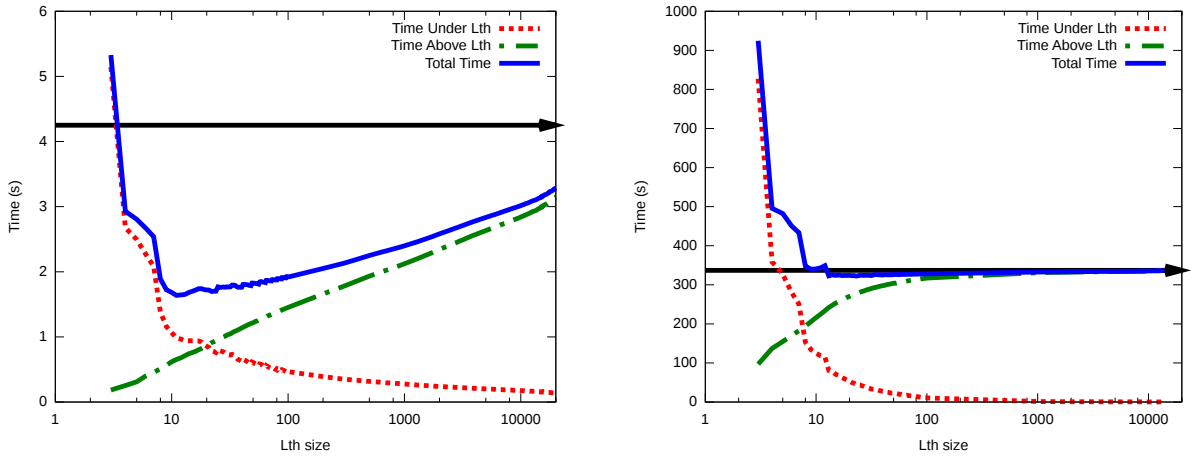
2.2.3 Simulation

Before an actual implementation, we perform some simulations in order to assess the effectiveness of our approach. A simulator relying on the performance model described above was written in the *Python* programming language. The objective of the simulation was to understand and illustrate the behaviour of Algorithm 2.2 if we suppose that we do not stop the iterations. Figure 2.3 shows the results obtained for two different matrices generated from a finite-difference discretization. The 2D matrix uses a 9-point stencil on a square and the 3D matrix uses an 11-point stencil on a cube. We have chosen these two matrices because they represent typical cases of regular problems, with very different characteristics. The assembly tree related to the 2D matrix contains many small nodes at its bottom while nodes at the top are comparatively not very large. On the other hand, the assembly tree related to the 3D matrix contains nodes with a rapidly increasing size from bottom to top. Simulations consist in estimating the time spent under and above $\mathcal{L}_{th}$, as well as the total factorization time, for all layers possibly reached by the algorithm (until the leaves).

The horizontal axis corresponds to the number of nodes contained in the successive $\mathcal{L}_{th}$ layers and the vertical axis to the estimated factorization time. The horizontal solid arrow represents the estimated time that would be spent by using fine-grain node parallelism only (as in Section 2.1) and will be referred to as *pure node parallelism*. As expected, the curve representing the time under (resp. above) $\mathcal{L}_{th}$ decreases (resp. increases) when the size of $\mathcal{L}_{th}$ increases. The solid curve giving the total time (sum of the dotted and solid-dotted curves) seems to have a unique global minimum (sometimes requiring a zoom on the curve). We have run several simulations on several matrices and this behaviour has been observed on all test cases.

The best $\mathcal{L}_{th}$ is obtained when the solid curve reaches its minimum. Hence, the difference between the horizontal arrow and the minimum of the solid curve represents the potential gain provided by the proposed AlgTime algorithm with respect to pure node parallelism. This gain heavily depends on the kind of matrix. Large 3D problems such as the one from Figure 2.3 show the smallest potential for $\mathcal{L}_{th}$-based algorithms to exploit tree parallelism: the smaller the fronts in the matrix, the larger the gain we can expect from tree parallelism. This also explains

the gap between the horizontal arrow and the solid curve at the right-most part of Figure 2.3(a) for the 2D problem, where $\mathcal{L}_{th}$ contains all leaves. This gap represents the gain of using tree parallelism only on the leaves of the tree.



(a) 2D (1000×1000 grid) (b) 3D ($100 \times 100 \times 100$ grid)

Figure 2.3: 2D vs. 3D: Simulated time (`Intel2x4`) as a function of the number of nodes in the $\mathcal{L}_{th}$ layer for two matrices of order 1 million for the AlgTime algorithm. The horizontal black arrows represent the estimated time with pure node parallelism (without tree parallelism).

2.2.4 Implementation of the factorization phase

1: **1. Process nodes under $\mathcal{L}_{th}$ (tree parallelism)**
2: **for all** subtrees $\mathcal{S}$, starting from the costliest ones, in parallel **do**
3: **for all** nodes $\mathcal{N} \in \mathcal{S}$, following a postorder **do**
4: Assemble, factorize and stack the frontal matrix of $\mathcal{N}$, using one thread (Algorithms A.1, A.2 and A.3)
5: **end for**
6: **end for**
7: Wait for the other threads
8: **2. Perform computations above $\mathcal{L}_{th}$ (node parallelism)**
9: **for all** nodes $\mathcal{N}$ above $\mathcal{L}_{th}$, following a postorder **do**
10: Assemble, factorize and stack the frontal matrix of $\mathcal{N}$, using all threads (Algorithms A.1, A.2 and A.3)
11: **end for**

Algorithm 2.3: Factorization phase using the $\mathcal{L}_{th}$-based algorithms.

The factorization algorithm (Algorithm 2.3) consists of two phases: first, $\mathcal{L}_{th}$ subtrees are processed using tree parallelism; then, the nodes above $\mathcal{L}_{th}$ are processed using node parallelism. At line 2 of the algorithm, each thread dynamically extracts the next most costly subtree. A static variant following the tentative mapping from the analysis phase is also implemented. It can be used to achieve bit-compatibility and determinism of the results when the number of cores remains constant; a study of bit-compatibility of multifrontal methods on a varying numbers of cores has been performed in [68]. One important aspect of our approach is that we are able to call the same computational kernels (assembly, factorization and stacking) and memory management routines from our existing solver. A possible risk is that, because those

kernels already use OpenMP, calling them inside an OpenMP parallel region could generate many more threads than the number of cores available. This can be avoided by suppressing nested parallelism (using *omp_set_nested*) or, if nested parallelism is enabled, by enabling OMP_DYNAMIC (using *omp_set_dynamic*), which will still avoid creating too many threads inside a nested region. Finally, if OMP_DYNAMIC needs to be disabled (as it will be the case in Chapter 4), we can explicitly set the number of threads to one inside the loop processing $\mathcal{L}_{th}$ subtrees (using *omp_set_num_threads*).

We now discuss memory management. In our environment, as is the case in many sparse solvers, one large array is allocated once and used as workspace for all frontal matrices, factors and the stack of contribution blocks. Keeping a single work array for all threads (and for the top of the tree) is not a straightforward approach because the existing memory management algorithms do not easily generalize to multiple threads. Firstly, the stack of contribution blocks is no longer a stack when working with multiple threads. Secondly, the threads under $\mathcal{L}_{th}$ would require synchronizations for the allocation of their private fronts or contribution blocks in the work array [11]; because of the large number of fronts in the assembly tree, these synchronizations might be costly. Thirdly, smart memory management schemes including in-place assemblies and compression of factors have been developed in order to minimize the memory consumption, that would not generalize if threads work in parallel on the same work array. In order to avoid these difficulties and to use the existing memory management routines without any modification, and possibly be more cache-friendly, we have created a private workspace for each thread under $\mathcal{L}_{th}$. We still use the same shared workspace above $\mathcal{L}_{th}$ (although smaller than before since only the top of the tree is now concerned). This approach raises a minor issue. Before factorizing a front, the contribution blocks of its children must be assembled into it (Algorithm A.1). A node immediately above $\mathcal{L}_{th}$ may have children that are handled by different threads. We thus need to keep track of which thread handles which subtree under $\mathcal{L}_{th}$, so that we can locate the contribution blocks in the proper thread-private workspaces. This modification of the assembly algorithm is the only modification that had to be done to the existing kernels implementing Algorithms A.1, A.2 and A.3. When all contribution blocks in a local workspace have been consumed, it could be worth decreasing the size of the workspace so that only the memory pages containing the factors remain in physical memory. This can be done without copy, depending on the platforms, using the *realloc* routine to reallocate the local workspace with a smaller size.

2.2.5 Experiments

We present in Table 2.2 the factorization times on Intel2x4. We observe (columns 5 and 6) that $\mathcal{L}_{th}$-based algorithms improve the factorization time for all matrices, compared to the sole use of node parallelism (see Section 2.1), whose results are reported in column 4. For each matrix, these results correspond to the black arrows of Figure 2.3. $\mathcal{L}_{th}$-based algorithms applied in a pure shared-memory environment also result in a better performance than when message-passing is used: column 3 represents the best combination of number of MPI processes and number of threads per MPI for the same total number of available cores, where both node and tree parallelism are used at the MPI level, but only node parallelism is used with threads. As could be expected, this best combination consists in using more MPI processes when tree parallelism is needed (matrices with many small nodes), and more threads when node parallelism is efficient (matrices with large nodes).

We also observe that the gains of the proposed algorithms are very significant on matrices whose assembly trees present the same characteristics as those of the 2D matrix presented above, namely: trees with many nodes at the bottom and few medium-sized nodes at the top. Such matrices arise, for example, from 2D finite-element and circuit-simulation problems. For the entire 2D GeoAzur set, the total factorization time has been divided by a factor close to two;

on those matrices, the relative gain offered by $\mathcal{L}_{th}$-based algorithms does not depend too much on the size of the matrices. In the case of matrices whose assembly trees present characteristics similar to those of the 3D case, the proposed $\mathcal{L}_{th}$-based algorithms still manage to offer a gain, although much smaller than in the 2D case. The reason for this difference is that, in 3D-like cases, most of the time is spent above $\mathcal{L}_{th}$ where the fronts are very large, whereas in 2D-like cases, a significant proportion of the work is spent on small frontal matrices under $\mathcal{L}_{th}$. The QIMONDA07 matrix (from circuit simulation) is an extreme case is the one of , where fronts are very small in all the regions of the assembly tree, with an overhead of node parallelism leading to speed-downs. Thus, AlgTime is extremely effective on such a matrix (4.26 seconds). On this matrix, the AlgFlops algorithm aiming at balancing the work under $\mathcal{L}_{th}$ was not able to find a good layer, leading to 27.3 seconds. However, using 8 MPI processes leads to a good exploitation of tree parallelism[1] (5.54 seconds), although not as good as AlgTime. More generally, we observe that, as expected, AlgTime is more efficient and more robust than AlgFlops, especially on irregular matrices leading to unbalanced trees, and that AlgTime leads to better results than MPI.

Matrix	Serial reference	Best MPI results [*]	Pure node parallelism	$\mathcal{L}_{th}$-based algorithms	
				AlgFlops	AlgTime
3Dspectralwave	2062.0	371.9 [1×8]	371.9	343.6	**339.8**
AUDI	1270.2	249.2 [1×8]	249.2	225.8	**210.1**
sparsine	314.6	61.9 [1×8]	61.9	59.5	**57.9**
ultrasound80	441.8	89.2 [1×8]	89.2	**77.1**	77.9
dielFilterV3real	272.0	47.9 [4×2]	59.3	46.1	**44.5**
Haltere	691.7	115.2 [2×4]	120.8	102.2	**99.5**
ecl32	3.00	0.93 [2×4]	1.05	3.07	**0.72**
G3_circuit	17.0	3.61 [8×1]	8.73	8.82	**3.02**
QIMONDA07	25.8	5.54 [8×1]	28.5	27.3	**4.26**
GeoAzur_3D_32_32_32	75.7	15.8 [1×8]	15.8	13.0	**12.9**
GeoAzur_3D_48_48_48	410.8	69.7 [2×4]	73.0	64.1	**62.5**
GeoAzur_3D_64_64_64	1563.0	254.4 [1×8]	254.4	228.7	**228.1**
GeoAzur_2D_512_512	4.48	1.40 [4×2]	2.33	0.88	**0.84**
GeoAzur_2D_1024_1024	31.0	6.21 [8×1]	11.7	5.37	**5.02**
GeoAzur_2D_2048_2048	227.1	43.3 [4×2]	64.8	35.5	**34.6**

Table 2.2: Experimental results with $\mathcal{L}_{th}$-based algorithms on `Intel2x4` (8 cores). [*]: [number of `MPI` processes × number of threads per process]. Bold numbers indicate, for each matrix, the best result obtained.

As predicted by the simulations of Section 2.2.3, the loss of time due to the synchronization of the threads on $\mathcal{L}_{th}$ before starting the factorization above $\mathcal{L}_{th}$ is compensated by the gain of applying single-threaded factorizations under $\mathcal{L}_{th}$. This is a key aspect. It shows that, in multicore environments, making threads work on separate tasks is better than making them collaborate on the same tasks, even at the price of a strong synchronization.

In the case of homogeneous subtrees under $\mathcal{L}_{th}$, the difference in execution time between the AlgFlops and the AlgTime algorithms is small. Still, AlgTime is more efficient and the gap grows with the problem size. We now analyse the behaviour of the AlgFlops vs AlgTime

[1]The MPI implementation is also based on the Geist-Ng algorithm but the criteria to obtain the layer under which tree parallelism is exploited has been tuned on many classes of matrices over the years, resulting in a better implementation than AlgFlops.

algorithms in more detail by making the following three observations.

- AlgTime produces a better load balance of the threads under $\mathcal{L}_{th}$ than AlgFlops does. For the AUDI matrix, the difference between the completion time of the first and last threads under $\mathcal{L}_{th}$, when using AlgFlops, is $90.22 - 72.50 = 17.72$ seconds, whereas the difference when using AlgTime is $94.77 - 82.06 = 12.71$ seconds. This difference is valuable since less time is wasted in synchronization.

- The $\mathcal{L}_{th}$ layer obtained with the AlgTime algorithm is higher in the elimination tree than that of the AlgFlops algorithm. For the AUDI matrix, the $\mathcal{L}_{th}$ of AlgFlops contains 24 subtrees, whereas that of AlgTime only contains 17 subtrees. This shows that AlgTime naturally detects that the time spent under $\mathcal{L}_{th}$ is more valuable than that spent above, as long as synchronization times remain reasonable.

- AlgFlops offers a gain in the majority of cases, but can yield catastrophic results, especially when elimination trees are unbalanced. One potential problem is that the threshold set in AlgFlops could not be reached, in which case, the algorithm will loop indefinitely until $\mathcal{L}_{th}$ reaches all the leaves of the tree. In order to avoid this behaviour, one method is to limit artificially the size of $\mathcal{L}_{th}$. The problem is that this new parameter is difficult to tune and that very unbalanced $\mathcal{L}_{th}$'s could lead to disastrous performance. The AlgTime algorithm is much more robust in such situations and brings significant gains. For the ecl32 matrix, for example, the factorization time of AlgFlops is 3.07 seconds whereas that of the algorithm shown in Section 2.1 was 1.05 seconds. In comparison, AlgTime decreases the factorization time to 0.72 seconds.

These results show that AlgTime is more efficient and more robust than AlgFlops and brings significant gains.

To conclude, we have presented an algorithm called AlgTime and shown that it makes efficient use of both tree and node parallelism in a multicore environment. Its characteristics make it robust on a wide class of matrices, and we have shown that it leads to better factorization times than the sole use of node parallelism and is better than a variant based on floating-point models instead of benchmarks. AlgTime is also more efficient than the MPI implementation from MUMPS 4.10.0 which currently mixes both tree and node parallelism, even though a strong synchronization is required by AlgTime when switching from tree parallelism to node parallelism. We will discuss in Chapter 4 simple strategy to further reduce the overhead due to such a synchronization. Before that, we study in Chapter 3 the case of NUMA architectures.

Chapter 3

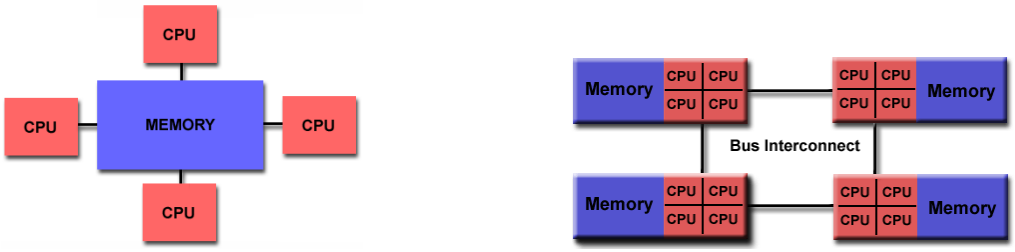
Impact of NUMA architectures on multithreaded parallelism

In Chapter 2, we proposed algorithms to exploit efficiently tree parallelism on multicore architectures. However, we did not take into account the memory irregularities of these computers. In the present chapter, we will extend the previous work to the case of NUMA architectures.

In Section 3.1, we will describe the effect of memory locality and affinity on the performance of multifrontal factorizations, while in Section 3.2, we will describe how to take into account resource sharing and racing in the performance model of dense factorizations. The proposed extensions will be shown to have a strong impact on performance.

3.1 Impact of memory locality and affinity

Two main architectural trends have arisen in the design of multicore computers: SMP and NUMA architectures (Figure 3.1).



(a) SMP (b) NUMA

Figure 3.1: Multicore architectures

On the one hand, **SMP** (**S**ymmetric **M**ulti**P**rocessor) architectures aim at making all cores access any memory address with the same cost. Their advantage is that any core can access the data mapped anywhere in memory without harming performance, but their drawback is that the number of interconnections between cores grows exponentially with the number of cores as they need to be fully connected, which makes this design efficient on small numbers of cores but hard to maintain on large numbers of cores.

On the other hand, **NUMA** (**N**on-**U**niform **M**emory **A**ccess) architectures make each core access memory addresses with different costs depending on their relative locations. Memory locality is thus an important issue on such architectures, since processors are partially connected and memory is organized hierarchically. Such a paradigm is more suited to large numbers of

cores.

In order to scale with the number of cores per chip, recent architectures try to merge SMP and NUMA paradigms. They contain sockets interconnected in a NUMA fashion, each socket being composed of local memory banks and a multicore processor with its cores arranged in an SMP fashion. All cores in a given socket preferably access their local memory (minimal cost) and access foreign memory banks with a penalty (worse cost), depending on the interconnection distance between the sockets.

Let us try to understand the effects of SMP and NUMA architectures on dense factorizations, in order to understand the modifications needed for sparse factorizations.

3.1.1 Impact of memory allocation policies on dense factorization performance

The more a machine has NUMA characteristics, the higher the differential cost of memory accesses. Thus, the way a dense matrix is mapped over memory banks and the mapping of the threads that will handle it over CPU cores may have important consequences on the performance of the factorization.

In order to highlight such effects, we have extracted in Table 3.1 the main results of an experiment consisting of factorizing with Algorithm A.2 a matrix of size 4000×4000 with $npiv = 1000$ variables to eliminate and a Schur complement of size $ncb = 3000$ with all possible combinations of cores used and memory allocation policies on `Intel2x4`. As `Intel2x4` contains two sockets, we use core 0 for sequential factorizations and cores 0 to 3 for multithreaded factorizations on socket 0. We also use core 1 for sequential executions since we want to compare core 0 and core 1. Similarly, core 4 is used for sequential factorizations and cores 4 to 7 for multithreaded factorizations on socket 1. We also use different allocation policies: `membind` is a policy forcing the system to allocate memory on a particular socket; `localalloc`, which is the default policy used on Linux, consists in allocating memory on the socket to which the core running the thread asking for memory is mapped on; `interleave` is a policy consisting in allocating memory in a round-robin fashion among different sockets.

	Core ID	`membind` `0`	`membind` `1`	`localalloc` (OS default)	`interleave` `0, 1`
socket 0	0	4.77	4.82	4.78	4.79
	1	4.74	4.78	4.73	4.75
	0...3	1.39	1.44	1.39	1.37
socket 1	4	4.75	4.71	4.71	4.72
	4...7	1.44	1.39	1.39	1.37
sockets 0,1	all	1.10	1.11	1.09	**0.79**

Table 3.1: Effect of the `localalloc` and `interleave` memory allocation policies with different core configurations on `Intel2x4`. The factorization times (seconds) are reported for a matrix of size 4000 with $npiv = 1000$ variables to eliminate. `membind` 0 (resp. 1) are memory allocations policies forcing data allocation on the memory bank of socket 0 (resp. 1). The bold number shows the best result.

Firstly, core 0 is less efficient than any other core, irrespective of the mapping of the matrix being factorized, because it plays a special role as it is the core in charge of handling I/O events and IRQ requests, having to process them by executing OS or drivers routines, which removes resources from the factorizations. Secondly, we note that the `localalloc` policy is the best

policy when dealing with serial factorizations. However, the `interleave` policy becomes the best when dealing with multithreaded factorizations, even when threads are mapped onto cores from the same socket. This effect is partially due to the fact that the experiment was performed on an unloaded machine: taking advantage of unused resources from the idle neighbour sockets is preferable to overwhelming local socket resources (memory bank and memory controller, memory bus, caches, ...). When running the experiment on all cores, the `interleave` policy is by far the best compared to local policies. We obtain a result of 0.79 *seconds*, which is to be compared with 1.10, 1.11, 1.09 *seconds* obtained with other policies. Further experiments that were made on the `AMD4x6` computer with various matrices confirm the advantage of using the `interleave` policy.

As suggested by this study and supported in Section 3.1.3, it seems that the best solution when working under $\mathcal{L}_{th}$ with concurrent single-threaded factorizations is to allocate thread-private workspaces locally near each core. On the other hand, when working above $\mathcal{L}_{th}$ with multithreaded factorizations, it will be preferable to allocate the shared workspace using the `interleave` policy.

3.1.2 Adaptation of $\mathcal{L}_{th}$-based algorithms for NUMA architectures

Controlling the memory allocation policy used in an application can be done in several ways. A first non-intrusive one consists in using the NUMACTL utility. However, this utility sets the policy for all the allocations in the program, which does not meet our requirements. A second, more intrusive, approach consists in using the LIBNUMA or HWLOC [27] libraries to dynamically change the allocation policy within the program. This is necessary in order to apply the `interleave` policy only on the workspace shared by all threads above $\mathcal{L}_{th}$, while keeping the default `localalloc` policy for the other allocations, in particular for the private data local to each thread under $\mathcal{L}_{th}$ (See Figure 3.2). We explain in detail in Appendix A.1 how to apply the `interleave` policy effectively on the shared workspace.

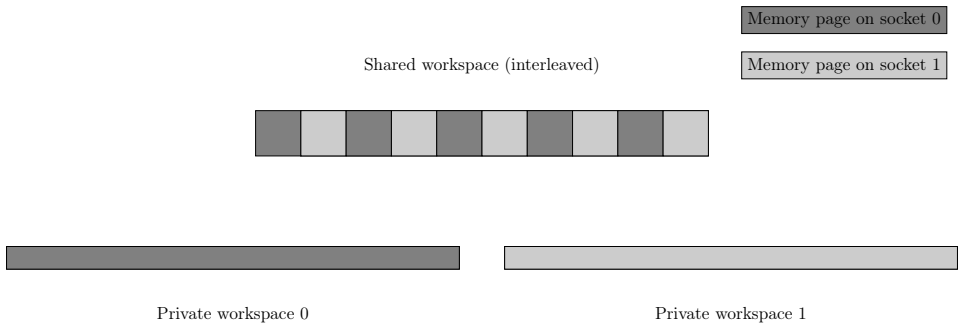


Figure 3.2: Illustration of the adaptation of the ALGTIME factorization phase for NUMA architectures on an example with two threads under $\mathcal{L}_{th}$, each mapped on a core of different sockets, and both working together above $\mathcal{L}_{th}$.

Figure 3.2 is an illustration of the application of the different memory policies in our context. We consider the example of the `Intel2x4` machine, with two sockets, each with a processor linked to a specific memory bank. We consider two threads under $\mathcal{L}_{th}$, each mapped onto a separate processor, and both working collaboratively above $\mathcal{L}_{th}$. We have thus allocated each thread's private workspace under $\mathcal{L}_{th}$ on the memory bank related to its processor, and have allocated the shared workspace above $\mathcal{L}_{th}$ on both memory banks, in a page-wise interleaved fashion.

3.1.3 Effects of the interleave policy

The last two columns of Table 3.2 show the factorization times obtained on `Intel2x4` with and without the use of the `interleave` policy above $\mathcal{L}_{th}$ in the AlgTime algorithm. The gains are significant for all matrices, except those with too small frontal matrices anywhere in the multifrontal tree, since, as expected, interleaving does not help in such cases.

	AlgTime +	
	Interleave	
Matrix	off	on
3Dspectralwave	339.78	295.84 (7.0)
AUDI	210.10	187.57 (6.8)
sparsine	57.91	48.60 (6.5)
ultrasound80	77.85	67.29 (6.6)
dielFilterV3real	44.52	40.88 (6.7)
Haltere	99.51	98.29 (7.0)
ecl32	0.72	0.70 (4.3)
G3_circuit	3.02	3.03 (5.6)
QIMONDA07	4.26	4.35 (5.9)
GeoAzur_3D_32_32_32	12.87	12.64 (6.0)
GeoAzur_3D_48_48_48	62.48	61.34 (6.7)
GeoAzur_3D_64_64_64	228.12	224.56 (7.0)
GeoAzur_2D_512_512	0.84	0.85 (5.3)
GeoAzur_2D_1024_1024	5.02	4.97 (6.2)
GeoAzur_2D_2048_2048	34.56	34.01 (6.7)

Table 3.2: Factorization times (seconds) without/with the interleave policy on the factorization time with the AlgTime algorithm on `Intel2x4`, 8 cores. The numbers in parenthesis correspond to the speed-ups with respect to sequential executions. The reference column ("Interleave off") is identical to the last column of Table 2.2.

In Table 3.3, we analyse further the impact of the `interleave` memory allocation policy on the `AMD4x6` platform, which has more cores and shows much more NUMA effects than `Intel2x4`.

The first columns correspond to runs with node parallelism only, whereas the last two columns use AlgTime. Parallel `BLAS` ("Threaded `BLAS`" columns) in Algorithm A.2 may be coupled with an `OpenMP` parallelization of Algorithms A.1 and A.3 ("Threaded `BLAS` + `OpenMP`" columns). The first observation is that the addition of `OpenMP` directives on top of threaded `BLAS` brings significant gains compared to the sole use of threaded `BLAS`, and that the `interleave` policy does not help when only node parallelism is used. Then, we combined AlgTime with the `interleave` policy (above $\mathcal{L}_{th}$). By comparing the last two columns, we observe very impressive gains with the `interleave` policy.

The `interleave` policy, although beneficial on medium-to-large dense matrices, is harmful on small dense matrices (as will be shown in Table 3.4), possibly because when cores of different NUMA nodes collaborate, the cost of cache coherence is more important than the price of memory accesses. The $\mathcal{L}_{th}$ layer separating the small fronts (bottom) from the large fronts (top) provides the benefits from interleaving without its negative effects. This is why the `interleave` policy alone does not bring much gain, whereas it brings huge gains when combined with the AlgTime algorithm.

Table 3.4 further illustrates this aspect on the AUDI matrix, by showing the times spent under and above $\mathcal{L}_{th}$, with and without interleaving, with and without $\mathcal{L}_{th}$. Although an $\mathcal{L}_{th}$

layer is not required in the first two columns corresponding to node parallelism, we still measure the times below and above $\mathcal{L}_{th}$ based on the layer computed by AlgTime (last three columns). Without interleaving, we can see that $\mathcal{L}_{th}$-based algorithms improve the time spent under $\mathcal{L}_{th}$ (from 109.8 seconds to 36.0 seconds) because of tree parallelism; above, the time is identical (122.0 seconds) since only node parallelism is used in both cases. When using the `interleave` policy, we can see that the time above $\mathcal{L}_{th}$ decreases a lot (from 122.0 seconds to 74.0 seconds) but that this is not the case for the time under $\mathcal{L}_{th}$. We observe that using the `interleave` policy both with and without $\mathcal{L}_{th}$-based algorithms is disastrous for performance under the $\mathcal{L}_{th}$ layer (from 109.8 seconds to 151.5 seconds without $\mathcal{L}_{th}$-based algorithms and from 36.0 seconds to 100.9 seconds with $\mathcal{L}_{th}$-based algorithms). This confirms that the `interleave` policy should not be used on small frontal matrices, especially when they are processed serially.

Matrix	Serial reference	Node parallelism only: Threaded `BLAS`		Node parallelism only: Threaded `BLAS` + `OpenMP`		AlgTime algorithm	
		Interleave		Interleave		Interleave	
		off	on	off	on	off	on
3Dspectralwave	2365.2	375.4	362.3	323.0	342.9	288.3	**174.7**
AUDI	1535.8	269.8	260.8	231.8	225.5	158.0	**110.0**
conv3D64	3001.4	518.5	563.1	497.5	496.9	439.0	**303.6**
Serena	7845.4	1147.6	1058.0	1081.4	1006.7	893.6	**530.6**
sparsine	365.4	64.9	67.4	57.1	58.6	60.2	**35.9**
ultrasound80	516.1	104.5	100.8	93.9	90.6	74.5	**44.9**
dielFilterV3real	324.5	81.1	80.8	68.4	69.9	36.2	**25.3**
Haltere	867.5	142.9	145.0	133.6	135.2	80.9	**56.5**
ecl32	3.91	2.10	2.12	1.74	1.70	1.10	**0.88**
G3_circuit	24.9	16.2	16.1	14.9	14.7	3.39	**2.81**
QIMONDA07	31.8	54.2	51.8	52.6	55.5	**4.23**	4.30
GeoAzur_3D_32_32_32	88.4	17.7	17.5	15.8	15.7	10.7	**8.54**
GeoAzur_3D_48_48_48	479.8	75.7	74.1	70.3	66.7	52.3	**37.5**
GeoAzur_3D_64_64_64	1774.6	240.4	239.9	221.7	225.5	195.7	**119.8**
GeoAzur_2D_512_512	5.28	18.4	18.2	21.2	21.1	1.91	**1.88**
GeoAzur_2D_1024_1024	39.9	86.7	102.2	97.2	152.8	**15.7**	19.4
GeoAzur_2D_2048_2048	309.6	98.8	158.2	96.3	436.7	**44.7**	55.3

Table 3.3: Factorization times in seconds and effects of the `interleave` memory allocation policy with node parallelism and with AlgTime on the 24 cores of `AMD4x6`.

	Node parallelism only (Threaded `BLAS` + `OpenMP` directives)		$\mathcal{L}_{th}$-based algorithm (AlgTime)		
Time	without interleaving	with interleaving under and above $\mathcal{L}_{th}$	without interleaving	with interleaving under and above $\mathcal{L}_{th}$	with interleaving above $\mathcal{L}_{th}$
Under $\mathcal{L}_{th}$	109.8	151.5	36.0	100.9	36.0
Above $\mathcal{L}_{th}$	122.0	74.0	122.0	74.0	74.0
Total	231.8	225.5	158.0	174.9	110.0

Table 3.4: Interactions of $\mathcal{L}_{th}$ and memory interleaving on `AMD4x6`, for matrix AUDI. Times are in seconds. The serial reference is 1535.8 seconds.

On huge matrices, such as Serena (Table 3.3), the sole effect of the AlgTime algorithm is quite not significant (1081.4 seconds down to 893.6 seconds) but the effect of memory interleaving

without $\mathcal{L}_{th}$ is even smaller (1081.4 seconds down to 1006.7 seconds). Again, the combined use of AlgTime and memory interleaving brings a huge gain: 1081.4 seconds down to 530.6 seconds (increasing the speed-up from 7.3 to 14.7 on 24 cores). Hence, on large matrices, the main benefit of AlgTime is to make interleaving become very efficient by only applying it above $\mathcal{L}_{th}$. This also shows that in our implementation, it was critical to separate the work arrays for local threads under $\mathcal{L}_{th}$ and for the more global approach in the upper part of the tree, in order to be able to apply different memory policies under and above $\mathcal{L}_{th}$ (`localalloc` and `interleave`, respectively).

3.2 Resource sharing and racing

A careful study of the results obtained when applying AlgTime reveals a discrepancy between the simulated times under $\mathcal{L}_{th}$ based on the performance model and the effective experimental times observed on the solver. Simulated estimations are somewhat optimistic on multithreaded runs while slightly pessimistic on sequential runs. This seems to be exacerbated by the machines' architectural characteristics. In sequential (single-threaded) runs, the sequence of dense matrix factorizations coupled with assembly and stack operations leaves the caches in quite a good state, which makes factorizations more efficient than those used in benchmarks. On the other hand, with multithreaded runs, two reasons may explain this phenomena. Firstly, the estimation only comprises the factorization time, whereas the actual time also includes assemblies and stack operations. However, the amount of time spent in assemblies and stack is very small compared to the factorization time, and so is not enough to explain the discrepancy. Secondly, we ran the benchmarks on unloaded machines in order to obtain precise results. For instance, when we have run the single-threaded benchmarks, only one CPU core was working, all the others being idle. Whereas, in $\mathcal{L}_{th}$-based algorithms, all the cores are active under $\mathcal{L}_{th}$ at the same time. In such a case, resource sharing and racing could be the reason for the observed discrepancy under $\mathcal{L}_{th}$.

3.2.1 Impact of machine loading on dense factorization performance

We want in this section to analyse and illustrate the effect of machine load on parallel performance. Machine loading depends on many parameters out of our control, making it difficult to model precisely.

In order to quantify this effect we ran experiments on *concurrent single-threaded* factorizations. This consists in running independent threads mapped onto different cores. Each thread factorizes different instances of identical matrices. We could then vary the number and mapping of threads to control machine loading and measure the effects of such configurations on performance for different matrix sizes.

We show in Table 3.5 the results of an experiment comparing the performance of multithreaded and concurrent single-threaded runs, for different matrix characteristics, using all 8 cores of `Intel2x4`. We can see that, on the `Intel2x4` computer with 8 cores, concurrent single-threaded factorizations are more efficient than multithreaded ones. Moreover, concurrent single-threaded factorizations are efficient irrespective of the characteristics of the matrix; whereas multithreaded ones need matrices of large sizes to reach a good efficiency.

We also note that the performance for both types of factorization decreases as the number of eliminated variables increases (going from partial to total factorizations: increasing $npiv$ for a constant $nfront$). In particular, $npiv = nfront$ leads to the poorest performance for a given $nfront$. One explanation is that there is no large TRSM/GEMM at lines 12/13 of Algorithm A.2 and that all BLAS3 calls operate on matrices with limited size in one dimension (the panel size),

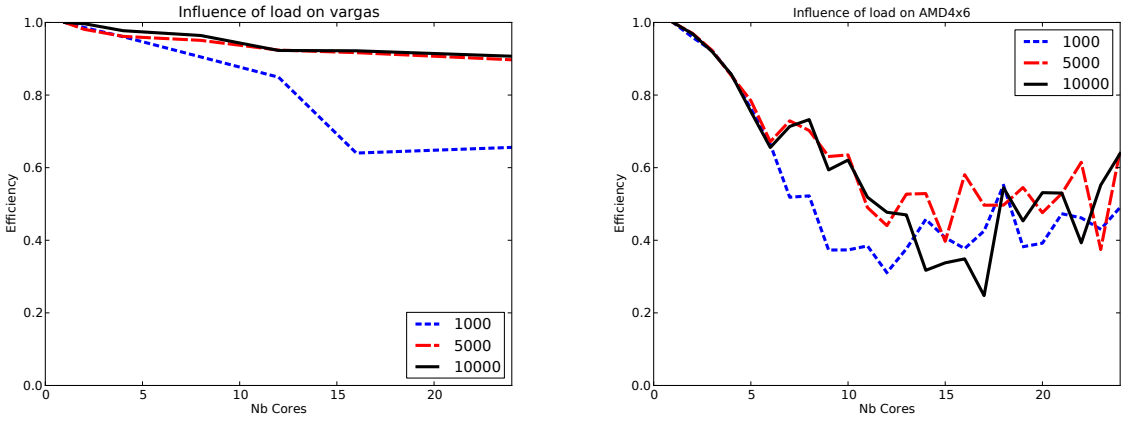
with higher load on the memory controllers. Thus, partial factorizations do not require access to all parts of the matrices simultaneously, as memory pages are able to be reused; whereas complete factorizations induce a memory access pattern that requires all the memory pages to be accessed simultaneously and frequently. One extreme case is that of the 15000×15000 matrix, where the performance of concurrent single-threaded factorizations decreases (75.0%) and becomes even lower than the efficiency of the corresponding multithreaded factorization (87.0%)! This contradicts common sense, but shows the limits of the concurrent single-threaded approach. It looks like, in this case, there are so many costly memory accesses that locality is better with multithreaded factorizations (when only one matrix is allocated) than with single-threaded factorizations (when a huge amount of memory is allocated). Note that, although the 15000×15000 matrix fills almost all the memory of `Intel2x4` when using a concurrent single-threaded factorization (1.8 GB $\times$ 8 threads = 14.4 GB out of 16 GB of total physical memory), we were able to check that no swap occurs.

		Sequential	Multithreaded		Concurrent single-threaded	
Matrix size ($nfront$)	eliminated variables ($npiv$)	time (seconds)	time (seconds)	efficiency (%)	time (seconds)	efficiency (%)
100	100	2.41×10^{-4}	1.72×10^{-4}	17.5	2.51×10^{-4}	96.0
1000	100	2.00×10^{-2}	4.28×10^{-3}	58.4	2.33×10^{-2}	85.8
	1000	7.94×10^{-2}	1.81×10^{-2}	55.0	9.80×10^{-2}	81.0
10000	100	2.10×10^{0}	2.96×10^{-1}	88.6	2.26×10^{0}	92.7
	1000	1.88×10^{1}	2.69×10^{0}	87.3	2.03×10^{1}	92.4
	10000	7.63×10^{1}	1.14×10^{1}	83.9	8.50×10^{1}	89.8
15000	100	4.72×10^{0}	6.52×10^{-1}	90.6	5.12×10^{0}	92.3
	1000	4.35×10^{1}	6.04×10^{0}	90.0	4.72×10^{1}	92.1
	10000	2.37×10^{2}	3.41×10^{1}	87.1	2.61×10^{2}	91.2
	15000	2.57×10^{2}	3.70×10^{1}	87.0	3.43×10^{2}	**75.0**

Table 3.5: Factorization times on `Intel2x4`, with varying matrix sizes and varying number of eliminated variables. 'Sequential' is the execution time for one task using one thread; 'Multithreaded' represents the time spent on one task using eight cores, and 'Concurrent single-threaded' represents the time spent on eight tasks using eight cores.

Overall, the degradation in performance in the majority of cases ranges between 5% and 20% under $\mathcal{L}_{th}$ when applying an $\mathcal{L}_{th}$-based algorithm, but can attain 33% in very unusual cases.

In Figure 3.3, we show the effects of the increase in the number of cores used in concurrent single-threaded runs for different sizes of matrices on machines with different characteristics. We choose the cores in increasing order of their ID, such that the first n cores have ID $0, .., n-1$. This shows the effect of load on cores of the same processor and also the effect of using cores of different processors. We can see how the architecture influences the performance, with an increase in machine load. `vargas` is a NUMA machine but could be considered as an SMP machine, since it has nearly uniform memory access. The degradation of performance is thus progressive with the number of cores. Also, even if the degradation is significant for small matrices (1000×1000), it becomes negligible for large ones (5000×5000 and 10000×10000). On `AMD4x6`, however, we observe a very rapid degradation of performance irrespective of the size of the matrices. When the working cores are located on the same processor (first 6 cores), the degradation is very smooth and predictable. However, once cores of different processors (sockets) are used, the trend of degradation becomes extremely chaotic, due to NUMA effects.



(a) Influence of load on `vargas` (SMP)

(b) Influence of load on `AMD4x6` (NUMA)

Figure 3.3: Influence of machine loading on a SMP machine (`vargas`, 32 cores) and on a NUMA machine (`AMD4x6`, 24 cores). The X-axis represents the number of cores involved in the "concurrent single-threaded" factorization, and the Y-axis represents the efficiency compared to an unloaded machine.

3.2.2 Adaptation of the performance model for NUMA architectures

As said before, the performance model is slightly optimistic under $\mathcal{L}_{th}$ because of the effect of load and, to a minor extent, because assembly and stack operations were not taken into account in the dense benchmarks we have based our model on. Moreover, the performance model is pessimistic above $\mathcal{L}_{th}$ because it does not consider the `interleave` policy. Table 3.6 shows the effects of modifying the performance models by taking into account the load under $\mathcal{L}_{th}$ and an "interleaved" benchmark above $\mathcal{L}_{th}$. The results are those for the AUDI matrix on the `AMD4x6` machine, for which the penalty ratio for the load was set experimentally to 1.4.

$\mathcal{L}_{th}$ layer defined with	Time under $\mathcal{L}_{th}$		Time above $\mathcal{L}_{th}$		Total Time
	Predicted	Observed	Predicted	Observed	Observed
no load + normal benchmark	28.08	36.93	137.56	73.37	110.30
load + interleaved benchmark	26.28	25.76	94.39	82.68	108.44

Table 3.6: Impact of better estimation of performance to define $\mathcal{L}_{th}$ layer (load under $\mathcal{L}_{th}$ and interleaved benchmark above) on `AMD4x6` on the AUDI matrix. AlgTime is used with `localalloc` and `interleave` policies under and above $\mathcal{L}_{th}$, respectively.

Because the performance model is used by AlgTime to define $\mathcal{L}_{th}$, the choice of $\mathcal{L}_{th}$ changes when taking into account machine load or the interleaved benchmark. The improved accuracy of predictions allows the AlgTime algorithm to make better choices of the $\mathcal{L}_{th}$ layers. For the AUDI matrix, the $\mathcal{L}_{th}$ layer contains 42 subtrees when taking into account load and interleaved benchmarks, and only 29 subtrees when ignoring them. This means that the $\mathcal{L}_{th}$ layer has been moved down the tree when using the "load+interleaved" benchmark. This is expected because single-threaded kernel performance (under $\mathcal{L}_{th}$) decreases with load and multithreaded kernels performance (above $\mathcal{L}_{th}$) increases with memory interleaving. These modified performance models result in a smaller observed time under $\mathcal{L}_{th}$ (25.76 seconds vs 36.93 seconds) and a larger observed time above (82.68 seconds vs 73.37 seconds). Finally, the total factorization time is decreased from 110.30 seconds to 108.44 seconds.

Concerning the accuracy of the model, we observe that the predicted times under and above $\mathcal{L}_{th}$ are more accurate when taking into account the machine load and the interleaved benchmark. For the AUDI matrix, the error of the prediction under $\mathcal{L}_{th}$ is 24% when ignoring load and 4.6% when taking it into account. Similarly, the error of the prediction above $\mathcal{L}_{th}$ is 47% with normal benchmark and 13% with the interleaved benchmark. It may look surprising that the interleaved benchmark still leads to pessimistic estimates. This may be due to the sequence of parallel assembly, factorization and stack operations that keep the caches and TLB's in a better state than in the benchmarking code.

Further improvements to the performance model require modelling cache effects, assembly and stack operations. Also, the performance on a matrix with given characteristics may vary depending on the machine state and numerical considerations such as pivoting. In the next chapter, we will present another approach, more dynamic, where we show that it is possible to be less dependent on the accuracy of the benchmark and on the precise choice of the $\mathcal{L}_{th}$ layer.

3.3 Summary

In this chapter, we have studied the multithreaded factorization algorithm AlgTime from Chapter 2, using both node and tree parallelism, on NUMA architectures. We have shown that the combined use of the `localalloc` and `interleave` memory allocation policies has a strong impact on performance, and that it is possible to take into account the load effects and the memory allocation policy in the benchmarks on which the algorithm is based. In both Chapters 2 and 3, we have observed that some cores become idle when switching from tree to node parallelism. In Chapter 4, we will show how to take advantage of these idle cores.

Chapter 4

Recycling Idle Cores

In the present chapter, we will push further the limits of our approach, by reducing as much as possible inactivity periods of CPU cores during the multithreaded multifrontal factorizations previously described in Chapters 2 and 3.

4.1 Motivation

When using $\mathcal{L}_{th}$-based algorithms, the $\mathcal{L}_{th}$ layer represents a barrier where cores must pay the price of synchronization by becoming idle after they finish their share of work under $\mathcal{L}_{th}$ and before starting work above $\mathcal{L}_{th}$. The greater the imbalance among threads under $\mathcal{L}_{th}$, the greater the synchronization cost. Figure 4.1 shows an execution GANTT chart of AlgTime for the `AUDI` matrix on `Intel2x4`. The X-axis represents machine cores; the Y-axis represents the time spent by each core in computations. Green/blue regions represent the activity of cores under/above $\mathcal{L}_{th}$, each core being mapped onto a separate/same thread(s), respectively. Black regions represent the time during which a core is idle. The sum of all black regions is wasted time that we will try to avoid in this chapter, by exploiting idle cores under $\mathcal{L}_{th}$ in order to reduce the total application execution time.

So far, we have used BLAS either with one core or with all cores, although an arbitrary number of cores could be used instead. Additionally, the efficiency of sequential BLAS is high but progressively decreases with the number of cores used. For example, on `AMD4x6`, the efficiency of Algorithm A.2 is about 76% on 6 cores but goes down to 51% on 24 cores. Consequently, $\mathcal{L}_{th}$-based algorithms make good use of sequential BLAS but not of parallel BLAS. This suggests that applying a smoother transition than that of our $\mathcal{L}_{th}$-based algorithms, between the bottom and the top of the tree, could make better use of BLAS.

Moreover, on numerically difficult problems, due to numerical pivoting during dense factorizations, performance could decrease on some fronts, making the work under $\mathcal{L}_{th}$ heavily unbalanced among threads. The dynamic scheduling of subtrees over threads is not enough in such cases to limit imbalance. Numerical difficulties could thus lead to even larger synchronization costs, increasing the idle period of certain cores.

In this chapter, we will use the term *thread* to identify application threads which we can explicitly control; whereas we will identify interchangeably by *core* both machine cores and threads implicitly managed by external libraries like multithreaded BLAS that create one thread for each machine core. Hence, when we say that a thread can use n cores, it simply means that this (application) thread may make a multithreaded BLAS and OpenMP region call that will use these n cores.

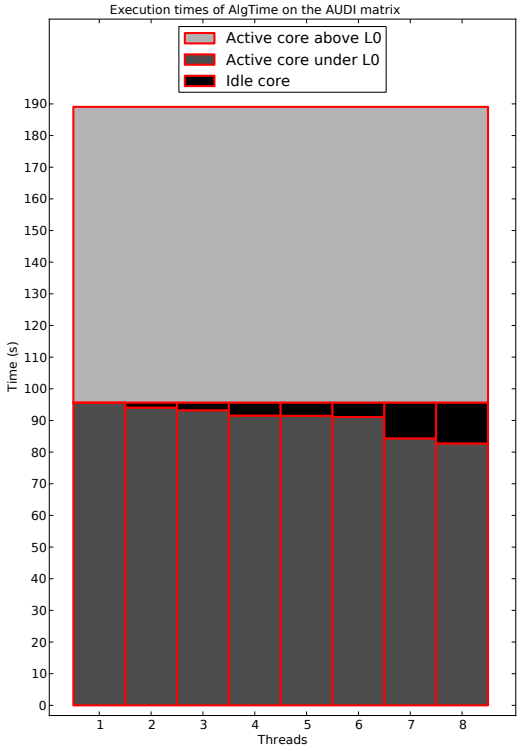


Figure 4.1: Wasted time under $\mathcal{L}_{th}$ on the `AUDI` matrix on 8 cores of the `Intel2x4` computer.

4.2 Core idea

In order to reduce CPU idle times, we could think of two strategies: either make idle cores start working above $\mathcal{L}_{th}$; or make them help the others under $\mathcal{L}_{th}$.

The first solution is attractive but presents some complications. Firstly, because most memory is consumed on the large nodes above $\mathcal{L}_{th}$, the traversal of the elimination tree above $\mathcal{L}_{th}$ is usually constrained to be a given postorder that most often optimizes memory consumption [84]. Starting the work above $\mathcal{L}_{th}$ too soon would constrain the order of processing of the nodes on the $\mathcal{L}_{th}$ layer: one must guarantee that each time a new node starts being processed above $\mathcal{L}_{th}$, all its child nodes on the $\mathcal{L}_{th}$ layer have already been processed. This would lead to a constrained scheduling of the subtrees under $\mathcal{L}_{th}$ that could lead to a loss of time which may not be compensated by the gains from reuse of idle cores.

Secondly, once a multi-threaed BLAS call is initiated, the number of threads cannot be modified. Since nodes above $\mathcal{L}_{th}$ are larger than those under $\mathcal{L}_{th}$, it is very likely that many threads working under $\mathcal{L}_{th}$ would complete their work before those working above in large BLAS calls (typically, at lines 12 or 13 of Algorithm A.2, where the non-fully summed rows are updated). Hence, new idle cores may not be used to help the thread(s) that started working on a large BLAS operation on a node above $\mathcal{L}_{th}$. They may then remain idle for longer than expected. Moreover, because the granularity is generally coarser the closer the node is to the root, if some threads start working on large tasks above $\mathcal{L}_{th}$, it will be increasingly costly to synchronize all threads higher in the tree.

Thirdly, we are here aiming at adapting a distributed-memory code to make use of multicore processors without a deep redesign. If we introduce tree parallelism above $\mathcal{L}_{th}$ in order to remedy the two previous problems, a whole new memory management strategy would have to be designed in order to be able to make many threads work in a single shared workspace.

The latter solution, which consists in reusing idle cores to help active threads under $\mathcal{L}_{th}$, is simpler and more natural. However, two main questions arise:

1. *who will decide on the attribution of idle cores to active threads?*
2. *how to dispatch idle cores to active threads?*

To answer the first question, we could think of two strategies: either the threads that finish their work under $\mathcal{L}_{th}$ decide how to dispatch their cores over the other working threads; or the newly available cores may be left in a pool of idle cores where each remaining active thread under $\mathcal{L}_{th}$ could choose to pick up some of them. The drawback of the latter approach is that, in order to access (read and modify) the total number of available cores, each thread must enter in a critical section, whose cumulated cost among all threads could be significant, and which could be exacerbated by the fact that in order to be reactive enough on the availability of new cores, active threads should check it quite often. The advantage of the first solution is that only one synchronization is done by each thread that terminates its work under $\mathcal{L}_{th}$. This is thus the solution we choose.

The second question is tougher. Many variants can be adopted for the assignement of idle cores to active threads. We can decide to attribute all idle cores in priority to the most loaded thread, i.e., the thread on the critical path of execution. This makes sense as, no matter how fast the other threads are, the completion time under $\mathcal{L}_{th}$ will depend on the critical one. However, it is hard to predict which thread is the most loaded in a dynamic scheduling environment, with numerical pivoting at runtime. When using node parallelism, the efficiency decreases with the number of processors. Thus, we could choose to achieve the fairest possible core dispatching, using a strategy consisting in assigning repeatedly each new idle core to the thread (or thread team) that has the minimum number of cores at its disposal. We describe the solution in Section 4.3.

A further optimized algorithm can be designed on NUMA architectures.When assigning idle cores to threads, we can, in case of equality of already assigned cores, assign preferentially idle cores to threads whose current cores are mostly mapped on the same processor (or NUMA node). Note that this requires knowledge of the thread-to-socket mapping; this may be done by initially binding threads to processors. For example, on `Intel2x4` which contains two (quad-core) processors, we map threads 0 to 3 on the first processor and threads 4 to 7 on the second.

4.3 Detailed algorithm

The Idle Core Recycling (ICR) algorithm consists of two mechanisms: the assignment of idle cores to active threads (Algorithm 4.1B) and the detection and use of new available cores by busy threads (Algorithm 4.1C). Algorithm 4.1B ensures a fair distribution of idle cores on active threads over time. It is executed in a critical section, exactly once by each thread finishing its share of work under $\mathcal{L}_{th}$ and discovering that the pool of unprocessed subtrees is empty. From time to time, when a thread starts the factorization of a new panel in a front, or between two BLAS calls, it applies Algorithm 4.1C, to check whether the number of cores at its disposal has changed, by comparing its entry in an array called *nb_ cores* (indexed by *my_ thread_ id*) with its current number of cores stored in the private variable *my_ nb_ cores*. If this is the case, it then updates its variable *my_ nb_ cores* and updates the number of cores to be used in future BLAS

calls and in OpenMP regions with a call to *omp_set_num_threads*. In our implementation, we do not use a mutex (or lock) because we assume atomic (exclusive) unitary read/write of the small integer (aligned 32-bit word) entries in the array *nb_cores*. This greatly simplifies synchronization issues, and allows threads to check for the availability of new cores as often as needed. If there is no atomic unitary read/write, we can replace the critical section from Algorithm 4.1B by a mutex or by using the *OMP_ATOMIC* primitive, then use that mutex in the algorithm detecting available cores. The mutex should be inside the *if* block, at line 3 in order to limit its cost: it is only used when *nb_cores*(*my_thread_id*) is being, or has been, modified.

Thus, with the hypothesis of atomic unitary read/write, the only possible (but unlikely and acceptable) disadvantageous situation could happen when Algorithm 4.1B (Mapping of idle cores to threads) is executed by a thread (one thread only since it is applied in a critical section) and Algorithm 4.1C (Detection of available cores) is simultaneously executed by another thread. In such a situation, the thread wishing to update its number of cores could read a new number of cores which is being increased simultaneously by the other thread, so that it could miss some idle cores for the computation of the current front, and use them only for the computation of the next BLAS call on the next OpenMP region.

One important thing to mention is that, when a thread gives its cores to other threads, it actually only notifies them of the availability of a certain number of additional cores. The newly available cores could be used in practice by any thread, which could be different from those expected. Indeed, the operating system and the OpenMP scheduler choose the effective mapping of cores to threads, even though some configurations using libraries like libnuma or hwloc [27] may be done in order, either to fix mappings, or to define preferable mappings, of cores to threads.

Figure 4.2 illustrates an execution of the ICR algorithm for the factorization of the AUDI matrix on `Intel2x4`. The X-axis represents the threads working under $\mathcal{L}_{th}$ while the Y-axis represents cores put at their disposal. Each sub-figure represents one step in the evolution of the ICR algorithm, where one thread, after having finished its work, gives its cores to others.

4.4 Implementation

The implementation of Algorithm 4.1 is straightforward. However, dynamically changing the number of resources inside a parallel region is not, mainly because of the interactions between OpenMP and BLAS libraries. In Algorithm 4.1C, the *omp_set_num_threads* function sets the number of cores for future OpenMP parallel regions of the calling threads. Nested parallelism should be enabled, and this can be done because of the *omp_set_nested* function. Moreover, the *omp_set_dynamic* function must be used in order to disable automatic dynamic control of the number of threads. When enabled, it typically prevents *omp_set_num_threads* from increasing the number of threads within a nested region beyond a value larger than the original number of threads available for the inner parallel regions. Even after using these functions, we still had problems within the `MKL` BLAS library, which still automatically limits the effective number of cores used. After setting the `MKL` dynamic behaviour to "False" by calling *mkl_set_dynamic* with the proper argument, we finally observed that BLAS kernels took advantage of the extra cores. This shows that the portability of such approaches is still an issue with current technologies.

In summary, we had to do the following: *omp_set_nested(TRUE), omp_set_dynamic(FALSE), mkl_set_dynamic(FALSE)*. Concerning BLAS, we were only able to implement the ICR mechanism with the MKL library.

A: Initialization:
shared:
    $nb\_cores \leftarrow (1, \ldots, 1)$ {Number of cores of each thread}
private:
    $my\_thread\_id \leftarrow$ Id of the current thread
    $nb\_threads \leftarrow$ Total number of threads
    $my\_nb\_cores \leftarrow 1$ {Number of cores currently used by the current thread}

B: Mapping of idle cores to threads:
Begin critical section
while $nb\_cores(my\_thread\_id) > 0$ **do**
  {Find thread with least number of cores}
  $id\_thread\_to\_help \leftarrow 0$
  for $i = 1$ **to** $nb\_threads$ **do**
    if $i \neq my\_thread\_id$ **and** $nb\_cores(i) \neq 0$ **and**
    $(id\_thread\_to\_help = 0$ **or** $nb\_cores(i) < nb\_cores(id\_thread\_to\_help))$ **then**
      {thread i is not me and has not finished yet and is the first we encountered or has less cores than the current thread we wish to help}
      $id\_thread\_to\_help \leftarrow i$
    end if
  end for
  if $id\_thread\_to\_help \neq 0$ **then**
    {Notify thread $id\_thread\_to\_help$ that it has more cores}
    $nb\_cores(id\_thread\_to\_help) \leftarrow nb\_cores(id\_thread\_to\_help) + 1$
    $nb\_cores(my\_thread\_id) \leftarrow nb\_cores(my\_thread\_id) - 1$
  else
    Break
  end if
end while
End critical section

C: Detection of available cores:
if $my\_nb\_cores \neq nb\_cores(my\_thread\_id)$ **then**
  $my\_nb\_cores = nb\_cores(my\_thread\_id)$
  $omp\_set\_num\_threads(my\_nb\_cores)$
  {Change the number of cores to be used by the current thread (my_thread_id)}
end if

Algorithm 4.1: $\mathcal{L}_{th}$ with idle core recycling.

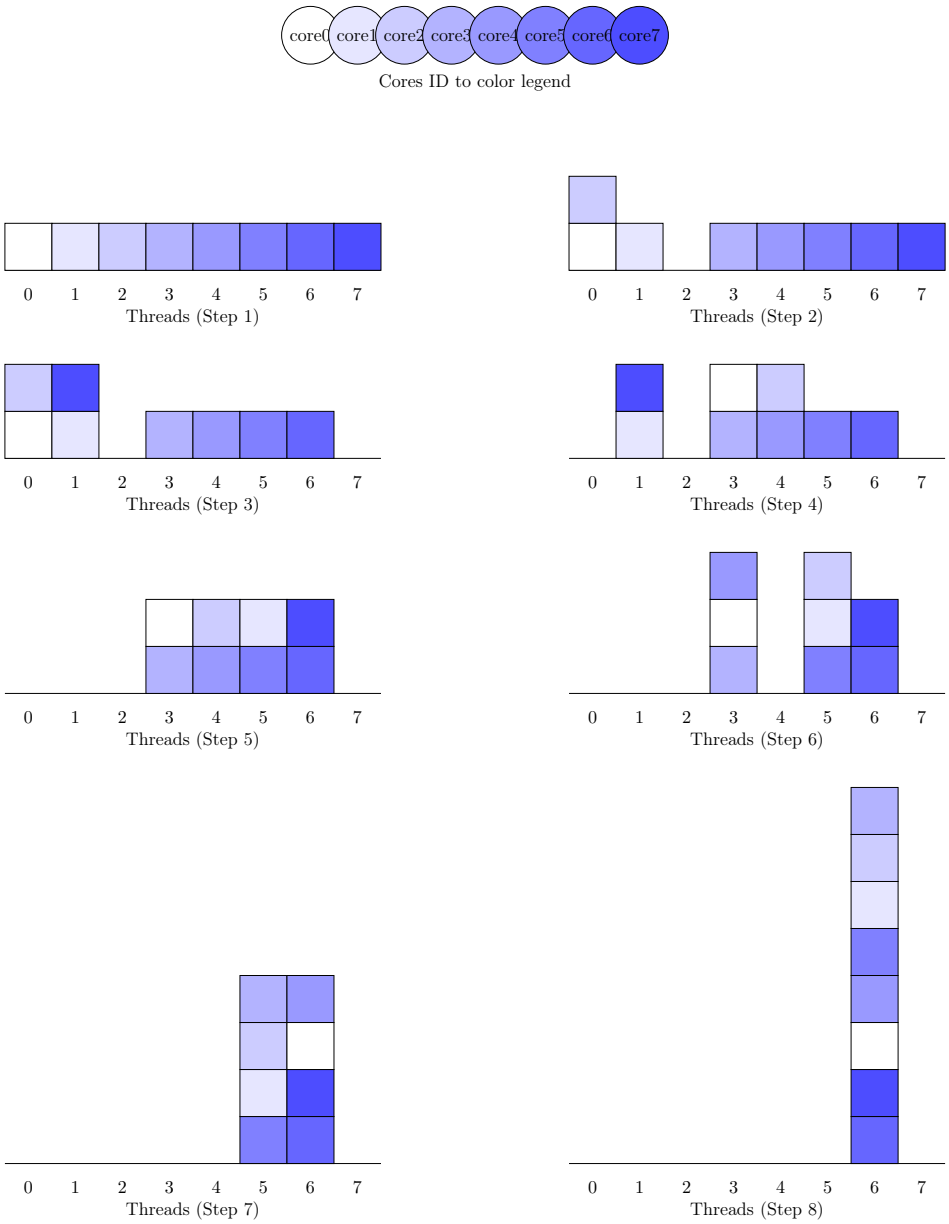


Figure 4.2: Example of evolution mapping of idle cores to active threads during the execution of the ICR algorithm on `Intel2x4` (8 cores) on matrix AUDI.

4.5 Optimization for earlier core detection

When comparing the time spent under $\mathcal{L}_{th}$ by each thread, we observed that several threads finished earlier when using Algorithm 4.1 than when not doing so. However, we also observed two problems:

- Firstly, several threads never used the additional cores available. In some cases, the time spent in the root of a subtree under $\mathcal{L}_{th}$ is between one third and one half of the time spent in the whole subtree. Thus, when the first thread finishes its subtrees, it often happens that others have already started the computation of the root of their final subtree, or even the last large BLAS call at the end of the factorization (line 13 of Algorithm A.2).

Therefore, they did not realize that new cores were actually available for them, due to the granularity of their computations. In less extreme cases, some threads did notice, at first, the availability of new cores, but failed to notice it afterwards. As we can see in Figure 4.2, the number of cores available for the last working thread (in our example the seventh (thread number 6)) converges progressively to the total number of cores of the machine. However, when this last thread started its last BLAS calls at the root of its last subtree, only two cores were available at that time; yet, more cores became available after that (step 4 then 8). We could observe that, on well-balanced $\mathcal{L}_{th}$'s, this phenomenon occurs frequently, whereas on badly-balanced $\mathcal{L}_{th}$'s, the algorithm is more efficient since more threads are able to notice the availability of new cores.

- Secondly, it happens that many threads take advantage of available idle cores and finish sooner than expected, but that the time spent under $\mathcal{L}_{th}$ remains unchanged. This could be explained by the fact that not enough cores were given to the thread with the largest amount of remaining work (like a critical path of execution), which determines the execution time under $\mathcal{L}_{th}$.

Two main phases may be distinguished in our dense partial factorization kernels: the first consists in computing the partial LU factorization by eliminating pivots; the second consists in updating the Schur complement by a call to TRSM and GEMM. The origin of the problem of not detecting (or detecting too late) the availability of new idle cores comes from this last phase, as it is the most time consuming part of the factorization which, once started, cannot be stopped or paused to dynamically increase the number of cores it uses. Hence, to test for the availability of idle cores more often, we decide to **split** these two last BLAS calls into pieces: we first update a block of rows of L, then update the corresponding block of rows of the Schur complement, then work on the next block of L, followed by the next block of the Schur complement, etc (See Figure 4.3). Such a split may provide better locality between TRSM and GEMM calls, at the cost of smaller blocks on which BLAS performance may be slightly slower. Finally, we decided to split the last two BLAS calls in only two pieces and only at the roots of the $\mathcal{L}_{th}$ subtrees; we did not observe an impact on BLAS performance. This decision is strengthened by the fact that the situation where idle cores can help active threads often happens only when all these threads work on the roots of $\mathcal{L}_{th}$ subtrees.

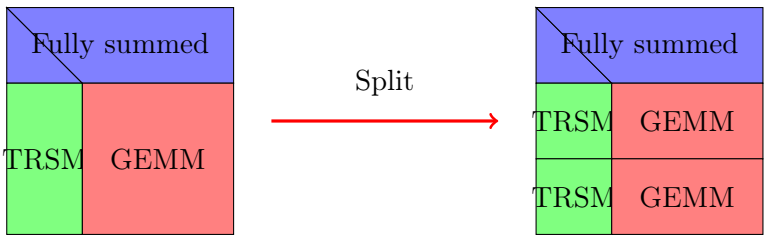


Figure 4.3: Splitting of the TRSM and GEMM calls in the update of the non-fully-summed rows of a front. A first TRSM followed by a GEMM is applied to the same set of rows, in order to increase data locality. A second TRSM followed by a GEMM is applied to the second set of rows. The detection of newly available cores is performed between each consecutive BLAS call.

4.6 Experimental study

4.6.1 Impact of ICR

Figure 4.4 shows times spent under $\mathcal{L}_{th}$ by threads for the AUDI matrix on `Intel2x4`, without the use of ICR, with the original ICR variant and with the early idle core detection variant. The X-axis represents threads under $\mathcal{L}_{th}$; the Y-axis represents their completion times. The threads are sorted by completion time (the order is the same for all variants). There is little variation between the different executions, as the first thread should finish at the same time in all cases, but its completion time actually varies a little between each of them. As can be observed, when applying the original ICR variant, the effect of idle core recycling is visible only on thread 6, which finishes significantly earlier, and on threads 1, 4 and 5, which finish earlier. The total execution time under $\mathcal{L}_{th}$ is thus not significantly improved for the reasons discussed in Section 4.5. With the early idle core detection variant plus splitting, we can see that nearly all threads take advantage of the available idle cores, which improves the execution time under $\mathcal{L}_{th}$ (92 seconds instead of 95 seconds for this (well-balanced) example).

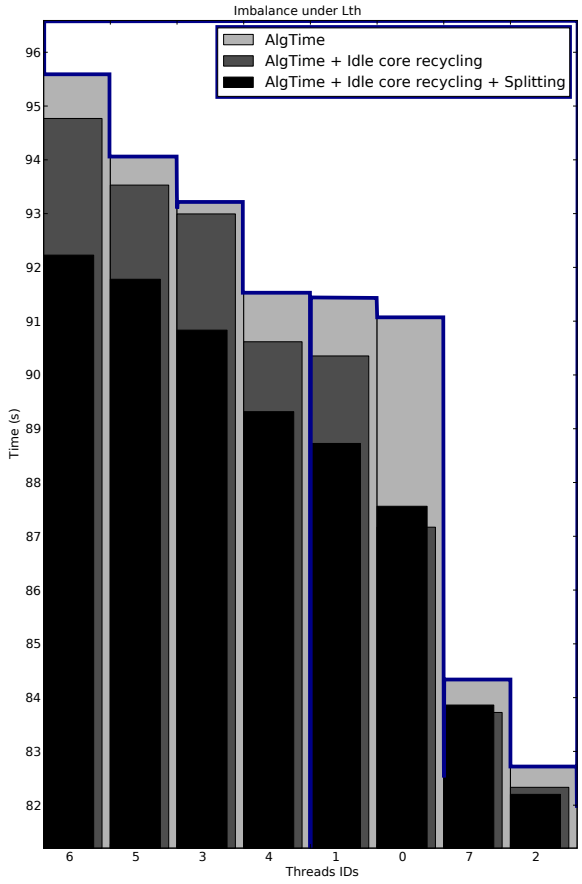
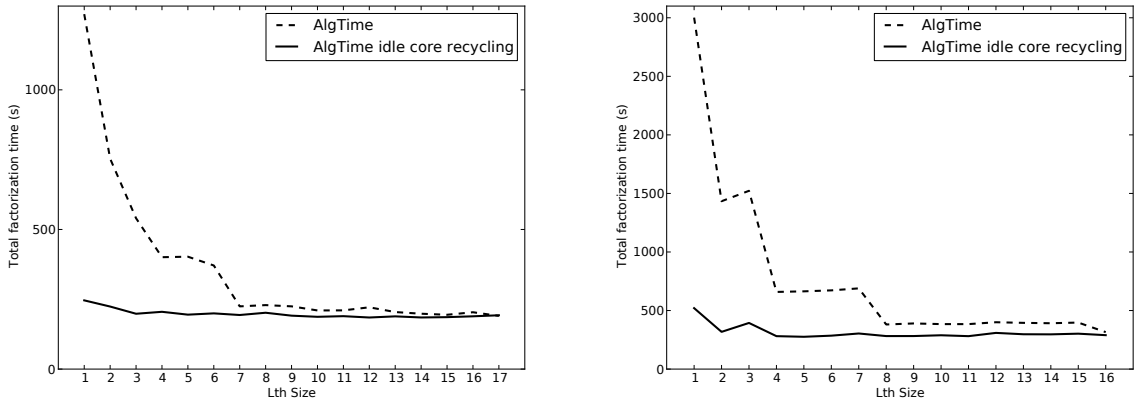


Figure 4.4: Time spent under $\mathcal{L}_{th}$ by each thread on the AUDI matrix on `Intel2x4` (8 cores), with and without the ICR algorithm and with the early idle core detection variant. In the last two approaches, when a thread finishes its last subtree, the corresponding cores are reassigned to other threads.

4.6.2 Sensitivity to $\mathcal{L}_{th}$ height

A consequence of reusing idle cores in Algorithm 4.1 is that the performance of the factorization is less sensitive to the height of $\mathcal{L}_{th}$. The algorithm may even work better when choosing an $\mathcal{L}_{th}$ higher than that found by AlgFlops or AlgTime: if we do this, the nodes in the tree handled sequentially with a normal $\mathcal{L}_{th}$ will still be handled sequentially, but some of the nodes that were originally treated using all cores will now be treated with less cores, leading to higher efficiency. Doing so gives Algorithm 4.1, whose initial aim was to improve load balance under $\mathcal{L}_{th}$, the additional goal of efficiently mixing tree parallelism with node parallelism.



(a) AUDI matrix on `Intel2x4` (b) conv3D64 matrix on `AMD4x6`

Figure 4.5: Robustness of the idle core recycling algorithm when $\mathcal{L}_{th}$ is raised. The leftmost parts of the dotted and plain curves ($\mathcal{L}_{th}$ composed of one node) correspond to the serial time and to the time with node parallelism only, respectively.

Figure 4.5 shows the total factorization time for AUDI on `Intel2x4` and for conv3D64 on `AMD4x6`, for $\mathcal{L}_{th}$ layers of different sizes, with and without the use of Algorithm 4.1. Using Algorithm 4.1, we see that a higher than normal $\mathcal{L}_{th}$ layer does not bring gain on AUDI, but at least provides a stable performance: Algorithm 4.1 allows to recover from imbalanced $\mathcal{L}_{th}$ layers.

On the conv3D64 matrix with more cores, there is some loss for certain sizes of $\mathcal{L}_{th}$, but gains with others. For example, with an $\mathcal{L}_{th}$ of size 16, the total computation time is 303.61 seconds; whereas with an $\mathcal{L}_{th}$ of size 5, the total computation time decreases to 275.87 seconds. Even though this gain of time is valuable, raising the $\mathcal{L}_{th}$ differs from what we have studied so far: such an $\mathcal{L}_{th}$ contains fewer subtrees (5 subtrees) than there are cores on the machine (24 cores), which means that most nodes under $\mathcal{L}_{th}$ have been treated with roughly 5 cores each. In such a situation, whole subtrees, and more particularly small nodes in the bottom of these subtrees, are computed with more than one core, which is not ideal for efficiency in the general case. This means that the gains are obtained higher in the tree: the side effect of raising the $\mathcal{L}_{th}$ layer is that tree parallelism is used higher in the tree, whereas less node parallelism is used, with a smoother transition between tree and node parallelism. Since the scalability of the dense factorization kernels we experimented with is not ideal on 24 cores, we can expect higher gains on this matrix.

4.7 Conclusion

In this chapter, we have described how to reduce the time wasted under the $\mathcal{L}_{th}$ layer. After dealing with issues of detection and assignment of idle cores, we have shown how to make idle cores help active threads finish their computations more quickly.

Our proposed Idle Core Recycling algorithm has a similar objective as work-sharing and work-stealing algorithms. Both aim at taking advantage of idle resources to dynamically (re)balance workload as much as possible. ICR can be seen as an alternative approach to work-stealing. Indeed, work-stealing algorithms follow an active approach, where idle computation resources explicitly make scheduling decisions on which ready tasks to steal from others' task queues. On the contrary, the ICR algorithm is an alternative that follows a passive approach, where idle computation resources delay scheduling decisions, letting active resources take the decisions for them and exploit them instead. Idle resources are thus assigned work from busy threads by joining their teams, instead of stealing work from their ready task queues. The work we have described in this chapter is similar in objective and approach to other works carried out in the context of runtime systems [69].

We believe that the algorithm we have presented has the potential to be applied to a much wider spectrum than the special case of our study. It may be applied to any application relying on malleable tasks, i.e., tasks that can be handled by many processes in parallel and whose number of active processes may change during their execution.

Chapter B

Conclusion of part I

To conclude Part I, we summarize the work in Section B.1, present some new results on applications in Section B.2, and finish with a discussion in Section B.3.

B.1 Summary

In this first part of the thesis, we have shown how to take advantage of optimized multithreaded libraries inside parallel tasks, while taking advantage of task-based parallelism on sequential tasks.

The proposed algorithms rely on a separation of two kinds of parallelism − tree and node parallelism −, in order to apply them to the parts of the task graph where they are the most efficient: tree parallelism is applied at the bottom of the tree and node parallelism at the top of the tree. The choice of the layer where to switch from one type of parallelism to the other relies on measures of the amount of work and we showed that performance models of the factorization kernels allow for more accurate decisions than the use of floating-point operations.

We have also shown how to adapt an existing multifrontal solver to implement the algorithms and have shown the impact of the proposed approaches on performance on real-life matrices. Although our implementation is simple and handcrafted, a more advanced one could be carried out in the future using emerging technologies like the task model of the OpenMP4 standard.

Moreover, we have shown that memory allocation policies could have a strong impact on the efficiency of the computations. To obtain good performance, each type of parallelism requires a specific type of memory allocation of the data it works on. Thus, on sequential tasks, it is more appropriate to work on data local to each thread; whereas in multithreaded tasks, as all threads work on the same set of data, these data would better be mapped onto all of the threads memory, in an interleaved way.

Furthermore, in order to leverage the main bottleneck of the previous approach based on work-sharing, we have shown how to increase the parallelism even further, by recycling idle cores and thus limiting the effect of the synchronization arising between threads. The resulting algorithm can be seen as an alternative to work-stealing, and some future perspectives are further discussed in Section B.3. We will present in that section some applications and possible extensions of the algorithmic work described in the previous chapters.

B.2 Real life applications

B.2.1 Electromagnetism

On the set of test matrices used by the FLUX software, in collaboration with University of Padova [9], we have applied `MUMPS` both without and with the AlgTime algorithm with the memory allocation techniques described above.

Test matrices arise from modelling induction heating industrial devices: heating of a susceptor by pancake coils and gear induction hardening. In the case of Pancake, starting from the same geometry, meshes are refined in order to solve problems of different sizes (Pancake 1, 2, or 3).

Table B.1 shows the order (N) of the matrices and the number of nonzero entries (NNZ LU) in the LU factors, and summarizes the experimental results obtained on `Intel2x4`.

Matrix	N	NNZ LU	MUMPS without AlgTime	MUMPS with AlgTime + Interleave
Pancake 1	320K	990M	223	194
Pancake 2	630K	2.1G	607	538
Pancake 3	1M	5.2G	2150	1974
Gear 1	370K	700M	104	82

Table B.1: Results (in seconds) comparing `MUMPS` with and without the AlgTime algorithms on matrices from the FLUX software, in collaboration with University of Padova.

Our approach brings a valuable gain in time on all matrices, even if this gain tends to decrease proportionally on large matrices. This is due to the fact that, on large problems, the portion of work in the top of the tree (above $\mathcal{L}_{th}$) increases relatively to the workload in the bottom of the tree. Still, this gain will always be present and will increasingly be due to the use of the *interleave* memory allocation policy. However, we remind (see Chapter 3) that even when most of the work is near the top of the tree, the use of the AlgTime algorithm is still necessary, because using the *interleave* policy without it tends to have catastrophic effects (on the bottom of the tree).

B.2.2 Partial differential equations and low-rank solvers

The work presented in this thesis exclusively concerns traditional full-rank factorizations. However, low-rank approximation techniques prove to be effective in reducing the overall amount of computations in multifrontal factorizations. The idea of low-rank compression is to compress off-diagonal blocks of the frontal matrices, as depicted in Figure B.1. The low-rank form can be used to perform the partial factorization of the fronts, resulting in large computational gains.

On matrices arising from finite difference or element methods, most of the full-rank computations occur at the top of the multifrontal tree; similarly, in low-rank, most of the compressions (gains) are observed on those same large fronts near the top of the tree. Thus, by combining $\mathcal{L}_{th}$-based algorithms with low-rank approximation techniques, the amount of work above the $\mathcal{L}_{th}$ layer decreases while that under $\mathcal{L}_{th}$ remains unchanged. The relative effect of $\mathcal{L}_{th}$-based algorithms is thus much more important when the time above $\mathcal{L}_{th}$ has been reduced because of low-rank compression.

To illustrate our discussion, we consider the *GeoAzur_2D_2000_2000* test case on `crunch` (a 32-cores Intel Sandy-Bridge machine). We compared the results obtained when applying: (*i*) the standard version of our solver; (*ii*) the AlgTime algorithm; and (*iii*) the low-rank version of the solver. The computation time when using the standard version of the solver is 64 *seconds*;

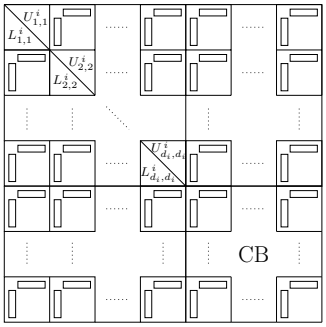


Figure B.1: Low Rank representation of a front.

whereas that when using the AlgTime algorithm is 21 *seconds*, where 13 *seconds* are spent under $\mathcal{L}_{th}$ and 8 *seconds* are spent above it. This means that the application of AlgTime improves the computation time of the fronts under $\mathcal{L}_{th}$, from $64-8=56$ *seconds* to 13 *seconds*. Assume that the use of the low-rank version of the solver gives a typical compression of the work of 80%. As most compression occurs on the large fronts above $\mathcal{L}_{th}$, we may consider that the time reduction due to low-rank compression is $8 \times 80\% = 6.4$ *seconds*, both without and with AlgTime, leading to a total time of 14.6 *seconds* with AlgTime versus 57.6 *seconds* without AlgTime. Thus, the use of AlgTime exacerbates the effect of low-rank compression.

B.3 Discussion

B.3.1 Multithreaded solve phase

One extension of our work is to apply the same ideas we used in the factorization phase to the solve phase. Indeed, in many industrial applications, very many solves are performed for each sparse matrix factorization. The number of operations during the solve then becomes significant with respect to the factorization, while the solve phase suffers from smaller GFlop/s rates than the factorization, especially in multithreaded environments, and even in the case of multiple right-hand sides (RHS).

In such cases, the exploitation of tree parallelism should also be considered. One could either re-use the $\mathcal{L}_{th}$ layer determined at factorization time, or determine a new $\mathcal{L}_{th}$ layer specific to the solve (which we believe will be higher than that for the factorization).

B.3.2 Scheduling threads in multithreaded BLAS

The results of the ICR algorithm presented in Chapter 4 opened the door to new perspectives. Indeed, dynamically reusing idle cores, to help active ones, has transformed our way of using multithreaded BLAS. Instead of using the BLAS either in a purely sequential or in a fully multithreaded way, the algorithm tends to use a smoother transition, from 1 to n cores per BLAS. The side effect of this technique, in addition to exploiting idle times, is that it improves performance even more than expected, by extending the effect of $\mathcal{L}_{th}$-based algorithms, i.e, by using few cores per BLAS on small tasks and many cores per BLAS on large tasks. However, we have no control over this effect, which is a consequence of the dynamic load imbalance of the system. The major perspective of the work discussed in Part I of the thesis is thus to develop algorithms that explicitly control this effect by using varying numbers of cores per BLAS instead of the "one or all" approach. The obvious difficulty is the scheduling of tasks, given a performance model of these tasks for varying numbers of cores per task, and given the dynamic imbalance of the system. Moreover, some multithreaded BLAS implementations efficiently take

advantage of up to only eight cores per BLAS, which means that node parallelism is limited by BLAS capabilities while tree parallelism is limited by the shape of multifrontal trees. On modern multicore architectures, with more than 64 cores per computing node, the scheduling algorithms must take into account these parallelism limitations in order to scale in terms of number of cores. We are already able to do this as the solver we have considered in our study was originally designed for distributed-memory architectures. It is therefore possible to use several MPI processes even in a single shared-memory computer, as will be shown in Part II. Although the ICR algorithm could be extended to such a hybrid environment, this is not the main priority.

B.3.3 Path to shared-distributed memory environments

In Part I of this thesis, we have focused exclusively on shared-memory environments. In practice, some industrial applications rely exclusively on such systems and wish to extract maximum performance out of them. However, even if modern shared-memory architectures grow in terms of numbers of cores, they are sometimes still limited in terms of amount of available memory. In order to scale in terms of problem sizes and amount of computing resources, one must target hybrid shared-distributed memory environments, where many more processors and memory units are available. Furthermore, even on shared-memory systems, a distributed-memory paradigm can be useful to deal with large numbers of cores.

Figure B.2 illustrates our distribution of processes and threads in such environments. Firstly, $\mathcal{L}_{th}$-like algorithms are applied across processes, rather than across threads. An $\mathcal{L}_{ps}$ layer is then created, separating subtrees that will be treated separately by each process under it from fronts that will be treated by several MPI processes above it. Secondly, our proposed $\mathcal{L}_{th}$-based algorithms are applied to each process's subtrees, as explained before. The share of work under all the aggregated processes' $\mathcal{L}_{th}$ layers gradually shrinks with the number of processes, compared to what we have usually encountered in pure shared-memory environments. This is why, in hybrid environments, we must focus on other approaches. The work of Part II of this thesis will thus consist in optimizing the computations that occur above the $\mathcal{L}_{ps}$ layer.

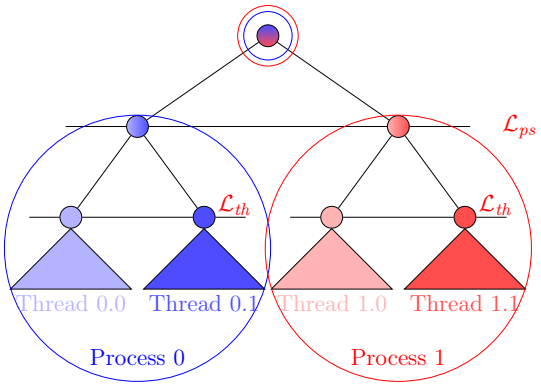


Figure B.2: Possible distribution of processes and threads over a multifrontal tree in an hybrid shared-distributed memory environments.

Part II

Hybrid-memory environments

Chapter C

Introduction to part II

The multifrontal factorization phase can be viewed as a traverse of a dependency graph called the elimination tree, or assembly tree, from the bottom to the top. At each node, an assembly phase followed by a partial LU factorisation, referred to as node factorization, is performed (see Section 1.3.3). In the first part of this thesis, a shared-memory processing of the assembly tree and both node and tree parallelisms have been considered. In this second part of the thesis, we consider distributed-memory and hybrid shared-distributed memory environments. Although many approaches exist to handle parallelism in multifrontal environments, the approach we rely on is the one described in [13, 15], and is thus related to our asynchronous environment. We will first study in Chapter 5 the distributed memory factorization of a frontal matrix and will extend this study in Chapter 6 to the frontal matrix factorization of a chain of nodes. Three bottlenecks of the distributed memory multifrontal phase and of its extension to general trees will be identified in Chapter 6. In Chapter 7, we will address the first one, related to reducing the volume of communication during the assembly process. In our asynchronous and limited memory environment, one can further limit synchronizations by introducing deadlock avoidance algorithms. This will be discussed in a very general framework in Chapter 8 and will be described in some more detail (both from algorithmic and implementation perspectives) in our asynchronous distributed memory context, in Chapter 9.

In this introduction, we first describe in some more detail the parallelism, synchronisation and communication mechanisms involved during the multifrontal factorization phase. We first briefly introduce in Section C.1 all types of parallelism involved during factorisation. A special focus is proposed in Section C.2 on the asynchronous node assembly process, since it is at the heart of the communication patterns and involves partial synchronisations between nodes of the tree. We conclude this introduction with the description, in Section C.3, of the distributed memory node factorization.

In this chapter, we use the notations of Table C.1 to refer to process nodes, and of Table C.2 to refer to messages.

C.1 Types of parallelism

Let us consider the processing of an assembly tree in a hybrid shared-distributed-memory environment (see Figure C.1). We can subdivide this processing into three categories.

Firstly, each leaf of the condensed assembly tree in Figure C.1 represents a subtree of the assembly tree, and is handled by one MPI process only, in a purely shared-memory environment, as described in Part I of this thesis. The selection of these subtrees is performed following an algorithm very similar to that described in Part I for the same purpose ($\mathcal{L}_{ps}$-based algorithm).

Nodes which are processed by one process only (with possibly many threads inside it) will be referred to as nodes of *"type 1"*, and the corresponding parallelism in the assembly tree will be referred to as *"type 1 parallelism"*.

Secondly, as already mentioned in Part I, the potential of tree parallelism decreases when going up towards the root of the tree, whereas that of node parallelism increases. Indeed, it has been observed [12] that more than 75% of the computations are most often performed in the top three levels of the assembly tree (mainly on 3D problems). Thus, it is necessary to obtain further parallelism within the large, nodes near the root of the tree. Hence, the remaining nodes of the tree, whose frontal matrices are sufficiently large are each handled by several MPI processes. Using the terminology of [14], such nodes will be referred to as *"type 2"* nodes and the corresponding parallelism as *"type 2 parallelism"*. This parallelism is obtained by doing a 1D block partitioning of the rows of these frontal matrices, and by applying pipelined factorizations on them. We say that this 1D partitioning is block *acyclic* (as opposed to block cyclic) because each process holds a single block of contiguous rows in the frontal matrix (as opposed to many blocks with a cyclic mapping of the blocks of rows to the processes). We associate with each of these fronts one process, called the *Master process (M)*, that will handle the factorization of the fully-summed (FS) rows. We also associate with each front a set of candidate processes [109] that may be involved in its computation but from which only a subset of processes, called *worker processes (S)*, will be dynamically selected by the master (before it starts the factorization of the front) to do the corresponding updates of the contribution block (CB) rows. The selection of workers is based on dynamic scheduling decisions that depend on the current CPU load and memory usage of each candidate process (and of the master).

Thirdly, a 2D block-cyclic partitioning of the frontal matrix at the root node of the assembly tree is performed, if it is large enough, using ScaLAPACK. The parallel root node will be referred to as a node of *"type 3"* and the corresponding parallelism as *"type 3 parallelism"*.

Figure C.1 illustrates the dynamic subdivision and mapping of type 1, type 2 and type 3 nodes in an assembly tree, where the subgraph corresponding to the pipelined factorization of type 2 nodes is only defined at runtime.

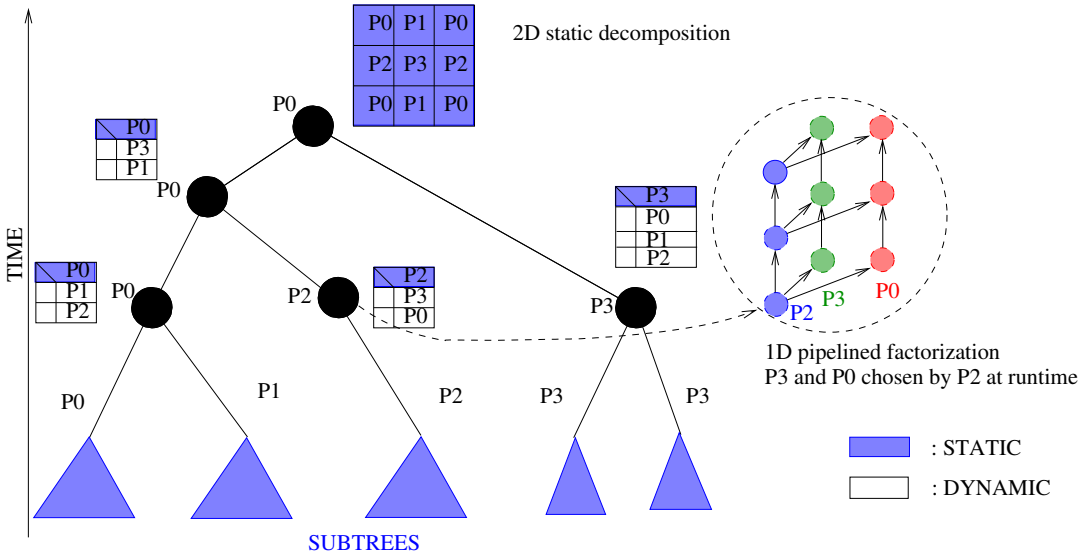


Figure C.1: Distribution of the computations of a multifrontal assembly tree on four processors P0, P1, P2, and P3.

C.2 Distributed-memory node assembly algorithm

Let us now consider the assembly of a parent front (P) from its children fronts (C_1, ..., C_n). Figure C.2 describes the four communication steps of the assembly algorithm in a distributed-memory environment.

In the multifrontal method, assembly trees must be traversed in a topological order. Thus, MP (Master process of the Parent front) must first receive from each MC_i (Master process of Child front i) the confirmation that they have finished their work in C_i, which is done through the reception of *M2M* (Master to Master) messages from each MC_i to MP. In addition to their use for synchronization, these messages contain a lot of relevant symbolic information, like the list of workers involved in the children (Cs), the mapping of rows of Cs, the list of delayed rows (non-eliminated rows), etc. Once MP has received all the symbolic information from the MCs, and has identified parts of the initial sparse matrix corresponding to front P, it will, first, build the structure of P, i.e.: the ordered list of the variables of P. Secondly, it will choose SPs (Workers of Parent front P) from the set of potential candidate processes that could be involved in P, based on an estimate of the workload and memory load of all processes. Thirdly, it will compute the mapping of the rows of P on the chosen SPs. Then, MP sends to each SP a *DB* message notifying it of its selection for the computation of P, alongside its mapping in P, i.e., the list of rows of P this SP will be in charge of updating. Upon reception of this message, each SP allocates the necessary resources for handling its part of the computation of P. Moreover, knowing both the mapping of SCs and SPs, MP sends a *MAPROW* (standing for *"mapping of the rows"*) message to each SC_i, notifying it of which rows of the contribution block of C_i it must send and to which SP. Each SC then sends the effective rows to the appropriate SPs through so-called *CBT2* ("ContriButions for Type 2 nodes") messages. Upon reception of such messages, SPs can then perform the numerical assembly (extend-add operations) in P, row by row.

C.3 Distributed-memory node factorization algorithm

Let us consider the factorization of a front (F). The distributed-memory dense partial factorization kernels we use rely on a **1D block acyclic pipelined dynamic asynchronous** algorithm (see Algorithm C.1 and Figure C.3). In order to partially factorize a front of size *nfront* with *npiv* fully-summed (*FS*) rows and *ncb* non-fully-summed or contribution block (*CB*) rows, with *nproc*MPI processes, one process, designated as the *master process* (M), will handle the factorization of the FS rows, and the *nproc*-1 (or *nworker*) other processes, called *worker processes* (S), will manage the update of the CB rows.

On the one hand, the master uses a blocked *LU* algorithm with threshold partial pivoting. Pivots are checked against the magnitude of the row (instead of column, usually) but can only be chosen within the first $npiv \times npiv$ block. After factorization of a panel of size *npan*, the master sends it to the workers, asynchronously and in a non-blocking way, through so-called *Block of Factors (BF)* messages, together with the column permutations due to pivoting. Then, depending on the adopted algorithmic variant, right-looking (RL) or left-looking (LL), the master immediately updates either the whole remaining non-factored rows using the just factorized panel (RL), or only the next panel, using all the preceding panels (LL). On the other hand, workers update in parallel all their rows after the reception of each new panel (RL).

Factorization operations rely on BLAS1 and BLAS2 routines inside panels, whereas update operations (both on master and workers) mainly rely on BLAS3 routines, where TRSM is used to update columns of newly eliminated pivots and GEMM is used to update the remaining columns.

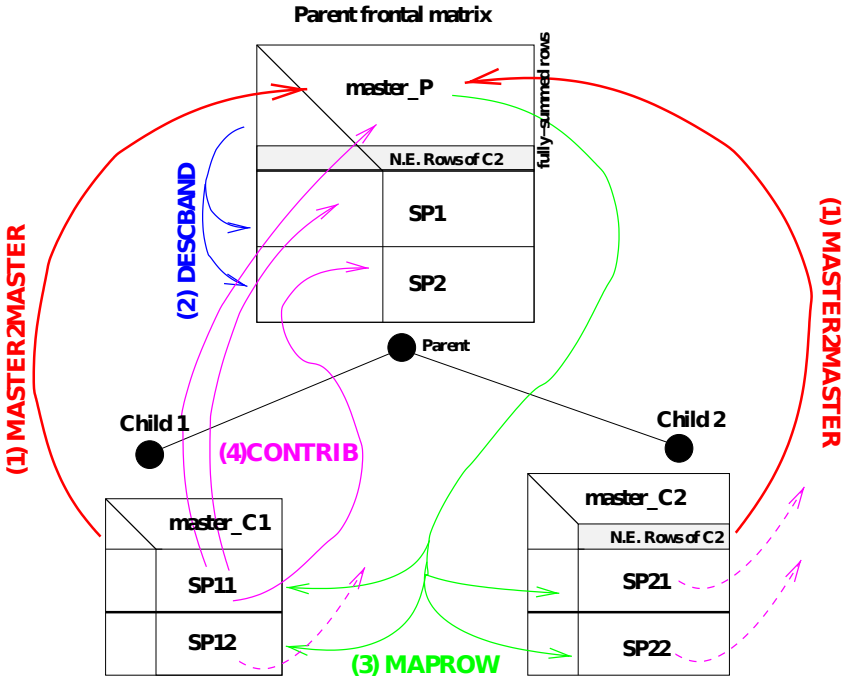


Figure C.2: Distributed-memory assembly of a parent front from its children fronts. Messages involved in the assembly operation are (1) M2M (Master2Master), (2) DB (DescBand), (3) MR (MapRow), (4) CBT2 (ContriBution Type 2).

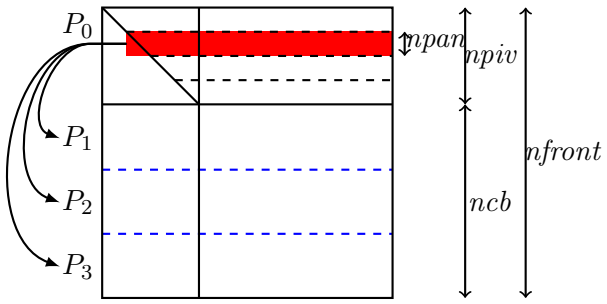


Figure C.3: Distributed-memory partial factorization of a front of size *nfront*, with *npiv* fully-summed variables to be eliminated by panels of *npan* rows, and $ncb = nfront - npiv$ rows to be updated. Messages involved in the factorization are BF messages.

C.4 Management of asynchronism and communication

Algorithm C.2 describes the scheduling algorithm (performed by all processes) which drives the asynchronous multifrontal solver. Each process tests the arrival of (any) new message, and treats it depending on its type (or tag), using the corresponding operation. If no message is available, the process starts local ready tasks (if any).

In our particular implementation of the asynchronous multifrontal method, processes have special communication buffers and specific ways of handling messages.

Each process has a cyclic buffer, called a *send-buffer*, dedicated to the transmission of messages and designed to store several messages simultaneously. However, one can store messages only on top of all the others, in a cyclic way. Thus, if some space becomes free between the locations of two other messages, this space cannot be used to store any new message (fragmentation

1: **Right-looking approach on the master side:**
2: **for all** horizontal panels $P = F(k : k + npan - 1, k : npiv + ncb)$ in the fully summed rows **do**
3: **while** a stable pivot can be found in columns $k : npiv$ of P **do**
4: Perform the associated row and/or column exchanges
5: Scale pivot column in panel (_SCAL)
6: Update panel (_GER)
7: **end while**
8: Send factorized panel to the workers **asynchronously**
9: Update fully-summed column block $F(k + npan : npiv, k : k + npan - 1)$ (_TRSM)
10: Update the fully-summed rows $F(k + npan : npiv, k + npan : npiv + ncb)$ (_GEMM)
11: **end for**

1: **Right-looking updates on a worker holding rows** *rowbeg* **to** *rowend***:**
2: Receive the next factorized panel from the master **asynchronously**
3: Apply column exchanges specified in the message, if any
4: Update fully-summed columns $F(rowbeg : rowend, k : k + npan - 1)$ (_TRSM)
5: Update Schur complement $F(rowbeg : rowend, k + npan : npiv + ncb)$ (_GEMM)

Algorithm C.1: Distributed-memory dense partial factorization of a frontal matrix F of order $npiv + ncb$ with $npiv$ variables to eliminate and a Schur complement of order ncb. $npan$ is the block size for panels. We assume that there are $npiv$ pivots to (try) eliminate. The matrix is distributed on the processes in a 1D acyclic way. The master has the fully-summed rows mapped onto it, and the workers have the rows of the contribution block distributed across them.

1: **while** (no global termination) **do**
2: **if** (some received messages can be processed) **then**
3: process them
4: **else**
5: check whether a new task can start and activate it (involving other processes)
6: **end if**
7: **end while**

Algorithm C.2: Asynchronous multifrontal scheme.

of memory).

Each process has another buffer, called a *receive-buffer*, dedicated to reception of messages. This buffer is meant to be large enough to store any BF message. CBT2 messages (assembly messages containing contributions of child fronts to be assembled in parent fronts) are usually too large to fit in the receive-buffer and are then split into separate pieces and transferred in sequence by the sender. All other messages described in Figure C.2 mostly hold symbolic information and are consequently much smaller, and will thus fit in the receive-buffer.

In distributed-memory environments and particularly in asynchronous environments, we will explain and illustrate in detail in Chapter 8 how messages could arrive too early (causality dependency is not guaranteed by MPI communication layer and must be handled at the application level). A process that has received a so-called **early message** might not be able to process it correctly as it has to receive all late messages upon which the early message depends to be able. Thus, in addition to the receive-buffer, each process may exceptionally and dynamically allocate another buffer, called a *temporary-buffer*, to temporarily store the early message. Unfortunately, the situation is significantly more complex because the processing of a late message might in-

volve sending messages in a context where the send-buffer might already be full. We see that asynchronism may then introduce recursivity that can be very demanding in terms of buffer memory and that could even be a source of deadlocks. Both issues will be deeply analysed and discussed in Chapter 8.

C.5 Experimental environment

Test cases We will mainly focus on separate dense fronts in order to optimize the kernels on them, before optimizing them over the whole multifrontal factorization at the end of the thesis. The matrices we choose for this purpose will be matrices mainly arising from 3D finite difference and finite element problems.

Test machines In this second part of the thesis, we rely on three multicore computers:

- `crunch`: a 4 × 8-core Intel Sandy Bridge Xeon E5-4620 2.20 GHz Processor, with 384 GigaBytes of memory and 2.7 TeraBytes of disk, from LIP-ENS Lyon;
- `devel`: 8 cluster nodes of a 2 Quad-core Nehalem Intel Xeon X5570 2.93 GHz Processor, with 24 GigaBytes of memory, from Plafrim in Bordeaux;
- `hyperion`: 368 cluster nodes of a 1 × 8-Core Intel Xeon X5560 2.80 GHz Processor, with 36 GigaBytes of memory, from CALMIP-CICT in Toulouse;
- `ada`: 332 nodes of a x3750-M4 quadri processor Intel Sandy Bridge 8-cores 2.70 GHz, with 128 GigaBytes of memory and a 4.1 Go/s InfiniBand network, from IDRIS, in Orsay;
- `eos`: 612 nodes of an Intel(R) IVYBRIDGE 10-cores 2.80 GHz Processor, with 64 GigaBytes of memory and 6.89 Go/s Infiniband Full Data Rate network, from CALMIP-CICT in Toulouse.

C.5.1 Summary of main notations used in PartII

Throughout the study, we will use the following notations in Table C.1, C.2 and C.3.

Notations for messages can refer to specific nodes, as in the examples below:

- BF(P): BF message relative to node P;
- $CBT2_{SC}^{SP}$: Contrib Block Type 2 message from SC to SP.

Notation can be combined, for example:

- SP, MC: worker of parent, master of child;
- TC, TP, TG: *IBcast* tree of C, P, G.

M:	**M**aster process
S:	**S**lave process
C:	**C**hild node
P:	**P**arent node
G:	**G**randparent node
F:	**F**ront
$\mathbf{F}_i$**:**	**F**ront indexed by **i**

Table C.1: Notations related to factorizations.

BR / BS:	communication **B**uffer of **R**eceives / **S**ends
BF:	Block of Factors message, containing factorized panels
MR:	MapRow message, containing mapping information of child rows in the parent
CBT2:	Contrib Block Type 2 message, containing contribution rows sent from a worker in a child (SC) to a process in the parent (MP or SP)
DB:	DescBand message, containing row distribution information, sent by the master of a node to its workers
M2M:	Master to Master message, from MC to MP
ENDNIV2:	EndNiv2 message (introduced later, in Section 9.2), notifying a master process of the completion of its computations by the sending worker process

Table C.2: Notations related to messages type.

T:	*IBcast* Tree
$pred_F(S)$**:**	Predecessor of process S in *IBcast* Tree of front F
$sucs_F(S)$**:**	Successors of process S in *IBcast* Tree of front F
$ancs(F)$**:**	Ancestor fronts of F (higher fronts in the chain / assembly tree reachable from F)
$desc(F)$**:**	Descendant fronts from F (lower fronts in the chain / assembly tree reachable from F)

Table C.3: notations related to multifrontal and broadcast trees (mainly in Chapters 8 and 9).

Chapter 5

Distributed-memory partial factorization of frontal matrices

5.1 Introduction

In this chapter, we present a study of 1D block pipelined asynchronous distributed-memory factorization kernels (See Section C.3). Firstly, we describe these kernels in an ideal computation and communication environment, in order to exhibit their inner characteristics, properties and limits. Secondly, we show adaptations of these kernels that scale in terms of the size of the targeted problems and of the size of the computers.

Even though distributed-memory multifrontal factorization kernels are meant to be used in asynchronous environments, where processes are usually active on several fronts simultaneously, a preliminary study consists in investigating first their performance bottleneck on single dense fronts before trying to improve them on whole sparse matrices. Moreover, very often on large problems − typically arising from finite-element domain decompositions, which represent an important part of the challenging problems we target −, most of the computations are spent on the large fronts at the top of multifrontal trees, where tree parallelism is reduced and where only a limited number of fronts may be computed in parallel. This gives even more importance to the optimisation of distributed-memory kernels on single fronts. Furthermore, when dealing with limited memory issues and when using memory-aware approaches, memory constraints tend to make schedulers reduce parallelism [98], which, in addition to serializing factorizations of some fronts, tends to map more processes on single fronts, making the scalability of single-front factorization an even more important issue.

Let us remind the reader that the main notations used in this chapter are described in Tables C.1, C.2 and C.3.

5.2 Preliminary study on an example

Preliminary experiments showed that our distributed-memory LU factorization (see Section C.3) is efficient enough on whole multifrontal trees, in the sense that many processes spend most of their time in the application doing computations, but that such a factorization shows poor performance on single fronts, even on shared-memory architectures where network communication issues do not matter. The aim of this section is to analyse some low-level issues that could partly explain this behaviour.

5.2.1 Experimental environment

We profile the factorization of a single front in the following way.

Firstly, in order to perform the factorization of a single front, we generate a sparse matrix whose multifrontal analysis phase leads to an assembly tree composed of a single front only. We have implemented a Python script (**genmtx.py**) that generates a sparse matrix from the shape of a multifrontal tree. To reduce storage and to minimize the time in the analysis phase, a front is represented as an arrowhead with values equal to 1 in the off-diagonal entries of the first row and column. To avoid numerical difficulties during the factorization phase of these matrices, each front has a dominant diagonal with values equal to $10 \times nfront$. This very general script will also be of use in the following chapters when testing factorization algorithms on whole multifrontal trees and when testing algorithms in specifically-targeted cases. Here, we will use it to generate fronts of arbitrary characteristics (for example, SimTest, as defined below).

Secondly, we denote by **SimTest** the test case we generate for our factorizations. The characteristics of the main front of SimTest are given in Table 5.1 (See also Figure C.3).

characteristics	values
nfront	10000
npiv	2155
ncb	7845
nproc	8
nthread (/proc)	1

Table 5.1: Characteristics of SimTest.

We have carefully chosen the ratio $\frac{npiv}{nfront}$ so that the amount of Flops in the master and in each worker is the same, in order to achieve good load balance. Moreover, due to our specific experimental environment (see Chapter C), as distributed-memory factorizations cannot be applied to leaves of the multifrontal tree, we add to the initial front two very small children fronts of size $nfront = 150$ ($npiv = 100$ and $ncb = 50$) that become the leaves of the multifrontal tree. The computation time for these two leaves is negligible and we therefore ignore them. Furthermore, in the multifrontal method, full dense factorizations are achieved on roots of the tree. Thus, as the front to be computed is the root of the multifrontal tree, we use an option (called the Schur option) of the solver to achieve a partial (rather than full) factorization.

Thirdly, we use the ITAC (Intel Trace Analyser and Collector) software to profile the behaviour of the solver. The Gantt-charts it produces are similar to those of traditional profiling tools such as TAU or VAMPIR (which it relies upon). ITAC represents processes with time lines, where red parts correspond to the time spent in the MPI layer and blue parts correspond to the time spent in the application layer. In our experiments, the first process at the top of each ITAC figure is the master and all the others are the workers.

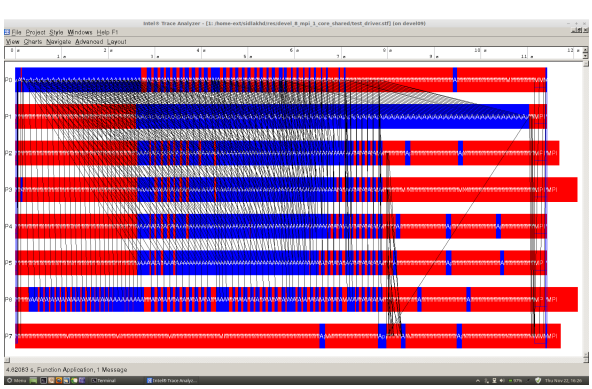
Fourthly, in order to ascertain that the network is not the cause of a possible bottleneck, we run the experiment on a shared-memory architecture; namely, one 8-cores node of the `devel` machine (see Section C.5), using 8 MPI processes (using IntelMPI) with one thread (core) each.

5.2.2 Description of the experimental results

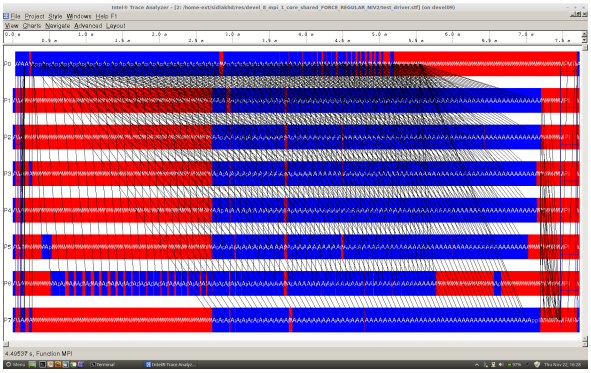
In order to have a reference for comparison, we previously ran an experiment using a single MPI process with 8 threads. We used the shared-memory factorization kernel Algorithm A.2. The resulting factorization time is 4.6 *seconds*, which gives us a reference for our distributed-memory kernels.

When we ran the experiment, the factorization time with 8 MPI processes using Algorithm C.1 increased to 12 *seconds*. The visual trace in Figure 5.1a evidences many problems. Firstly, not all available processes are involved in the computation. The last one (Process 7, at the bottom of the figure) is not used. It just remains in the MPI layer waiting for messages, only reacting to non BF (Block of Factors) messages because no BF message is sent to it. Secondly, when looking at the mapping of the rows of the front, we could observe that the amount of work was unbalanced across processes: the first worker has 2246 *rows* of the front to update; whereas each of the other workers has 1120 *rows*! Thirdly, all workers (but one) start their work very late, when the master has already done nearly all its work. Indeed, once the workers start to work, the master very often alternates between the application layer for computations and the MPI layer for sending panels, probably because its send-buffer (see Section C.4) is then full.

The first and second problems are understandable. They arise because the solver makes dynamic decisions on the mapping of processes on fronts. Its scheduling algorithms are well adapted to asynchronous environments on entire multifrontal trees, but not to single fronts. Thus, in order to avoid the first and second problems, we simply forced a static mapping that uses all the available processes and gives them the same amount of work (equal number of rows on all workers). The computation execution time then decreased to 7.3 *seconds*, instead of 12 *seconds* (see Figure 5.1b).



(a) Gantt-chart of workers default behaviour. Factorization time of 12 *seconds*. The master is Process 0.

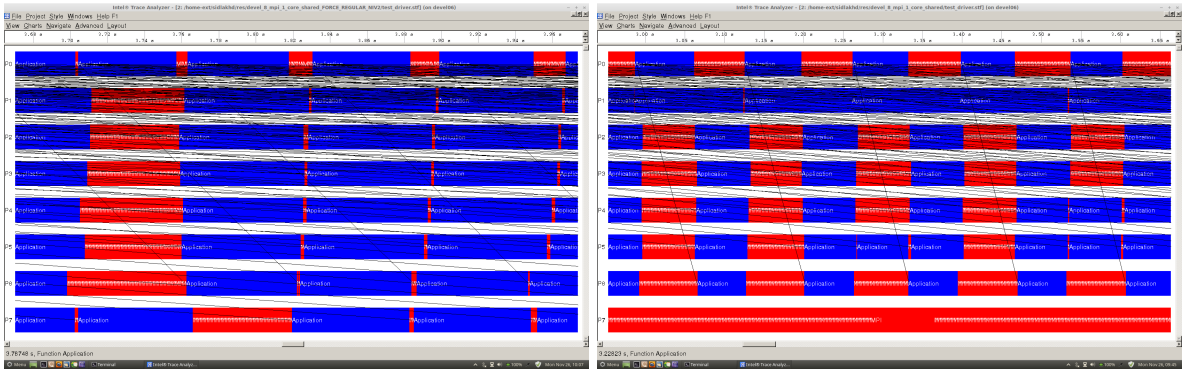


(b) Gantt-chart when forcing a static mapping with equal amount of work on all processes. Factorization time of 7.3 *seconds*. The master is Process 0.

Figure 5.1: Experiment of the SimTest case on the `devel` machine with 8 single-threaded MPI processes.

The origin of the third problem is harder to understand. It arises because there is no overlap between communications and computations. To understand its origin, we need to zoom into the Gantt-chart to observe more deeply what happens when the master computes a panel and sends it to the workers (Figure 5.2a and Figure 5.2b). We can see that the workers effectively receive panels (short red rectangles) only when the master enters the MPI layer at the same moment. When a worker misses the time window while the master is inside the MPI layer, it must wait (long red rectangle) for the next window to actually receive its message.

At the beginning (first half of Figure 5.1b), when the master starts its factorization, as it still has much work to do and much free space in its send-buffer (section C.4), it uninterruptedly does computations, spending (nearly) no time in the MPI layer as it makes calls only to the immediate asynchronous *ISend* MPI routine, which comes back as quickly as possible from the MPI layer to the application layer. This makes all workers (but one) starve, spending their time in the MPI layer, waiting for new panels. Only the second-to-last worker is lucky enough to receive some panels from the master. When its send-buffer starts to be full, by the middle of its factorization, the master must wait for it to free-up a little before being able to send newly computed panels and to continue its computations. Usually, the master looks for other tasks to complete on other active fronts it is mapped on. However, in our case, the master has nothing else to do. Thus, it enters a loop of active waiting in the MPI layer, using the *MPI_ TEST* or *MPI_ WAIT* routines, which forces the master to enter the MPI layer and makes previously submitted communications progress. From that moment, the master has much less work to do than the workers, and the time it needs to compute and to do the updates related to any future panel is less than the time needed by the workers to do the updates of any panel. Thus, each time the master tries to send a newly computed panel, it discovers that its send-buffer is still full. It must wait in the MPI layer for all the workers to receive the oldest panel sent (in order to free up some space in the communication buffer) to be able to store newly computed panels and continue the computations. This is why the more the master approaches the end of its computations, the more time it spends in the MPI layer. Thereafter, the workers start to receive and treat panels uninterruptedly, until the end of their computations.



(a) Zoom 1 (b) Zoom 2

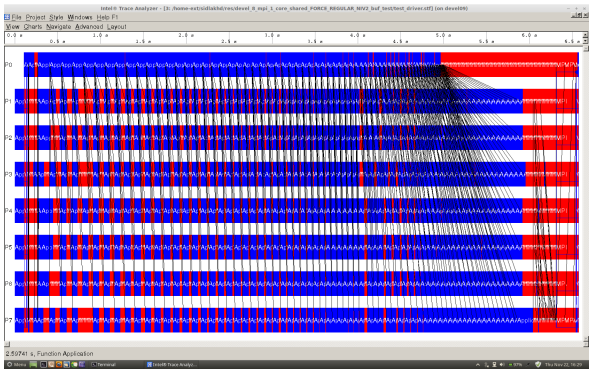
Figure 5.2: Non-overlap of communications and computations on the `devel` machine on 8 single-thread MPI processes (Zoom on Figure 5.1b)

This non-overlap phenomenon of communications and computations is well known. It is simply due to MPI implementations not having a special thread dedicated to them, which could handle MPI communications in parallel with computations [66]. The thread calling the MPI communication routine immediately returns to computations, without letting the MPI layer make communications progress, so that communications only take place when the process re-enters the MPI layer. Although some MPI-dependent solutions exist (e.g., setting MC_CSS_INTERRUPT to True on IBM machines), we have decided to activate a communication thread each time the code enters an intensive computation phase (typically, a BLAS3 call). One very temporary solution we have used consists in forcing the master to enter the MPI layer much more often. This is done inside panels after the factorization using each pivot (which is quite brutal), and between the calls to TRSM and the calls to GEMM, by calling the *MPI_ Test* routine for that purpose. As all CPU cores are intensively used in BLAS3 calls, this additional thread is invoked periodically (each millisecond) to make a call to *MPI_ TEST* on the

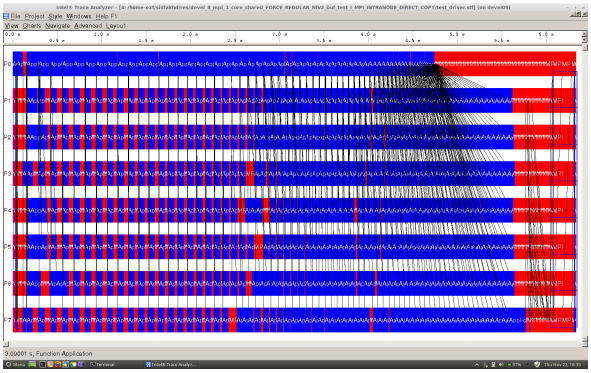
submitted requests, and directly returns to sleep, so as not to impact computation performance.

We can observe in Figure 5.3a that the workers start their work much sooner than before, immediately after the master sends the first panel. As a consequence, the computation time is now reduced to 6 *seconds*.

One minor additional improvement consists in optimizing the send/receive phases, by setting the *IntelMPI* environment variable *I_MPI_INTRANODE_EAGER_THRESHOLD* to the value of the largest possible panel, thus avoiding the copy of messages whose length is smaller than this size in shared-memory environments, and making the sending and reception of messages quicker across processes (see Figure 5.3b). As a consequence, the computation time is now further reduced to 5.7 *seconds*.



(a) Introduction of an artificial communication thread-like mechanism. Factorization time of 6 *seconds*.



(b) Improvement in the case of shared-memory machines. Factorization time of 5.7 *seconds*.

Figure 5.3: Real limitations of our distributed-memory kernel on the SimTest case on the `devel` machine on 8 single-threaded MPI processes.

One interesting point is that the amount of time spent in the application is nearly the same for the master as for the workers. We say "nearly" because, for the factorization of panels, the master not only performs efficient BLAS3 operations (that workers also do) but also computes some less efficient BLAS2 operations (that workers do not do). This shows that the flop measure is representative enough of the time on such a large matrix when using single-threaded processes, but the GFlop/s rate of the factorization is less important than that of the updates, which is understandable.

Other problems exist that can be observed in Figure 5.3, and which are the real limitations of the distributed-memory factorization kernels we consider. We will discuss these problems in detail in Section 5.3.2. For now, we will model the performance of the distributed-memory factorization kernels in order to find the most appropriate factorization algorithm for the master.

5.3 Right-looking and Left-looking 1D acyclic pipelined asynchronous partial factorizations

5.3.1 Model and Simulator

Let us define the model we will use to study distributed-memory kernels. The parameters of the front being factorized are *nfront*, *npiv* and *ncb*. The factorization algorithms we will consider are:

- the blocked right-looking (RL) variant defined by Algorithm C.1; and
- the blocked left-looking (LL) variant, for which we use a left-looking approach on the master, while RL is still applied on the workers.

An additional parameter such algorithms rely upon is *npan*, the panel size. The machine parameters to use are: *nproc*, the number of MPI processes; α, the GFlop/s rate of factorization; β, the GFlop/s rate of update; and γ, the network bandwidth.

In this section, we will consider the ideal case where computations always take place at machine peak and where communications are immediate. The reason why we start with this ideal case is to discover the inner characteristics of the two RL and LL variants. We already know that real computations and communications are far from this perfect case, and that they are an important bottleneck. We will thus study their impact further, both on communications, (Section 5.4) and on computation (Section 5.5).

Our first attempt consisted in modelling factorizations analytically. Figure 5.4 shows the context. We used the *MAPLE* software to help in this task, due to the complexity of the equations arising when solving problems such as finding optimal parameters of the factorizations.

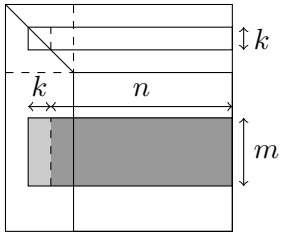


Figure 5.4: Illustration of the factorization of a panel of size $k \times (k+n)$ on the master and of the corresponding update on a worker. The light and dark gray areas represent the pieces of the front on a worker on which a TRSM and GEMM are applied, respectively.

Equation 5.1 represents the number of floating-point operations required to factorize a panel of k rows and $k+n$ columns:

$$Wf(k,n) \rightarrow \left(\frac{2}{3}\right) k^3 + \left(n - \frac{1}{2}\right) k^2 - \left(n + \frac{1}{6}\right) k \ . \tag{5.1}$$

This is the result of summing of the floating-point operations of the factorization of each row:

$$Wf(k,n) \rightarrow \sum_{i=k-1}^{0} i + 2 * i * (i+n) \ . \tag{5.2}$$

Equation 5.3 represents the number of floating-point operations required to process a block of rows on a worker (factorization of the L factors and update of the contribution part) of m rows and $k+n$ columns by a panel of k rows and $k+n$ columns (we thus assume a right-looking algorithm):

$$Wu(m,n,k) \rightarrow W_{TRSM}(m,k) + W_{GEMM}(m,n,k) \ , \tag{5.3}$$

with

$$W_{TRSM}(m,k) \rightarrow mk^2 \ . \tag{5.4}$$

and

$$W_{GEMM}(m,n,k) \rightarrow 2\,mnk \ . \tag{5.5}$$

MU_i, the time to perform an update related to the i^{th} panel by the master, is given by:

$$MU_i = \beta \times Wu\left(npiv - \min\left(npiv, i * npan\right), npiv + ncb - \min\left(npiv, i * npan\right), \min\left(npan, npiv - (i-1)\, npan\right)\right) \quad . \quad (5.6)$$

SU_i, the time for an update related to the i^{th} panel by a worker, is given by:

$$SU_i = \beta \times Wu\left(\frac{ncb}{nslave}, npiv + ncb - \min\left(npiv, i * npan\right), \min\left(npan, npiv - (i-1)\, npan\right)\right) \quad . \quad (5.7)$$

MF_{i+1}, the time of factorization of the $(i+1)^{th}$ panel (if it exists) by the master is given by:

$$MF_{i+1} = \alpha \times Wf\left(\min\left(npan, npiv - npan\ \min\left(i, floor\left(\frac{npiv}{npan}\right)\right)\right), npiv + ncb - (i+1) * npan\right) \quad . \quad (5.8)$$

We have attempted to describe the total factorization time of a RL factorization by Equation 5.9.

$$TRight = MF_1 + \sum_{i=1}^{ceil\left(\frac{npiv}{npan}\right)} \max\left(SU_i, MU_i + MF_{i+1}\right) \quad (5.9)$$

As distributed-memory factorizations are discrete-event-based, it is hard to describe them analytically, as shown by the above formulas.

Even though it is possible to build analytical formulas, albeit complicated, to express some properties, we have decided instead to consider the implementation of a Python simulator for distributed-memory factorizations. Our simulator is naturally able to simulate varying communication and computation models and to produce Gantt-charts of the factorization. In order to illustrate some inner aspects of the algorithms studied, we first consider, in this section (5.3) only, that communications take place at infinite bandwidth and that computations take place at a constant rate. As said in the model description, this allows us to reveal better some intrinsic properties of RL and LL algorithms. Notice that we use the RL variant in order to verify that the Maple formulas and the simulator give the same results.

In the following subsections, we compare the behaviour of both RL and LL variants.

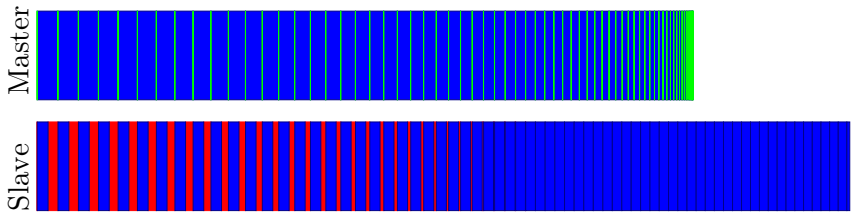
5.3.2 Gantt-charts with RL and LL factorizations on the master

Even if the sequence (RL or LL) of computations and communications theoretically has no impact on the completion time of the master process, it is important for the overall computation time, as they directly impact the scheduling of the workers.

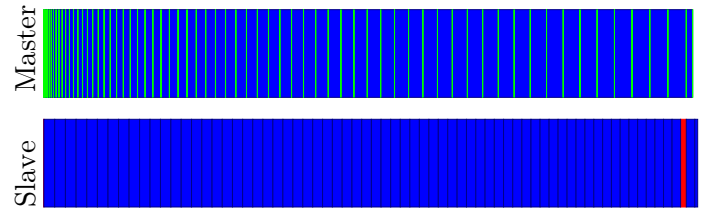
Figure 5.5 shows the Gantt charts of the simulated execution of the `SimTest` test case. Both right-looking (Figure 5.5a) and left-looking (Figure 5.5b) block factorization variants are used on the master, while workers perform their updates upon reception of each panel, thus always in a right-looking way. In each sub-figure, the Gantt chart at the top represents the activity of the master and the lower one, that of a single worker, all 7 workers theoretically behaving in the same way. Moreover, green, blue and red parts represent factorization, update and idle phases, respectively.

Figure 5.5a is unsurprisingly very similar to what we observed previously in the actual experiments (see Figure 5.3). It clearly illustrates the weakness of the RL approach. The

workers have many idle phases while the master does not, which makes the workers complete later than the master. Furthermore, under good load balance conditions, the workers' idle phases sum to the gap between master and workers completion times. In the RL variant, the size of the matrix to be updated by the master, at each successive panel, decreases by *npan* rows and *npan* columns. Whereas, in the case of the workers, it only decreases by *npan* columns, the number of rows to be updated remaining unchanged. Thus, the amount of update operations related to each panel is initially higher for the master but decreases faster than for the workers, where it remains nearly unchanged. We see then that, near the beginning of its factorization, after it has sent a panel to the workers, the master takes longer to perform the corresponding update and to factorize the next panel than the workers require for the update related to that panel. Thus, in practice, workers waste a lot of time, spinning in the MPI layer, waiting for new panels to arrive. However, as the factorization progresses, this waiting time decreases at each new panel until messages are available as soon as the workers need them. At that moment, the master has less rows to update and is able to feed the workers more quickly than they manage to perform the corresponding updates. The workers then work continuously. In practice, when the master finishes all its computations, it stays idle inside the MPI layer, trying to send panels to the workers as soon as they are ready to receive them, while they intensively pursue their work.



(a) right-looking.



(b) left-looking.

Figure 5.5: Gantt-chart of the RL and LL algorithms on the SimTest case. Factorization (green), updates (blue) and idle times (red).

Generally, when the work is well balanced between master and workers, the scheduling behaviour we would expect from a good factorization algorithm on the master side is to make master and workers work uninterruptedly and make them all finish at the same time. The bottleneck for the workers is the production of factorized panels by the master for their future use. It is thus critical for the workers that the master produces panels as soon as possible, delaying its own updates as much as it can. This is one of the aims of look-ahead algorithms [76] which make sure that the workers are always well fed. Their behaviour lies between the two RL and LL extreme variants. The application of the left-looking variant on the master results in the nearly perfect Gantt chart of Figure 5.5b. In the beginning, the master produces panels much faster than the workers can consume. Gradually, this advance tends to diminish until both master and workers terminate, almost at the same time.

5.3.3 Communication memory evolution

Figure 5.6 shows the evolution of the memory utilization in the send-buffer of the master in `SimTest`, assuming that panels are sent as soon as they have been computed. This buffer is the place in memory where factorized panels computed by the master are temporarily stored (contiguously) and sent using non-blocking primitives; when the workers receive a panel, its corresponding memory in the send-buffer can be freed.

Most of the time, the buffer in the RL variant only contains one panel, immediately consumed by the workers. When the master computations decrease (for the last panels), it rapidly produces many panels that cannot be consumed immediately by the workers. In contrast, the LL variant has always enough panels in reserve, ready to be sent. This is because, when the work is balanced over processes, RL is not able to correctly feed the workers, whereas LL is.

We remark that the peak of the memory buffer (to store panels) needed for RL is 36 MB while it is 41 MB for LL. The scheduling advantage of LL thus comes at the price of a slightly higher buffer memory. Nonetheless, this additional memory remains very reasonable compared to the total memory used by the master process for the factorization: $nfront * npiv * sizeof(double) =$ 172 MB. However, in practice, send-buffers may have a given limited size that is smaller than the peaks of Figure 5.6. Currently, in the factorization algorithm, if only a few panels can fit in buffer memory, the master must wait when the send-buffer is full, leading to some performance loss.

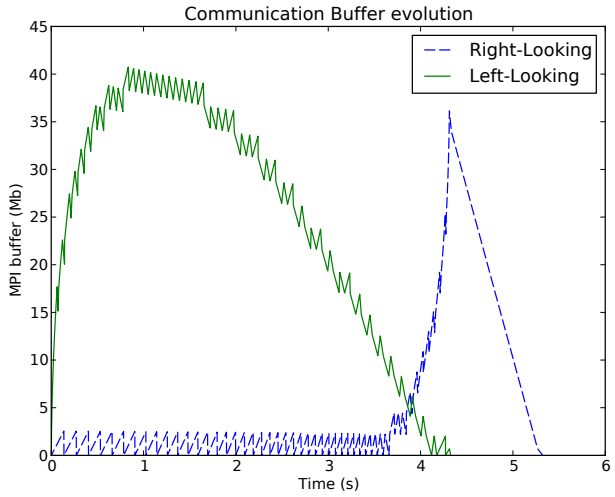


Figure 5.6: Communication buffer evolution using RL and LL algorithms.

One solution would consist in not using any intermediate send-buffer at all but instead pointing directly to the memory in the front where the panel is located so that MPI can access it. However, it would be complicated to do so as the panel in the front is not contiguous in memory (in our experimental setting/implementation), and because in our asynchronous environment, memory management algorithms may decide to move a front (garbage collection).

Another solution is to control the peak of buffer memory, by avoiding to post `MPI_ISend`s as soon as possible. This solution consists in keeping buffer memory under a predefined threshold by copying panels from the frontal matrix to the buffer only when enough space is available. This should be performed independently from the fact that many more panels might have already been computed. We will present a more appropriate solution later in this chapter (see Section 5.4).

5.3.4 Influence of the granularity parameter *npan*

Figure 5.7 shows the influence of the panel size *npan* on the execution times in `SimTest`. The red curve represents the execution time of the master process (identical for RL and LL, still under the hypothesis of constant GFlops/s rate), whereas the blue and green ones represent the total execution time of the RL and LL factorizations, respectively. In this simulation, the smaller the value of *npan* (compared to *npiv*), the more efficient RL and LL will be, thanks to a shorter start-up and termination of the pipelines in the factorization. In practice, however, in order to achieve good BLAS and network performance, we would prefer a higher *npan* value (32, for example), thus performing computations/communications on a few large blocks/messages rather than on many small ones. Even though LL is more sensitive to variations of *npan* (the green curve is not as smooth as the blue one), as long as *npan* remains small compared to *npiv*, performance of both variants remains stable and acceptable. For the largest value of *npan* ($npan = npiv = 2155$), a single panel is used, with no overlap between master and worker computations (and no pipeline).

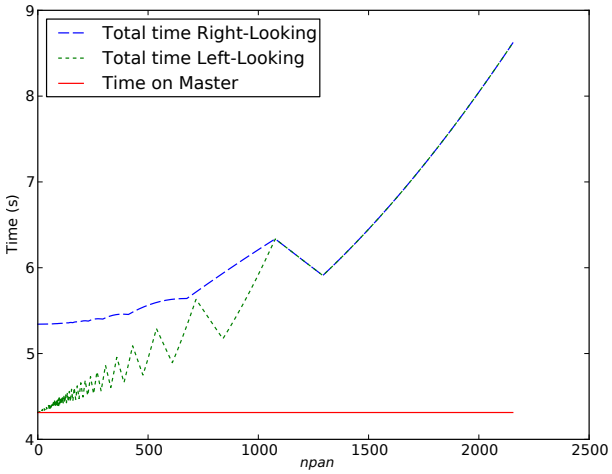


Figure 5.7: Influence of *npan* on RL and LL algorithms.

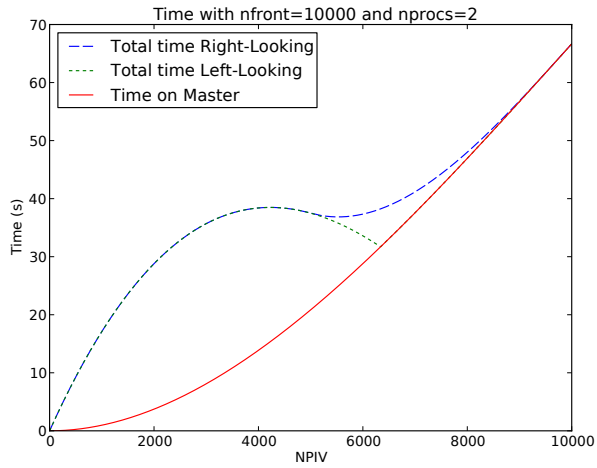
We note that simulations for other values of *nfront* (with the corresponding well chosen *npiv*) show that, the absolute value of *npan* does not matter, but that the ratio of $\frac{npan}{npiv}$ is the relevant parameter that dictates the behaviour of both algorithms.

5.3.5 Influence of *npiv* and *nfront* on the load balance

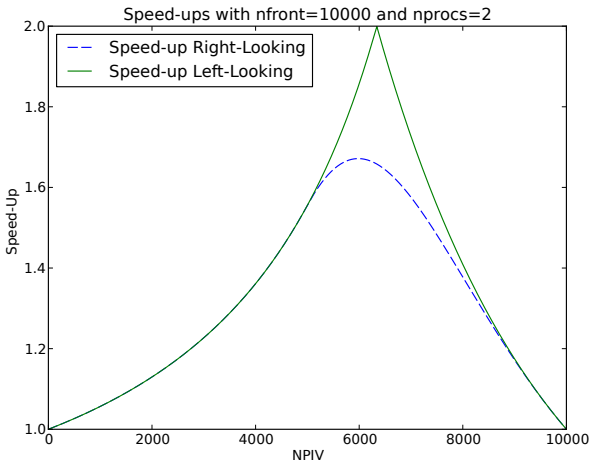
Although *npiv* and *nfront* are mainly defined by the sparsity pattern of the matrix to be factorized, we will illustrate in Chapter 6 that we have some leeway to modify these two values. For now, we study the influence of *npiv* for a fixed *nfront*.

Figure 5.8 shows the influence of *npiv* on the speed-ups for `SimTest`, with $npan = 1$ and $nproc = 2$ (instead of the default value 8). We chose $npan = 1$ to avoid artificial effects due to start-up and termination of the pipelined factorization, and $nproc = 2$ (one master, one worker) to highlight some effects more clearly. We divide the discussion in two, depending on the value of the ratio $\frac{npiv}{nfront}$. In the first part, for $npiv \leq x$ (where x is a certain value, here ≈ 5100), the LL and RL algorithms behave exactly the same: worker processes are the bottleneck, because they have much more work than the master. In both variants, the master is able to produce panels faster than the workers can consume them. In the second part, for $npiv > x$, LL improves. Intuitively, x is the threshold value of *npiv* above which the update time on the master process with respect to the first panel becomes higher than the corresponding update time on the worker, leading to some idle time on the worker. Both variants then attain their peak speed-up but for different ratios of $\frac{npiv}{nfront}$. In the right most part of the curve in Figure 5.8, for large values of

npiv, the execution time of both algorithms asymptotically tends to that of the master process. The latter becomes the bottleneck because it has much more work to do than the workers.



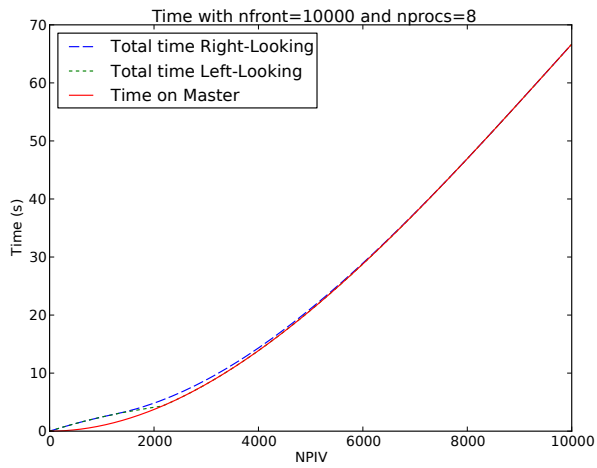
(a) Time with $nproc = 2$



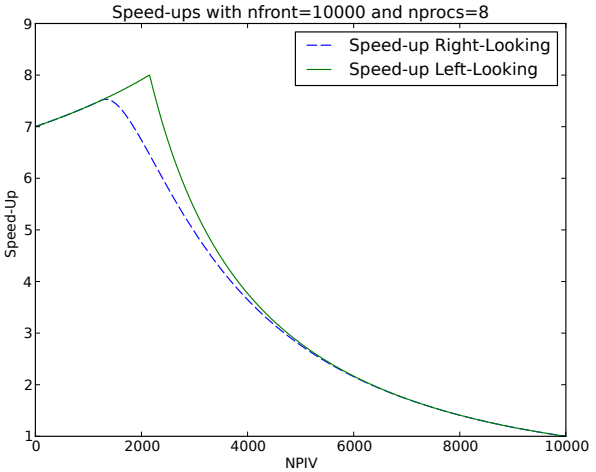
(b) Speed-up with $nproc = 2$

Figure 5.8: Influence of *npiv* on LL and RL algorithms with 2 processes on speed-up, with respect to the serial version. We note that, when *npiv* increases for a given *nfront*, the total amount of work increases. However, the speed-up (and GFlops rates proportional to the speed-up) only depends on the $\frac{npiv}{nfront}$ ratio.

5.3.6 Influence of the scalability parameter *nproc*



(a) Time with $nproc = 8$



(b) Speed-up with $nproc = 8$

Figure 5.9: Influence of *npiv* on LL and RL algorithms with 8 processes.

Figure 5.9 is similar to Figure 5.8, except that more processes are used ($nproc = 8$ instead of 2). The maximum speed-ups of RL and LL become closer for larger *nproc*. LL reaches its maximum speed-up when all processes (master and worker) get the same amount of computation, here when $npiv = 2154$. In this case, neither the master nor the workers slow down each other and an ideal speed-up of 8 is obtained. On the other hand, RL appears to reach its maximum speed-up when all processes (master and workers) are *roughly* assigned the same number of rows. This approximation is based on the fact that the more processes, the less relevant the master

time will be, compared to that of all workers, as one unit of time on the master is equivalent to *nworker* units of time on the workers. In order to never make worker processes wait, it is sufficient in RL to avoid their first idle period. This can be achieved by making the update time of the master process related to the first panel plus the factorization time of the second one the same as the update time of the worker processes for the first panel. For a given *nfront*, we denote by, *eqFlops* and *eqRows* (*Flops equilibrium* and *Rows equilibrium*) the value of *npiv* for which LL and RL reach their optimum, respectively.

5.4 Communication schemes

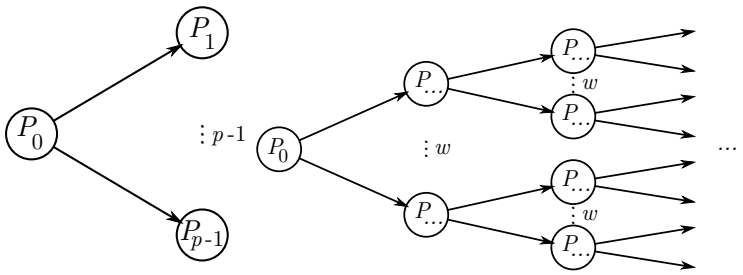
5.4.1 Sharing bandwidth for panel transmission (*IBcast*)

We found that our experimental results were very similar to those of the model, as long as the ratio between computation and communication remains large enough (*nfront* being relatively large compared to *nproc*). Strong scaling, i.e., increasing *nproc* for a given *nfront*, globally increases the amount of communications while keeping the amount of computations the same. The master process sends a copy of each panel to more worker processes, decreasing the bandwidth dedicated to the transmission of a panel to each worker: the maximal master bandwidth (aggregated master network channels bandwidths) is divided by *nworker* in this one-to-many communication pattern. The panel transmission rapidly becomes the bottleneck of the factorization. Although many broadcast algorithms exist, our need is to have an asynchronous, pipelined broadcast algorithm. For example, a binomial broadcast tree would not be appropriate because once a process has received a panel and forwarded it, it makes sense to keep the send bandwidth available for the next panel in the pipelined factorization. Many efficient broadcast implementations exist for MPI (see, e.g. [111]), and asynchronous collective communications are part of the MPI-3 standard. However the semantic of these operations requires that all the processes involved in the collective operation call the same function (`MPI_IBCAST`). This is constraining for our asynchronous approach which is such that each process can, at any time, receive and process any kind of message and task: we want to keep a generic approach where processes do not know in advance if the next message to receive in the receive-buffer (Section C.4) is a factored panel or some other message. For these reasons, we have implemented our own asynchronous pipelined broadcast algorithm based on `MPI_ISEND` calls. Instead of sending a panel to all worker processes (*nworker* processes), the master process only sends it to a restricted group of worker processes (w processes), which will in turn forward the panel to another group of worker processes. This scheme to parallelize communications can be represented by a broadcast tree rooted on the master process, as shown in Figure 5.10b. The shape of this tree is simply described by two parameters: the width w, which represents the number of worker processes to whom panels are relayed; the depth d, the maximal number of forwardings, from the master to worker processes. These two parameters are related, as setting one of them sets the other ($d = log_w(d)$).

5.4.2 Impact of limited bandwidth

The Gantt charts of Figure 5.11 show the impact of the communication patterns with limited bandwidth per process. With the baseline communication algorithm, workers are most often idle, spending their time waiting for broadcast communications to finish, before doing the corresponding computations, whereas the tree-based (here using a binary tree) shows perfect behaviour. When further increasing *nproc* or with more cores per process, we did not always observe such a perfect overlap of communication and computation, but the tree-based algorithm always led to an overall transmission time for each panel of $\frac{nfront \times npan \times w \times \log_w(nproc)}{\gamma}$ much smaller than that

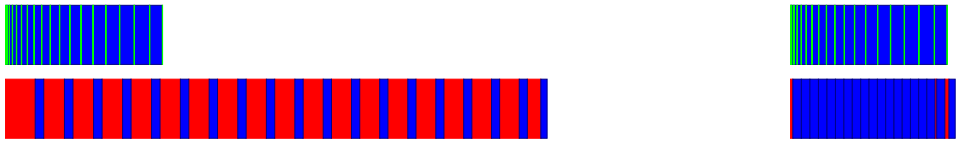
of the baseline algorithm $\frac{nfront \times npan \times (nproc-1)}{\gamma}$.



(a) Baseline algorithm.

(b) Tree-based algorithm.

Figure 5.10: One-to-many communication pattern used for sending blocks of factors from the master process to the worker processes. In the baseline algorithm, the master process sends data to every process (with a loop of `MPI_ISEND`). In the tree-based asynchronous broadcast, every process corresponding to an internal node of the tree sends data to w other processes. $w = p-1$ is equivalent to the baseline algorithm.



(a) Baseline algorithm (loop of Isends).

(b) Tree-based algorithm.

Figure 5.11: Influence of the IBcast communication pattern with a limited bandwidth per proc (γ=1.2 Gb/s, α=10 GFlops/s) on LL algorithm with $nfront = 10000$, $npan = 32$, $nproc = 32$ and $npiv$ chosen to balance work (idle times in red).

5.4.3 Experimental Results

On the one hand, choosing $w = nworker$ (and then $d = 1$) is equivalent to applying the baseline algorithm (flat tree). For each point-to-point communication, the bandwidth decreases and the transmission time increases. On the other hand, choosing $w = 2$ (binary tree) has opposite effects. Not only does point-to-point bandwidth increase, but more network links are used in parallel, so that a much higher overall bandwidth is exploited, decreasing the overall transmission time. However, the drawback is that, in the first case, all worker processes finished receiving each panel at the same time; whereas, in the second case, the cost of pipeline priming (start and end of the pipeline), and the gap between the time the first worker (a successor of the root in the broadcast tree) finishes receiving the first/last panel and the time the last worker (the deepest leaf of the broadcast tree) does, becomes significant. This cost must unfortunately be added to the execution time of the worker processes in the total computation time. Moreover, this creates load imbalance between worker processes. Experimentally, we have measured the factorization time for a front of size $nfront = 10^5$ with $npiv$ respecting $eqRows$ ($npiv = 3062$) and $nproc = 64$ with 8 cores per process. We have run on `ada`, as its network has a high bandwidth, which allows us to show the impact of *IBcast* even on such a powerful machine. The results are: 29 seconds without *IBcast*; 26 seconds with *IBcast* with a broadcast tree of depth 2; and 23 seconds with a binary broadcast tree. This shows the benefits of *IBcast* and the influence of the shape of the broadcast tree.

5.5 Computation schemes

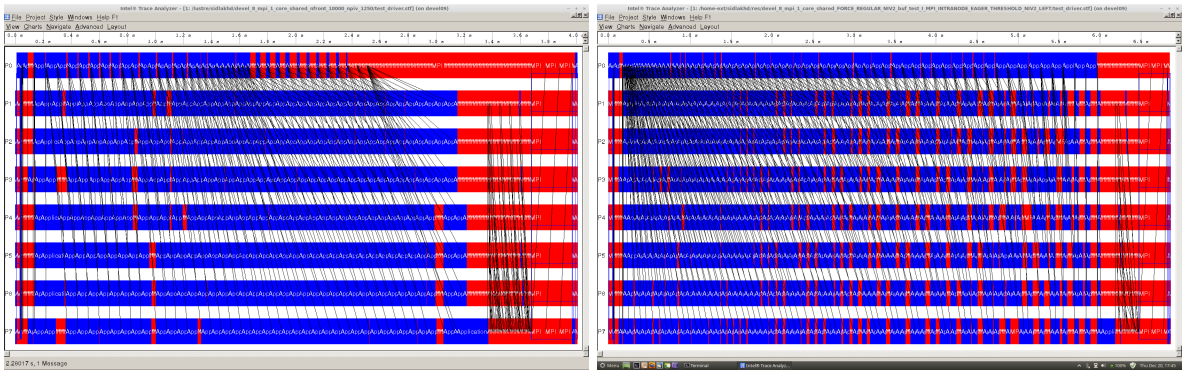
In this section, we consider realistic computations instead of the perfect model previously used, where all computations took place at machine peak. We now show how computations behave in practice, and how to overcome their limitations.

5.5.1 Impact of limited GFlops rate

The previous promising simulation results led us to implement the left-looking variant of the distributed-memory asynchronous factorization kernel. When running `SimTest` (see Table 5.1) on the `devel` machine (see Section C.5), the behaviour of RL (with *eqRows*) was as expected (Figure 5.12a), whereas that of LL (with *eqFlops*) was unexpected (Figure 5.12b).

In LL, we can see without surprise that the workers start at the same time as the master, and finish at the same time, too. However, instead of having significant idle periods at the beginning, which progressively vanish during factorization, the workers contrarily evidence progressive outcome of idle periods, that become more and more important as the computations progresses. Despite the valuable properties of LL, the master seems to be a bottleneck for the workers.

We might think that the fact that *eqFlops* is applied on LL while *eqRows* is applied on RL is the reason for this discrepancy. Indeed, the master is assigned an additional amount of work in LL compared to RL. However, this is only partly true. The underlying reason is that, even for the same amount of work given to the master (*eqFlops* in both RL and LL variants), the master is still slower in LL than in RL, requiring 5.6 *seconds* against 4.6 *seconds* (as shown in Section 5.2.2), and thus too slow compared to the workers. Thus, in LL, even if the master is able to feed the workers at the beginning of the factorization, this advantage is quickly lost, more quickly than expected, and the workers rapidly become starved, waiting longer and longer for panels. A deeper analysis shows that the factorization times of the successive panels in LL are the same as in RL, but that the corresponding update times (on the master) are unexpectedly longer. Two main reasons explain this phenomenon.



(a) right-looking algorithm with *eqRows* (b) left-looking algorithm with *eqFlops*

Figure 5.12: Behavior of the RL and LL algorithms using their best $\frac{npiv}{nfront}$ ratio, on `devel` machine with 8 MPI processes and 1 thread per process.

Firstly, there are far more TRSM operations compared to GEMM operations in LL than in RL. Each update on the master consists of a call to TRSM and a call to GEMM. In the case of RL, the ratio of work spent in TRSM compared to that spent in GEMM remains small for the updates relative to all panels. In the case of LL, this ratio increases at each new panel. As

the GFlops/s rate of GEMM operations is generally better than that of TRSM operations, this limits performance on the master with LL.

Secondly, the overall efficiency of TRSM and GEMM is always lower on the master than on the workers. Indeed, the workers perform their computations on the same large matrix block; whereas the master operates on different (increasing (LL) or decreasing (RL)) sizes of blocks as the factorization proceeds. This affects the performance of the master in RL but even more so in LL.

5.5.2 Multi-Panel factorization in shared-memory

The performance problem previously revealed is more a shared-memory related issue than a hybrid-memory one. In this section, we consider alternative computational approaches. This study aims at improving distributed-memory factorization kernels, but should naturally also improve the shared-memory kernels presented in the first part of the thesis.

5.5.2.1 Multiple levels of blocking

Dense linear algebra factorization algorithms usually use blocking for efficiency. This means that, instead of executing operations on single rows or columns, using BLAS2 routines, they execute operations on blocks of rows or columns, using more efficient BLAS3 routines. This is the case when whole panels (blocks) are factorized before ithey are uses for updates (Algorithm C.1). In order to push this approach one step further, we can also use multiple levels of blocking. The factorization of a single block relies on a second level of blocking (or more), which extends the use of BLAS3 routines even further and restricts the use of BLAS2 routines only to the bottom level of the blocking.

Let us measure the effect of this latter approach on partial factorization. Our experiment consists in measuring the factorization time of fully-summed rows only, without taking into account the update time of the trailing sub-matrices. We have used a separate code written in C. We have used matrices of different *nfront* and *npiv* values and we have tried different levels of blocking (1, 2 and 3), with different block sizes (*npan*1, *npan*2 and *npan*3) on each level, using the RL variant at each level of the blocking. We have chosen the values of *nfront*, *npiv*, *npan*1, *npan*2 and *npan*3 from the set $\{2^i, i \in [1..13]\} \cup \{10^i, i \in [1..4]\}$. As special effects could be observed when powers of two are used, we added powers of ten to this set. Since we did not need a distributed-memory machine but only a single shared-memory node, we have implemented the experiment on the `crunch` computer (see Section C.5), with 1 core and 8 cores, always mapped on the same socket to avoid NUMA effects.

When using only one level of blocking, we observe that the computations with small (large) values of *npan*1 are inefficient. *npan*1 is either too small or too large to limit the use of BLAS2 operations and, at the same time, enable BLAS3 to reach good performance. The values of *npan*1 that allow us to meet each of these criteria are very different. It is thus necessary to introduce another level of blocking.

Tables 5.2 and 5.3 show the best computation times obtained and the corresponding best block sizes (among all possible values of *npan*'s from the above-defined set), both in the sequential and the multithreaded cases, respectively. The performance gets better when using two levels of blocking instead of one, and gets even slightly better when using three levels, although the major improvement comes from two levels of blocking. The best times are obtained for larger value of *npan*1 and smaller values of *npan*2 when using two levels of blocking compared to the

corresponding optimal values of *npan*1 when one panel is used. Similarly, when using three levels of blocking, the best times are obtained for even larger values of *npan*1 and even smaller values of *npan*3, whereas *npan*2 takes intermediate values.

nfront	*npiv*	Nb of panel levels				
		1	2		3	
		Time	Time	Ratio	Time	Ratio
512	128	$5.25*10^{-3}$	$5.16*10^{-3}$	0.98	$5.14*10^{-3}$	0.98
	256	$9.09*10^{-3}$	$8.55*10^{-3}$	0.94	$8.46*10^{-3}$	0.93
	512	$1.89*10^{-2}$	$1.73*10^{-2}$	0.92	$1.72*10^{-2}$	0.91
1000	256	$1.52*10^{-2}$	$1.42*10^{-2}$	0.93	$1.40*10^{-2}$	0.92
	512	$3.86*10^{-2}$	$3.55*10^{-2}$	0.92	$3.50*10^{-2}$	0.91
	1000	$1.00*10^{-1}$	$9.04*10^{-2}$	0.90	$8.98*10^{-2}$	0.90
2048	512	$5.08*10^{-2}$	$4.48*10^{-2}$	0.88	$4.42*10^{-2}$	0.87
	1024	$1.54*10^{-1}$	$1.35*10^{-1}$	0.88	$1.33*10^{-1}$	0.86
	2048	$4.50*10^{-1}$	$4.00*10^{-1}$	0.89	$3.96*10^{-1}$	0.88
10000	2048	$2.96*10^{+0}$	$2.62*10^{+0}$	0.89	$2.58*10^{+0}$	0.87
	4096	$1.03*10^{+1}$	$9.30*10^{+0}$	0.90	$9.14*10^{+0}$	0.89
	8192	$3.25*10^{+1}$	$3.04*10^{+1}$	0.94	$3.03*10^{+1}$	0.93
	10000	$4.40*10^{+1}$	$4.14*10^{+1}$	0.94	$4.12*10^{+1}$	0.94

(a) 1 core

nfront	*npiv*	Nb of panel levels				
		1	2		3	
		Time	Time	Ratio	Time	Ratio
512	128	$6.44*10^{-3}$	$6.44*10^{-3}$	1.00	$6.44*10^{-3}$	1.00
	256	$9.67*10^{-3}$	$9.60*10^{-3}$	0.99	$9.43*10^{-3}$	0.98
	512	$1.35*10^{-2}$	$1.24*10^{-2}$	0.92	$1.20*10^{-2}$	0.89
1000	256	$1.31*10^{-2}$	$9.91*10^{-3}$	0.76	$9.84*10^{-3}$	0.75
	512	$1.86*10^{-2}$	$1.43*10^{-2}$	0.77	$1.43*10^{-2}$	0.77
	1000	$3.10*10^{-2}$	$2.72*10^{-2}$	0.88	$2.50*10^{-2}$	0.81
2048	512	$3.17*10^{-2}$	$2.65*10^{-2}$	0.84	$2.64*10^{-2}$	0.83
	1024	$7.06*10^{-2}$	$5.61*10^{-2}$	0.79	$5.47*10^{-2}$	0.77
	2048	$1.76*10^{-1}$	$1.35*10^{-1}$	0.77	$1.29*10^{-1}$	0.73
10000	2048	$9.41*10^{-1}$	$4.63*10^{-1}$	0.49	$4.33*10^{-1}$	0.46
	4096	$2.49*10^{+0}$	$1.51*10^{+0}$	0.61	$1.45*10^{+0}$	0.58
	8192	$6.80*10^{+0}$	$4.69*10^{+0}$	0.69	$4.53*10^{+0}$	0.67
	10000	$8.29*10^{+0}$	$6.32*10^{+0}$	0.76	$6.16*10^{+0}$	0.74

(b) 8 cores

Table 5.2: Computation times of partial RL factorizations using 1, 2 and 3 levels of panel with static sizes. Ratios are comparisons between the times with several levels of panels and the times with one level of panel.

nfront	*npiv*	Nb of panel levels					
		1	2		3		
512	128	16	16	8	128	32	8
	256	32	32	8	64	16	8
	512	32	100	16	100	32	8
1000	256	16	64	16	64	16	10
	512	32	100	16	64	16	10
	1000	32	100	16	128	32	8
2048	512	32	100	16	100	32	16
	1024	32	100	16	100	32	8
	2048	100	128	16	256	32	8
10000	2048	64	128	16	128	32	8
	4096	100	256	16	256	32	4
	8192	128	256	16	256	64	8
	10000	128	256	16	256	32	4

(a) 1 core

nfront	*npiv*	Nb of panel levels					
		1	2		3		
512	128	128	128	128	128	128	128
	256	16	256	16	64	32	8
	512	16	64	10	128	100	8
1000	256	16	16	10	256	32	4
	512	16	32	8	32	8	8
	1000	32	32	8	100	32	8
2048	512	16	32	4	64	16	4
	1024	16	64	8	128	32	4
	2048	32	128	8	128	32	8
10000	2048	64	128	1	128	10	1
	4096	64	128	8	128	32	4
	8192	100	256	10	512	64	4
	10000	100	256	8	256	32	1

(b) 8 cores

Table 5.3: Panel sizes of partial RL factorizations using 1, 2 and 3 levels of panel with static sizes

The parameters *nthread*, *nfront* and *npiv* impact the multilevel approach in the following way.

1. For a given ratio $\frac{npiv}{nfront}$ and for a given *nthread*, the gain of the approach increases with *nfront* up to a certain optimal value before decreasing asymptotically.
2. The optimal value increases with *nthread*.
3. The gain increases as *nthread* becomes large and the ratio $\frac{npiv}{nfront}$ becomes small.

The explanation for all of this is that the approach takes advantage of the sum of the caches of the cores. For example, on full factorizations ($npiv = nfront$), we observe a 10% gain with

nfront = 1000 and 6% gain with *nfront* = 10000 in the sequential case, whereas these gains become, 19% and 26% in the multithreaded case respectively. Moreover, with *nthread* = 8 and *nfront* = 10000, we observe a 54% gain with *npiv* = 2048 and a 26% gain with *npiv* = 10000. All of this shows the potential of the multilevel approach on multifrontal factorizations.

5.5.2.2 Choice of block sizes

The next question concerns the automatic determination of the optimal panel sizes.

This problem has already drawn some attention in the past. Due to the high number of factors that influence the choice of optimal block sizes, the developers of the LAPACK library isolated this choice in a separate routine *ILAENV* (c.f.: *LAPACK Users' Guide: Third Edition*[18]) returning default block sizes (usually 32, 64 or 128) observed to lead to good performance on a reasonable number of test machines. Users are encouraged to tune these sizes according to their particular environment. Because relying on manual configurations is tedious, the developers of the *ATLAS* library [113] have chosen to rely on auto-tunning techniques to determine machine-dependent block sizes, together with other algorithmic variants.

Two interesting ideas are available in the literature. The first approach, described in [108], consists in finding the optimal panel size in the RL variant in order to minimize the amount of I/Os from primary to secondary memory. Equation(5.10) gives the optimal panel size, given the size M of primary memory and the number of row m and column n of the matrix being factorized.

$$Optimal\_Panel\_Size(M,m,n) = \begin{cases} max(M/n, sqrt(m)) & if M/3 \geq m \\ max(M/n, sqrt(M/3)) & if M/3 < m \end{cases} \tag{5.10}$$

The generalization of this approach, known as the *cache-aware* approach and described in [22], considers not only the case of two memory levels but the case of cascading memory hierarchies. However, only sequential execution is considered.

The second approach, known as *Parallel Cache Assignment* (*PCA*) [29], consists in considering the amalgamated L1 caches of a multicore processor as a unique L1 cache in order to speed-up panel factorizations in the multithreaded case. However, only the first cache level (L1) is considered and not the other cache hierarchies nor the other memory hierarchies.

The idea we propose, under the name **cache-friendly** approach, is a mix of the two previous ones. In our approach, the depth of the blocking level is the depth of the memory hierarchy on the test machines. The size of memory at each cache or memory level is the sum of the amalgamated caches or memories of all the cores under that level. Usually, L1 caches are private to each core. L2 caches are either private or shared by two cores. L3 caches are shared by all the cores of a given multicore processor (socket). The cache-friendly approach consists in applying Equation(5.10) at each level of blocking while considering the amalgamated memory related to that level, in order to get the optimal block sizes. Then, it consists in using sequential BLAS calls on matrix pieces located on memory locations private to the cores, while using multithreaded BLAS calls on matrix pieces located on shared caches or memories, as done in PCA. The cache-friendly approach is expected be even more applicable powerful on GPUs. On such architectures, cache hierarchies are more complex and communication costs between caches of the same (or different) level(s) are much costlier than they are on CPUs, as is the cost of data transfer between CPU and GPU. For now, however, in our current implementation and we do not worry about the cache affinity as much as we should because we call multithreaded BLAS rather than calling many separate sequential BLAS.

A comparison between the computation times previously shown and those of the runs executed using the cache-friendly approach shows that it is complicated to find a general rule that allows

us to choose the panel sizes. However, the cache-friendly approach is able to yield reasonably good block sizes. For example, on a matrix with $nfront = 10000$ and $npiv = 10000$, the cache-friendly approach leads to a time of 6.37 *seconds* with $npan1 = 209$, $npan2 = 26$ and $npan3 = 3$. This is to be compared with the best time of 6.15 *seconds* with $npan1 = 256$, $npan2 = 32$ and $npan3 = 1$, using exhaustive trials. One drawback of the approach is that it does not take into account the effect of block sizes on BLAS3 and BLAS2 efficiency. For example, we have observed that GER, which is a BLAS2 routine, is not parallelized in practice (at least, in the MKL library we use), so that it is better to use GEMM instead, which is its BLAS3 equivalent. We have thus replaced the calls to the GER routine by calls to the GEMM routine. BLAS libraries developers invest most of their efforts in the optimization of BLAS3 routines, GEMM particularly, over BLAS2 routines. Thus, the best results in the multi-panel approach are obtained when the innermost panel size is lower than that predicted by Formula 5.10.

We compared our performance with that of the recursive factorization algorithm [108], meant to be theoretically optimal in minimizing the amount of I/O. We have decided not to use the recursive approach in our application as it presents some negative side effects in distributed-memory factorizations, as will be shown in the next section. Its computation time for the full factorization of a front with $nfront = 10000$ and $nproc = 8$ is 6.45 *seconds* (with the best inner-panel size threshold) while our three levels cache approach required 6.37*seconds*. Also, the performance of the recursive algorithms is much less stable than that of the multipanel approach. The reason is probably that much more work is done in TRSM than in GEMM in the recursive approach.

We also compared our results with the GETRF routine of MKL. We have similar performance on large matrices, but slightly worse on small matrices.

5.5.2.3 BLAS performance on RL-like vs LL-like patterns

In order to understand the influence of the multi-panel approach on RL and LL computation schemes, we have run a benchmark of the GEMM routine. Table 5.4 shows the Gflops/s rate on matrices $a[m,k]$, $b[k,n]$ and $c[m,n]$ ($c \leftarrow c - a \times b$) on the `crunch` and `hyperion` machines, the sizes m, n and k correspond to the parameters of the GEMM routine illustrated in Figure 5.13. These sizes are chosen to represent typical master panel and update block sizes.

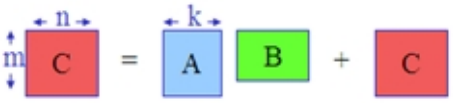


Figure 5.13: Representation of the m, n and k parameter of the GEMM routine

One point worth noticing is that the larger the panels (k), the higher the GEMM efficiency. On `crunch` (Intel Sandy Bridge) with a large panel size (512), LL performance is only slightly worse than RL with 1 core, but much significantly worse with 8 cores (77.75 GFlops/s vs. 91.59 GFlops/s). This result is counter intuitive as LL does more (and cheaper) read operations and less write operations.

We also observe that the results are very much machine-dependent, so that it is difficult to conclude on which variant is the best. On `hyperion`, the LL performance is slightly worse than RL, except on 8 cores with a panel size of 32, where LL is much better! This might also be due to different versions of the MKL library.

In [81], the authors observe that LL is better than RL in `SuperLU_MT`. However, one difference with our work is that `SuperLU_MT` relies only on sequential BLAS while we use multithreaded

nbcores	pattern	m	n	k	GFlops/s	
					`crunch`	`hyperion`
1	LL	32	10000	1000	10.68	8.72
	RL	1000	10000	32	11.38	9.52
	LL	512	10000	1000	15.40	11.86
	RL	1000	10000	512	15.98	12.12
8	LL	32	10000	1000	29.49	65.98
	RL	1000	10000	32	30.91	29.91
	LL	512	10000	1000	77.75	80.89
	RL	1000	10000	512	91.59	81.27

Table 5.4: Performance of GEMM in Gflops/s on `crunch` and `hyperion`, both sequential and multithreaded, on both RL-like and LL-like configurations. m, n and k are the parameters of GEMM describing the size of the updated and updating matrices. The *hwloc-bind* utility is used to bind all threads on the same socket (*hwloc-bind socket:0 ./gemm*).

BLAS.

5.5.2.4 Experimental results on sparse matrices

Table 5.5 shows the shared-memory factorization times (in seconds) of some matrices from acoustic physical problems [1], using 8 cores of `crunch` and using the RL variant only. The choice of the panel sizes follows Equation(5.10). Moreover, in order to be able to compare the effects of multiple levels of blocking, we have run the executions with no tree parallelism, which means without the work presented in the first part of the thesis.

Matrix	one level of blocking	two levels of blocking
acoustic_pml_DF	428.94	430.20
acoustic_RC1	1242.43	1241.81
acoustic_RC3	634.66	622.45
TM_bypass	2311.29	1960.16
TM_inlet	7571.52	5950.51
VA_pl (*)	4179.85	4136.35
VA_RC4	105.90	103.84

Table 5.5: Times (in seconds) using the 2-level panel approach on FFT matrices. (*) Much numerical pivoting inducing large computation times.

We see that huge improvements are obtained on matrices TM_bypass and TM_inlet. After analysis of the front sizes, we observed that these matrices have large fronts ($nfront \approx 30000$), on which the 2-level panel approach has more effects. On matrices with only small fronts (typically, acoustic_pml_DF, acoustic_RC1 and acoustic_RC3), very little improvement is observed. In order to improve the performance on such matrices, tree parallelism should be used (See Part I of the thesis).

[1] provided by the FFT company

5.5.3 Multipanel factorization in hybrid-memory

From the previous observations and from the work done on shared-memory dense linear algebra, the theoretically ideal solution to overcome LL performance issues would be to use a *recursive blocking* scheme [108]. However, doing this in our environment would be disastrous as shown in the simulated execution of Figure 5.14. The drawback of this computation scheme is that, at the first (outermost) level of recursion, the update of the second block by the first one would take too much time on the master, making workers become idle for too long (red rectangle). In practice, however, it is possible to use recursive techniques only inside panels and not on the whole matrix. Although one can expect them to be more portable (performance-wise) as they are *cache oblivious*, in the sense that they do not have to bother with the choice of block sizes, one can also expect them to be less efficient than multiple embedded panels, where a careful choice of the block sizes is done.

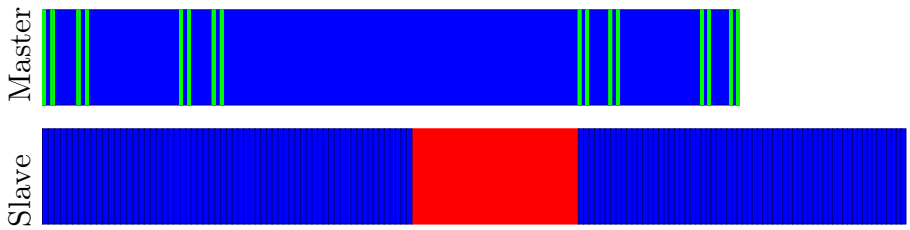


Figure 5.14: Effect of the recursive blocking scheme applied on the master on the total execution time. A significant idle period appears in the workers while the master updates its outermost trailing matrix.

We have implemented the multipanel approach in a distributed-memory environment. We stopped at two levels of blocking because of the complexity required to handle pivoting issues on higher levels. We denote by *npan1* and *npan2* the sizes of the innermost and outermost panels, respectively. We use the more efficient RL variant on internal smaller panels and compare in the following the RL and LL approach on external panels both with *eqRows* to be able to compare their resulting execution times.

npan1 and *npan2* directly impact the performance of master and workers. Regarding the workers, a compromise must be found. For the same amount of data sent by the master, it is more efficient for network communications to send or make computations on a few large blocks than many small ones. Similarly, for a given amount of data used by workers for their updates, it is more efficient for BLAS3 computations to make computations on a few large blocks than many small ones. Thus, sending large external panels may improve the overall performance. However, as explained in Section 5.3.4, sending small panels is theoretically better for pipelining. Hence, the choice of the level of panel to be sent (whether internal or external) depends on the relative size of *npan* compared to *npiv*. We could also desynchronize the computation of panels from their transmission, which would require dedicated mechanisms, like a communication thread, to handle communications separately from computations.

Table 5.6 shows the influence of *npan1* and *npan2* on the factorization time on `hyperion` (and `ada`) of a front with $nfront = 64000$ (10^5 on `ada`), *npiv* in *eqRows*, $nproc = 8$ (64 on `ada`), and 8 cores per process.

The best panel size combination obtained on `hyperion` (and `ada`) is $npan1 = 128$ (64 on `ada`) and $npan2 = 64$ (32 on `ada`). We see that optimal results are obtained when sending external panels instead of internal ones. Moreover, the application of the multi-panel approach in the case of distributed-memory factorizations seems to have quite a significant impact on performance.

Furthermore, we see that it is hard to choose the best panel size. However, for a given machine, we can deduce good panel sizes by running a benchmark such as that in Table 5.6, and by using the best sizes obtained by default on that machine (as done classically in the linear algebra packages).

E/I	*npan1*	*npan2*	`hyperion`		`ada`		E/I	*npan1*	*npan2*	`hyperion`		`ada`	
			RL	LL	RL	LL				RL	LL	RL	LL
I	32	32	173	203	21.8	22.0	E	32	32	203	204	21.9	22.2
		64	202	203	21.7	21.9			64	101	100	**20.5**	**19.8**
		128	202	203	22.0	22.1			128	97	97	21.0	21.4
		256	204	145	21.8	21.5			256	125	118	24.7	25.0
		512	145	203	22.6	21.8			512	129	129	30.6	30.8
	64	64		120	21.3	**21.1**		64	64		120	21.7	21.8
		128	111	109	21.2	21.4			128	**97**	**93**	22.4	22.1
		256	118	107	21.2	21.4			256	124	118	25.3	25.7
		512	**102**	101	**21.2**	21.4			512	136	147	32.8	33.8
	128	128	117	104	30.9	31.0		128	128	119	110	31.2	31.2
		256	109	**99**	30.6	31.2			256	132	129	33.1	33.0
		512	107	101	31.2	30.8			512	131	121	36.7	36.7
	256	256	135	147	51.1	51.6		256	256	135	129	51.1	51.5
		512	151	133	51.4	51.8			512	149	137	54.7	55.3
	512	512	205	209		92.0		512	512	222	209		92.5

Table 5.6: Execution time (in seconds) of factorization of a front with $nfront = 64000$ on `hyperion` and $nfront = 10^5$ on `ada`, $nproc = 8$, and varying *npan1* and *npan2*, with external (E) or internal (I) panel send. *eqRows* is used in order to have identical computation costs between RL and LL. The best results are highlighted in each column, corresponding to one variant (RL or LL), one machine (`hyperion` or `ada`) and one panel sending strategy (internal or external).

5.6 Conclusion

In this chapter, we have studied the inner characteristics of the algorithms we use in our distributed-memory dense partial LU factorizations. After exploring the theoretical behaviour of the RL and LL algorithms, we have determined the most appropriate way and the ideal configurations to take advantage of them, and have shown the superiority of the LL scheme over the RL one. Then we have targeted the practical issues that limits them on large numbers of processors. Communications from the master process to the worker processes was the first bottleneck. We have shown how typical broadcast communications could be exploited to overcome the previous limit imposed by the master network bandwidth, by using instead, and by exploiting the overall network bandwidth. Moreover, the efficiency of the computations on the master was the other issue that slows down the whole factorization, which seemed to be critical in a multithreaded environment, especially in the LL approach. We then proposed the use of multilevel blocking schemes to enhance panel computations. The result was then to free the LL approach from the master computation barrier, thus restoring back its advantage over RL.

The remaining main bottleneck discussed in this chapter is the dependence of the efficiency of our factorizations over the shape of the fronts and the number of processes used. This is what we will target in the next chapter.

Chapter 6

Limitations of splitting to improve the performance on multifrontal chains

6.1 Introduction

The model developed in Chapter 5 showed that front factorizations are efficient when the ratio $\frac{npiv}{nfront}$ respects (roughly) the *eqRows* and *eqFlops* properties for RL and LL, respectively. Hereafter, unless explicitly stated, the term *optimal ratio* will refer to *eqRows*, when dealing with RL, and *eqFlops*, when dealing with LL. The goal of the present chapter is to extend the work of the previous chapter, both by modifying fronts with arbitrary shape and by consequently adapting the mapping of processes. Our aim here is to preserve performance independently from the characteristics of the targeted problems.

In practice, the true optimal ratios might differ slightly from those predicted by the model. Moreover, the model showed that any divergence from the predicted optimal values could rapidly decrease the efficiency. Unfortunately, the multifrontal method generates trees of frontal matrices whose shape usually do not lead to ratios having the desired values as the number of processes mapped on a given front directly influences its optimal ratio. However, the construction of a multifrontal tree does not depend *a priori* on the number of processes used in the factorization, but only depends on the original sparse matrix and the chosen ordering. Fortunately, multifrontal trees are not rigid entities. They can be reshaped, using mainly two standard operations known as a *amalgamation* and *splitting* that will be used in the present chapter to improve our algorithms.

On the one hand, *amalgamation* consists in merging a child and its parent. The amalgamated front will contain more fully-summed rows than will the parent front for a same contribution block size. We could use *amalgamation* to increase the aforementioned ratio when necessary, in order to give more work to the master process. *Amalgamation* has the advantage of generating larger fronts, which increases the efficiency of the factorization, sometimes at the cost of extra fill-in that increases computation and memory.

On the other hand, *splitting* consists in cutting a single front into two or more fronts, resulting in a chain of fronts in the multifrontal tree referred to as a *multifrontal chain* or a *split chain* in the following (See Figure 6.1). Its characteristic is that the Schur complement of a child front, which includes fully summed variables, is considered as a new parent front, the Schur complement of the last (top) front being that of the original front.

With *splitting* as a possible tool for reshaping multifrontal trees, we now turn to the following crucial question: which mapping of processes is best to use in conjunction with this tool in order to improve performance? Indeed, in contrast to the 1D cyclic and 2D block cyclic distributions, the 1D acyclic distribution (Section C.3) needs some remapping strategy in order to be efficient. In this chapter, we will propose a remapping strategy, which we will refer to as the *restart mapping*. It relies on a compromise between improving computation at the price of remapping communications.

The present chapter is organized as follows:

In Section 6.2, we propose splitting algorithms. We show the results obtained when applying these algorithms using both the RL and LL variants.

In Section 6.3, we give a detailed description of our *restart mapping*. In particular, at the end of that section, we will discuss the limitations of the *restart mapping* and we will point out three important bottlenecks, each of which will be the subject of a specifically dedicated chapter (namely, Chapters 7, 8, and 9).

Let us remind the reader that the main notations used in this chapter are described in Table C.1, C.2 and C.3.

6.2 Splitting

The 1D acyclic pipelined factorization algorithm (See Algorithm C.1) will behave differently depending on the value of $\frac{npiv}{nfront}$. When this ratio is lower, the critical path is on the workers and the master wastes time, whereas the workers are always active. On the contrary, when it is higher, then the critical path is on the master and it is now the workers that "waste" time. The key point to consider here is that one unit of CPU time lost on the workers is equivalent to *nworker* units lost on the master! To target this issue, we split fronts with a higher than optimal ratio to make the changed ratios as near to the optimum as possible on each front of the resulting chain (See Figure 6.1).

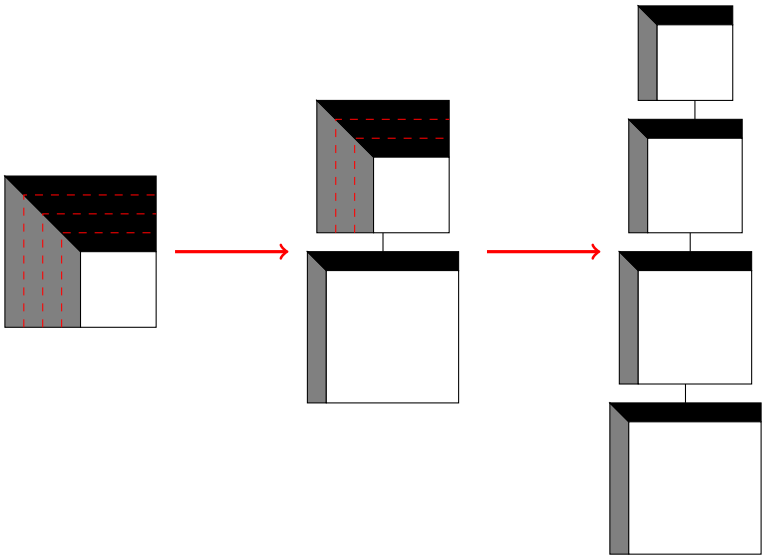


Figure 6.1: Splitting of a front (left) into a split chain (right) to be processed bottom-up (L and U factors in gray and black respectively, Schur complements in white).

6.2.1 Splitting algorithms

The *static splitting algorithm* (See Algorithm 6.1) starts from an initial front of size $nfront_{init}$ with $npiv_{init}$ pivots and a given fixed number, *nproc*, of available processes. It computes the optimum number $npiv_1$ of pivots. The initial front is then split into a first (bottom) front of the split chain of size $nfront_{init}$ with $npiv_1$ pivots, and a second (top) remaining front of size $nfront_{init} - npiv_1$ and with $npiv_{init} - npiv_1$ pivots. This latter front is recursively being as an initial front and the same cut operation is applied as long as the ratio relative to the considered front is higher than the optimal one for this front.

Input: $nfront_{init}, npiv_{init}, nproc$
Output: $nfront[], npiv[]$
1: $nfront[1] \leftarrow nfront_{init}$
2: $npiv[1] \leftarrow npiv_{init}$
3: $i \leftarrow 1$
4: **while** $\frac{npiv[i]}{nfront[i]} > optimal\_ratio(nfront[i], npiv[i], nproc)$ **do**
5: {*optimal_ratio* being *eqRows* for RL and *eqFlops* for LL }
6: $npiv_i \leftarrow npiv_{opt}(nfront[i], npiv[i], nproc)]$
7: $nfront[i+1] \leftarrow nfront[i] - npiv_i$
8: $npiv[i+1] \leftarrow npiv[i] - npiv_i$
9: $npiv[i] \leftarrow npiv_i$
10: $i \leftarrow i + 1$
11: **end while**

Algorithm 6.1: Static splitting algorithm.

The splitting algorithm as presented in this static variant depends on *nproc*, the number of available processes, and particularly on the assumption that it remains unchanged during the whole factorization of the chain. This assumption remains true as long as only a single separate split chain is considered or as long as a static process mapping and scheduling is considered on the initial multifrontal tree (before splitting). Obviously, the performance of the factorization of chains could be negatively impacted if these split chains, which had been initially crafted for a given number of processes, were to be treated by a different number of processes. Nevertheless, the asynchronous and dynamic nature of our environment makes it possible for the number of available processes to vary between one front and another of a split chain. Processes may be assigned to other fronts in other subtrees, whereas newly available processes may be assigned to future fronts in the chain. Consequently, the way a front will be split into a chain of fronts is strongly influenced by the evolution of the number of available processes. Therefore, in such a dynamic environment, the length of a chain and the shape of its fronts should not be determined *a priori*. A *dynamic splitting algorithm* should be considered instead. This algorithm consists in determining the shape of the next front to be computed, factorizing it first, then, repeating these steps again with the new updated value of *nproc*. It is a generalization of the static splitting algorithm, as it gives the same splitting when *nproc* remains unchanged.

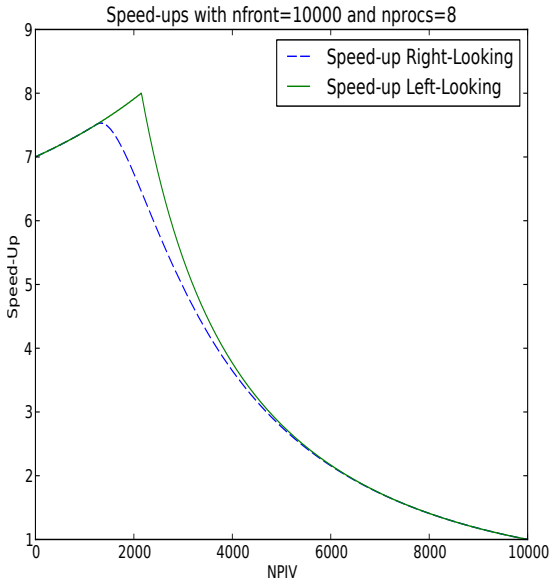
6.2.2 Results of splitting

Figures 6.2a and 6.2b show the simulated time and speed-up, respectively, on the `simul` test case with varying *npiv* when the splitting algorithm is used. The dashed-blue/solid-green curves show the behaviour of the RL/LL variant with *eqRows*/*eqFlops* applied as load balance criteria for the splitting of each front.

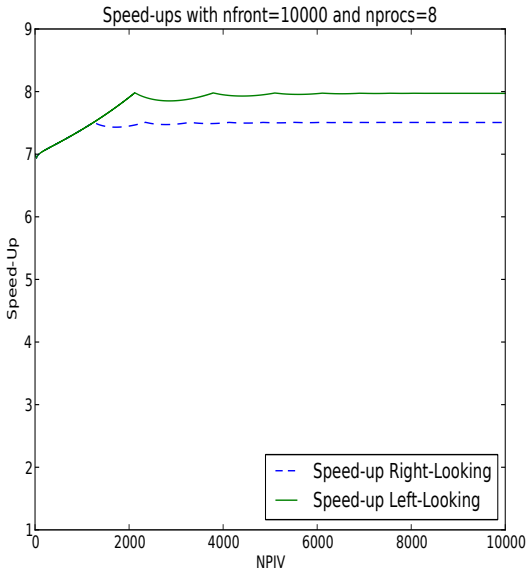
For both RL and LL, the speed-up grows rapidly to its optimal value, exactly as it would have done if splitting were not applied. The reason is that, as mentioned earlier, splitting is

applied only on a front whose $\frac{npiv}{nfront}$ ratio is higher than the optimal one. Then, the speed-up locally slightly decreases from the optimum and slightly re-increases to the optimum, describing a saw-tooth curve (examples of two local optima for LL are $npiv \approx 2200$ and $npiv \approx 3800$). Such a behaviour can be explained by the fact that, when the optimal ratio is reached on each front of the chain, the same optimal speed-up is reached on each front, and hence, on the whole chain. Likewise, local decreases/re-increases from/to the maximal speed-up could be explained by the fact that, each time the last front of the chain needs to be split, because of an increase in *npiv* value, the ratio of this new front is different from the optimal one, which thus induces some performance loss on the master's side, which in turn impacts the global speed-up.

Compared to the case of Figure 6.2a when no splitting is applied, it now seems that splitting makes the factorization much less sensitive to the value of the $\frac{npiv}{nfront}$ ratio and brings a valuable speed-up independent of it. In this model, however, we assume an infinite network bandwidth and do not take into account communications between fronts in the chain, which is an unrealistic hypothesis.



(a) Speed-ups without splitting

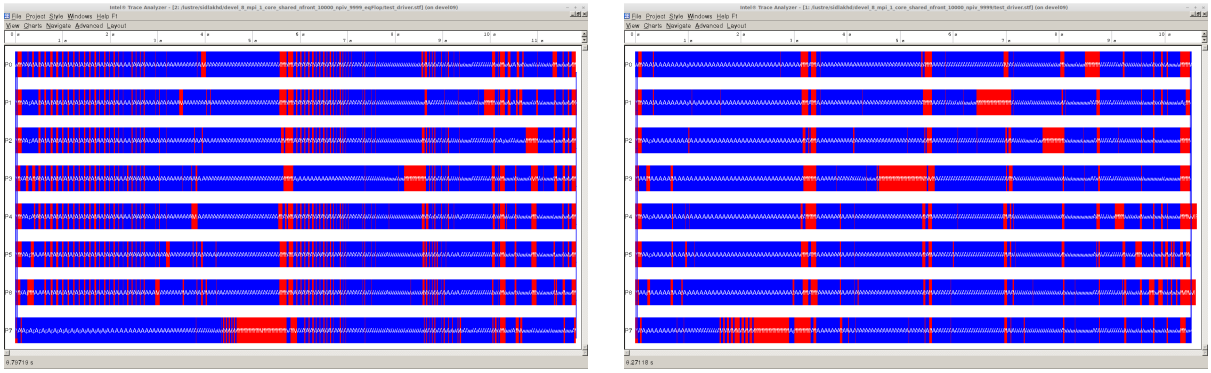


(b) Speed-ups with splitting

Figure 6.2: Splitting on a matrix with *nfront* =10000 and varying *npiv* with 7 workers and 1 master. LL (resp. RL) uses *eqFlops* (resp. *eqRows*).

Figure 6.3 shows the Gantt-chart of the complete factorization of a front with *nfront* = *npiv* = 10000 and *nproc* = 8 on the `devel` machine (See Section C.5). The *static* splitting algorithm is applied with the RL variant with both *eqFlops* (Figure 6.3a) and *eqRows* (Figure 6.3b). A comparison of the two sub-figures shows that the overall execution time is, as expected, better when *eqRows* is applied (10.5 *seconds*) instead of *eqFlops* (11.75 *seconds*). This experiment however reveals an important flaw in the use of splitting as it tends to lead to many assembly phases (large continuous red parts common to all processes in the figures). Moreover, the ratio between the cost of assembly and the cost of factorization tends to increase in hybrid shared-distributed memory environments, as the factorization time shrinks with an increase in the number of cores per process while the assembly time remains unchanged. Typically, after splitting a front with *nfront* = 100000 and *npiv* = 100000 (which would correspond to a root in a multifrontal tree) and with *nproc* = 64, the ratio between the assembly time of the second front of the chain and

its factorization time is only of 2% with 1 *core* per process but increases to 10% with 8 *cores* per process. Furthermore, the optimal *npiv* for a given *nfront* decreases when *nproc* increases. Hence, when the number of processes increases, the split chain becomes longer, incurring an even larger total assembly cost. Nevertheless, we note that the use of *eqFlops* leads to shorter chains, which makes the LL variant even more attractive over the RL one.



(a) *eqFlops* (11.75*seconds*) (b) *eqRows* (10.5*seconds*)

Figure 6.3: Application of the static splitting algorithm on the full factorization of a matrix with *nfront* = 10000 and *nproc* = 8 on `devel`, with RL both with *eqFlops* and *eqRows*

6.3 (Re)Mapping

6.3.1 Remapping strategies

In the LU factorization of a frontal matrix, it is worth adding computational resources until the cost of communications overtakes the benefit of the additional computational power. The point at which it happens depends on how the matrix is distributed over the processes. It is reached more quickly with 1D acyclic distribution than with 1D cyclic or 2D block cyclic distributions. We can, however, solve this issue in the 1D acyclic case by using careful process mapping and remapping strategies.

The mapping we have used so far, which we will denote by *standard mapping*, consists in remapping all the available processes from each child front C to each parent front P of a chain. Changing the process mapping from one front to another comes at the price of an assembly operation. As this operation is potentially costly for large *nproc*, the use of such a mapping by itself will not scale when *nproc* is large.

From the results described previously in Section 6.2, it is clear that we must rethink this mapping of processes. The *communication-free mapping* (CFM) consists instead in avoiding any assembly operation by simply avoiding any remapping. However, the obvious drawback of this approach is that one process (the master of the child front, MC) is lost during the transition.

A compromise must thus be found between losing some computational resources, on the one hand, and increasing the number of assembly communications, on the other hand. This is the purpose of *restart mapping*, which combines the two previous approaches, trying as much as possible to retain their advantages and to eliminate their drawbacks.

Figure 6.4 shows the typical behaviour of standard, communication-free and restart mappings. It shows the gain offered by the restart mapping. The restart mapping consists in using a communication-free mapping until the cost of the communication caused by a remapping onto a larger number of processes is more than compensated for by the computational benefit of

the extra processes, in which case a standard mapping is applied. We denote this remapping operation as a *restart operation*.

Because it keeps the mapping of the workers of C unchanged in P, CFM necessarily implies that *eqRows* is used in P, and likewise in its ancestors. Indeed, applying *eqFlops* on P would make the MP (master of P) have more rows than the workers of P, which would *a fortiori* make MP have more rows in C than the other workers in C (MP being the first worker in C). As MP will have more work than the workers in C, MP would make them wait for it to start working in P, which is not acceptable for performance. Similarly, this holds for all the workers in C, which potentially will be masters of ancestor fronts in the split chain. However, *eqFlops* can be, and still is, applied to the first (bottom) front of each subchain (namely, C) after each restart operation. This is in order to make better use of the master process on this particular front, to make more computations on it and thus to reduce the length of the split chain and the total number of assembly operations.

The question that now arises is to determine which strategy should be adopted at each new front in the chain, knowing that the shape of the rest of this chain will be closely related to this mapping decision. To fix the ideas, let us state the restart mapping problem as follows: given a front F in the chain,

Case 1: should one lose a process for the parent node (CFM)?

Case 2: or would it be worthwhile to remap some or all of the processes, including the previously lost ones in the successive previous CFMs?

We study in this section the restart problem.

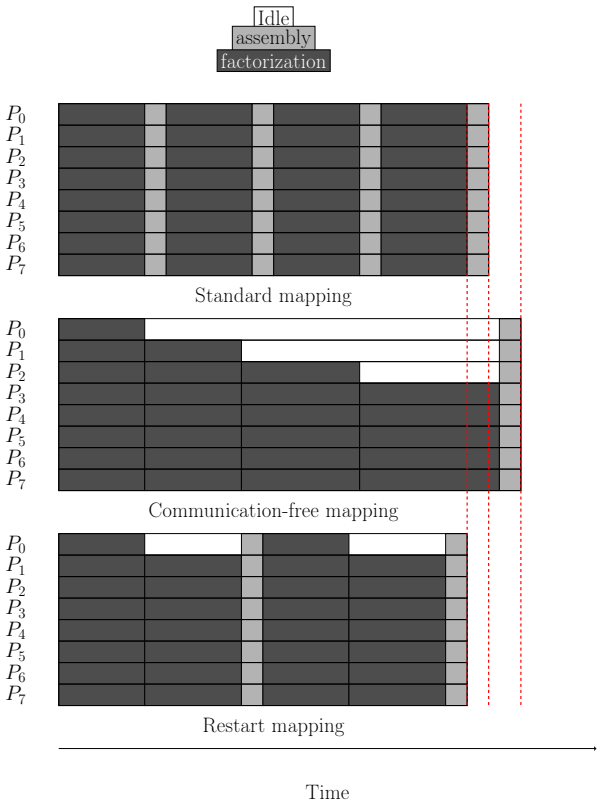


Figure 6.4: Standard, communication-free and restart mappings.

The decision of restarting made at each front of the chain impacts not only the structure of the parent front but also the whole remaining chain. Whether to lose a process or to remap is

thus a combinatorial problem, which depends on the model of communication and computation, which we measured experimentally.

6.3.2 Remapping model

In order to find algorithms choosing the appropriate remapping strategy (standard, communication-free mapping, restart) at each front in a chain, the objective of this section is to first define the computation and communication models, given the per-process bandwidth γ and the per-process GFlops/s rate α.

An illustration for each of the time of assembly, factorization, update and pipeline is given in Figure 6.5, which should ease the understanding of the formulas below.

Some processes involved in P must receive their rows from C, as they were either working on a different set of rows in C or not involved in the factorization of C. Because we use *eqRows* (see Section 6.3.1), the maximal total number of rows some processes have to receive is $\frac{nfront_P}{nproc_P}$ rows of P. The time of assembly communication can thus be estimated by:

$$Time_{Assembly} = (\frac{nfront_P^2}{nproc_P})/\gamma + TimeLatency \tag{6.1}$$

Moreover, the total time required for a front to be factorized not only depends on the time needed by one worker to perform its computations but also depends on the broadcast pipeline start and end times. Consequently, the total computation time of a front is given as follows:

$$Time_{Factorization} = Time_{Update}(\frac{ncb}{nproc-1}, nfront - \frac{ncb}{nproc-1}, npiv) + Time_{Pipeline}(nfront, npan, nproc, w) \tag{6.2}$$

with

$$Time_{Update}(m,n,k) = (2\,mnk + mk^2)/\alpha$$

and

$$Time_{Pipeline}(nfront, npan, nproc, w) = \frac{nfront * npan}{(\frac{\gamma}{w * log_w(nproc)})}$$

In these equations, w is the width of the broadcast tree, $Time_{Update}$ is the time necessary for a worker to complete all its computations (see also Equation(5.3)) and $Time_{Pipeline}$ is the time necessary for the first panel, transmitted from the master, to reach the last worker in the broadcast tree. These terms are explained in Section 5.3.1 and Section 5.4.1. We will use this model in the rest of this chapter.

6.3.3 Restart algorithm

Algorithm 6.2 is obtained by adapting the static splitting algorithm from the case of the simple standard mapping to the case of the more complicated restart mapping.

In this algorithm, *criterion* is a Boolean function corresponding to the two cases discussed at the end of section 6.3.1. It returns *True* if it judges that the communication-free mapping should be applied locally to the next front to be split (case 1) or *False* if it judges that a restart operation is necessary (case 2). *criterion* is meant to be the implementation of the strategy we wish to use. In other words, if *criterion* always returns *False*, that will correspond to the standard mapping whereas *criterion* always returning *True* will correspond to the communication-free mapping. Finally, the mixed case will correspond to the restart mapping which we consider next.

A natural way to solve the restart problem is to apply dynamic programming techniques. The computation of the optimal decision between case 1 and case 2 with the parameters $nfront_{init}$,

Input: $nfront_{init}, npiv_{init}, nproc_{init}, nthread, criterion$
Output: $nfront[], npiv[]$

1: $nfront[1] \leftarrow nfront_{init}$
2: $npiv[1] \leftarrow min(max(1, \frac{nfront_{init}}{nproc_{init}}), npiv_{init})$
3: $nproc[1] \leftarrow nproc_{init}$
4: $time \leftarrow 0$
5: $nass \leftarrow npiv[1]$
6: $i \leftarrow 1$
7: **while** $nass < npiv_{init}$ **do**
8: {still fully summed variables left}
9: **if** $nfront[i] \leq 1$ **then**
10: **return** $time + Time_{SeqFactorization}(nfront[i], npiv[i])$
11: **end if**
12: $nfront[i+1] \leftarrow nfront[i] - npiv[i]$
13: **if** $(nproc[i] > 2)$ **and criterion**$(nfront, npiv, nproc, nthread, i)$ **then**
14: {CFM (case 1): lose one process}
15: $nproc[i+1] \leftarrow nproc[i] - 1$
16: **else**
17: {Restart (case 2): use all available processes}
18: $nproc[i+1] \leftarrow nproc_{init}$
19: $time \leftarrow time + Time_{Assembly}(nfront[i+1], nproc[i+1])$
20: **end if**
21: $npiv[i+1] \leftarrow min(max(1, \frac{nfront[i+1]}{nproc[i+1]}), npiv_{init} - nass)$
22: $time \leftarrow time + Time_{Factorization}(nfront[i+1], npiv[i+1], nproc[i+1], nthread)$
23: $nass \leftarrow nass + npiv[i+1]$
24: $i \leftarrow i + 1$
25: **end while**
26: **return** $time$

Algorithm 6.2: Splitting algorithm with restart mapping.

$npiv_{init}$ and $nproc_{init}$ requires computing the cost (execution time) of all the sub-configurations with:

$$\begin{cases} 0 < nfront \leq nfront_{init} \\ 0 < npiv \leq, npiv_{init} \\ 2 \leq nproc \leq nproc_{init} \end{cases}$$

However, this would lead to the computation of the optimal solution for all possible fronts smaller than our initial front size $nfront_{init}$, for all possible numbers of pivots smaller than $npiv_{init}$ and for all numbers of processes smaller than $nproc_{init}$. Doing so would be much too costly to compute ($O(nfront_{init} \times npiv_{init} \times nproc_{init})$).

Relying on the observation that only two choices are possible at each new front of the chain, a recursive solution could seem more suitable. However, a purely recursive solution requires too much time to compute.

It is possible to work out a very efficient way to find the optimal mapping strategy on the whole chain. This can be done if we take some care in detecting already computed solutions and if we avoid performing the same recursion several times by checking, during the recursions, whether a solution has already been computed for a given configuration. This improved recursive algorithm will be particularly useful for long split chains. However, the following different but cheaper strategy produces the same optimal solution in all the simulations we have run.

Indeed, after experimenting with several heuristics, we have retained a *greedy heuristic* which consists in choosing between case 1 and case 2 by locally considering pairs of child and parent fronts rather than considering the whole chain. As this *greedy heuristic* gave in simulations the same (optimal) mapping as the one given by the improved recursive algorithm for all simulated cases, we have implemented and used it in practice.

6.3.4 Theoretical results

Table 6.1 shows some simulation results comparing the standard, communication-free and restart mappings, applied to an initial front of size $nfront = npiv = 100000$, on different simulated machines, and using different numbers of MPI processes and threads per process.

Firstly, unsurprisingly, when the computational power is reduced (fewer processes), the Standard mapping is better than the Communication-free mapping; whereas when the computational power is large (many processes), the Communication-free mapping becomes better.

Secondly, when increasing the computational speed of the processes, by increasing the number of threads in them, while the network bandwidth remains unchanged, the limit on the number of processes where the Communication-free mapping becomes better than the Standard mapping decreases (see the dashed lines in Table 6.1).

Thirdly, we can see the potential of a restarting strategy. Indeed, it becomes advantageous to use such a strategy before reaching the aforementioned limit on the number of processes. Although we do not report the results here, we also compared the greedy heuristic with dynamic programming for many cases and noticed that it produces mappings resulting in quasi optimal performance (except in very special cases with very large numbers of processes where we reach the limit of the 1D approach, see below).

Fourthly, when the Communication-free mapping becomes better than the Standard mapping, we reach the limit of the 1D factorization approach. The optimal restarting strategy yields mappings very similar to the Communication-free mapping, and the greedy heuristic starts producing mappings resulting in worse performance (see results with $nthread = 32$, $nproc = 2048$ on Machine 2, for example).

Finally, we have not reported timing results for constructing the mapping itself, but we have observed that finding the optimal mapping using dynamic programming is computationally intensive and very resource consuming, as the dynamic programming and recursive techniques require a lot of time and memory, particularly when applied to large numbers of processes and, to a lesser extent, when applied to large fronts.

Thus (assuming the model is good enough), the right approach to choosing the best mapping will consist in choosing the mapping by comparing the simulated results for the Standard mapping, Communication-free mapping and the Greedy heuristic.

6.3.5 Experimental results and bottlenecks of the approach

After having implemented the restarting strategy (*greedy heuristic*), we have run experiments both on `hyperion` and `ada`. We see in Table 6.2 the benefits of the restarting strategy over the standard one. This table presents the execution times with $nfront = 64000$ on `hyperion` and $nfront = 10^5$ on `ada` respectively, with $npiv = nfront$ (full Factorization) and $nproc = 64$ using $nthread = 8$, on both `hyperion` and `ada`.

Unfortunately, the GFlops/s rate per core with 512 ($= nproc \times nthread$) cores is only 3.38 and 6.85 on `hyperion` and `ada` respectively. These results are worse than the theoretical results (3.38 instead of 5.77, and 6.85 instead of 9.79), mainly due to large discrepancies between the theoretical peak performance used and the achievable peak.

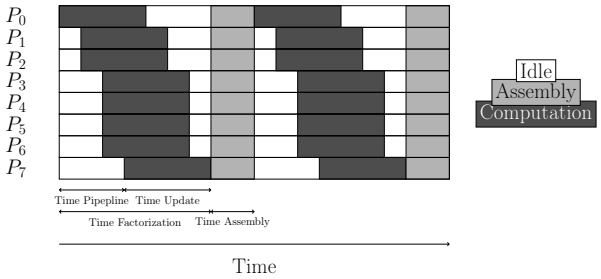


Figure 6.5: Bottleneck of our 1D pipelined approach on split chains.

The analysis of the results obtained so far, when scaling both in terms of front size and in terms of number of threads and processes, show new bottlenecks of our approach on split chains. Figure 6.5 illustrates the three most important bottlenecks, each of which is briefly described below and each of which will be the subject of a specific chapter.

The first bottleneck is the cost of assembly operations (except for CFM). This cost is illustrated in Figure 6.5 by the width of the light gray bands. Even though the restart mapping diminishes the need for assemblies compared to the standard mapping, these are usually still needed. To target this issue, we will study in Chapter 7 the influence of the relative position of the processes between child and parent fronts over the position of the rows they are mapped on, in order to minimize the amount of data each process has to send/receive, and thus, to decrease the cost of assembly operations.

The second bottleneck is the cost of pipeline starting and ending at each front of a chain. This cost is illustrated in Figure 6.5 by the white areas which represent process idle periods. Due to the use of *IBcast*, with deeper and deeper broadcast trees as the number of processes increases, pipeline starting and ending becomes more and more costly, and may even become more costly than the effective per-process computation time on each front. It is important to point out that the pipeline is broken at each front. Therefore, we are not paying the price of pipeline starting and ending only once during the computation of the whole chain. Instead, as shown in Figure 6.5, we pay this price repeatedly at each front. Indeed, assembly operations can be viewed as some kind of smooth but inevitable synchronization, which tends to impact pipelines. Moreover, the use of *IBcast* in a dynamic asynchronous environment may introduce deadlocks. This is the reason why synchronization barriers have been introduced between each consecutive pair of fronts in a chain. The side effect is to break up the pipelines. Consequently, this issue persists even when the restart mapping is applied. Indeed, although the restart mapping normally reduces the number of light gray areas in Figure 6.5, it still needs to apply synchronization barriers. Therefore, this problem persists.

The next step is to avoid such costly synchronizations. However, trying to remove them opens Pandora's box. This will require a complete rethinking of the synchronization strategies in asynchronous environments. This issue will be tackled in Chapter 8 in which we will try to identify the minimal synchronization mechanisms needed in order both to guarantee a deadlock-free code and to avoid breaking up unnecessarily pipelines at each front of a chain.

The third bottleneck is related to some technical issues. In particular, our asynchronous communications use the `MPI_PACK` and the `MPI_UNPACK` routines, so that the time for serial copy operations is not negligible when many threads are being used. Those copies could be parallelized or even avoided. This issue will be discussed in the first part of Chapter 9.

We believe that significantly better speed-ups will be obtained after these issues are resolved.

Machine	*nthread*	*nproc*	Standard		Communication-free		Restart (Greedy)	
			Time	Speed-Up	Time	Speed-Up	Time	Speed-Up
Machine 1	1	8	14796.5	7.51	20548.0	5.41	14796.5	7.51
		1024	168.5	659	188.6	589	160.5	692
		2048	182.9	607	137.9	805	157.5	706
		4096	308.3	360	165.5	671	214.1	519
		8192	633.1	175	292.3	380	292.3	380
	8	8	1852.6	7.50	2568.6	5.41	1852.6	7.50
		256	69.5	200	86.4	161	65.7	212
		512	56.2	247	52.1	267	48.9	284
		1024	73.5	189	45.8	303	55.0	252
		4096	284.6	48.8	129.5	107	129.5	107
Machine 2	1	8	8876.8	7.51	12328.8	5.41	8876.8	7.51
		1024	82.7	806	105.3	633	81.0	823
		2048	70.2	949	65.5	1018	65.2	1023
		4096	98.6	676	61.0	1092	80.0	833
		16384	401.2	166	178.4	374	178.4	374
	8	8	1110.5	7.50	1541.1	5.41	1110.5	7.50
		64	**133**	62.6	196	42.6	**133**	62.7
		512	24.8	336	27.8	300	23.1	361
		1024	25.7	324	19.7	424	21.6	385
		2048	41.7	200	22.6	369	27.4	304
		4096	84.3	98.8	39.5	211	39.5	211
	32	8	278.4	7.48	385.3	5.41	278.4	7.48
		256	12.6	165	13.7	152	11.3	185
		512	12.6	166	9.4	221	10.1	205
		1024	19.6	106	10.5	198	12.9	161
		2048	38.7	53.9	18.0	116	18.0	116

Table 6.1: Results of different remaping strategies during the factorization of a front of size $nfront = 100000$ with varying *nproc* and *nthread*. Machine 1 corresponds to a GFlops/rate of 6 and a bandwidth of $1.2GB/s$ and Machine 2 corresponds to a GFlops/rate of 10 and a bandwidth of $4.1GB/s$. Results in bold are to be compared with the effective experimental results. Speed-ups are relative to *nproc* =1, with the value of *nthread* unchanged. The dashed lines are explained in Section 6.3.4.

Machine	Standard		Restarting (Alg 6.2)	
	Time (s)	GFlops/s	Time (s)	GFlops/s
`hyperion`	404	1650	385	1732
`ada`	227	2937	190	3509

Table 6.2: Influence of the restart mapping in split chains with $nfront = 64000$ on `hyperion` and $nfront = 10^5$ on `ada`, with $npiv = nfront$ using $nproc = 64$ and $nthread = 8$.

Chapter 7

Minimizing assembly communications

7.1 Introduction

The increasing number of processors in modern computers and the increasing size of problems to be solved make the amount of communications involved in data remapping and migration increase as well. These communications relate, in the multifrontal method, to assembly operations between child(ren) and parent fronts. As shown in the end of Chapter 6, they may rapidly become critical to performance. In the present chapter, we target this issue by reducing such communications as much as possible. In a synchronous environment, minimizing the maximum per-process volume of communications would be the right metric, as the assembly would directly depend on it. In our asynchronous environment however, as processes are not synchronized by the starting time for their assembly communications, minimizing the total volume of communications is a better objective.

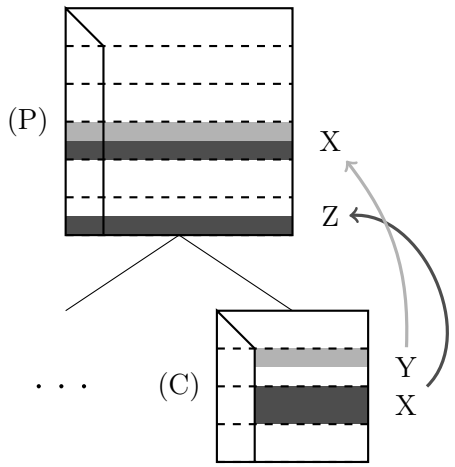


Figure 7.1: Illustration of some communications arising during the assembly operation between a child front (C) and a parent front (P). Process X sends rows to Z and receives rows from Y.

Before considering reducing assembly communications, let us first find their origin. Figure 7.1 is an illustration of a distributed-memory assembly operation between a child front (C) and a parent front (P). On the one hand, we see that some of the rows mapped on process X in C are remapped to process Z in P. Thus, X must send these rows (dark gray) to Z. On the other hand, we see that some rows mapped on X in P are mapped instead on Y in C. Thus, X must

receive these rows (light gray) from Y. Note that if X had simply kept its rows of C in P, it would not have generated any assembly communication. From this simple example, we see the effect of the distribution of the rows and the mapping of the processes in a front, on the volume of assembly data exchanged.

7.1.1 Problem definition

Let us formally define the problem as follows:

Definition 7.1 (Minimizing assembly communications between children and parent fronts)**.** *Given a parent front P to be mapped on a set of $nproc(P)$ processes and its children fronts $C_1, \ldots, C_n$, each mapped to a set of $nproc(C_i)$ processes, what should be the distribution of rows and the mapping of processes in P to minimize the total assembly time or, at least, to minimize the total amount of assembly data exchanged, while still ensuring a good load-balance in the parent?*

This problem may be viewed as two entangled subproblems:

Subproblem 1 (OrderRows): *Order the $nfront(P)$ rows of the parent front P in an appropriate manner. (We will discuss later what an appropriate order should be, in order to obtain a solution to the problem of Definition 7.1.)*

Subproblem 2 (MapRowSets): *Given a solution to subproblem* **OrderRows***, find the mapping of $nproc(P)$ sets of contiguous rows on the $nproc(P)$ processes of the parent front minimizing assemblies cost.*

Because of the multifrontal environment, we impose two additional constraints, one for each subproblem:

Constraint 1 (fsFirst): The *fully-summed rows*, which will be eliminated (factorized) in the parent front must appear higher in the front than do the *contribution rows*, which will be updated during the factorization of the parent front, before being assembled in the grandparent front and factorized in an ancestor. We will also consider that the process on the frontier between fully-summed and contribution rows may have both kinds of rows mapped on it.

Constraint 2 (EqRows): The $nproc(P)$ processes in the parent should all be assigned the same number of rows, that is, $\frac{nfront(P)}{nproc(P)}$ rows. We impose this constraint in order to ensure a good load-balance (see Definition 7.1) and to obtain a well-defined problem.

Notice that we consider that several processes can be mapped on the fully-summed rows, as this corresponds to the application of the splitting algorithm described in Chapter 6 (here with *eqRows*).

7.1.2 Assignment problem

Our problem may be expressed as follows:

Definition 7.2 (Assignment problem)**.** *Let us consider a number $nproc(P)$ of processes and a number $nfront(P)$ of rows to be mapped on them. Any process may have any row mapped on it, but at a varying cost, depending on the process-row association. We must map exactly $\frac{nfront(P)}{nproc(P)}$ rows on each process and exactly one process to each row with the objective of minimizing the total cost of the assignment.*

Formally, given two sets $Procs$ and $Rows$ of processes and rows, together with a weight function $S : Procs \times Rows \to \mathbb{R}$ representing, for example, the volume of assembly communications generated when mapping a given process p on a given row r, we must find a mapping function $map : Rows \to Procs$ such that the cost function

$$\sum_{r \in Rows} S(map(r), r)$$

is minimized.

The weight function, S, is usually viewed as a real-valued matrix S of $nproc(P)$ rows and $nfront(P)$ columns, called the *score* matrix. The cost function may then be written as

$$\sum_{r \in Rows} S_{map(r),r}$$

In our context, the cost of mapping a row to a process function (as used in Definition 7.2) will depend on the communication it involves for that row between the children and the parent.

We note that this assignment problem can be expressed as a standard linear program with the following objective function

$$\sum_{p \in Procs} \sum_{r \in Rows} S(p, r) x_{pr}$$

subject to the following constraints

$$\begin{cases} \sum_{r \in Rows} x_{pr} = \frac{nfront(P)}{nproc(P)} \text{ for } p \in Procs \ , \\ \sum_{p \in Procs} x_{pr} = 1 \text{ for } r \in Rows \ , \\ x_{pr} = 0 \text{ or } 1 \text{ for all } p, r \in Procs, Rows \ . \end{cases}$$

Finally, we define S_{pr} as the cost of mapping row r on process p. In practice, if row r corresponds to rows already mapped on p (assuming that process p is involved with the children), then $S_{pr} = 0$. Otherwise, S_{pr} is the cost of sending to process p the rows of the children that must be assembled in row r.

The integer variable x_{pr} represents the mapping of process p on row r of the parent, taking the value 1 if the assignment is done and 0 otherwise. The first constraint in the linear programming formulation means that every process is mapped on exactly $\frac{nfront(P)}{nproc(P)}$ rows, and the second constraint means that every row is mapped on exactly one process.

Therefore, classical linear programming techniques could be used to solve this problem. However, we describe in this chapter algorithms that are more specific to our remapping problem. Remark that the above assigment problem (and linear programming) formulation does not take **fsFirst** into account.

7.1.3 State of the Art

Our objective of minimizing migration costs during a remapping operation between children and parent nodes of a multifrontal tree belongs to the more general problem of *dynamic load balancing*. The dynamic load balancing problem does not consider the minimization of migration costs only. It rather aims to minimize the whole application execution time; namely: the sum of computation, communication, migration and repartitioning costs.

Among earlier work targeting repartitioning, Catalyurek et al. [30] present a dynamic load balancing technique based on hypergraph partitioning. The vertices of the hypergraph represent the computational load associated with the data. The hyperedges of the hypergraph represent

data dependencies. When a partition cut crosses a hyperedge, this leads to a communication cost, *i.e.*: a migration cost. Such a technique assumes, however, an unchanged number of processes during the remapping phase.

Vuchener and Esnard [110] have extended such techniques, allowing a change in the number of processes during the repartitioning. This extension can be applied but, unfortunately, only to the unconstrained version of our problem, as it does not take into account our constraints.

Also related to our problem, Hérault et al [65] consider the optimal redistribution cost from an arbitrary distribution to a target one, when the processes in the target distribution can be permuted to limit the volume of communication. In that case, the number of processes is the same in the original and target distribution. We also note that ScaLAPACK provides functionality (e.g., the `PxGEMR2D` routine) to redistribute a matrix from a 2D block cyclic grid of processors to another one, when the mapping is given.

7.1.4 Overview

In this chapter, we present mapping algorithms aiming to minimize communications of assembly operations in the multifrontal method. We focus on the relative positioning of processes in fronts over the positions of rows in these fronts in order to maximize the amount of common rows that processes have between parent and children fronts. We first present, in Section 7.2, a heuristic for an approximate mapping of the multifrontal tree. We then present in Section 7.3 an algorithm for optimal mappings on split chains. Finally, Section 7.4 provides simulation results obtained on typical application cases and shows the gains to be expected from such mappings over the random or canonical mappings that we have used until now. These gains depend on the structural characteristics and mapping of the children fronts being assembled.

Let us remind the reader that the main notations used in this chapter are described in Tables C.1, C.2 and C.3.

7.2 Algorithm to minimize assembly communications in the general case

In this section, we present, first, an algorithm producing an optimal solution of the **MapRowSets** subproblem, given a solution to subproblem **OrderRows** (see Section 7.2.1). We then present in Section 7.2.2 a potential approximation to the solution of subproblem **OrderRows**, leading to an algorithm to solve the our communication minimization problem (from Definition 7.2). We finally present, in Section 7.2.3, an optimal algorithm to solve this problem (ie, both the **OrderRows** and **MapRowSets** subproblems) without taking into account the **fsFirst** constraint. Although this variant cannot be applied in our context (since the **fsFirst** constraint is not satisfied), it provides a lower bound that will help in evaluating the quality of the solution of the constrained problem.

7.2.1 Hungarian Method on Blocks of rows (HMB)

Let us assume that we have a solution to subproblem **OrderRows** under the constraint **fsFirst**, i.e.: we have an ordering of the rows in P (that we will present in Section 7.2.2).

In order to find a solution to the **MapRowSets** subproblem while respecting the **eqRows** constraint, we must divide the set of rows of P into $nproc(P)$ contiguous blocks of equal size (to within a rounded error), before identifying the mapping of each of them to a process involved in P. Since we now have to map as many blocks of rows as there are processes (namely, $nproc(P)$) instead of having to map many more rows than processes, we can then apply the *Hungarian*

method [75] (also known as the Kuhn–Munkres algorithm or the north-west-corner method) to find an optimal mapping. The aim of this method is to find a maximum matching in a weighted bipartite graph. Indeed, this method solves the (bipartite) 1-to-1 version of the assignment problem. The entry $S_{p,b}$ of the (now square) score matrix used by this method represents the cost of matching block b on process p. This cost is simply computed by summing the costs of mapping each row r in the block b on the process p. The cost of each row r is computed by considering the presence or absence of rows in the children C_i that will be assembled in row r of the parent along with whether the process holding such a row in a child C_i is the same as p or not. The cost of mapping a block b of rows on a process p is more precisely given by

$$S_{p,b} = \sum_{r \in b}^{nb} \sum_{i=1}^{children} ncb(C_i) \times y(p, r, C_i), \tag{7.1}$$

where

$$y(p, r, C_i) = \begin{cases} 0 & \text{if } r \notin C_i \vee\ mapping(r, C_i) = p\ , \\ 1 & \text{if } r \in C_i \wedge\ mapping(r, C_i) \neq p\ . \end{cases}$$

and where $ncb(C_i)$ is the size of the contribution block of C_i that will be sent to the parent.

In the above formula, the notation $r \in C_i$ actually means that a row of C_i must be assembled in row r of the parent front. In that case, if that row is not mapped on p in the child (i.e., $mapping(r, C_i) \neq p$), it induces a communication volume $ncb(C_i)$ (order of the contribution block of C_i), corresponding to the number of entries in that row in C_i (all entries of the row are sent and all rows in C_i have the same numer of entries).

Although the worst-case complexity of applying this Hungarian Method on Blocks (*HMB*) is $O((nproc(P))^3)$, several efficient implementations of the Hungarian method exist. In our evaluations (see Section 7.4), we used an existing implementation available in Python libraries.

7.2.2 Fronts characteristics and ordering of the rows

In this section, we propose a solution to the **OrderRows** subproblem by considering some properties of the structure of the front at a parent node.

In the multifrontal method, a parent front has a number of children fronts that depends on: (*i*) the initial sparse matrix from which the assembly tree is built; (*ii*) the ordering algorithm which was used to generate the corresponding elimination graph; and (*iii*) the symbolic factorization, if performed separately from (*ii*). Most often (although not always), when nested dissection techniques are applied (see Section 1.3.1.3), there are two children because these techniques dissect one set of variables into two disjoint subsets by identifying a set of *separator* variables that correspond to the fully-summed variables of the front. Moreover, cases with a single child often correspond to cases of artificial (split) chains (see Chapter 6) that we will consider in detail in a specific section (Section 7.3). Hence, we focus in this section on the case of two children. However, this discussion could be generalized to an arbitrary number of children.

When solving subproblem **OrderRows**, constraint **fsFirst** still leaves a significant degree of freedom regarding the choice of the relative positioning of the rows within the subset of the fully-summed rows and within the subset of the contribution rows.

Intuitively, using a random distribution of rows would most likely lead to a high volume of communications. Indeed, in this case, any given block of contiguous rows in P would have its rows already mapped onto the children C_i on several *distinct* processes. Therefore, there would

be no process that could be preferentially associated with any block. Moreover, we would not be able to say much, in general, on the structural properties of P as a function of the structure of the children. However, if we group rows by their common characteristics (like the sets of contributing children/processes), we will see that we can distinguish some patterns appearing in the front P. We would then like to see the impact of fixing a local order between the rows of a parent P, that takes into account the origin and order of those rows in the child nodes.

Figure 7.2 displays a schematic representation of a parent P depending on C_1 and C_2 in a very general case. In this figure, rows to be eliminated in P appear first and contribution rows of P appear last. In each group, we have organized subgroups of rows to separate rows common to C_1 and C_2, rows coming exclusively from C_1, and rows coming exclusively from C_2. We have not considered in this figure rows/variables that appear exclusively in P (parts of the original matrix), because those do not involve communication between the children and the parent. Such rows give some freedom in the mapping as they do not involve communication from the children. For example, they may be mapped preferentially on newly available processes that are not involved in computations of either C_1 or C_2.

In the figure, we have the property that (i) fully-summed rows to be eliminated come before contribution rows (**fsFirst**) but also that (ii) inside the fully-summed block and the contribution rows block, rows with the same characteristics are grouped together. For example, rows appearing only in the contribution block of C_1 come before rows appearing only in the contribution block of C_2. Moreover, delayed rows due to numerical pivoting issues appear last in the list of fully-summed rows of the parent.

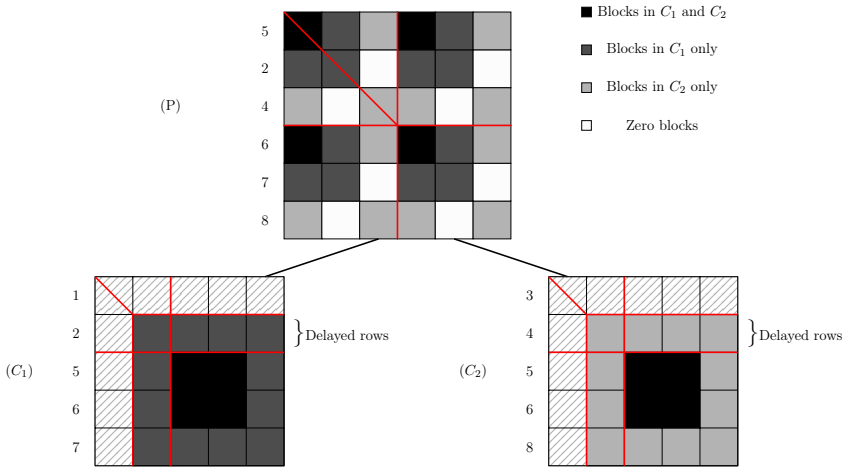
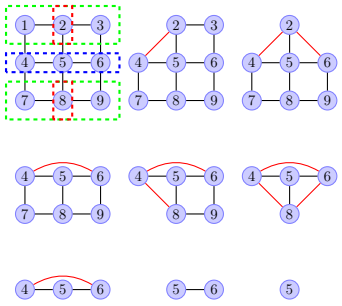


Figure 7.2: Structure of a parent front with respect to the structure of its two child fronts, in the general case. A row containing only entries in C_i assembles rows from C_i only. A row containing an entry in both C_1 and C_2 assembles rows from both C_1 and C_2. Rows corresponding to index 2 in child C_1 and to index 4 in child C_2 have their elimination delayed from the children to the parent because of numerical difficulties.

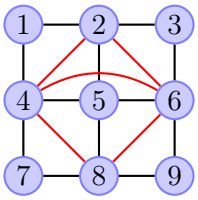
By using such groups and subgroups of contiguous rows instead of a random distribution of rows, processes mapped onto the children might be able to keep some, or all, of their rows in P, given an appropriate solution to subproblem **MapRowSets**. Rows common to C_1 and C_2 may be exchanged only between processes they are mapped on in Cs or may be mapped on new processes in P as, inevitably, they will be the source of assembly communications. Moreover, fully-summed rows in P coming exclusively from C_1 or C_2 are most often *delayed pivots* resulting from numerical difficulties during the factorization of C_1 or C_2. Along with the other set of rows only present in C_1 or C_2, they may be remapped exclusively on the processes on which they

were mapped before, without generating any assembly communication.

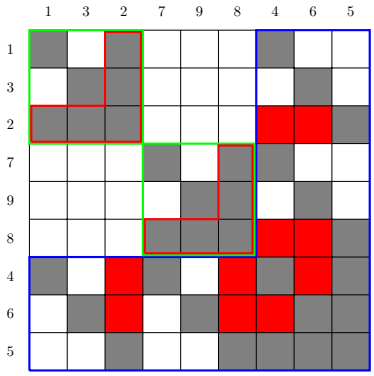
We can make more precise observations when nested dissection based orderings are used to generate multifrontal trees. The rows in the front corresponding to variables from the separator of the nested dissection method are common to both C_1 and C_2. They will be the (or part of the) fully-summed variables to be eliminated in P. The other rows corresponding to the two disjoint subsets are completely disjoint between contribution blocks C_1 and C_2. They will be the (or part of the) variables of the contribution block in P. We repeat in Figure 7.3 Figure 1.4 from Chapter 1, which illustrates this structure for a simple example based on nested dissection. In this figure, variable 2 is the separator between variables 1 and 3. Variable 2 is thus fully-summed in the parent front and variables 1 and 3 are fully-summed in the two corresponding children fronts (those fronts are the three left-most fronts of Figure 1.6). Variable 2 is common to the contribution blocks of both children and variables 4 and 6 appear in the parent but are each present in just one child.



(a) Elimination of variables.



(b) Filled graph.



(c) Sparsity pattern of the LU factors.

Figure 7.3: Nested-dissection ordering (fill-in in red).

More generally, we can build an accurate structure of the parent front P such as the one presented in Figure 7.4. In this figure, the ordering of the rows consists in grouping disjoint contribution rows of C_1 in one part and those of C_2 in another part. Inside each part ($C_1 \cap C_2$ rows, C_1 only rows or C_2 only rows), we order the rows in the parent in the same order as they appear in the children. Such an ordering presents some advantages.

A first advantage is that, even if processes will still potentially have to exchange contributions related to the eliminated variables of P, this will not necessarily be the case on contribution variables of P, everything depending on process mapping. For example, in the case where a proportional (or equivalent) mapping is considered and where $\frac{ncb(C_1)}{nproc(C_1)} \approx \frac{ncb(C_2)}{nproc(C_2)}$, there will be potentially few assembly communications on contribution rows of P.

A second advantage is that such row distributions allow us to have only one large contiguous block of nonzeros in the contribution block of P corresponding to each of C_1 and C_2 (one for each of them). With an appropriate process mapping, not only would this allow processes mapped on the children to avoid any assembly communication, but would even make the assembly process indirection-free, in the sense that contiguous entries in a child row could also be contiguous in the parent row. Furthermore, an implementation may make it possible to reuse the same memory locations from C for the computations in P, thus avoiding needless copies and maximizing memory locality.

A third advantage of such distributions is that they lead to one large block of zeros corresponding to each of C_1 and C_2. To our knowledge, these off-diagonal zero blocks in assembled fronts have not be exploited in multifrontal solvers. However, in some symmetric solvers, the existence of zeros on diagonal blocks of frontal matrices are exploited when factorizing frontal matrices using oxo or tile pivots [47]. In our case, off-diagonal zero blocks are filled during the factorization of the front but could be exploited in low-rank sparse direct solvers (see, e.g. [8]). Indeed, in such techniques, where a block is compressed (approximated) using the product of two smaller rectangular blocks, the compression that can be achieved to a block with respect to a given precision threshold depends on the rank of the rows (or columns) constituting it. Thus, when update operations are applied on a block of zeros, the resulting updated block will potentially have a low rank which will be in all cases smaller than $npiv(P)$.

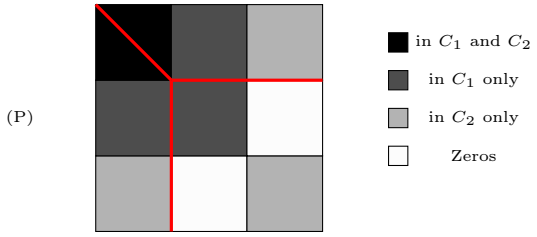


Figure 7.4: Shape of a parent front P with two children C_1 and C_2 in the case of an ordering based on nested dissection, without delayed pivots. (The general case was shown in Figure 7.2.)

We believe that we can make similar observations to those presented in this section in the case where a parent front has more than two children, even if these observations would be harder to visualize because of the combinatorial explosion of the classes of blocks when increasing the number of children.

To conclude, this section gives an approximate solution to problem **OrderRows**. Since we have a solution to the **MapRowSets** subproblem (Section 7.2.1), we thus obtain an approximate solution to our initial problem.

7.2.3 Hungarian Method on Rows (HMR)

From the results of Sections 7.2.1 and 7.2.2, we have an optimal solution to the **MapRowSets** subproblem, but only an approximate solution to **OrderRows**. The solutions however respect the **fsFirst** and **eqRows** constraints. In order to measure how far our solutions are from optimal, it is interesting to be able to compare them with an optimal solution of the unconstrained version of our problem. We discuss in this section an algorithm to solve the unconstrained problem. In fact, **eqRows** will still be maintained, but **fsFirst** is suppressed, so that any row can be mapped on any process of the parent without constraint.

In order to obtain such a solution, we do not apply the Hungarian method on blocks of rows as was done previously in the HMB approach. Instead, we will apply the Hungarian method to map *each* row of the parent front. However, as the method requires equal numbers of rows and processes, we must artificially replicate each process into as many virtual processes as there are rows per process. This results in a cost matrix S in the associated assignment problem that is square, with $nfront(P)$ rows for the virtual processes and $nfront(P)$ columns representing the rows in the parent P. Compared to the HMB approach where the cost matrix was of order $nproc(P)$, this Hungarian Method on Rows (HMR) is thus costlier, with a $O((nfront(P))^3)$ worst-case complexity. As explained before, this approach is however not meant to be used in practice because it does not respect **fsFirst** (although **eqRows** is respected); we only discuss it

because it will provide a lower bound on the volume of communication, and will help evaluating the quality of the HMB heuristic.

7.3 Special case of split chains

In this section, we present an optimal remapping algorithm for the special case of a child C and a parent P that are part of a split chain (see Chapter 6) of a multifrontal tree. The central characteristic is then that the structure of the parent P exactly matches the structure of the contribution block of its child C, and that the order of the rows is given, both in the child and the parent. Therefore, there is no need to solve the **OrderRows** subproblem.

The number of processes in the parent is $nproc(P)$ and the number of processes in the contribution block of the child is $nproc(C) - 1$, since one process in the child is dedicated to the master part of the child containing the rows eliminated at the child.

At each node of the split chain, the **eqRows** constraint must be ensured. Therefore, each process of P (resp. C) is assigned $\frac{nfront(P)}{nproc(P)}$ rows (resp. $\frac{nfront(C)}{nproc(C)}$ rows, assuming **eqRows** is also applied in the child – see also Chapter 6). Concerning the child, we are more precisely interested in the fact that the blocks all contain $\frac{nfront(C)-npiv(C)}{nproc(C)-1} = \frac{ncb(C)}{nproc(C)-1}$ rows.

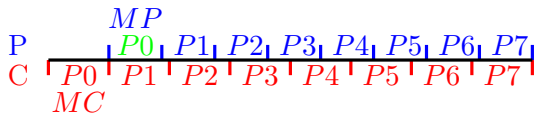
As described in Chapter 6, the restart mapping is a succession of communication-free mappings (which require no assembly operation, $nproc(P) = nproc(C) - 1$) and restart operations (which require mapping more processes on the parent P than there are in the contribution block of the child C, $nproc(P) > nproc(C) - 1$). We will thus focus on this configuration, in which we consider that the set of processes in C, noted $Procs_C$, is included in that of P, $Procs_P$.

7.3.1 Illustrative Example

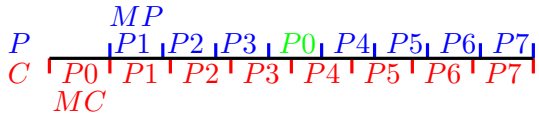
Let us consider a split chain not as a succession of fronts in a chain, but as a succession of contribution blocks inside the initial front (see Figure 6.1 of Chapter 6). From this perspective, as we consider a 1D matrix distribution over processes, successive fronts C and P in the chain may be represented as line segments (the segment representing P thus being a subsegment of C) and processes' rows sets as subsegments of these segments. An example is given in Figure 7.5, where the front of P is mapped on processes MP= $P0$, $P1$,...,$P7$, and the contribution block of C is mapped on the processes $P1$, $P2$, ..., $P7$.

In the case represented in the figure, we have the equality $nproc(P) = nproc(C) = 8$ and the problem consists in remapping the $ncb(C) = nfront(P)$ rows from 7 to 8 processes. Clearly, the master of C (MC, or in this case, $P0$) has no common rows between C and P: $P0$ must receive $\frac{nfront(P)}{nproc(P)}$ rows and does not send any. Let us first consider the natural mapping of the processes, where processes in P are mapped in the same order as processes in C, as shown in Figure 7.5a. We observe that the workers in C ($P1$ to $P7$) all have some common rows between C and P. For example, $P7$ only sends a few rows to $P6$ while it does not have to receive any row. $P1$ must send many rows (here to $P0$) and receive many rows (from $P2$) but still has a few common rows for which no communication is required. However, this configuration globally implies too many row exchanges and does not lead to an optimal overall amount of communication. In contrast, inserting the master of C (MC = $P0$) in the middle of the list of the processes of the parent and choosing the first worker of C ($P1$) as the master of P (MP) leads to a mapping (Figure 7.5b) with far less rows received by processes in the parent P (or equivalently, far less rows sent by processes in the child C). This mapping is indeed optimal: only the mapping of P1, P2, P3, P5, P6, P7 shown in Figure 7.5b maximizes the number of common rows for those processes between child and parent. Two blocks remain to be mapped in the parent, on P0 and P4. This leaves two possibilities for P0 and P4: the one in the figure and the one where P0 and P4 are swapped.

In both cases, the number of common rows is maximized for P4. Therefore, both mappings are equivalent and optimal.



(a) Standard mapping.



(b) *MinAsmComm* mapping.

Figure 7.5: Illustrative example of mappings in a split chain.

7.3.2 Remapping algorithm

Since the order of rows in the case of split chain is given, we have a solution to the **OrderRows** subproblem. Then, the HMB method presented in Section 7.2.1 yields an optimal remapping of the processes in P, for all values of $nproc(P)$ and $nproc(C)$. However, our specific context of split chains makes it possible to design a simpler algorithm, specific to our case. We will refer to this algorithm as the *MinAsmComm* algorithm (resulting in Figure 7.5b) and describe a possible implementation in Algorithm 7.1.

The goal of the *MinAsmComm* algorithm is to make each process keep as many rows as possible between child and parent. We note that, related to this work, Vuchener and Esnard [110] have also studied a problem equivalent to this one. They give a formal characterization of optimal solutions and prove their optimality. The sketch of proof behind the optimality of *MinAsmComm* relies in our context on the observation that, in the case where $nproc(P) \geq nproc(C)$ (restart mapping), the pattern encompassing the subsegment of P shown in Figure 7.5 will just be repeated $GCD(nproc(C), nproc(P))$ times (with GCD being the Greatest Common Divisor). Finding an optimal mapping on P reduces to finding an optimal mapping on each of these subsegments. We can see that the number of common rows between blocks in C and blocks in P is maximal on the borders of the subsegment of P, and tends to decrease until the middle block, where only half the rows of the C block are shared with a P block. Applying the *MinAsmComm* algorithm on this subsegment will first remap the processes that are near the borders of C near the borders of P, then will progress to the middle until all the processes in the subsegment of C are remapped. Then, as they have no row of P already mapped on them, the newly available processes, which are assigned in P along with MC, are mapped randomly on the remaining unmapped middle blocks. Finally, applying *MinAsmComm* directly on all blocks in P will result in the same behaviour and the same result as when applied separately on each subsegment.

In the case of the example of Figure 7.5, Algorithm 7.1 is such that the loop at line 7 maps processes from $Procs_P$ in the order $P1$, $P7$, $P2$, $P6$, $P5$, $P4$, $P0$ (depending on tie breaking).

Input: $Blocks_P$: the blocks of rows in the parent P,
$Blocks_C$: the blocks of rows in the child C,
$Procs_P$: the set of processes in the parent P,
$Procs_C$: the set of processes in the child P,
$Mapping_C : Procs_C \rightarrow Blocks_C$, the mapping of C
Output: $Mapping_P : Procs_P \rightarrow Blocks_P$, the mapping of P

{Find maximum common rows between child and parent blocks}
for all $p \in Procs_P \cap Procs_C$ **do**
  $b_C \leftarrow Mapping_C(p)$
  $MaxCommonRows(p) \leftarrow \max_{b_P \in Blocks_P}(|b_P \cap b_C|)$
end for
{Map processes that can be mapped ideally}
for all $p \in Procs_P \cap Procs_C$ in decreasing order of $MaxCommonRows(p)$ **do**
  $b_C \leftarrow Mapping_C(p)$
  Find an unmapped $b_P \in Procs_P$ such that $MaxCommonRows(p) = |b_P \cap b_C|$
  if b_P found **then**
    $Mapping_P(p) \leftarrow b_P$
  end if
end for
Map remaining processes from $Procs_P$ arbitrarily.

Algorithm 7.1: *MinAsmComm* algorithm.

7.4 Simulation results

7.4.1 Results in the general case

We now present simulation results comparing *HMB* and *HMR* on typical test cases involving a parent and two children.

Figure 7.6 represents an example of score matrix relative to *HMR*. It corresponds to a test case where: $nfront(P) = 900$ and $nproc(P) = 15$; $ncb(C_1) = 700$ ($rows \in [1, 100] \cup [301, 900]$) and $ncb(C_2) = 300$ ($rows \in [1, 300]$); $nproc(CB_1) = 3$ and $nproc(CB_2) = 7$ ($lines \in [1, 900]$) (with $nproc(CB_i) = nproc(C_i) - 1$ being the number of processes mapped onto the contribution block of child i). The X-axis represents the rows of the parent front. The Y-axis represents the virtual processes to be mapped on this parent front. Gray scale intensity represents the amount of communication induced by the mapping of each row on each process, from black (no communication) to white (maximum communication).

We can see, for example, that $rows \in [667, 900]$ are mapped only onto process 3 in C_1. They can thus be mapped on it in P without introducing any communication (black area). However, mapping them on any other process would mean that process 3 had to send them to it, which would imply some communication (light gray area).

We can also see that it is cheaper for any process to be mapped in P on $rows \in [101, 300]$, which corresponds to the contribution block of the small second child front (C_2, as the corresponding colour is dark gray), than to be mapped on $rows \in [301, 900]$, which correspond to the contribution block of the large first child front (C_1, as their corresponding colour is light gray). Indeed, $rows \in [101, 300]$ are of size 300 (number of columns related to them), while $rows \in [301, 900]$ are of size 700.

Moreover, $rows \in [1, 100]$ are shared by both children. This is why the cost of their mapping on any process, which is not used in the children, will cause maximum communication (white

area), i.e., communications from processes handling them in C_1 and from those handling them in C_2. Similarly, processes on which these rows are already mapped in one child will need communications from the processes handling them on the other child (dark gray area corresponding to processes mapped on C_1 and light gray areas to processes mapped on C_2).

The obvious observation that we can make from the figure is that the artificial processes required by *HMR* cause a great amount of redundancy in the matrix; consider for instance that each process, from 1 to 15 in the Y-axis, actually corresponds to $900/15 = 60$ virtual processes, or 60 rows in the cost matrix. Consequently, we expect that *HMB* would take advantage of such blocks which are identical.

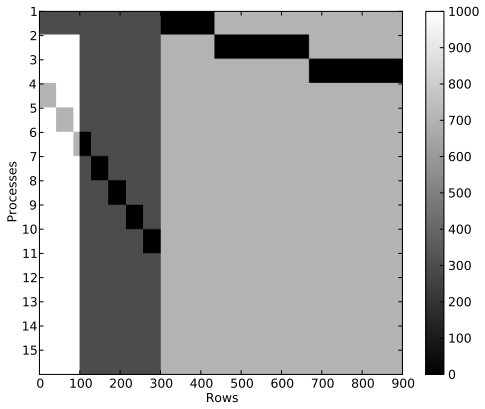


Figure 7.6: Example of a score matrix for *HMR*. The test case correspond to a first child front with a contribution block with 700 rows, with 3 processes mapped onto them, and a second child front with 300 contribution rows and 7 processes mapped on them. The first hundred rows are common to the contribution blocks of both children. The X-axis represents each row of the parent front. The Y-axis represents each (virtual) process involved in the computation of the parent front. Each point thus represents the cost of mapping the corresponding row on the corresponding process in terms of the total number of scalars to be communicated by all processes to the current process in order to assemble that row. The colour intensity represents the value of the cost. Dark means low cost, whereas light means high cost.

Table 7.1 shows the total amount of data transferred during assembly communications when mapping parent front processes using different algorithms and for different test cases.

We use *Random*, *Canonical*, *HMB* and *HMR* mappings to identify the mapping of processes between children (C_1 and C_2) and parent (P) fronts. The *Random* mapping consists in mapping processes randomly on rows, while the *Canonical* mapping consists in sorting the processes in the order of their IDs and assigning them onto the parent front rows in that order. The HMB and HMR mappings are the ones from Sections 7.2.3 and 7.2.1.

Columns *ncb* and *nfront*(P) represent the size of the contribution blocks of the children, and the size of front P, respectively. Column *nproc* CB and *nproc*(P) give the number of processes mapped on the contribution blocks of the children and on the parent front P, respectively. Different front sizes, contribution block sizes and numbers of processes are used. They are chosen to correspond to typical cases encountered when treating 2D and 3D finite difference and finite element matrices. The rows common to the two children nodes are usually those that will be eliminated in P (fully-summed rows), they are located in the first positions in the contribution block of each child, respecting a local order on the variables between the Cs and P

(as in Figure 7.4). The number of these common rows is thus $ncb(C_1) + ncb(C_2) - nfront(P)$.

Test case						Algorithms			
ncb		*nproc CB*		*nfront*(P)	*nproc*(P)	Random	Canonical	*HMB*	*HMR*
C_1	C_2	C_1	C_2						
1000	1000	5	5	1000	10	1812	1800	1000	1000
1000	1000	7	3	1000	10	1792	1720	1051	994
1000	1000	5	5	1500	10	1784	1400	950	700
1000	1000	7	3	1500	10	1782	1018	885	699
250	150	5	5	300	10	75.5	67.5	35.5	34
250	150	7	3	300	10	75.95	53	31	24.5
300	200	5	5	400	10	117.3	92	42	42
300	200	7	3	400	10	117.2	46.4	44.8	28.2
400	300	5	5	500	10	222.8	195	106	96
400	300	7	3	500	10	227.3	153.4	106.4	86.2
600	600	5	5	1000	10	643.8	372	300	192
600	600	7	3	1000	10	655.8	486	321	240
400	400	5	5	700	10	288.8	132	124	64
400	400	7	3	700	10	287.6	227.2	139.2	90.4

Table 7.1: Simulated results for different mapping strategies for minimizing assembly communications. For each size and number of processes on each of P (Parent), C_1 (Child 1) and C_2 (Child 2), the *Random*, *Canonical*, *HMB* and *HMR* strategies are applied, and the resulting total amount of communications (in terms of number of entries sent x1000) is shown.

Firstly, it is important to notice that the *Random* and *Canonical* approaches, which do not take into account the targeted problem, behave poorly. They almost always lead to high volumes of assembly communications. Indeed, only if the appropriate process is mapped on the appropriate set of rows on the parent front may the related assembly communications be avoided. The choice of any other process for that set of rows inevitably causes data transfers.

Secondly, both *HMB* and *HMR* lead to better results. Since the *HMR* yields optimal mappings, we can conclude that the *HMB* leads to worthwhile approximations.

Thirdly, the computation time of *HMB* is much reduced compared to that of *HMR*. For example, for the case corresponding to the first row of results in Table 7.1, *HMB* finds a mapping in only 2.29×10^{-5} seconds while *HMR* requires 4.84×10^{-2} seconds. Moreover, the time and space required to build the score matrix along with the computation time of *HMR* and *HMB*, grows polynomially with the problem size and the number of processes used, respectively. Since the size of the matrices is usually much larger than the number of processes, the construction of the score matrix and the computation time of *HMR* may rapidly become a challenge. This method will become rapidly unusable, while *HMB* may still be used.

All these remarks show that *HMB* may be very useful in practice.

7.4.2 Results in the special case of split chains

As noted before, the HMB approach will result in an optimal total volume of communication in the case of a child and a parent that are part of a split chain. However, we use here Algorithm 7.1 (also leading to an optimal mapping) for its better time complexity. We now present simulation results that show both a reduction in the total and in the per-process communication volume.

7.4.2.1 Total remapping communication volume

Figure 7.7 shows the total volume of data transfered during assembly communications with and without applying the *MinAsmComm* mapping between nodes C and P. The size of front C is $nfront(C) = 100000$ and $nproc(C)$ and $nproc(P)$ vary between 2 and 128. We recall that when $nproc(P) \geq nproc(C)$, the set of processes in the child is included in that of the parent. When $nproc(P) = nproc(C) - 1$, which corresponds to the case of communication-free mapping (CFM), the processes in P are the workers of C, i.e., the ones mapped on the contribution rows of C. Here, we also consider the case where $nproc(P) < nproc(C) - 1$, and we consider in this case that the processes in P are also part of C. The latter case is more general than the case of split chains where $nproc(P) \geq nproc(C) - 1$ and depends on which processes from C are in P. In our setting, we consider that the common processes in C and P are the first workers of C. In the figure, the X and Y axes represent the number of processes in P and C, respectively. The color intensity represents the total amount of communications in terms of number of scalars (red for high and blue for low).

We observe in Figure 7.7a that the default restart mapping without the *MinAsmComm* algorithm behaves poorly whenever $nproc(C) \neq nproc(P) + 1$, as it induces very large amounts of communication (red areas). In contrast, as shown in Figure 7.7b, the *MinAsmComm* algorithm is often efficient, particularly when there are as many, or more, processes in P than in C (blue and green areas above the diagonal), which corresponds exactly to the case of restart operations that we target. We note that, when $nproc(C) = nproc(P) + 1$, there are no communications, which corresponds to the case of CFM. This corresponds to the communication-free mapping discussed previously in Section 6.3.1. Henceforth, the fact of reducing the global volume of communications implies that there will be potentially less network contention, which may in turn reduce the assembly times even further.

We observe in Figure 7.7b that the lower triangle is not the transpose of the upper one; so that the picture is not symmetric. This is because, when there are less processes in P than in C, the communication volume also depends on which subset of C is being used in P. In our case, we have considered that the mapping algorithm has chosen $Procs_P$ to be the first workers in C. Other choices would lead to different communication volumes. Let us however recall that this case is less interesting for us: we are mostly targeting the upper triangle corresponding to our context of split chains.

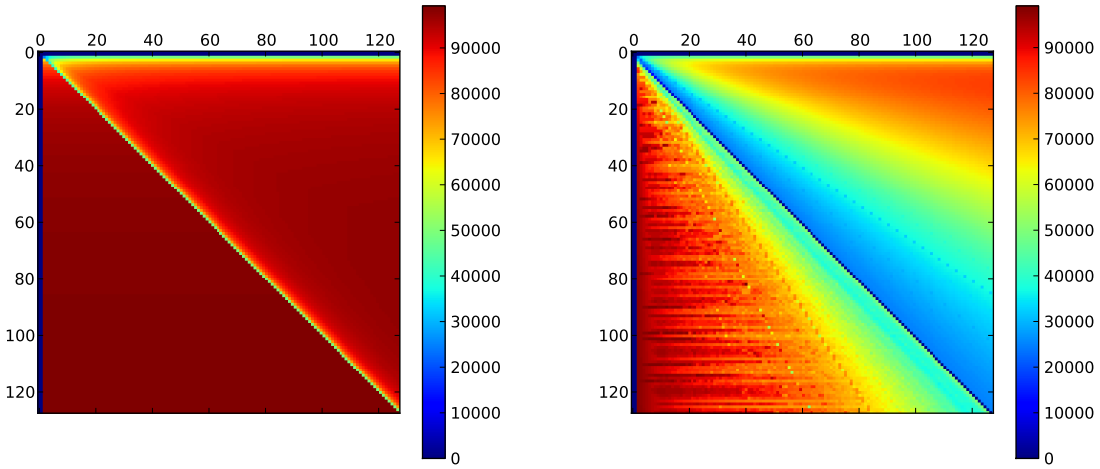
7.4.2.2 Per-process remapping communication volume

Although Algorithm 7.1 aims at minimizing the total communications volume, we now illustrate its impact on the per-process communication volume, which can be also of interest in practice.

Figures 7.8 and 7.9 show the sent per-process remapping communication volumes obtained when using the standard and the restart mappings, with and without the use of the *MinAsmComm* algorithm. The X-axis represents the numbers of rows sent by processes, whereas the Y-axis represents the number of processes sending that numbers of rows. We consider here two simulations with identical and different numbers of processes used between parent and children fronts.

Figure 7.8 concerns the assembly between fronts C and P with $nfront(C) = 100000$, $nfront(P) = ncb(C) = 98437$ ($eqRows$) and $nproc(C) = nproc(P) = 64$ (standard mapping).

In the case of Figure 7.8a, we have used a natural mapping, which consists in ordering the processes the same way in C and P. We can observe that each process has to send a different number of rows to other processes: from no rows at all (in the case of the master of C) to a



(a) Without *MinAsmComm*.

(b) With *MinAsmComm*.

Figure 7.7: Total volume of communications in assembly step as a function of the number of processes, with $nfront(C) = 100000$ and $nproc$ varying between 2 and 128. The X-axis corresponds to $nproc(P)$ while the Y-axis corresponds to $nproc(C)$. The colour intensity corresponds to the volume of communication (number of scalars sent from child to parent).

maximum number of rows, corresponding in this case to the number of rows to be sent by the first worker of C.

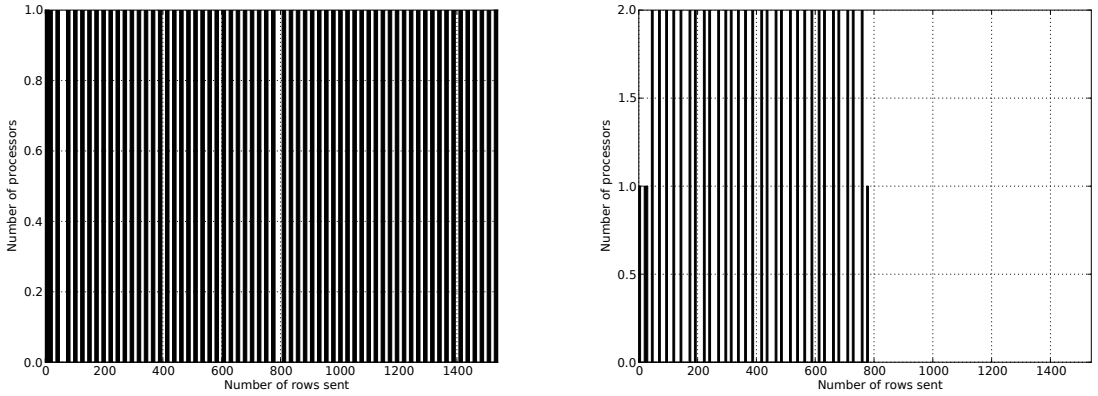
In contrast, in the case of Figure 7.8b, we have applied the *MinAsmComm* mapping. Looking at the square with coordinates $0 < x < 800$ and $0 \leq y \leq 1$ in Figures 7.8a and 7.8b, we now observe that half of the processes still send the same amount of rows as before. However, the volume of data sent by the other half has been roughly divided by two.

Figure 7.9 concerns the assembly between fronts C and P with $nfront(C) = 100000$, $nproc(C) = 56$ and $nproc(P) = 64$ and corresponds to the case of restart mapping described in Section 6.3.1.

In the case of Figure 7.9a, the new processes are mapped at the top of front P and the workers of C at the bottom of front P. We observe that 48 processes (out of 56) send the maximum possible number of rows, that is, all their rows.

In contrast, in the case of Figure 7.9b, we have applied the *MinAsmComm* mapping. We now observe that all processes send less than half of the number of rows that they had in C.

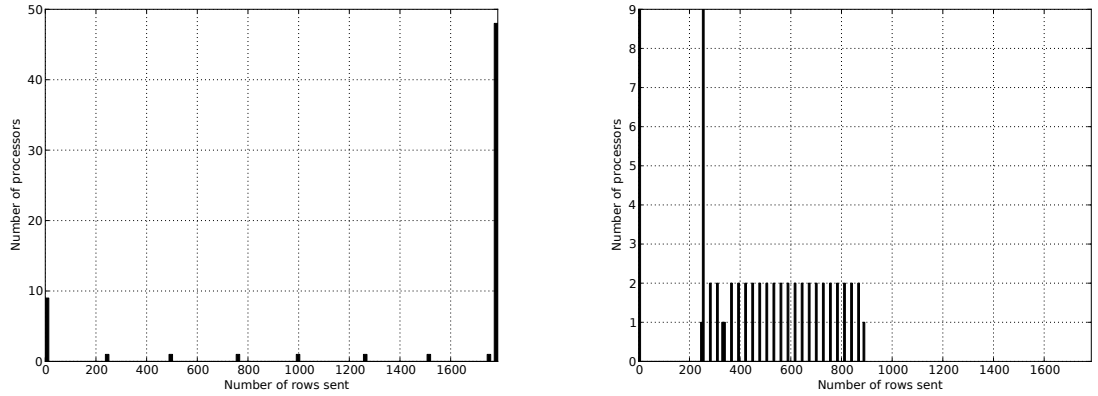
In summary, the important point to notice here is that the maximum number of rows any process has to send is divided by two when applying the *MinAsmComm* algorithm, compared to applying a regular mapping. Thus, on most processes, the time of assembly operations can be radically reduced, at least divided by two, when using *MinAsmComm* mapping instead of when not using it.



(a) Without *MinAsmComm*.

(b) With *MinAsmComm*.

Figure 7.8: Per-process remapping communication volumes during the assembly between fronts C and P with $nfront(C) = 100000$ and $nproc(C) = nproc(P) = 64$.
X-axis: numbers of rows sent by processes. Y-axis: number of processes in the child sending that number of rows.



(a) Without *MinAsmComm*. 48 processes of the child send all their rows.

(b) With *MinAsmComm*. No process sends more than half its rows.

Figure 7.9: Per-process remapping communication volumes during the assembly between fronts C and P with $nfront(C) = 100000$ and $nproc(C) = 56$ and $nproc(P) = 64$.
X-axis: numbers of rows sent by processes. Y-axis: number of processes in the child sending that number of rows.

7.5 Conclusion

It is difficult to measure precisely the actual assembly costs in practice. In our asynchronous environment (see Chapter C), assembly operations start and complete at different times on each process and there is some overlap between computation and communication. Thus, in order to measure the effect of the *MinAsmComm* mapping, we have synchronized artificially all processes after they had finished their computations in C and after they have done the actual assembly operations. On split chains, when $nproc(C) = nproc(P)$ for example, we have observed on ITAC Gantt-charts that the *MinAsmComm* mapping effectively divides the total amount of assembly communication by two and that the results match the simulation. Unfortunately, the

total assembly time remained unchanged. The processes which do not have any row of P already mapped on them still need to receive all their rows of P from other processes. Examples of such processes are MC or previously lost ones during former factorizations of descendant fronts.

Fortunately, in our asynchronous environment (without the artificial synchronizations – See Figure 6.5), such processes have usually finished their computations by the time we start the assembly operation and are thus the first to be ready to receive contribution blocks from workers of C. This, coupled with the aforementioned benefits of overall reduction of assembly communication volumes and reduction of network contention, should make the *MinAsmComm* mapping reduce the total execution time for the whole application.

To conclude, we have presented in this chapter techniques aimed at reducing the total volume of data exchanged during assembly operations, both on multifrontal chains and general trees. Even though the time of assembly operations depends on the per-process time of assembly, in asynchronous environments, reducing the total volume of communications helps to reduce the total time of assembly, due to unsynchronized communications and reduced contention in the network. One advantage of the results obtained in this chapter, which we have not mentioned yet, is that the reduction of communication on networks greatly helps to reduce energy consumption. Indeed, on modern computers and *a fortiori* on future extreme scale computers, communications contribute the greatest share of total energy consumption. Reducing communication, even without a significant reduction in execution time, is still very worthwhile.

Moreover, in this chapter, we focused exclusively on 1D matrix distributions. It would be interesting to analyze whether the proposed approaches could also be applied or extended to the case of 2D (possibly, block-cyclic) distributions.

Chapter 8

Synchronizations in distributed asynchronous environments

8.1 Introduction

Fully asynchronous environments are double-edged swords. They allow us to reach high performance but at the price of introducing complex dependencies between tasks that need to be handled with much more care to avoid **deadlock** situations. Deadlocks correspond to cycles in the process dependency graph that are often related to resource dependencies, in a limited memory context (as in our case). There exists two main families of ways to address deadlocks: **deadlock prevention** and **deadlock avoidance**. Both approaches will be considered in this chapter.

In order to avoid deadlocks, we may be constrained to decide, statically or at run time, to reduce the asynchronism by introducing some synchronizations or to increase the size of the communication memory.

We have observed in Chapter 6 that the synchronization occurring during the transition between child and parent fronts in multifrontal chains is harmful for performance, as it brutally breaks the computation pipeline and the communication flow of processes. It is thus critical to limit as much as possible synchronisations introduced to prevent or avoid deadlocks as they represent a bottleneck to performance. Applied to our multifrontal factorization, we will see that we must compromise between memory and time performance, while preventing deadlocks.

One typical and simple situation that we want to avoid is described in the following. Let us assume that a process receives a message "too early" with a simple broadcast communication scheme (loops of *MPI_ISends*) during a factorization. If this process has exhausted all its communication memory and still has to receive a message upon which the early message depends, then we have reached a deadlock situation.

The communications related to the factorization of a given front (See Algorithm C.1) iteratively repeats the same pattern of communication (even with tree-based broadcasts). MPI guarantees in that case that messages will be received in the same order that they have been sent. As messages are sent from the master in a defined order, they are also received in the same order by the workers. The abovementioned case of deadlock thus cannot result from such messages only. However, when we consider the communications of the whole multifrontal factorization, there is *a priori* no reason for the messages to be received in the order they have been sent, as they may come from different processes and as MPI does not guarantee a causality dependency property (illustrated in the next paragraph). The previous case of deadlock may thus happen even with single loops of *MPI_ISend*.

To explain this more clearly, let us consider the following example (See Figure 8.1) to illustrate where causality dependency is not respected. Let A, B and C be three processes. A sends message msg1 to C. Then, it sends message msg2 to B, which will in turn sends message msg3 to C. Given the order of send events, C may need to receive msg1 before msg3. Unfortunately, in an asynchronous environment, the opposite may happen, i.e. C may receive msg3 before receiving msg1. This situation is problematic if C is not able to treat msg3 prior to having received and treated msg1. This may be the case for example when the task related to msg3 depends on the completion of the task related to msg1. Two approaches may then be adopted to solve this problem.

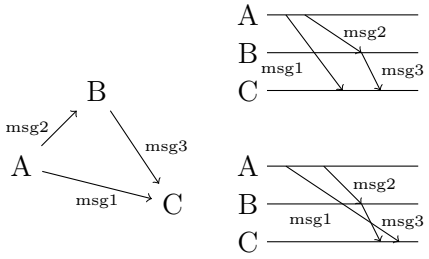


Figure 8.1: Example of a situation where messages are received in a different order from how they were sent.

On the one hand, C may continue in a fully asynchronous spirit to receive any message. It then takes the risk of receiving an early message that differs from the expected message msg1. In the case where the memory dedicated to communications is limited, C may run out of memory without having received msg1 which would lead to a deadlock.

On the other hand, C may decide to post a blocking reception on the missing message msg1. When choosing this approach, C must have the guarantee that msg1 has already been sent. In our case, we know that if C receives msg3 from B, it means that B has previously received msg2 from A. Thus, as we know that A sends msg2 only after having sent msg1, C may safely do a blocking reception on msg1.

From the previous example, and more generally to guarantee that performing a blocking receive will not introduce deadlocks (referred to as *Safe blocking receive property* in the following), sent messages need be ordered to respect causality and temporary memory.

With a tree-based broadcast communication scheme, it is much more difficult to guarantee the safe blocking receive property since some relay processes are involved in the communication, making difficult to guarantee that the causality dependency is also respected by the intermediate processes.

In order to illustrate this, let us first explore a simple example leading to a deadlock. Let x, y, $S1$, $S2$, a and b be processes involved in the computation of two independent fronts F_1 and F_2 with broadcast trees TF_1 and TF_2, respectively. Figure 8.2 shows branches of TF_1 and TF_2, together with the corresponding dependency graph between processes, exhibiting a cycle (in red) between $S1$ and $S2$.

Let us assume that, similarly to the model in Section C.4, each process has one 'receive' buffer and one 'send' buffer. For the sake of simplicity, each buffer will have enough space for one message only (associated with some panel of the factorization of the front – See Algorithm C.1). If $S1$ receives a message from x and if $S2$ receives a message from y, then both $S1$ and $S2$ will copy their newly received message from their receive-buffer to their send-buffer to relay it to $S2$ and $S1$, respectively. However, if $S1$ receives another message from x before the one from $S2$

and if $S2$ receives another message from y before the one of $S1$, then the receive/send buffer of both $S1$ and $S2$ will be full and none of these processes will be able to relay/receive its message to/from each other. Both processes are then in deadlock.

Now let us assume that more memory is available for communications, such that each buffer can store more than one message at a time (say n messages). In this case, we may (only temporarily as explained later) escape from this deadlock.

Intuitively, we can predict that if the master of a front F_1 produces panels quicker than that of a front F_2 – because F_1 is smaller than F_2 and/or because the master of F_1 is less loaded than F_2 (the latter possibly working on other fronts in parallel) or if the network link between a relay and its predecessor in TF_2 is saturated or slower than with its predecessor in TF_1 – then, a relay in TF_1 and TF_2 may be flooded by messages of F_1 which will lead to a deadlock similar to the previous one, no matter how much (limited) memory is dedicated to communications on this relay process. Intuitively, we can predict that dedicating more memory to communications can only help stepping aside from deadlocks; the extreme case being to have infinite memory available for communications, in which case deadlocks cannot occur.

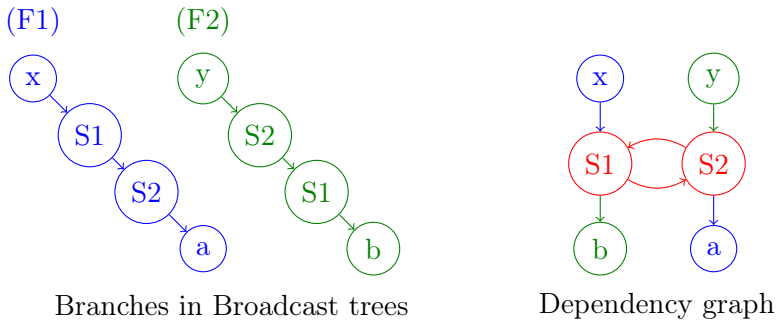


Figure 8.2: Example of a deadlock.

8.1.1 Motivation

The previous example shows that the size of the memory for communications directly affects deadlock situations. However, when scaling in terms of matrix sizes, the number of columns assigned to each process will increase. Moreover, when scaling in terms of number of processes, the number of assigned rows per process will reduce even if the number of rows in each panel ($npan$) may remain constant. Thus, when both the problem size and the number of processes increase, the relative size of memory for communications (to store panels) might increase. Typically, on a front in double complex arithmetic, with $nfront = 10^5$, $nproc = 128$ and $npan = 128$ (see notations in Section C.5.1), the memory needed to store the part of the front for each process is $nfront^2/nproc = 1.16\ GB$ while the memory needed to store only a single panel might be (in case of local equilibrated memory based scheduling) $nfront * npan = 190\ MB$, namely 16% of the memory for the front, which is not negligible. The total memory for communications may even be larger, as we would like to use more than a panel to overlap communications and computations.

8.1.2 Goal

In this chapter, for a given controlled/limited communication memory constraint per process, we explain how to preserve parallelism by pushing the asynchronism paradigm as far as possible while still preventing / avoiding deadlocks.

The prime purpose of this study is to improve asynchronous sparse multifrontal parallel solvers. However, its impact is more general and could for example be useful in the case of runtime systems. Limited communication memory and related deadlocks are critical issues that have to be addressed in runtime systems and we hope that the proposed work can also be of interest in this context.

8.1.3 Theory of deadlocks

Before diving into the heart of our problem, let us first present the basics of the theory of synchronizations and deadlocks. Deadlocks are related to resource racing and sharing, even though the notion of processes holding resources in distributed-memory is different from that in shared-memory. Indeed, since the resources are spread among different processes in distributed-memory, a process x can still access its local resources but cannot directly use a resource r located on a distinct process y. Nevertheless, the semantic of x getting r still exists. It means that y saves resource r for x by doing a blocking receive on a message from x that will be stored in r

In the general theory of deadlocks, the four conditions, known as the *Coffman conditions* [35], are necessary and sufficient to lead to deadlocks when they hold simultaneously.

1. **Mutual exclusion:** *A resource can be held by at most one process.* In our distributed asynchronous context, resources correspond to communication buffers. When a process x can receive messages from many processes, the buffer on x is not blocked for any of them and thus there is no mutual exclusion in this case. However, when x is constrained to do a blocking receive on a message from y, its buffer, say resource r, will be exclusively dedicated to y, and cannot be used for any other purpose until the message arrives. Here lies the mutual exclusion on resource r which means that process x reserves its resource r so that r cannot be used to communicate with any other process (note that freeing resource r depends on an event produced by process y).

2. **Hold and wait:** *Processes that already hold resources can wait for another resource.* This is always true in our context since many processes may do receives on messages from the same source x. For instance, in a distributed-memory environment, one such process, let us say y, may allocate a resource that it dedicates to (receiving a message from) x, which is thus *held* by x. Meanwhile, x may *wait* for another process, let us say z, and allocate a resource thus dedicated to z. Therefore, x holds a resource of y while it waits for a resource of z.

3. **Non-preemption:** *A resource, once granted, cannot be taken away.* This is also always true in our context, as processes do not use the same buffer to receive simultaneously two messages.

4. **Circular wait:** *Two or more processes are waiting for resources held by one of the other processes.* This condition is probably the most important one to consider as it lies at the heart of deadlocks. It typically corresponds in our asynchronous distributed memory context to loops of blocking receives.

Also in the general theory of deadlocks, four handling strategies exist:

1. **Ignorance:** Ignore the problem and assume that a deadlock will never occur.

2. **Detection:** Let a deadlock occur, detect it, and then deal with it. This usually means aborting and restarting one or more processes causing the deadlock. Fault tolerant and

resilient algorithms like checkpointing or replication may be used to handle such situations, if we consider that a deadlock is a kind of failure.

3. **Prevention:** Make a deadlock impossible by granting requests so that one of the necessary conditions for deadlock does not hold. As the first to third conditions are often defined by the environment, the condition we usually try to forbid is the existence of cyclic dependencies.

4. **Avoidance:** Choose resource allocation carefully so that deadlock will not occur. Resource requests can be honored as long as the system remains in a safe (non-deadlock) state after resources are allocated.

Deadlock prevention is a static approach that does not take into account the specificities of the system, while deadlock avoidance is a more dynamic approach that tries to take advantage from the state of the system to make a better use of the resources.

8.1.4 Models and general assumptions

Although the four aforementioned solutions work in the more general context of our asynchronous multifrontal solver with all features already introduced (other messages, sophisticated remapping, ...), for the sake of simplicity, we focus on communications related to the factorization step. Thus, without loss of generality, the only communications that we consider are the asynchronous intra-tasks communications, so called *Block of Factors* messages (that we denote by BF – See Algorithm C.1), which follow the broadcast tree pattern described in Section 5.4.

For our parallel tasks, we consider the *moldable* model of tasks. Moldable and malleable tasks [79] are tasks that can be executed in parallel on several processes. The difference is that the number of processes used in moldable tasks is defined before the tasks start and cannot be changed once started; whereas malleable tasks can change their processes during execution. Moreover, for our communication buffers, we assume they can be used for receiving as well as for sending messages. Furthermore, one key notion in the following is the *resource dependency graph*. A *process dependency graph* links processes depending on the connections between them in terms of communication dependencies. A resource dependency graph more precisely describes the links between the resources used by the processes. Of course, when all the processes have only one resource each, the two graphs perfectly overlap. In the following, both process and resource dependency graphs will be used and more precisely defined.

Finally, we assume the following hypotheses that will be further referred to in this chapter:

(H1) Computation and relay operations associated with a message are atomic. In particular, a message arriving too soon is not relayed before local operations are done.

(H2) At each node of a broadcast tree, if memory is available on all successors, the message is sent to all of them in the broadcast tree (send to all or to none of them).

(H3) If m_1 is sent from P_i to P_j before m_2, then m_1 is received by P_j before m_2.

Let us remind the reader that the main notations used in this chapter are described in Table C.1, C.2 and C.3.

We first propose in Section 8.2 families of solutions to the problem of deadlocks. We introduce deadlock prevention and deadlock avoidance families, both under minimal and limited and controlled amounts of resources. We then study in Section 8.3 ways of increasing performance for deadlock-free solutions and focus on the construction of broadcast trees that maximize the pipeline parallelism.

8.2 Deadlock issues

We give a spectrum of solutions, covering different levels of resource requirement. These solutions belong to the two main families of approaches: deadlock prevention and deadlock avoidance. In the following, by "resource", we will denote the working memory, including buffer memory, needed by a process to execute all of its (active) tasks.

8.2.1 Deadlock prevention solutions

The idea behind deadlock prevention is to ensure that, at least, one of the four necessary conditions for deadlock will not hold. We will focus on the *circular wait* condition and on the existence of cycles.

We present two types of solution. The first aims at building broadcast trees so as never to create cycles in the resource dependency graph. The second aims at breaking cycles whenever they occur. Within this second type, we first present a solution which is simple both from a theoretical and a practical but which is potentially more resource-consuming. Then, we present an improved solution which is more complex but more resource-friendly.

8.2.1.1 Global order on processes in broadcast trees

We know that trees (particularly broadcast trees) are acyclic. Moreover, we have observed that, under our assumptions, cycles occurring in the dependency graph result from the merging of all the broadcast trees. Consequently, eliminating cycles can be achieved by building broadcast trees in such a way that the overall dependency graph remains acyclic.

In order to reach this goal, we can set a global order on processes where all broadcast trees would represent a partial order respecting the global one. Basically, this means that, when considering separately any branch of any broadcast tree, from the root to a leaf, the order of processes in this tree must respect the global order (See Figure 8.3). If each process has a unique ID, this could be a simple global order of the processes.

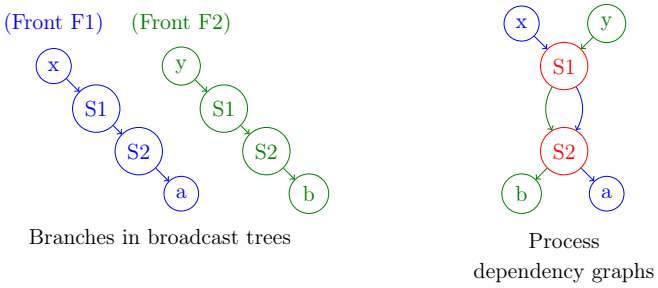


Figure 8.3: Effect of a global order on processes in the construction of broadcast trees. In the example of deadlock presented in the introduction, if we consider that S1 is of higher order than S2, then the global order on processes was not respected in the broadcast tree of front F2, as S2 is a predecessor of S1 there, thus creating a cycle between them. Now, when swapping S1 and S2 in that branch of the tree, no more cycle remain in the process dependency graph.

Even though this technique would prevent deadlocks, it is quite constraining. For instance, if the master process of a task − which is the root of the corresponding broadcast tree of this task − is not higher in the order than some processes involved in that task, the global order could not be respected. This thus enforces a very strong constraint on the mapping of the masters of each task. Moreover, for performance reasons, as will be shown in Section 8.3, the

relative position of processes in broadcast trees could be made dependent on their mapping in the fronts, as presented in Chapter 7. In that case, respecting a global order on processes will become inappropriate. However, the global order property is very interesting. It allow us to minimize the number of cycles in the resource dependency graph and thus to minimize the potential risk of deadlocks. Therefore, we still try to enforce the global order property as much as we can so long as it does not conflict with important properties for performance.

8.2.1.2 One buffer per active front on each process

Cycles in the resource dependency graph most often occur when a process uses the same resource for communication for more than one front. Thus, one straightforward solution for eliminating potential cycles is to make each process allocate one buffer per active front in which it is involved and to dedicate it to communications related to that front. The underlying idea is that, no matter whether cycles exist or not in the process dependency graph, there will be no cycle in the resource dependency graph. This comes from the fact that there are no cycles in individual broadcast trees. There will be dependencies only on resources related to the same front (See Figure 8.4).

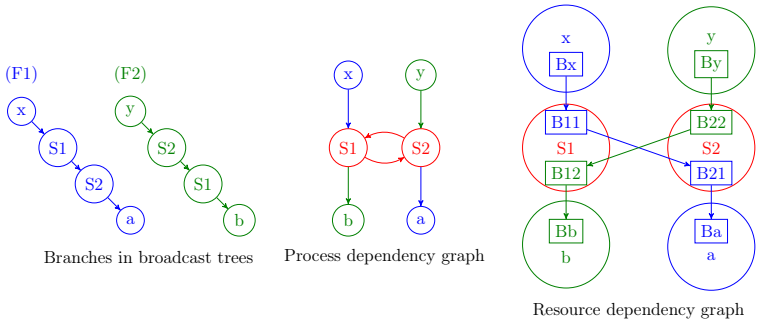


Figure 8.4: Deadlock prevention by allocating one buffer per active front on each process. Cycles in the process dependency graph do not exist any more in the resource dependency graph. B_{ij}: buffer of S_i dedicated to front j.

The obvious drawback of this solution is that it may use more memory than necessary or waste memory that could be better used. If a process receives messages at a higher rate from one front compared to the others (for example, because the master of that front is faster or because the bandwidth from the predecessor in the associated broadcast tree is higher), the fact of using only the buffer dedicated to that front is globally inefficient.

8.2.1.3 One additional buffer per cycle

Let us improve the previous solution by reducing the memory requirement. Avoiding cycles in the process dependency graph is very hard, particularly in a distributed asynchronous environment where scheduling and mapping decisions are taken locally and dynamically. However, given some freedom on the number of resources to be allocated on processes, it is possible to avoid cycles in the resource dependency graph. Indeed, duplicating the memory resource of a process present in a cycle is equivalent to duplicating **virtually** this process on two virtual processes. Thus, as no dependency may exist between these two virtual processes, the duplication has the effect of **breaking** the cycle (See Figure 8.5).

Based on this observation, in order to break all the cycles, our strategy consists in allocating an one memory buffer per cycle on a well-chosen process. This buffer is allocated in addition to one buffer that must be allocated per process.

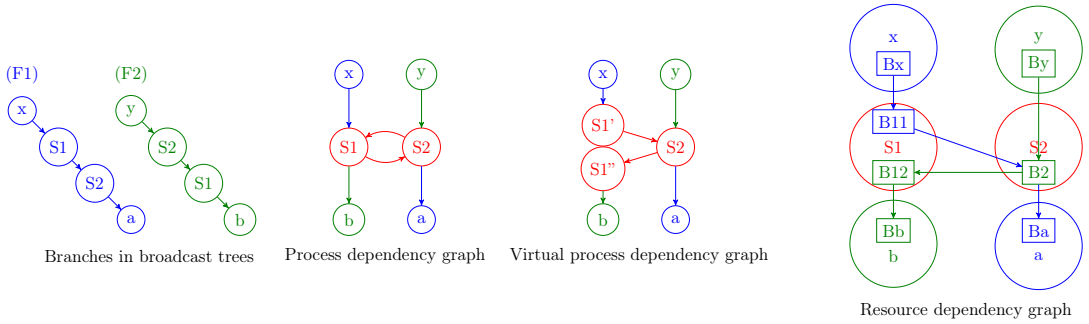


Figure 8.5: Deadlock prevention by allocating one additional buffer per cycle. Process 1, which is involved in a cycle, is allocated one additional buffer. We can thus see this operation as virtually splitting process 1 into two virtual processes, 1' and 1", thus breaking the cycle.

A first problem that arises is the **detection of cycles**. Simple algorithms exist in shared-memory environments, such as the Tarjan algorithm [107] which detects strongly connected components of a graph. In distributed-memory environments, however, more complicated algorithms must be used. One algorithm by Boukerche and Tropper [26] finds whether or not a node in a distributed graph is in a cycle. Another algorithm by Manivannan and Singhal [88] has the extra capability of finding the set of nodes in the cycle.

A second problem encountered, after the detection of a cycle, is the **election of a leader** process within this cycle. This leader will be the one which allocates an extra memory buffer to break the cycle. We could decide to make each process in the cycle allocate one extra memory buffer. However, this would be closer to the previous deadlock prevention solution (one buffer per active front on each process) and unnecessarily memory consuming. In distributed-memory, having enough aggregated global memory on all processes does not necessarily mean that enough per-process memory is available. Thus, it is preferable to allocate the additional memory buffer on a process located on a machine node with the most (or enough) available memory. The Bully Election Algorithm and the Invitation Election Algorithm are leader-election algorithms, proposed by Garcia-Molina [51] and improved by Stoller [106]. These algorithms rely on a notion of a priority attributed to processes. They set this priority as the (unique) ID of processes (sort of global order on processes). We may thus take advantage of this priority mechanism by setting process priorities to the amount of memory available on their related machine nodes.

8.2.2 Deadlock avoidance solutions

The idea behind deadlock avoidance is to grant a resource request only if this allocation may not lead to a deadlock. In other words, this means that there will always exist a way to avoid potential deadlocks in the future, even though it may be costly. Formally, we define the safe and the unsafe states. A safe state is one where no deadlock will ever happen; whereas an unsafe state is one where deadlocks may happen. Deadlock avoidance solutions always ensure a safe state and so avoid any risk of deadlock.

In the following, we present one deadlock avoidance solution which may naturally adapt to memory constraints while still reaching good overall performance. We start by presenting the key property upon which we will rely to guarantee that our system is always in a safe state. We then explain the implications on performance and parallelism of this property when there are various levels of resource constraints.

8.2.2.1 Global order on tasks

In order to guarantee that we can avoid any potential deadlock, we introduce the *global order* property as the fundamental property which will enable us to avoid deadlocks and which we must always respect to ensure a safe state.

Property 8.1. *Let $T_{i\in\{1..n\}}$ be n independent tasks. Let $<$ be a global order on them. Let $P_{i\in\{1..p\}}$ be p processes on which the tasks T_i are mapped.*

If, each time a process P_i realizes that only one communication resource remains free (others being busy), it will dedicate that resource only to messages (reception, relay) associated with the smallest task (relatively to the global order $<$) on which it is mapped, then deadlocks will be avoided.

Proof. One necessary condition for the occurrence of deadlocks is the presence of cycles in the processes dependency graph. Since no cycles exist within a single broadcast tree, a cycle may occur only between distinct broadcast trees. Thus, in any cycle, there is at least one process x that is involved in communications related to two broadcast trees simultaneously. If x respects an order between tasks when it has critically low buffer memory, it will avoid communication in the broadcast tree related to the biggest task. It thus removes the corresponding dependency, so long as the one related to the smallest task still exists. Thus, x breaks the cycle. Therefore, there will be no deadlock. □

When infinite resources are available on a process, that process is guarantee never to be a source of deadlock: any resource already used will simply not be requested until it is freed, hence avoiding any circular dependency. However, when only one resource is available on a process, any circular wait dependency on it will inevitably lead to a deadlock. In *rescue mode*, i.e. when respecting a global order on the tasks to grant resources, processes are guarantee not to create a circular dependency on their resources, thus remaining in a safe state. Moreover, as we have a solution to avoid deadlocks when only one resource remains, we are not compelled to follow the global order on tasks when two or more resources are available. This extra degree of freedom can be (and will be) exploited for performance purposes.

The spirit of the global order property is similar to that of the classical deadlock prevention mechanism [62] negating the circular wait condition by fixing a global order on the different kinds of available resources that processes must follow when requesting these resources.

The idea behind the global order property is similar to the one used in memory-aware approaches [98] in multifrontal methods, which, instead of dealing with communication memory, deal with the working memory for computations. When enough memory resources are available, there is some freedom on the choice of the tasks to be executed. Whereas, when only a minimal amount of memory is available, the order of the execution of tasks is constrained to those ensuring minimal overall memory consumption [57, 58, 84, 98]. An example of such an order is the postorder defined by Liu for this purpose in the sequential case.

8.2.2.2 Application of global task order and impact of task graph topology

In order to illustrate the behaviour of our deadlock avoidance approach, we now show how it can be applied in practice on different types of task graphs. In the examples below, we limit the number of buffers to only one per process, in order to reach more rapidly critical cases usually leading to deadlocks, and to show how the global oder may be efficiently applied to exploit the parallelism arising from different task-graph topology, even with this lowest level of resources.

Series task graph (Chain) When tasks belongs to a series graph (or are linked in a chain in elimination trees), a global order comes naturally from the graph dependency (topological order in multifrontal method). Because of this global order, a process is then forced to finish the work in the preceding (child) task before it can start the work in the (parent) task. It may even not be allowed to start the successive task in the task graph immediately after the current one because of other processes not having finished their work in preceding tasks (independent constraints of transparent remapping). Accordingly, using an all-to-all mapping (all available processes on all tasks) is better than an arbitrary mapping (arbitrary sets of processes mapped on arbitrary tasks), as it is better to make all processes work all the time rather than to make some of them idle some of the time. However, this holds only as long as we consider a model of moldable tasks, where adding computational resources on a task helps decrease its computation time. Indeed, this may generally be the case up to a certain limit on the number of processes, where communications start to cost more than computations. Therefore, in series graphs, the constraint on parallelism is not solely fixed by the synchronization need but is inherent to the graph structure.

Parallel task graph (Independent) When tasks are independent, in a parallel task graph (or in branches of the elimination tree), any ordering on them could be chosen as a global order. The level of process overlapping between tasks depends on the mapping of the processes onto the tasks. On one extreme case where disjoint sets of processes are mapped onto each task (e.g.: proportional mapping), tasks are fully independent as there is no link between them and as no link may exist between processes treating them. Furthermore, even in the other extreme case where all processes are mapped on all tasks (all-to-all mapping), some parallelism may still exist, which is a pipelined parallelism. Typically, the process at the root of the broadcast tree of the first task may start the successive task(s) well before the processes at the leaves of that broadcast tree finish their work in the first task. An illustration of this phenomenon when all *IBcast* trees are the same is shown in Figure 8.6. The degree of parallelism depends both on the depth of the broadcast trees and on the number of panels and the time to transmit these panels by the processes. Hence, in a hypothetical example where only one panel is to be relayed in each task and where broadcast trees of all tasks are the same, the number of tasks the root process is ahead on (compared to a leaf process) is simply the number of processes separating them in the broadcast tree (roughly the height of the tree). Therefore, in parallel graphs, the sole synchronization need does not necessarily represent a heavy constraint on parallelism.

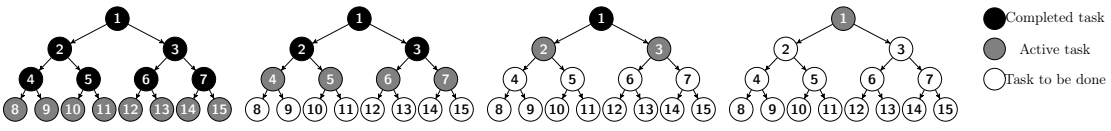


Figure 8.6: Pipelined parallelism on the computation of four independent tasks by processes mapped on all of them using an all-on-one mapping. We can see that, when the processes at the leaves of the broadcast tree of the first (left-most) task are still working on it, the root of that broadcast tree is much ahead on them, and is already computing the last (right-most) task. All the processes in between in the broadcast tree are also active on middle tasks.

Series-Parallel task graph (Tree) From the results on series graphs and parallel graphs, we can deduce that any topological order in a series-parallel graph is a valid global order. The relative order between children tasks does not matter, as long as each of them is ordered before its parent task. Moreover, any mapping, which separates in disjoint sets the processes to be

mapped on different branches of a parallel graph and which maps all processes on all tasks on a series graph, is desirable (e.g.: proportional mapping or Prasanna-Musikus mapping [95]).

8.2.2.3 Deadlock avoidance vs. deadlock prevention approaches

As we have seen, granting only one extra communication buffer to processes is enough to avoid deadlocks thanks to Property 8.1. However, it is not ideal for performance as it does not allow for an overlapping of communications and computations. Using two buffers instead of one, a technique known as *double buffering* will be enough to offer potential for overlapping.

Deadlock avoidance offers some valuable advantages compared to deadlock prevention. Firstly, when communication memory is limited rather than being controlled, it is not always possible to grant the necessary communication memory on all processes in a deadlock prevention approach, as the number of active tasks per process is not limited. In contrast, the deadlock avoidance approach is able to adapt itself to the amount of available memory on each process. The approach simply consists in letting each process receive messages, as soon as they come, from any predecessor, in any broadcast tree, to let the system be as fluid as possible, until the moment where only one resource remains available. Processes must then switch to a rescue mode where they dedicate their last resource to the highest priority task. The deadlock avoidance approach is able to consume less overall resources than deadlock prevention approaches. Secondly, deadlock avoidance makes better use of the memory resource than deadlock prevention, since in deadlock avoidance, a given memory buffer of a given process could be used for communications related to any front, while the deadlock prevention approach tends to dedicate buffers to active fronts on cycles. This makes deadlock avoidance more reactive and more asynchronous, able to adapt to dynamic variations of the system (network congestions, ...).

An inconvenience of deadlock avoidance is that, when a process is in rescue mode, it could block and delay many other processes working on other common and less prioritized tasks. We expect this situation to be very temporary, and to end as soon as more resources become available. In the next section, we target performance issues that could further improve this situation.

8.3 Performance issues

In the previous section, we focused on the deadlock problem, the solutions of which introduced some constraints that can be viewed as partial synchronization constraints. In this section, we focus on performance issues under these previously introduced constraints.

Forcing processes to follow a global order on tasks is a way of solving the deadlock problem. However, it can reduce the performance, as it forces an order on message transmission, and more importantly, on their reception too. Thus, when a process has little remaining communication memory and receives too soon a message of a low priority task, it is forced to wait for the reception of messages from higher priority tasks (both in order to continue its computations and in order to be able to treat the early message) and to free its related buffer. However, the process has no idea whether these messages have already been sent or not, nor whether they will be sent soon or not. In this context and in order to still reach good overall performance, we must bound the penalty resulting from the global order.

From a local (per process) point of view, we want to reduce as much as possible the time a process has to wait for a given message. Ideally, we would like the following property (closely related to the safe blocking receive property introduced in Section 8.1) to hold for any process at any time.

Property 8.2 (No wait)**.** *A process is allowed to do a blocking reception (when it needs to) on a specific message only if it has the guarantee that the message has already been sent.*

Hence, rather than waiting for a message indefinitely, without knowing when it will be sent, a process will wait for it with the guarantee that it has already been sent. The major advantage of enforcing such a property is that the time a process will have to wait for a message is then bounded by the transmission time of the message, which does not depend on the dependency graph of the application but only on the network topology, bandwidth and load. Moreover, on messages smaller than a certain user-defined threshold, another advantage from using MPI is that, when these messages are sent by a sender process before the corresponding receives are posted by the receiver process, MPI mechanisms may effectively transmit the messages from the sender side to the receiver side, and store them temporarily in local MPI buffers. Hence, when the receiver process finally posts the receive operations, the messages are already there and can be copied into user-defined buffers. However, we cannot rely on such a mechanism on large messages like panels or contributions as we will not be able to control the memory allocated by MPI.

From a global point of view, the main bottleneck to performance in our asynchronous pipelined factorization, as observed in previous chapters, is the breaking of the processes' execution flow and communication pipeline.

In order to eliminate this, we first make (see Section 8.3.1) Property 8.2 be respected, without introducing any side effects on performance. A second step (see Section 8.3.2) will consist in carefully choosing broadcast trees whose construction and shape will impact positively the execution flows of processes and their communication pipelines. One way to achieve this is to make processes transitions between different tasks as smooth and as fast as possible.

8.3.1 Impact of broadcast trees on performance

8.3.1.1 Effect of broadcast trees on pipelining

Let us first illustrate the importance of the relative shape of broadcast trees of successive tasks on the pipelining of computation and communication.

Figure 8.7 shows a chain with a child C and a parent P. Two different sets of corresponding broadcast trees are presented, TC1 and TP1, or TC2 and TP2. Moreover, in order to avoid remapping, a communication-free mapping is considered (see Section 6.3).

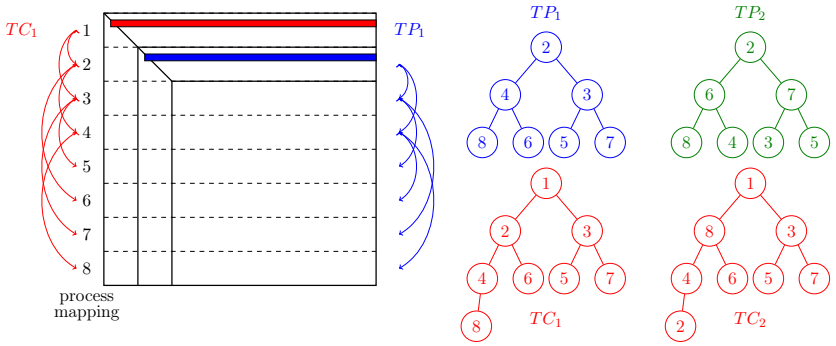


Figure 8.7: 1D pipelined factorization and broadcast trees: TC_1 and TC_2 for the child, TP_1 and TP_2 for the parent. We assume here that process mapping remains unchanged between C and P so that the root of TC does not work in TP.

The fact that red panels must be computed and treated before blue ones is naturally represented by a causality link between TC and TP, formally defined as follows.

Definition 8.1 (Causality link)**.** *Let TC and TP be two broadcast trees. We define the child-parent causality link between TC and TP by the relation: $\forall P_i \in TP$, if $P_i \in TC$, then all activities of P_i in TC must be finished before any activity of P_i in TP can start.*

In other words, a process cannot treat nor relay messages from P before all messages from C have been treated. This causality link is the expression of a global order between C and P resulting from the multifrontal task dependency graph. It may thus naturally be extended to the case of the artifical dependencies induced by the global order on tasks. The necessity of respecting the causality link may depend on the mapping of the processes. If we use a communication-free mapping, a process keeps the same rows between C and P and cannot start working in P before it ends its work in C. It must thus respect the causality link. Using other mappings, a process may have no common row between C and P. It could thus start working in P before finishing working in C. We thus assume that the causality link should be respected and will explain why this will influence the shape of the broadcast trees. An important point to notice is that the causality link does not represent an order on all the tasks of C and P (strong synchronization between all processes) but is rather an order on their related local tasks (on each process).

Figure 8.8 shows the Gantt-charts of executions when using each set of broadcast trees.

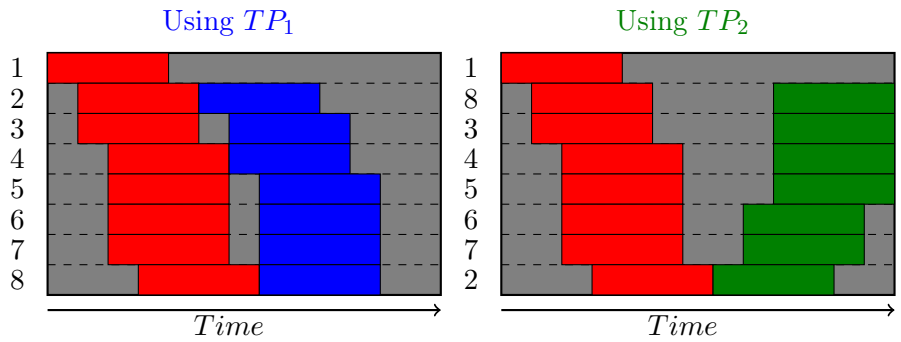


Figure 8.8: Gantt-charts of successive front factorizations: child (red); parent (blue or green); idle periods (gray).

On the one hand, the use of TC_1 and TP_1 makes a smooth transition between C and P for all processes. Indeed, none of them could have started in P earlier. The only reason why process 3 (for example) waits a little is because it ended its computations in C at the same time as process 2, and must thus wait for the computation and relay of the first panel of P by process 2.

On the other hand, the use of TC_2 and TP_2 is disastrous for the pipelines. All processes must wait for the last process in the pipeline of C (process 2) before starting a new pipeline in P. Thus, even if we use the most advanced deadlock prevention or avoidance techniques, the use of TC_2 and TP_2 is the same as applying a strong synchronization between the processes in C (see Section 6.3.5).

This example shows the impact of the shape of successive broadcast trees on performance. Some more general observations can be drawn from the aforementioned example.

On the one hand, as process 2 is the root of TP_1 and the direct successor of the root in TC_1, it is able to start its work in P very soon, making BF(P)s available to other processes as soon as possible. On the other hand, if process 2 is the root of TP_2 but also the deepest leaf in TC_2, it makes all the processes wait for it before the can start their work in P.

The master process of a front is always the root of the corresponding broadcast tree. Hence, it must be ready to start before any other process in the front. Moreover, the time it takes a process to finish its work in a front depends on its position in the broadcast tree: the higher, the sooner. Thus, in order to make MP start as soon as possible, we must always set MP in C as the direct successor of MC in TC.

Furthermore, the case when process 2 makes all other processes wait is an extreme case of a more general situation where a node in a low position in TC is mapped higher in TP.

A process H mapped high in TC would terminate in C before a process L mapped low in TC. Thus, if L is a predecessor (direct or not) of H, then H may lose time if it has to wait for L to finish in C before starting relaying in P. Hence, it is preferable to match the mapping of processes in TC and TP. In other words, processes which are mapped high in TC should also be mapped high in TP, and *vice versa*.

Additionally, because of the structure of TC_1 and TP_1, we are sure that process 4 will not receive a Block of Factors message from P, BF(P), in TP_1 before the last BF(C) is received. Property 8.2 is guarantied, since the predecessor of process 4 in TP_1 (process 2) is the same as its predecessor in TC_1, and since MPI ensures that messages sent from the same source to the same destination arrives in order. Property 8.2 also holds for process 6 in TP_1 even if $pred_P(6) \neq pred_C(6)$. Indeed, let us suppose that process 6 needs to preserve the causality link, and thus to receive a BF(C) while it has just received a BF(P). As process 2 starts relaying in P to process 4 (and process 4 to process 6), it means that process 2 has already relayed all its BF(C)s in TC (particularly to process 6), because to respect the causality link, as it is the master of P, it just has to wait to finish all its treatments in C before deciding to start its work in P. However, there is no way to ensure Property 8.2 in the case of TC_2 and TP_2. We can already see that broadcast trees, if well chosen, could help ensure Property 8.2. We will see that it is always possible to do so.

8.3.1.2 Characterisation of compatible broadcast trees

Let us characterise what makes broadcast trees respect Property 8.2.

Definition 8.2 (IB-compatibility)**.** *Let TC and TP be two broadcast trees. TP is said to be* **IB-compatible** *with TC if, $\forall N \in TP \cap TC \setminus \{root(TP)\}, \exists A \in \{$ancestors of N in $TP\}$, s.t. $A \in$ subtree in TC rooted at $pred_{TC}(N)$, the predecessor of N in TC.*

As demonstrated in the following, IB-compatibility is a property of broadcast trees guaranteeing that, whenever a process does a blocking receive on a specific message, the corresponding send operation is guarantieed to have already been submitted.

Property 8.3 (Local no wait)**.** *Let C and P be a child and parent such that TP is IB-compatible with TC. If a process P_i in TP performs a blocking receive on a message in TC to respect causality links, the expected message has already been sent.*

Proof. Let P_i be a process mapped on $NC \in TC$ and on $NP \in TP$ (NC being a node on C and NP a node on P), which has received a message from $pred_T P(NP)$, but has not yet finished its work in TC. Respecting causality links implies that, as soon as P_i has only a single buffer

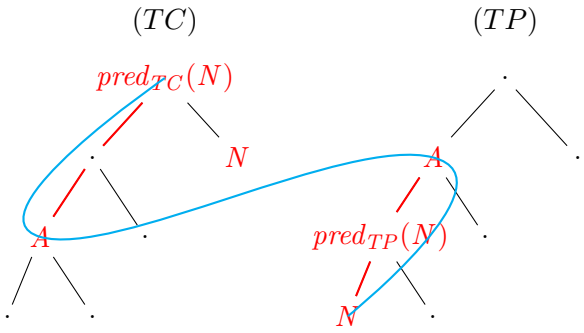


Figure 8.9: Illustration of IB-compatibility between the broadcast tree of a child (TC) and that of its parent (TP)

resource available, it must post the reception and must treat messages `msg` in TC (coming from $pred_{TC}(NC)$). The only way to guarantee that a message `msg` has already been sent is to find a path linking this event "P_i has posted the reception of `msg` from $pred_{TC}(NC)$" with the event "$pred_{TC}(NC)$ has sent `msg` to NC". As the reception of a message of TP from $pred_{TP}(NP)$ by P_i means that all the ancestor processes of P_i in TP have relayed all the messages in TC (respect of causality link) and as one of P_i's ancestors (A) in TP is also mapped in TC in the subtree rooted at $pred_{TC}(NC)$ (IB-compatibility of TP with TC), this implies that all the processes between this ancestor and $pred_{TC}(NC)$ in TC have relayed all the messages in TC (in particular `msg`). Hence, `msg` is guaranteed to have been sent already. More precisely, it has been sent by $pred_{TC}(NC)$ to the sibling of the NC subtree that contains A, and thus – thanks to hypothesis **(H2)** (see Section8.1.4) – it has also been sent to NC. □

Property 8.3 applies on parent and child nodes. Nonetheless, its validity extends to a whole chain of fronts, or to any succession of fronts respecting a global order on them.

Property 8.4 (Global no wait)**.** *Let $(F_i)_{i\in1,\dots,n}$ be a chain of fronts and let $(TF_i)_{i\in1,\dots,n}$ be their corresponding pairwise IB-compatible broadcast trees. If a process P_k in TF_j performs a blocking receive on a message in TF_i after having received a message in TF_j, to respect causality links (F_i being thus a smaller task than F_j), then the message in TF_i has already been sent.*

Proof. Firstly, P_k cannot be the root of the broadcast tree of any task F_x with $F_i < F_x < F_j$. Indeed, if this were the case, P_k could not have had received a message from F_j, as this front could not have been activated before F_x. Hence, P_k has necessarily at least one ancestor in the broadcast tree of each task F_x with $F_i < F_x < F_j$. Secondly, as all broadcast trees are pairwise IB-compatible, it means that there exists in TF_j an ancestor A_j of P_k that is also in the subtree of TF_{j-1} rooted at $pred_{j-1}(P_k)$. If P_k receives a message from $pred_j(P_k)$ in TF_j, this thus means that $pred_{j-1}(P_k)$ has already sent all the messages of F_{j-1} to P_k. Thus, by applying the same reasoning to all the broadcast trees from TF_{j-1} to TF_i, we prove that $pred_i(P_k)$ has already sent to P_k all the messages in TF_i. □

Furthermore, the cases where the notion of IB-compatibility applies are not limited to the case of chains. The property applies to any chain in a tree, corresponding at each level to one parent front and one among many of its children. Thus, broadcast trees of a parent P and its child C1 can be IB-compatible while those of P and C2 are not.

8.3.2 *A*synchronous *B*road*C*ast (ABC_w) Trees

With IB-compatibility at hand as a criterion to ensure Property 8.2, we define **ABC$_w$ trees** by:

Definition 8.3 (ABC_w trees)**.** *ABC_w trees are broadcast trees in a series of fronts (either a simple series or in a series inside a tree) respecting the following properties:*

(#1) IB-compatibility of the broadcast tree of the next front with that of the current front;

(#2) Fixed width w (which could be determined by network topology);

(#3) Minimal height and balanced subtrees (difference in the number of nodes of at most 1 between subtrees at the same level);

(#4) Maximum overlap between successive child and parent pipelines (minimum idle time between two successive fronts).

ABC$_w$ trees are broadcast trees whose purpose is not to optimize *single broadcast* operations, but rather to optimize *a sequence of broadcasts* of successive node factorizations in multifrontal trees.

By its definition, it is not possible to characterise the broadcast tree of a single front as an ABC_w tree, since IB-compatibility only makes sense on broadcast trees of successive fronts (in multifrontal chains or trees). However, all the other properties are intrinsic to each broadcast tree.

(#2) comes from the fact that the right width of a broadcast tree depends on the network topology, among other things. We could imagine that when each machine node of a computer has n network links, it could be advisable to choose $w = n - 1$ (for all nodes except the root of the broadcast tree, one link for the reception of BFs and $n-1$ links for its relay). However, some network contention may happen so the best broadcast tree width could be different. Indeed, the processes we have to relay to are not necessarily processes mapped on machine nodes directly linked to the current one. Moreover, it is not always possible to guarantee that all subtrees will have exactly w children as the total number of processes may not be a power of w. In such cases, each set of sibling leaves in the tree may have a cardinality smaller than w. In the remainder of this chapter, we will illustrate examples of ABC_w trees with $w = 2$, without loss of generality.

(#3) comes from the simple remark that when using pipelined communications, the total execution time is the sum of the start and stop pipeline costs plus the effective time of a single process communication. By minimizing the height of broadcast trees, start and stop costs are minimized. Moreover, coupled with (#2), this property is equivalent to balancing all subtrees of the broadcast tree. Otherwise, if one subtree is much deeper than another, the processes on the shortest subtree could be forced to wait for the others, which would be contrary to (#4).

(#4) means that the time a process has to wait between a parent and a child task should be minimal. This property was illustrated in Figure 8.8. If (#4) is respected, the overall computations could be finished much sooner than in other cases. It is thus of paramount importance.

We explain in Section 8.3.2.1 and Section 8.3.2.2 how to build such ABC$_w$ trees, or at least pipeline-friendly broadcast trees, in multifrontal chains and trees, respectively.

8.3.2.1 ABC_w trees in multifrontal chains

Let us now show how to create ABC$_w$ trees on multifrontal chains, depending on the mapping of the processes in them.

ABC_w trees and communication-free mapping. When the communication-free mapping is used in a chain, at the transition from a child front C to its parent front P, the root of TC is systematically lost and the root of TP is necessarily a direct successor of the root of TC in TC.

Moreover, starting from the definition of IB-compatibility (cf. Definition 8.2), we know that $\forall P_i \in TP$, $pred_{TP}(P_i)$ should be in $subtree_{TC}(pred_{TC}(P_i))$. Hence, $pred_{TP}(P_i)$ may be one level higher than P_i in TC, thus being $pred_{TC}(P_i)$, but cannot be in a higher level.

Furthermore, setting $pred_{TP}(P_i)$ in a lower level in TC than P_i could harm performance. Indeed, P_i would then finish its work in C earlier than $pred_{TP}(P_i)$, and would then have to wait for it to start working in P.

Thus, to find the right balance when building TP from TC, we guarantee that $pred_{TP}(P_i)$ is either $pred_{TC}(P_i)$ or, at least, a sibling of P_i in TC, thanks to the **Ascension** mechanism:

Definition 8.4 (Ascension)**.** *An* **Ascension** *is an operation on broadcast trees that consists in first choosing the longest chain of nodes in the tree starting from the root to a leaf, then, promoting each node of this chain (i.e. replacing each parent node by its child node in the chain).*

Let us show that ascensions applied on an ABC_w tree of C effectively generate an ABC_w tree for P.

Property 8.5. *Given a child C and a parent P, if TP is built from TC through an ascension, then TP is IB-compatible with TC.*

Proof. Firstly, the nodes on the chain of promoted nodes (in red in Figure 8.10) all have the same predecessor, except of course the child of the former root which becomes the new root. Secondly, the nodes having no link with the nodes in the chain (in green in Figure 8.10) will stay unchanged and will keep the same predecessor. Finally, the processors that are not in the chain but whose predecessor is in the chain (in blue in Figure 8.10) will have as new predecessor their sibling in the chain. In all cases, IB-compatibility is thus maintained. □

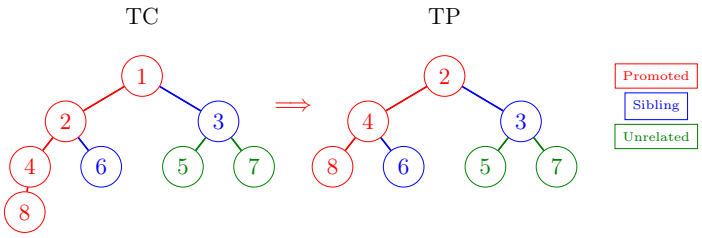


Figure 8.10: Example of Ascension

We have shown that the ascension mechanism maintains (#1) between child and parent broadcast trees. Moreover, it also respects (#4) because most processes in TP will keep the same predecessor as in TC, and the few whose predecessor changes have as their new predecessor their siblings in TC, which will make them wait only a little during the transition between C and P. Furthermore, it also maintains (#2) and (#3) as the promotion all the processes on a single and longest chain does not change the shape of the tree (except for the leaf of the chain, which is removed but does not harm (#2) nor (#3)).

So far, we have not taken into account the relative position of processes in broadcast trees with respect to their position in fronts, even though a tight link exists between them. Indeed, the master of a front is at the same time the process mapped on the first rows of the front and the root of the corresponding broadcast tree. When n successive communication-free mappings are

applied to a split chain, the position of processes in the successive broadcast trees must be such that the process mapped at the n^{th} position in the initial front is the root of the n^{th} broadcast tree in the chain (after n successive ascensions). One way to meet this condition is to create the structure of the broadcast tree of this first front, and apply as much ascensions as there are processes. The mapping of the processes in the first front will then dictate the mapping of the root process of each broadcast tree, and thus, of all the processes in the initial broadcast tree.

ABC_w trees and restart remapping. After a restart operation, lost or unused processes are inserted in the parent front P in the chain.

Firstly, what is the effect of the insertion of a new process in an ABC_w tree on IB-compatibility?

The process to be inserted may be a completely new process, never involved before in any front of the chain. It may have also been previously involved in a former front of the chain, necessarily being the root of the corresponding broadcast tree at its removal, as it would not have been dropped otherwise. In either case, this process does not need IB-compatibility of TP with TC as it has no message from older fronts to wait for.

Moreover, a new process may be inserted anywhere in TC to build TP without harming the IB-compatibility of TP with TC, because this does not break existing paths between processes and their ancestors in TC but only extends them.

Furthermore, for the same reason, inserting any number of newly available processes anywhere in TC will result in a tree TP which still is IB-compatible with TC.

Secondly, where to insert newly available processes when building an ABC_w tree for P?

In order to build TP from TC, we could first apply an ascension on TC to remove MC (Master of Child) and to build an intermediate IB-compatible broadcast tree. In this tree we will then insert all the newly available processes, one by one. At each insertion, some care must be taken to respect properties (#2) and (#3), while any random insertion will still respect (#4). Indeed, by inserting new processes, they will either wait or make the others wait in P, depending on whether they are located at the top or at the bottom of the tree. No configuration can make all the processes start as soon as they can, due to the extension of the size of the pipeline with the increase in the number of processes in the new broadcast tree. However, as new processes are usually less loaded than the others, it would be preferable to insert them at the top of the new broadcast tree, as they will be potentially more reactive than others for the treatment and relay of messages of P, and as they will give time for the others to finish their remaining work in C, if needed. In practice, however, given the link between processes mapping in the fronts and in the corresponding broadcast tree, the position of newly available processes in the broadcast tree is dictated by the restart mapping resulting from the *MinAsmComm* algorithm discussed in Chapter 7.

8.3.2.2 ABC_w trees in multifrontal trees

The construction of ABC_w trees in multifrontal trees is harder than in multifrontal chains, if not impossible. We now present two operations, that we denote by **merge** and **fusion**, which try to build, whenever possible, ABC_w trees in multifrontal trees or, at least, broadcast trees with smooth transitions between children and parent fronts.

The merge operation The number and mapping of processes in the broadcast trees of the children fronts $C_1, \cdots, C_n$ influence the feasibility of an ABC_w tree for the parent front P.

In the general case, where even if only a subset of processes of P is spread over the Cs in a random way, it is not always possible to find an ABC_w tree for P. Indeed, if X is a predecessor of Y in C_1 and Y a predecessor of X in C_2, for example, it is not possible to build an ABC_w for P: even if the produced tree is IB-compatible with that of one child, it will necessarily not be with that of the other one. In the special case of mappings separating processes of each multifrontal subtree under P into disjoint sets (e.g.: proportional mapping), one solution to build TP is to **merge** the broadcast trees of the children.

Let us consider the case of a parent P with only two children C_1 and C_2, with $w = 2$. We would like to merge TC_1 and TC_2 into a single tree TP (See Figure 8.11). We first apply an ascension on the biggest TC, let us say TC_1, and note the resulting temporary tree as TC'_1. Instead of throwing away the root of TC_1, as we would normally do, we will reuse it as the future root of TP (denoted by root(TP)). This is possible only if $root(TP) = MP$. We then insert both TC'_1 and TC_2 as the subtrees of root(TP) to obtain TP.

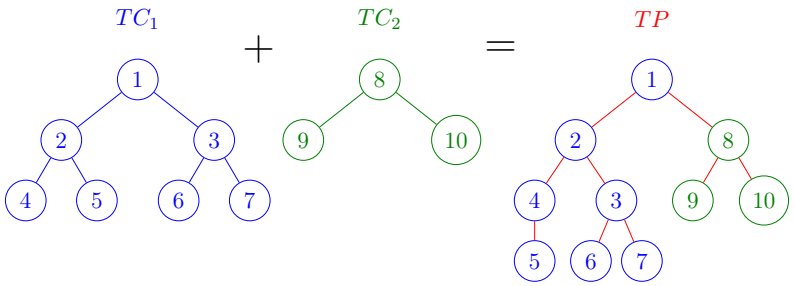


Figure 8.11: Merge operation between child broadcast trees TC_1 and TC_2 to produce parent broadcast tree TP

The broadcast tree produced this way respects (#1), by construction, but does not necessarily respect (#2), (#3) and (#4). If the number of processes is unbalanced between the Cs, TP will not respect (#3). Although we could remove processes from the heaviest tree (TC_1) and insert them in the other tree (TC_2), this would make property (#3) be respected but would harm property (#1).

The case of two children fronts and of binary broadcast trees is the simplest one. The merge operation would necessarily be more complicated in other cases. We will not enter into the details of its adaptation in these cases, but it could still be applied there.

The fusion operation From the previous remarks, we abandon trying to find a method for building ABC_w trees on multifrontal trees, as such methods may exist only in special cases. We will thus not try to respect the IB-compatibility property. We could then either not try to ensure Property 8.2 at all, or, we could use minimal additional synchronizations between parent and children fronts, when necessary, in order to still ensure it. We will not enter into the details of these mechanisms, as their explanation is not mandatory for the understanding of the remapping. The idea is to artificially add the missing causality links that would ensure Property 8.2. As the master process of each front knows the broadcast tree structure of that front, all child masters could send their broadcast tree to the parent master. This master will then construct the missing causality links between all the processes involved in P and all the processes involved in the Cs. It would then notify all the processes in the Cs and in P about the existence of these missing links, so that each process in a child C could send a special notification message to a corresponding process in P, notifying it as to whether it is authorized to relay the messages in P. The same, processes in P would have to wait for all the notifications from processes in the Cs to be able to start relaying the messages in P.

Whatever solution we adopt, we could build broadcast trees that reduce as much as possible the break of pipelines (trying to respect (#4)). A simple observation is that, in order to make smooth transitions between children and parent broadcast trees, processes mapped high in a child broadcast tree should be mapped high in the parent broadcast tree too. If they are mapped low in another child broadcast tree, they should be mapped low in the parent broadcast tree, as finishing the work in a child is more critical than starting the work in a parent. The fusion operation (Algorithm 8.1) relies on this observation to build TP from the TCs (see Figure 8.12).

1: $lst \leftarrow \{\}$
2: **for** $depth \in [|MaxDepthTCs, \cdots, 1|]$ **do**
3: **for** $C \in Cs$ **do**
4: **for** $proc \in$ [list of processes at depth $depth$ in TC] **do**
5: **if** $proc \notin lst$ **then**
6: $lst.append(proc)$
7: **end if**
8: **end for**
9: **end for**
10: **end for**
11: $lst.revert()$
12: Build a structure of TP which is well balanced and whose width is w
13: **for** $node \in TP$ following a breadth-first traversal **do**
14: $TP(node) \leftarrow lst(node)$
15: **end for**

Algorithm 8.1: Construction of TP from TCs preserving (#2) and (#3)

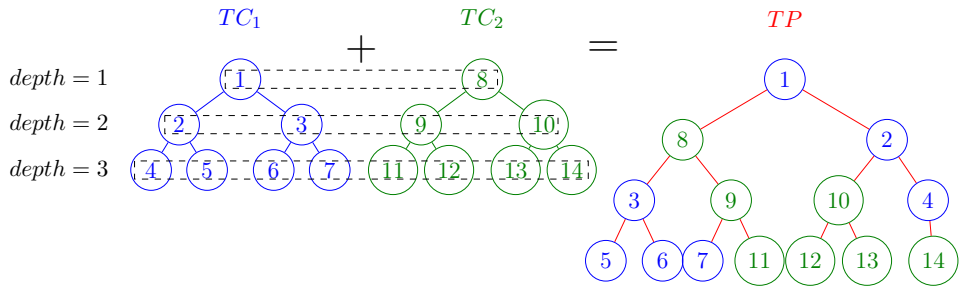


Figure 8.12: Fusion operation between child broadcast trees TC_1 and TC_2 to produce parent broadcast tree TP. At step 11 of Algorithm 8.1, the *lst* list contain $[1, 8, 2, 3, 9, 10, 4, 5, 6, 7, 11, 12, 13, 14]$

The resulting broadcast tree produced by Algorithm 8.1 has the advantage of respecting (#2) and (#3), and tries to respect (#4) as much as possible, but does not respect (#1). This means that the algorithm tries to make communications and computations overlap well from a globally point of view, but that this is not guarantied from a local point of view.

8.4 (Re)Mapping-aware synchronization issues

8.4.1 Static mapping with static remapping

In the previous sections, we supposed that there was no remapping, or that the remapping was hidden, handled by a transparent mechanism. In this section, we consider its relationship with task computations.

One important observation concerning remapping, or, in the multifrontal terminology, assembly, is that it involves pairs of processes only. Assemblies neither follow any broadcast tree, nor any structured pattern. An assembly operation occurs between two processes, a and b. Usually, a sends a contribution from task T_i to b working on task T_j, with T_i having to be computed before (of higher priority than) T_j. When $a \equiv b$, the assembly is said to be local, no communication is needed and only local copies are involved. In the general case, remappings could follow patterns. However, we will focus hereafter only on the multifrontal case.

8.4.1.1 Deadlock prevention

A simple approach to prevent deadlocks during remapping communications would be to isolate the two types of communication: those from remappings and those from computations. This could be done by physically separating the resources used by each of them. Hence, no matter what solution is chosen from the previous ones to make computation communications deadlock-free, additional separated memory buffers should be allocated and dedicated to remapping communications. This is a reasonable request because remapping messages are symbolic information of small size.

Due to their communication patterns, cycles may occur between any pair of processes a and b. This happens when a sends contributions to b and *vice versa*. We assume that during remapping messages, processes send and exploit directly (without relaying) remapping received messages. The following property can then be established.

Property 8.6. *If each process dedicates one 'send' and one 'receive' buffer for remapping purposes, remapping messages will introduce no deadlock.*

Proof. As the 'receive' (resp. 'send') buffer of any process is exclusively dedicated to receptions (resp. emissions), no cycle can result. Dependencies in the process' resource dependency graph only flow in (resp. out of) it. As this is the case for all processes, no deadlock may happen because of remapping. □

8.4.1.2 Deadlock avoidance

One may want to use the same buffer resources both for computation and remapping communications. In this case, some order must be respected between BFs and CBT2s. In addition to the order on BFs dictated by the global order on tasks, an order on CBT2s could be defined by giving the priority to sends of CBT2s of the highest priority task. However, as BFs require one buffer only while CBT2s require two (one for sending and the other for receiving), and as there is no interleaving between BF sends and CBT2 sends for a given front, it is natural to ask whether this second buffer for CBT2s could be used for BF sends as well. The critical issue here is to ensure that, if used by BFs, it will be freed later in case it is needed for CBT2s. If we cannot ensure this necessary condition, then it will have to be left unused when sending BFs.

If a process is to use the second CBT2 buffer for BF purposes, it means that it will find itself in rescue mode, as only one buffer will be remaining. Thus, it will be compelled not only to receive and relay BFs but also to send and receive CBT2s of higher priority tasks than that of the messages currently waiting to be sent and consuming memory buffers. If two processes must exchange CBT2s and are both in this same situation, then, there will be a deadlock. Hence, if a process uses the second buffer on a task that is not the most prioritized, a deadlock may occur, as all the CBT2s that had to be sent or received first had not been treated. On the other hand, if the process consumes its second buffer while working on its highest priority task, it is just a matter of time before this buffer is freed. Moreover, as long as the second buffer is not freed and even if the process is ready for, it should not start to send CBT2s relative to the current (highest

priority) task. However, it is advisable to concentrate exclusively on the reception of any CBT2s from other processes, in order to help them free their own buffers to pursue their work. Forcing an order of CBT2s with BFs may not be a good solution. In many cases, one would want to send them only when it becomes necessary, because they will consume memory on the receiver side, not in buffers where they will only be transient, but in computational memory. Hence, if the task that needs their data is not yet active (or activable), they will consume unnecessary extra memory. Special messages could be used to inform the processes when to send CTB2s relative to a given task. Hence, we may avoid enforcing an order between BFs and CBT2s. The condition that allows us to use the second CBT2 buffer for BFs is to have already sent all the CBT2s on tasks of higher priority than the ones that will be used by the second buffer.

8.4.2 Dynamic mapping with dynamic remapping

In the previous section, we assumed the mapping of processes on tasks was static. However, in many applications, this is not the case. Indeed, a dynamic mapping is often needed to respond to discrepancies between the performance model of the application and the actual computations. Even though a dynamic mapping could rely on a static mapping, corrections may be necessary, for example due to numerical pivoting issues.

Usually, dynamic mapping requires the introduction of new kinds of messages. However, such messages can be treated immediately and are usually small enough so we need not worry about the communication memory they require.

One problem that arises when the mapping is dynamic, and hence not known *a priori*, is that it is no longer possible to guarantee a deadlock-free solution through a global order on tasks. Processes having a local vision of their mapping, could decide to dedicate the last of their communication resources to the current highest priority task they are mapped on, before discovering afterwards that they have been mapped dynamically on other tasks of still higher priority in the global order.

In such a situation, the mechanism used to notify processes of their mapping on a given task should be extended to notify the other processes that they are not mapped on that task. Thus, each process initially supposes that it could be mapped on any task. Hence, it must respect the global order on tasks, but can gradually remove them from its list of potential tasks to be able to use its safety memory buffer on the true highest priority task it is mapped on. Moreover, in applications like `MUMPS`, a notion of candidate mapping is used, where the list of potential tasks a process can be mapped on is much reduced compared to that of all possible tasks. This may greatly help processes get a correct vision of the order of tasks each one must respect.

8.5 Conclusion

In this chapter, we have targeted the main issue of the asynchronous approach, that is to say: deadlocks. After having identified the sources of this problem in our environment, we have proposed different solutions within the deadlock prevention and deadlock avoidance families of solutions, with varying levels of complexity and of resource requirements. Then, we have targeted the efficiency issue under the constraints raised by the previous solutions. Without careful attention regarding the management of communications, even the most advanced deadlock solution proves to be no more efficient than a naive strategy. We have thus identified properties ensuring minimum levels of efficiency and then characterized communication patterns guaranteeing them together with some ways to achieving these properties.

The work of this chapter aims to be more general than the scope of this study. It targets the context of asynchronous environments. In a short term perspective, it may be applied effec-

tively to the case of symmetric multifrontal factorizations, where more complex computations and communications take place. In the longer term, it may also be applied to asynchronous runtime systems. Indeed, for the moment, such environments use large amounts of resources, ensuring them to be deadlock-free. To target larger problems on modern computers (which have a decreasing amount of memory per processor), we believe that the analysis and the approach proposed in this chapter could also be of interest for limited memory runtime scheduling in asynchronous distributed-memory environments.

Chapter 9

Application to an asynchronous sparse multifrontal solver

9.1 Introduction

Many strategies have been adopted in the past to prevent and avoid deadlocks in the asynchronous multifrontal environment of MUMPS described in Chapter C, which corresponds to the MUMPS solver. We will describe in the present chapter the most relevant ones, and try to explain them in the light of the theory of deadlocks previously discussed in Chapter 8. Then, we will explain the new mechanisms we have adopted to apply the minimal synchronization strategies that we presented in Chapter 8. Finally, we will give preliminary experimental results obtained on both single fronts and on whole multifrontal trees, after applying various contributions that have been described in this thesis.

Let us remind the reader that the main notations used in this chapter are described in Tables C.1, C.2 and C.3.

9.2 Former synchronization solutions in MUMPS

The two major sources of deadlocks encountered were the existence of cycles between processes during the partial factorization of parallel fronts and that some messages could overtake each other. In this section, we present the solutions that were previously adopted.

In the traditional asynchronous approach, any message could be received from any process and at any time. Given that only one buffer is dedicated to receptions (see Chapter C), only messages that could be immediately processed should be received. This is the case either when their processing is self-contained or when all the other messages on which they depend have already been received.

However, in an asynchronous environment, this cannot be guaranteed as the order of reception of messages cannot be controlled due to the non-determinism of distributed-memory communications. Thus, when some message *msg* arrives too early, we are obliged, first, to allocate dynamically a temporary buffer and, second, to receive and process one or more specific messages required for the treatment of *msg*. Otherwise, this could lead to a situation in which one might need to allocate a second, third, or more additional buffers.

In order to control the memory of the receive-buffer, we cannot continue to allow for the reception of random messages. We are then forced to do blocking receives, on the right messages, from the right processes and in the right order. However, this will inevitably lead to deadlocks if the process is involved in a cyclic dependency. We will end up exactly in the critical situation

previously mentioned in Chapter 8 and which was solved by entering a safety mode.

An example of such a situation is when a worker receives prematurely a panel of a certain front from its predecessor in the corresponding broadcast tree while the assembly operations related to this front on this worker have not yet been finished or have not even started. Contribution blocks of that front must be received and assembled before the process is able to treat the panel.

The fundamental rule we have relied on so far for avoiding deadlocks while waiting for specific messages is Property 8.2 which we recall as:

A process is allowed to do a blocking receive on a specific message when it needs to only if it has the guarantee that the message has already been sent.

This way, no matter how much time the message will take to arrive, we are ascertained that it will arrive. Respecting this rule is not always feasible, given that, in distributed memory and particularly in an asynchronous environment, we cannot know whether a foreign process has already sent a given message. The only way to do so from the perspective of a given process is to try to deduce it, knowing which messages this process had already received in the past and knowing the deterministic path of execution all processes must follow during the processing of fronts (as described in Chapter C, the introduction of Part II of this thesis).

Figure 9.1 is a Gantt chart showing the causality links between events representing message transmission and reception on related processes for related fronts. More precisely, it is the DAG of events linking master (M) and worker (S) processes of child (C), parent (P) and grand-parent (G) fronts of a chain. The X-axis represents the time, or the evolution of the system. The Y-axis represents the processes MC, MP, MG, SC, SP and SG. Circles represent events. The blue ones correspond to the emission of messages, whereas the green ones correspond to their reception. The kind of a message, its source and destination are noted in the center, bottom right and upper right of the circle, respectively. For example, $M2M^{MP}_{MC}$ is an M2M messsage, sent by MC and received by MP. The purpose of the figure is to illustrate the path that connects events, allowing a process to link the reception of a certain premature message with the fact that the desired message has already been sent.

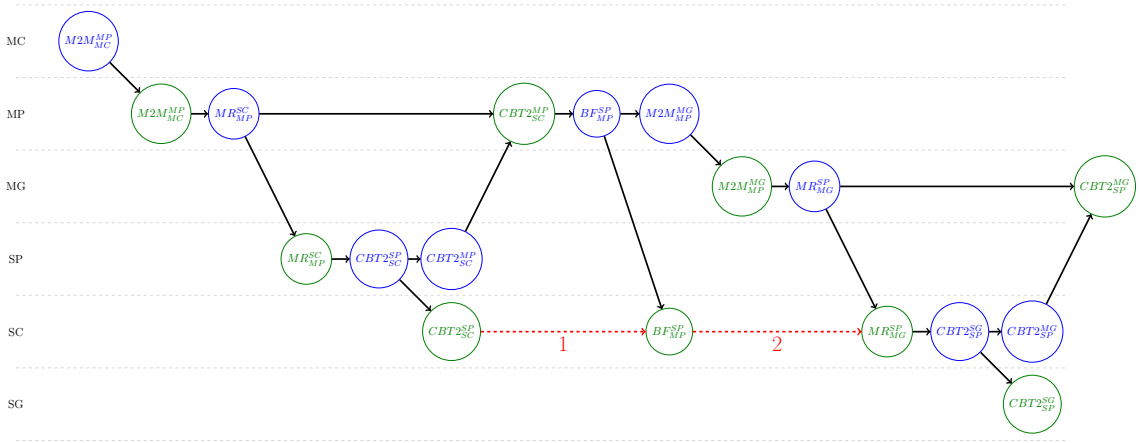


Figure 9.1: Causality links (black) between events representing messages emission (blue) and reception (green). A red dashed arrow between reception events A and B means that, if reception B occurs before reception A, then it is possible to safely perform a blocking reception on A, as the message is guaranteed to have already been sent.

Let us consider the interaction of BF (Block of Factors) and CBT2 (ContributionBlock) messages in the two cases of panel transmissions, i.e. without and with the IBcast communication scheme. In the former case, we consider both flat and binary broadcast trees.

- **Before the use of asynchronous broadcast communications**, i.e. when the master was sending the panels to all its workers, we were able to guarantee Property 8.2 for the blocking reception of missing contribution blocks after the premature reception of a panel by a worker (SP). In other words, there was a path (dashed red arrow number 1) linking the premature reception of BF_{MP}^{SP} (BF message from MP to SP) to the transmission of $CBT2_{SC}^{SP}$ (CBT2 messages from SCs to SP). Indeed, when looking at Figure 9.1, we can see that the reception of BF_{MP}^{SP} by SP implies its emission by MP. This means that MP has received all the $CBT2_{SC}^{MP}$ from SCs. This also means that all SCs sent them. To create the required missing link, we added to SCs the extra constraint of sending $CBT2_{SC}^{MP}$ only after having sent all $CBT2_{SC}^{SP}$ to all SPs, which is the condition we are looking for. Therefore, SP can safely post a blocking reception on the missing CBT2 message(s) and is guaranteed to receive it (them).

- **After the introduction of asynchronous broadcast communications**,

  - the DAG of causality links has changed. Instead of having only one direct link between MP and all SPs, for the communication of BF_{MP}^{SP}, we now have some SPs (SP1, or direct successors of MP) which still have this link. However, other workers (SP2) now have this link being replaced by a link from SP1 to SP2, for the transmission of BF_{SP1}^{SP2}, corresponding to workers relaying BF messages to other workers. We thus break the previous chain of causality links, and can no longer enforce Property 8.2. The solution that was previously adopted was to create artificial synchronization messages, called ENDNIV2. The last relays of each broadcast tree must send such a message to their master after having received and relayed all their BF messages. It is only when receiving such messages that MC is allowed to send the M2M message authorizing MP to start its work. This technique prevents the creation of dependency cycles between successive fronts (series graph);
  - however, cycles may still exist between independent fronts (parallel graph). The typical situation here is when two processes X and Y are both mapped on two independent fronts F_1 and F_2, X being a predecessor of Y in F_1 and Y being a predecessor of X in F_2, thus creating a cycle. In order to avoid this situation, the solution that was previously adopted was to limit the depth of broadcast trees to two levels [98], i.e with one level of relay only. Doing this, and because leaves of broadcast trees may not generate cycles, the only possibility is to have a cycle between a master and its direct successors. Thus, a rule was added, that consists in avoiding choosing as a relay a process that could be a master on a future front. Since a relay in F_1 may currently be a master on a front F_2 which started earlier, this rule ensures that the master of F_1 is not a relay in F_2.

We now consider another example of a problematic situation corresponding to the interaction of MR (MapRow) and BF (Block of Factors) messages. Because of the non-determinism of message receptions, an MR message can be received prematurely by SPs while some missing BF messages have not yet arrived, on which the SPs must thus do blocking receptions.

- When using the **non-broadcast** communication scheme, we can see in Figure 9.1 that a causality link (dashed red arrow number 2) exists between the reception of MR_{MG}^{SP} and the fact that all the BF_{MP}^{SP} have already been sent. Indeed, the reception of MR_{MG}^{SP} means its transmission. This, in turns, means that MG received $M2M_{MP}^{MG}$. This also means that MP has finished sending all the BF_{MP}^{SP} messages, which guarantees Property 8.2. This implies that SPs can safely do blocking receptions on them.

- However, when using the **broadcast** communication scheme, the transmission of BF_{MP}^{SP1} does not necessarily imply the relay of BF_{SP1}^{SP2}. In this case, the introduction of the ENDNIV2 messages mentioned in the previous section, once again artificially creates the missing causality link which ensures Property 8.2.

An example of another problematic situation, independent from the aforementioned non-determinism of distributed-memory communications, is the overtaking of messages. Even though MPI guarantees that:

Property 9.1. *Messages sent from a process X to a process Y with a tag TAG on an* MPI *communicator COMM are received in the same order as that with which they have been sent.*

This problem of message overtaking is caused by the relative order of treatment of messages compared to their order of reception. Typically, when a process X is unable to relay a BF message msg_1 to a process Y, because its send-buffer is full, it enters then in recursion, trying to receive and treat any message, until some space in its send-buffer becomes free. During this recursion, it may happen that some space becomes available in its send-buffer so that it receives another BF message, M2, to relay. It will thus relay M2, before leaving the recursion, going back to the point where it finally relay M1. Thus, because of the way of handling memory difficulties with a recursion scheme, message overtaking could still occur even with Property 9.1. The simple solution that was previously adopted in this case was to prohibit the reception and relaying of any BF message if another one had already been attempted (and failed) to be sent.

9.3 Proposed minimal synchronization solutions in `MUMPS`

The ENDNIV2 mechanism induces heavy synchronizations which are harmful for performance. In this section, we propose an implementation of new synchronization solutions to solve the problems described above. These new solutions are based on the discussion in Chapter 8. Firstly, we need to be able to uniquely identify messages. Secondly, we must be able to control at a fine grain level the management of communication buffers. Thirdly, we must be able to take the correct local scheduling decisions.

9.3.1 Identification of messages with fronts

In order to implement deadlock-free solutions, and particularly to respect the global order described in Section 8.2.2.1, we must adapt the way of receiving BF messages and be able to do blocking receptions on specific messages. Indeed, the asynchronous mechanism of receiving any message, of any type and from any process inevitably leads to deadlocks, as the amount of communication resources cannot be controlled in this way.

Firstly, we must be able to receive only BF messages and from specific sources. Instead of receiving messages from MPI_ANY_TAG and MPI_ANY_SOURCE, the semantic of MPI allows us to specify the BF tag and the source from which to receive a specific BF message. Secondly, we must be able to receive BF messages related to specific fronts only. Indeed, if processes X and Y are both working on two parallel fronts F_1 and F_2, and if X is the direct predecessor of Y on both fronts, then Y cannot specify to receive BF messages from the front F_1 only, as BF messages from X could correspond either to that from F_1 or to that from F_2. As the MPI standard specifies only 32 different tags, even though implementations we are aware of allow at least 65536 different tags, the MPI tag mechanism is not sufficient, as the number of fronts in multifrontal trees may be much larger. In our current implementation, the number of parallel fronts (treated by several processes) is much lower than the total number of fronts,

making this solution acceptable for the moment. We propose in Appendix A.2 another solution, which allows us to uniquely identify BF messages of specific fronts for a given pair of processes. This latter solution has the advantage of further reducing the number of different tags used.

9.3.2 Communication buffer management system

In our earlier implementation, only a single communication buffer is dedicated to the reception of messages and one buffer is used for the transmission of messages. The size of the receive-buffer is chosen to accommodate the largest BF message. However, CBT2 messages, will potentially be too large for the buffer. They are thus split into smaller chunks and sent separately.

In order to implement new synchronization solutions as presented in Chapter 8, we must rethink the management of communication buffers. The life cycle of a BF message may be summarized as follows: first, a BF message is received in the receive-buffer; second, if it is not yet ready to be treated, a new dynamic buffer is allocated, where the message is temporarily copied; third, if (when) it is ready to be treated, the message is unpacked from its current buffer to a temporary memory zone, where it will be used for computations; fourth, if the message needs to be relayed to other workers, the message is packed back into the send-buffer and is sent. As we can see, a BF message is currently copied too many times, which is harmful for the locality of data and in terms of cost of those copies. To avoid this, we want to use the same buffers for the reception, computation, and transmission of BF messages. We still use the former buffer management for all other messages except BF messages for which we prescribe a new type of management discussed next.

Instead of relying on two large statically allocated buffers, we now rely on several small dynamically allocated ones for handling BF messages. Since, at any given moment, many fronts are active on each process, we define on each process and for each active front a data structure *BF_STRUC_T* (see Figure 9.2), containing a set of buffers associated with that front (defined in a *BUFF_STRUC_T* data structures) and some management data. In the remaining, we explain in detail each of these two data structures. Whether we dedicate buffers to specific fronts or use them on any front, depends on the kind of synchronization we adopt. In order to have a proof of concept code, and for the simplicity of its implementation, we have chosen the deadlock prevention solution with at least as many buffers as there are active fronts per process (Section 8.2.1.2). However, the mechanisms we present here are easily adaptable to any other solution and, in particular, to deadlock avoidance solutions (Section 8.2.2).

The following Fortran code represents the two main data structures that allow us to handle buffers: *BF_STRUC_T* and *BUFF_STRUC_T*.

Each buffer is represented as an array of integers (which we call *BUFFER*), even though it will store not only integers but also arithmetic dependent scalars (e.g. double precision floats). Instead of using the *MPI_PACK* and *MPI_UNPACK* routines, we copy messages into the buffers directly in a row format using the intrinsic FORTRAN procedure *TRANSFER*.

Moreover, with each buffer an array of MPI_REQUESTs (which we call *REQUESTS*) is associated. Depending on the state of the buffer, the entries of this array store either receive or send requests. When the process posts an *MPI_IRecv* on a BF message, the resulting MPI_REQUEST associated with the MPI_IRecv call is stored in the first field of the array.

```
TYPE BUFF_STRUC_T
   INTEGER, POINTER, DIMENSION(:)  :: BUFFER
   INTEGER, POINTER, DIMENSION(:)  :: REQUESTS
   CHARACTER                       :: STATE
END TYPE BUFF_STRUC_T

TYPE BF_STRUC_T
   TYPE (BUFF_STRUC_T), POINTER, DIMENSION(:)  :: BUFFERS
   INTEGER  :: TAG
   LOGICAL  :: IS_MASTER
   LOGICAL  :: IS_ASSEMBLED
   LOGICAL  :: IS_ZOMBIE
   LOGICAL  :: IS_IRECV_TO_BE_POSTED
END TYPE BF_STRUC_T
```

Figure 9.2: *BF_STRUC_T* and *BUFF_STRUC_T* data structures.

When a message is already in the buffer (either because the associated process is a master that directly copied a panel into it, or because it is a worker that has already received a BF message and now wants to relay the message to other workers), all the MPI requests associated with each MPI_Isend are stored in the *REQUESTS* array.

Furthermore, in order to manage the BF buffers on all processes in an asynchronous way with a complete separation of computation handling and communication handling, we have decided to manage each buffer using an automata. A state automata of a system is a graph where each vertex represents a state of the system and where each edge represents an allowed action leading to a transition from one vertex to another. Indeed, we associate with each buffer a state (*STATE*) (states of the automata) on which management routines will rely to accomplish the correct transition operations (edges of the automata). All the buffer management mechanisms rely on this idea of automata depicted in Figure 9.3.

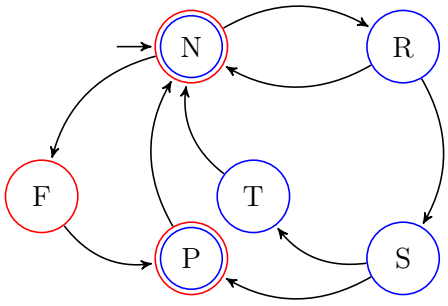


Figure 9.3: State automata describing the management of BF buffers. The state N marked with an incoming arrow is the entry point, or starting state, of the automaton.

The automaton associated with a buffer is different whether the process ithe buffer is allocated on is the root of a broadcast tree (master) or not (worker). In Figure 9.3, we represent, in red,

states associated with a root (master) of a broadcast tree and, in blue, states associated with a worker. The meaning of each state is as follows:

State	Explanation
F	I (master) am currently **F**illing the buffer
R	an MPI_I**R**ecv request has been posted on the buffer
S	the MPI_I**S**end requests have been posted on the buffer
P	the message in the buffer has been **P**rocessed, i.e. received and treated, and MPI_Isend operations have been posted (if any)
T	the message in the buffer has been received and the MPI_Isend operations have been posted (if any), but I still have to **T**reat the message
N	**N**ull: nothing to do and the buffer is free

Let us explain the different states of a buffer by describing the possible transitions. When a buffer is associated with a front (or a single message), it is initially empty (or may even not yet be allocated). It is in the state **N**.

When a master process wants to send a panel, it requests the management system to give it an empty buffer. If this request is granted, the given buffer passes from state **N** to **F**. Once the panel is copied into the buffer, the master sends the message to the workers. The buffer then passes from state **F** to **P**. The master does not have to care any more about the buffer, as the buffer management system will transparently pass the buffer from state **P** to state **N**, when all the send requests associated with it are complete. The buffer will then either be deallocated or be reused if the master makes another request.

When a worker needs to receive a BF message, after having been notified of its enrollement in the computation of a front (reception of a DB mesage) or the after its assembly of a fron being complete (last CBT2 message received), it requests the buffer management system for a buffer, and calls an MPI_Irecv request on it. The buffer passes from state **N** to state **R**. When the buffer management system discovers that the message arrived, it directly and transparently relays the message to other workers, passing the buffer from state **R** to state **S**. If the worker has treated the message while some MPI_ISend requests have not yet been completed, the buffer passes from state **S** to state **P**. When all the MPI_ISend request are completed, the buffer will pass from state **P** to state **N**. On the contrary, when all the MPI_ISend requests associated with the buffer complete while the message has not already been treated by the worker, the buffer passes from state **S** to state **T**. When the worker finishes the treatment of the message, the buffer will pass then from state **T** to state **N**.

Some additional information is also associated with each active front. As explained before, each process uses a specific TAG to handle communications of a given front. We also need to specify whether the process is a master (root of broadcast tree) or a worker, as the management automata will not be the same in each case. Moreover, it is not necessary for a worker to wait for a front to be fully assembled before posting an MPI_Irecv. Thus, as soon as a front is activated on a process, we can already post an MPI_IRecv. However, we cannot process messages until the front is fully assembled. Thus, we use the *IS_ASSEMBLED* variable to record the completion of the assembly. Furthermore, when a worker is ready to receive messages when all the buffers are busy, we use the *IS_IRECV_TO_BE_POSTED* variable to flag that, once a buffer is available, an MPI_Irecv must be posted. Finally, it is very likely that a process has finished all the activities related to a front, but that some messages it had started transmitting to its successors have not yet been completed. We use then the *IS_ZOMBIE* variable to notify the system that, as soon as all the remaining messages are sent, all the buffers dedicated to that front may be reused for other fronts or deallocated, and that all the data structures related to

that front may be deallocated.

9.3.3 Dynamic scheduling decisions

All this mechanism of buffer handling now allows the buffer management system to work as an independent entity, making communications progress independently from computations. However, now that we are able to identify the messages we want to receive, we have a degree of freedom on the choice of which communication to handle and which message to give priority to. From what we have seen in Chapter 8, if we decide to use the deadlock avoidance approach, when only minimal communication memory is available, we are obliged to dedicate the remaining memory to the highest priority front. However, when enough memory is available, we could decide to prioritize receptions from any front. The goal is to dedicate buffers and decide of the order of treatment of messages to optimize the efficiency. Let us suppose that we have additional memory to allocate one buffer, and we have to choose to dedicate it to one of two fronts F_1 and F_2, or let us suppose that two messages related to two different fronts are ready to be treated.

Many parameters may influence the availability of messages from F_1 and F_2. First, if the predecessor in the broadcast tree of F_1 is located on a machine node far from the node of the current process, while that in the broadcast tree of F_2 is a near neighbour, messages from F_2 should normally arrive sooner than that from F_1. Second, depending on the shape (ratio $\frac{npiv}{nfront}$) of F_1 and F_2, and on the load of the masters of F_1 and F_2, these masters could generate messages at very different rates from each other.

For the moment, we use the heuristic of favouring messages associated with tasks of high priority in the global order (that we defined in Section 8.2.2.1). However, it seems to us that such local decisions may have an important impact on the global efficiency of the system.

9.4 Preliminary experimental results

In order to assess the performance of the algorithms and kernels developed during this thesis, we have run some preliminary experiments, on both split chains, as shown in Section 9.4.1, and on whole multifrontal trees, as shown in Section 9.4.2.

9.4.1 Results on chains

Table 9.1 shows experimental results obtained when applying dense factorizations on single fronts (split chains) on the `ada` machine, whose machine peak is 20 GFlops/s. The fronts are of size 10000, 32000 and 64000. The number of processes is 1, 8 or 64, and the number of threads per process is either 1 (sequential processes) or 8 (multithreaded processes). In order to show some of the evolutions we came through, we run the first tests using the initial version of our software, without all the improvements explained in the thesis (we call it `Trunk`). We also use all the improvements described in Chapters 5, 6 and 7 (which we call `Synchro`). Then, we add to the `Synchro` version the improvements that reduce synchronizations, as explained in Chapters 8 and 9 (which we call `NoSynchro`). Finally, in order to have a point of comparison on the theoretical limits on performance, we compare our results with the MKL implementation of the ScaLAPACK [33] software library. This is the reason why all the factorizations of Table 9.1 concern exclusively full factorizations, while our multifrontal kernels are actually meant to be used mainly for partial factorizations, with numerical pivoting restricted to the fully-summed block.

In order to analyse both the weak and strong scalability of the different approaches, let us note that the computation times with `Synchro`, `NoSynchro` and ScaLAPACK on the factorization of

the front with *nfront* = 10000 are very similar on the single-process single-threaded executions ($\approx$ 32 seconds). All the other results can then be compared to this one.

We can observe, without surprise, that ScaLAPACK is better than `NoSynchro`, which in turn is better than `Synchro`, itself better than `Trunk`. However, the gap between them heavily depends on the parameters.

Firstly, for a given *nproc* and *nthread*, we can see that the performance of all the kernels improves when increasing *nfront*. This is because there are proportionally less communications than computations, and that the computations are more efficient as they take place on larger blocks. Secondly, for a given *nfront* and *nthread*, we see that the performance of `NoSynchro` remains close to that of ScaLAPACK; whereas that of `Trunk` and `Synchro` is poorer. This is due to the good communication pattern of the 2D block-cyclic distribution in ScaLAPACK and to the improved pipelining and asynchronous deadlock-free solutions in `NoSynchro`. Thirdly, for the same total number of cores, but for increasing *nproc* (and thus simultaneously decreasing *nthread*), we see that the performance of ScaLAPACK and `NoSynchro` remains stable (or even improves) when the front is large enough; whereas that of `Trunk` and `Synchro` becomes worse. The underlying reasons are similar to the previous ones, as this behaviour is due to an increase in the weight on communications compared to computations in multithreaded environments. Thus, for the same *nfront* and *nproc*, and for increasing *nthread*, we see that `NoSynchro` is more efficient than `Trunk` and `Synchro`. We observe by comparing the results with *nproc* = 64 and *nthread* = 8 on a matrix with *nfront* = 64000 that the GFlops/s rate of `NoSynchro` is 7.9, that of `Synchro` is 3.5 and that of `Trunk` is 2.29. This shows that `NoSynchro` benefits more from the hybrid shared-distributed-memory environments.

nfront	*nproc*	*nthread*	`Trunk`		`Synchro`		`NoSynchro`		ScaLAPACK	
			Time	GFlops/s	Time	GFlops/s	Time	GFlops/s	Time	GFlops/s
10000	1	1	36.44	18.29	32.29	20.64	32.29	20.64	31.99	20.84
	8	1	8.37	9.95	7.67	10.86	7.88	10.58	6.51	12.80
	8	8	2.00	5.21	1.52	6.86	1.47	7.09	1.64	6.34
	64	1	5.92	1.76	6.75	1.54	5.20	2.00	4.21	2.48
	64	8	3.38	0.39	5.55	0.23	3.28	0.40	1.61	0.81
32000	8	8	34.97	9.14	27.76	11.52	26.67	11.99	23.94	13.35
	64	1	94.95	3.37	48.49	6.59	31.44	10.17	27.26	11.73
	64	8	29.44	1.36	27.06	1.48	10.60	3.77	8.75	4.57
64000	8	8	250.67	10.90	189.78	14.39	187.64	14.56	173.19	15.77
	64	1	674.16	4.05	228.51	11.95	189.65	14.40	161.60	16.90
	64	8	149.24	2.29	97.48	3.50	43.21	7.90	34.87	9.79

Table 9.1: Experimental results on the `ada` machine for the full factorization of split chains of fronts of different sizes (10000, 32000, 64000), with different numbers of processes (1, 8, 64) and threads per process (1, 8), using the `Trunk` version of `MUMPS` (without the improvements presented in this thesis), the `Synchro` version (double panels, tree-based broadcasts, splitting with restart) and the `NoSynchro` version (`Synchro` version + relaxation of the synchronizations with deadlock prevention). We also provide the result of the execution using ScaLAPACK in order to have a point of comparison with the standard distributed-memory dense linear algebra kernel. Times are in seconds and GFlops/s are per core. The machine peak of `ada` is 20 GFlops/s.

The results we have with `NoSynchro` are preliminary, in the sense that some algorithmic improvements still need to be implemented, like the ABC_w trees that we described in Chapter 8.

9.4.2 Results on trees

We now present preliminary results on the factorization of a whole multifrontal tree. We have used for this experiment the `GeoAzur_128_128_128` matrix test case, which requires 1.225 10^5 GFlops for its factorization. It is one of the matrices in the set of `GeoAzur` matrices presented in Part I of this thesis. We have also relied on the `eos` computer (see Section C.5) to run our experiment.

When using $nproc = 90$ with $nthread = 10$, we took **162** seconds with the `Trunk` version, **151** seconds with the `Synchro` version and **120** seconds with the `NoSynchro` version.

This result is a rough result. It was obtained without any deep analysis or improvement to the `NoSynchro` version of our code. Many improvements may still be done to further reduce the execution time. First, the improvements targeting single fronts or chains of fronts will immediately impact computations on multifrontal trees; also, the ABC_w trees exposed in Chapter 8 should be implemented. Second, the fact of having removed, or at least relaxed, the synchronizations between processes has raised new scheduling challenges. For instance, we have observed that with the current scheduler in `NoSynchro`, as there is no limit to prohibit processes from starting new tasks as soon as they are able to do so, they tend to start computations on many more fronts in parallel than they did before. This makes them consume more memory for contribution blocks, which can be problematic on large problems. Consequently, some work has to be done to adapt the scheduling decisions of the processes to be able to use new degree of freedom on the asynchronism wisely.

9.5 Conclusion

In the light of the theory of deadlocks described in Chapter 8, we have presented in the present chapter the previous solutions that were used to manage deadlock issues in our asynchronous environment. Through the implementation of a prototype in an asynchronous multifrontal solver, we have demonstrated the feasibility of the solutions we have proposed. The preliminary experimental results we have obtained, both on fronts and trees, show the potential of our solutions in handling real-life test cases and confirm the validity of our asynchronous approach.

Chapter D

Conclusion of part II

The objective of the second part of this thesis was to improve asynchronous distributed-memory multifrontal factorizations. To reach this objective, we have mainly focused on the improvement of the parallel 1D pipelined dense factorization algorithms and kernels used in that context. The methodology we have followed consisted in successively identifying the major bottlenecks and in proposing solutions to each of them.

We started, in Chapter 5, by studying the theoretical behaviour of our algorithms. We showed the superiority of the LL approach over the RL approach in our context. Then, we identified the practical bottlenecks related to their application. We then showed the effect of the use of asynchronous broadcast schemes to resolve communication issues and showed the effect of extending the blocking approach to multiple levels to resolve computation issues. We then developed improved kernels, whose efficiency still depended on the shape of the fronts on which they were applied.

We then targeted this issue of fronts shape in Chapter 6. We took advantage of the flexibility offered by sparse direct methods of transforming multifrontal trees, that is, of replacing some fronts by a chain of fronts, where each new front in the chain has a shape for which our kernels reach their best performance. This transformation in turn introduced new mapping and remapping problems for the successive fronts, that we answered while taking advantage of the dynamic context, where new processes may become available. The chapter then exposed the bottlenecks to scaling the computations on large fronts and to large numbers of fronts.

The first problem was the increase of assembly operations earned by our solutions. We proposed in Chapter 7 solutions that adapt the mapping of the processes in the fronts to minimize the total volume of assembly communications in the case of chains of fronts. We also extended this work to the general case of multifrontal trees, where we proposed heuristics to reduce assembly communications as much as possible.

The second problem was the breaking of computations and communications pipelines due to synchronization. In Chapter 8, we analysed and characterized potential deadlock situations and discussed deadlock prevention and avoidance solutions to suppress them. Our approach is based on the observation that, as long as memory is available, deadlocks cannot occur, and thus, we just need to try to keep enough memory to guarantee that deadlocks can always be avoided. Moreover, such solutions require an adaptation of the construction of the asynchronous broadcast trees, in order to be used at their full potential. We then presented different families of solutions to minimize the impact of synchronizations on performance, by showing ways to smoothen the transitions of processes between successive fronts of the multifrontal tree to maximize the overlapping of communications and computations.

Finally, we have shown in Chapter 9 practical ways of implementing the aforementioned synchronization solutions. We then highlighted the impact of our work on typical chains of

fronts as well as on real life test case. This shows the feasibility of our asynchronous approach in a real multifrontal solver, concluding Part II of our thesis.

Even though our work is complete, it opens the door to some perspectives. The smoothening of the computation and communication pipelines through the implementation of ABC_w trees should improve the preliminary results we have obtained. A natural continuation of our work, in the short term, would be to pursue the profiling and experimentation process to adapt the solver to the enhanced asynchronous environment.

In the context of whole multifrontal factorizations, where many fronts are factorized in parallel and in different multifrontal subtrees, the main issue in the medium term will be the adequate scheduling of processes in order to take advantage of the flexibility offered by our improved kernels and of the increased asynchronism that our algorithms have made available.

Moreover, a direct application of our work on unsymmetric matrices is its application to the case of symmetric matrices. Due to more complex computation and communication schemes in this case, where each worker communicates portions of panels to the following ones [14], we could apply our methodology to this scheme and adapt the results of Chapters 8 and 9. The algorithms from Chapters 5, 6 and 7 should also be revisited to take into account the different shapes and distribution of symmetric fronts (see Figure D.1) compared to the unsymmetric ones we have considered in this thesis.

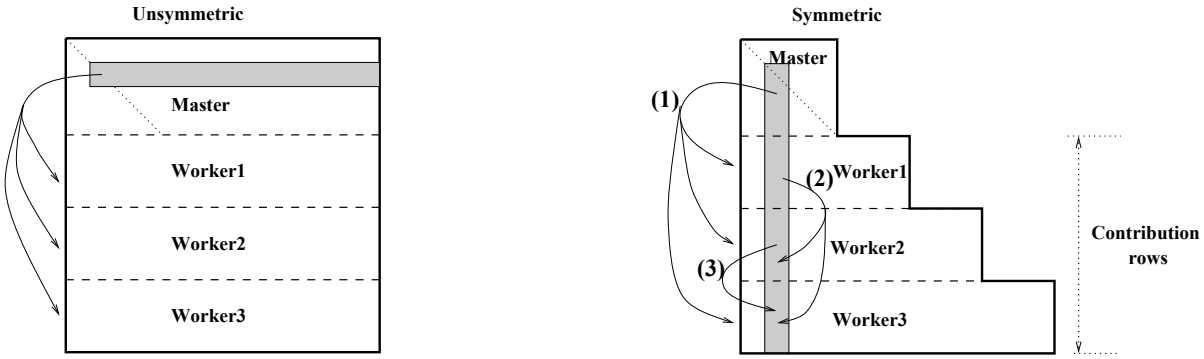


Figure D.1: Comparison of structures of unsymmetric and symmetric fronts with three workers. Arrows represent the transmission of blocks of factors between processes.

Similarly, it would be worth extending the type of work we have done for the factorization phase to the solve phase, as this phase may be costlier in applications with large numbers of right-hand sides. Unfortunately, the solve phase of sparse direct methods generally exhibits much smaller GFlops/s rates than those of the factorization phase. A characteristic of the context in Part II was an increase in the amount of communication with respect to the amount of computation, this phenomenon being exacerbated in the case of the multifrontal solve phase. There should thus be some potential to relax the synchronizations also of the solve phase and to design mapping and scheduling algorithms to take advantage of these relaxations.

We give more general, longer term, perspectives at the end of our general conclusion (Chapter D).

General Conclusion

Sparse linear systems lie nowadays at the heart of several application fields. With the never-ending increase in their size requiring an ever increasing amount of computing resources, the scalability of sparse linear solvers becomes critical. The goal of this thesis was two-fold: to target the solution of larger and larger sparse linear systems and to improve the implementation of these solutions on modern, large-scale, hybrid shared-distributed memory supercomputers. This double objective fits into the context of a multifrontal factorization method, based both on multithreaded kernels in shared-memory, and on 1D acyclic pipelined asynchronous kernels in distributed-memory.

We refer the reader to the conclusions of Part I and Part II (and to the conclusions of the corresponding chapters) for a detailed summary of our contributions. We only summarize the main points considered during the thesis. The contributions of our work are in the form of a step-by-step identification of bottlenecks followed by their resolution. This thesis is organized in two parts, the first one dealing with shared-memory environments and the second with distributed-memory environments. In the first part, we have shown the impact of a careful choice in parallelism granularity together with the impact of the respective location of computational resources and memory resources. In the second part, we have shown ways to target both the theoretical and technical limitations of our factorization algorithms and ways to address issues related to data distribution, process mapping and management of asynchronous parallelism with controlled memory for communications relying on message passing.

Some of the work in this thesis has been the object of communications to the scientific community, see Appendix B. We have published the work of Part I in the journal *Parallel Computing* [80], the work of Chapters 5 and 6 in the conference *VECPAR* [16], and some preliminary work corresponding to Chapter 8, in the SIAM workshop *CSC14* [6]. We have also contributed to scientific collaborations that have led to publications in the *COMPUMAG13* conference [5] and in the *SMIRT22* conference [2]. We also plan to submit the work of Chapter 7 to a conference. Furthermore, the work of Chapters 8 and 9 will be the basis for a submission to a journal.

A direct impact of this thesis concerns the improvement of the `MUMPS` solver. We will introduce the algorithms presented in the thesis in future releases of the software. This work will hopefully be of use for both the academic and the industrial users of the solver.

The arguments developed in this thesis sometimes diverge from and sometimes converge to existing ways in the literature of targeting similar problems.

We first discuss the main points of divergence, which come from different objectives and criteria, concerning both communication and computation. Concerning broadcast communications, where standard asynchronous implementations tend to use advanced communication patterns to optimize single broadcasts, we have instead successfully used simpler broadcast trees

in a pipelined way to optimize successions of broadcasts. Concerning computations, studies in dense linear algebra usually focus on the optimization of single matrix factorization. They thus recommend using a 2D block-cyclic matrix distribution to take advantage of its high asymptotic performance. The criteria we have used in our study differ in the case of sparse linear algebra, in the sense that many dense matrix factorizations occur simultaneously. In this context, we believe that the use of the 1D matrix distribution scheme is still possible for the nice numerical properties it brings, even at the price of its theoretical limitations: it allows for asynchronous parallelism with standard threshold pivoting, and pushing the limits of such an approach is of practical interest.

One point of convergence with the literature, among others, is the use of asynchronism. For instance, some runtime systems tend to rely like us on asynchronous approaches to speed up their computations and avoid idle times caused by strong synchronizations. However, as they target general problems, they target general task graphs. Due to the often large size of such graphs, runtime systems tend to have only limited visions (or windows) of those task graphs. They thus lack the full control we enjoy on our specific problems. Indeed, due to the particular shapes of our task graphs, which are trees of fronts whose computations can in turn be seen as very particular task sub-graphs, we can implement some interesting optimizations that would be harder to express with runtime systems.

We now give some perspectives to this work.

Of course, the overall performance of the 1D approach, remains lower than that of the 2D approach, at least when considering strong scalability. However, the improvement of the 1D approach that we have shown in our work could be useful for building a hybrid matrix distribution scheme that could mix both 1D and 2D improvements. Such a hybrid scheme could benefit from the numerical advantages of the 1D approach while taking advantage of the reduced limitations of the 2D approach. One such approach could consist in mapping fully summed rows of fronts following a 1D distribution and in mapping contribution rows in a 2D way.

Moreover, the evolution of computer architectures reduces the existing gap between shared-memory and distributed-memory. Indeed, shared-memory architectures contain more and more cores per node with inter-processor connections resembling distributed-memory networks (fat nodes). Moreover, with NUMA-link networks, distributed-memory networks may be viewed in a shared-memory way. Consequently, it would be interesting to merge the work described in Parts I and II of the present thesis by breaking the rigid way of approaching shared and distributed memory paradigms and by dynamically changing the number of processes together with the number of threads per process. The resulting flexibility would allow for a better adaptation, not only to the characteristics of the problems at hand but also to the characteristics of the machines on which the problems are solved. Typically, we would use many processes with few threads at the bottom of multifrontal trees, and progressively reduce the number of active processes, giving their threads to other active processes when going up the tree, until we use only one process with all computational resources at the root of the tree. One could also take advantage of some modern technologies, like runtime systems and PGAS languages, to facilitate this approach.

On modern computers, computations tend to become cheaper and cheaper, whereas data transfers, in general, and memory accesses and network communications, in particular, become comparatively more costly. Due to the inner limitations of the algorithms we rely on, we have often encountered this phenomenon. This will be more and more of an issue in the future, even on other algorithms that exhibits better asymptotic behaviors. This is aggravated by the complexity of memory and network hierarchies. Our work could thus be extended by following a *topology aware* approach, taking the topology of the computers into account at an algorithmic level. We have already started in Chapter 3 to explore this possibility by taking into account

locality issues. Moreover, early research in the field of collective communications has already been done [104]. This could be refined by making the thread mapping, the process mapping, and the data accesses fit as much as possible the topology of the memory and the network.

Furthermore, the work we have presented focused exclusively on performance and scalability. In order to target future industrial problems, this work should be coupled with other approaches, such as memory-aware approaches [98] and low-rank approximations [112]. This gives rise to new challenges. Firstly, these approaches increase the ratio of communications over computations. This is because memory-aware approaches tend to limit the total number of simultaneous active tasks, that are responsible for greater use of memory. For that, they must increase the number of processes in some tasks, in order to solve large problems with limited amounts of memory. Low-rank approximations decrease the amount of both computation and communication, but tend to decrease the amount of computation much more than they decrease the volume of communication. The work that we have done on scalability will thus be even more critical in these cases. Secondly, these approaches have a significant impact on the scheduling decisions of the system. Memory-aware approaches add new constraints to the scheduling by forcing the serialization of formerly parallel computations. Low-rank approximations make the behaviour of the computations unpredictable, since for some given required precision, the compression rates they induce heavily depend on the targeted problems. These approaches degrade static scheduling decisions and necessitate an increase in the dynamic part of the schedulers. In such a context, our work on the improvement of asynchronism in multifrontal trees, both on shared-memory (via a variant of work-stealing) and on distributed-memory (via minimal synchronization algorithms), could serve as a building block to target such issues. In all cases, we believe that asynchronism is of paramount importance to target next generation massively parallel computers.

Appendix A

Implementation remarks

A.1 Details on the application of the interleave policy for the use of $\mathcal{L}_{th}$-based algorithms on NUMA architectures

We show here how we did to effectively apply the interleave allocation policy on NUMA architectures, as described in Chapter 3.

We observed that applying the interleave policy only during the allocation of the shared workspace was not enough for the interleaving to be effectively applied. The reason is that, when calling the *malloc* function to allocate a large array, only its first page is actually allocated: the remaining pages are allocated only when an element of the array corresponding to that page is accessed for the first time. The effect of the interleave policy is such that each time a new memory page must be allocated, not necessarily involving the shared workspace, for any data structure, that memory page will be allocated in a round-robin fashion relative to the last allocated page. Moreover, as we must use local allocation policies for all other application data structures, we set the interleave policy just before the call to *malloc* only and reset a standard policy just after, which makes the interleave ineffective. In order to effectively obtain an interleaved memory in the shared array, one solution consists in accessing (writing one entry of) each page of the array immediately after the allocation. Another portable solution that does not depend on external libraries but only depends on the first touch rule is to make one thread of each socket access all the pages that will be allocated on that socket. Making this small modification leads to a memory mapping as illustrated in Figure 3.2 and dramatically improves the performance of the multifrontal factorization, as is discussed in Section 3.1.3.

The drawback of this approach, however, is that it forces one to allocate all the pages of the shared workspace at once, while the physical memory also needs to hold the already allocated private workspaces. This is not the case in the initial non architecture-aware approach where memory pages related to the shared workspace are allocated by the operating system progressively, only when needed. Additionally, we would like pages of thread-private workspaces to be gradually recycled to pages of the shared workspace, allowing for a better use of memory and a reduced maximum memory consumption (see remark on *realloc* in Section 2.2.4). This is made possible as the shared workspace will start to be accessed and used only when the work under $\mathcal{L}_{th}$ is completed, hence when the unused space in private workspaces previously used to hold temporary contribution blocks and active fronts and not being used anymore could be reused (recycled) by the system.

In order to overcome this drawback, we could benefit from the *first-touch* principle which states that, with the `localalloc` memory allocation policy, a page is allocated on the same node as the thread that first touches it. While working on the parallelization of assembly operations (Algorithm A.1), we observed that even without the *interleave* policy, it was possible to

have similar effects on the mapping of memory pages by parallelizing (with threads of different sockets) the initialization of the frontal matrices to zero. When accessing for the first time pages of the shared workspace, each of the threads involved in this initialization allocate these pages on its local memory, but as threads of different sockets do it, the final mapping is quite well balanced over memory of all the sockets. While this had no effect on the time spent in assemblies, significant gains were observed regarding the performance of the factorization (Algorithm A.2). Although these gains were not as important as those observed with the `interleave`, this approach is an interesting portable alternative.

A.2 How to associate message tags to tasks?

In the asynchronous multifrontal method, processes may receive any message, at any time and from anyone. In special cases however, mainly because of synchronization constraints, processes may be forced to receive specific messages. They must then be able to identify them.

The MPI standard provides 32 different tags to identify types of messages, although MPI implementations will provide until 65536 different tags. Currently in our solver, we use only one tag to identify the whole set of BF messages (Block of Factors, i.e. factorized panels). What we would like is to be able to identify the BF messages of a particular front uniquely. Using a unique tag for each front would be the natural way do it. Unfortunately, there would not be enough MPI tags to cover our need, as multifrontal trees might well have more fronts than there are tags available.

Thus, no matter how many fronts are in a multifrontal tree, the number of active fronts is limited or may be controlled, so one solution would be to reuse tags of terminated fronts on newly activated ones. A naive approach to do so would be to rely on a token server which would be responsible for assigning/collecting tags to/from new/old fronts through the front masters. This centralized approach should be avoided, however, as it harms asynchronism. A better approach would be to rely on decentralized semaphore or token management techniques [96] to make the masters activating new tasks select an unused tag among a set of available tags. Although these approaches would work in practice, they would still require the use of a large number of tags, as this is proportional to the number of active tasks in parallel, and may even constrain the parallelism to fit their requirements. These approaches are thus not scalable.

The solution we propose is a local one. Instead of assigning a tag to each front, we assign a tag to each pair of sender-receivers in the front's broadcast tree. Basically, each process allocates an array *mytags* of size *nproc* initialized to zero. When a process i realises that it is involved in the computation of front F and that it is a relay in the corresponding broadcast tree, then, for each successor j, it increases by 1 modulus $maxtags$ the current value of mytags(j) ($maxtags$ being the maximum number of available tags). The resulting value is the tag it will use to communicate BF messages of front F to process j. It then sends the value of this tag to process j through a special message before being allowed to send any corresponding BF message. If process j receives too soon a BF message from process i with an unknown tag, it thus knows that the special message has already been sent, being thus allowed to apply a blocking receive on it.

In the worst case of an all-on-one mapping (all processes assigned to all fronts), the maximum number of simultaneous tags used by the processes will be the same as that when using the previous global solutions, i.e. the number of active tasks in parallel. In the best case of proportional mapping (each process assigned to one front at a time), the maximum number of tags per couple of processes is only one. Thus, our local solution is more scalable as, depending on the mapping of processes, the increase of tree parallelism, and thus, the increase of the number of active tasks, is an advantage instead of being an issue. Moreover, the choice of a

tag between a pair of processes only is more asynchronous than in other solutions. Finally, the implementation of our local solution is much easier than that of other solutions.

Appendix B

Publications related to the thesis

Patrick R. Amestoy, Alfredo Buttari, Abdou Guermouche, Jean-Yves L'Excellent, and Wissam M. Sid-Lakhdar. **Exploiting Multithreaded Tree Parallelism for Multicore Systems in a Parallel Multifrontal Solver**. In *SIAM conference on Parallel Processing for Scientific Computing (PP12), Savannah, GA, USA*, **February 2012**.

Patrick Amestoy, Jean-Yves L'Excellent, and Wissam Sid-Lakhdar. **Characterizing asynchronous broadcast trees for multifrontal factorizations (extended abstract)**. In *SIAM Workshop on Combinatorial Scientific Computing, Lyon, France, July 21-23*, **2014**.

Patrick Amestoy, Jean-Yves L'Excellent, François-Henry Rouet, and Wissam Sid-Lakhdar. **Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver (regular paper)**. In *High-Performance Computing for Computational Science, VECPAR 2014, Eugene, Oregon, USA, June 30 - July 03*, **2014**.

Patrick Amestoy, Alfredo Buttari, Guillaume Joslin, Jean-Yves L'Excellent, Wissam Sid-Lakhdar, Clément Weisbecker, Michele Forzan, Cristian Pozza, Rémy Perrin, and Valène Pellissier. **Shared-Memory Parallelism and Low-Rank Approximation Techniques Applied to Direct Solvers in FEM Simulation**. *IEEE Transactions on Magnetics, Extended selected short papers from Compumag 2013 conference*, 50(2):517–520, **February 2014**.

Patrick R. Amestoy, Alfredo Buttari, Guillaume Joslin, Jean-Yves L'Excellent, Wissam M. Sid-Lakhdar, Clément Weisbecker, Michele Forzan, Cristian Pozza, Rémy Perrin, and Valène Pellissier. **Shared memory parallelism and low-rank approximation techniques applied to direct solvers in FEM simulation (regular paper)**. In *IEEE International Conference on the Computation of Electromagnetic Fields (COMPUMAG), Budapest, Hungary, June 30 - July 04*, **2013**.

Emmanuel Agullo, Patrick R. Amestoy, Alfredo Buttari, Abdou Guermouche, Guillaume Joslin, Jean-Yves L'Excellent, X. Sherry Li, Artem Napov, François-Henry Rouet, Wissam M. Sid-Lakhdar, Sheng Wang, Clément Weisbecker, and Ichitaro Yamazaki. **Recent advances in sparse direct solvers**. In *International Conference on Structural Mechanics in Reactor Technology (SMIRT-22), San Francisco, USA*. IASMiRT, **August 2013**.

Jean-Yves L'Excellent and Wissam M. Sid-Lakhdar. **A study of shared-memory parallelism in a multifrontal solver**. *Parallel Computing*, 40(3–4):34 – 46, **2014**.

Bibliography

[1] E. Agullo. *On the Out-of-core Factorization of Large Sparse Matrices*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.

[2] E. Agullo, P. R. Amestoy, B. Alfredo, A. Guermouche, G. Joslin, J.-Y. L'Excellent, X. S. Li, A. Napov, F.-H. Rouet, W. M. Sid-Lakhdar, S. Wang, C. Weisbecker, and I. Yamazaki. Recent advances in sparse direct solvers. In *International Conference on Structural Mechanics in Reactor Technology (SMIRT-22), San Francisco, USA*. IASMiRT, Aug. 2013. To appear.

[3] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In F. Wolf, B. Mohr, and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 521–532. Springer Berlin Heidelberg, 2013.

[4] P. Amestoy. *Méthodes directes parallèles de résolution des systèmes creux de grande taille* . Habilitation à diriger des recherches, Institut National Polytechnique de Toulouse, Toulouse, France, septembre 1999.

[5] P. Amestoy, A. Buttari, G. Joslin, J.-Y. L'Excellent, W. Sid-Lakhdar, C. Weisbecker, M. Forzan, C. Pozza, R. Perrin, and V. Pellissier. Shared memory parallelism and low-rank approximation techniques applied to direct solvers in FEM simulation. *IEEE Transactions on Magnetics, Extended selected short papers from Compumag 2013 conference*, 50(2), Feb. 2014.

[6] P. Amestoy, J.-Y. L'Excellent, and M. Sid-Lakhdar. Characterizing asynchronous broadcast trees for multifrontal factorizations (SIAM Workshop on Combinatorial Scientific Computing, Lyon, France, 21/07/2014-23/07/2014). 2014.

[7] P. R. Amestoy. *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment*. PhD thesis, Institut National Polytechnique de Toulouse, 1991. Available as CERFACS report TH/PA/91/2.

[8] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving multifrontal methods by means of block low-rank representations. Technical Report RT/APO/12/6, ENSEEIHT-IRIT, 12 2012. also appeared as INRIA technical report and submitted to SIAM J. Scient. Comput.

[9] P. R. Amestoy, A. Buttari, G. Joslin, J.-Y. L'Excellent, W. M. Sid-Lakhdar, C. Weisbecker, M. Forzan, C. Pozza, R. Perrin, and V. Pellissier. Shared memory parallelism and low-rank approximation techniques applied to direct solvers in FEM simulation (regular paper). In *IEEE International Conference on the Computation of Electromagnetic Fields (COMPUMAG), Budapest, Hungary, 30/06/2013-04/07/2013*, June 2013.

[10] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. Technical Report TR-94-039, CIS Dept., Univ. of Florida, 1994. Appeared in SIAM J. Matrix Analysis and Applications.

[11] P. R. Amestoy and I. S. Duff. Memory allocation issues in sparse multiprocessor multifrontal methods. Technical Report TR/PA/92/83, CERFACS, 1992.

[12] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. of Supercomputer Applics.*, 7:64–82, 1993.

[13] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[14] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[15] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[16] P. R. Amestoy, J.-Y. L'Excellent, F.-H. Rouet, and W. M. Sid-Lakhdar. Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver (regular paper). In *High-Performance Computing for Computational Science, VECPAR 2014, Eugene, Oregon, USA, 30/06/2014-03/07/2014*, 2014.

[17] P. R. Amestoy and C. Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24:553–569, 2002.

[18] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM, 1992.

[19] C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15:291–309, 1989.

[20] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23(2):187–198, Feb. 2011.

[21] H. Avron, G. Shklarski, and S. Toledo. Parallel unsymmetric-pattern multifrontal sparse LU with column preordering. *ACM Transactions on Mathematical Software*, 34(2):8:1–8:31, 2008.

[22] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *SIAM J. Scientific Computing*, 32(6):3495–3523, 2010.

[23] C. Balsa, R. Guivarch, D. Ruiz, and M. Zenadi. An hybrid approach for the parallelization of a block iterative algorithm (regular paper). In *International Conference on Vector and Parallel Processing (VECPAR), Berkeley, 22/06/2010-25/06/2010*, pages 116–128. Springer-Verlag, 2010.

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.

[25] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, 2011.

[26] A. Boukerche and C. Tropper. A distributed algorithm for the detection of local cycles and knots. *Parallel Processing Symposium, International*, 0:118, 1995.

[27] 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 M. Danelutto, J. Bourgeois, and T. Gross, editors, *PDP*, pages 180–186. IEEE Computer Society, 2010.

[28] A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. *SIAM Journal on Scientific Computing*, 35(3):C323–C345, 2013.

[29] A. M. Castaldo, S. Samuel, and R. C. Whaley. Scaling LAPACK panel operations using parallel cache assignment, 2013.

[30] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–11, March 2007.

[31] B. Chapman, G. Jost, R. Van der Pas, and D. J. Kuck. *Using OpenMP : portable shared memory parallel programming.* MIT Press, Cambridge, Mass., London, 2008.

[32] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. In *Proceedings of PMAA2006, Rennes, France*, oct 2006.

[33] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5(3):173–184, 1996.

[34] I. Chowdhury and J.-Y. L'Excellent. Some experiments and issues to exploit multicore parallelism in a distributed-memory parallel sparse direct solver. Research Report RR-4711, INRIA and LIP-ENS Lyon, Oct. 2010.

[35] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[36] T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse qr factorization. *ACM Trans. Math. Softw.*, 38(1):8:1–8:22, Dec. 2011.

[37] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18:140–158, 1997.

[38] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[39] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[40] F. Dobrian and A. Pothen. The design of I/O-efficient sparse direct solvers. In *Proceedings of SuperComputing*, 2001.

[41] J. J. Dongarra and D. C. Sorensen. Schedule: tools for developing and analyzing parallel fortran programs. Technical Report MCS-TM-86, Argonne National Laboratory, Nov. 1986.

[42] M. Drozdowski. Scheduling multiprocessor tasks: An overview. *European Journal of Operational Research*, 94(2):215–230, Oct. 1996.

[43] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel computing*, 3:193–204, 1986.

[44] I. S. Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *J. Comput. Appl. Math.*, 27:229–239, 1989.

[45] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[46] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

[47] I. S. Duff and J. K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 22(2):227–257, 1996.

[48] S. C. Eisenstat and J. W. H. Liu. A tree based dataflow model for the unsymmetric multifrontal method. *Electronic Transaction on Numerical Analysis*, 21:1–19, 2005.

[49] M. Faverge. *Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-coeurs.* PhD thesis, LaBRI, Université Bordeaux I, Talence, Talence, France, Dec. 2009.

[50] M. Faverge, X. Lacoste, and P. Ramet. A NUMA-aware scheduler for a parallel sparse direct solver. Research report RR-7498, INRIA, May 2010.

[51] H. Garcia-Molina. Election in distributed computing system. *IEEE Transactions on Computers*, pages 47–59, 1982.

[52] A. Geist and E. G. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.

[53] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ., 1981.

[54] J. A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10(2):345–363, 1973.

[55] L. Giraud, A. Haidar, and S. Pralet. Using multiple levels of parallelism to enhance the performance of domain decomposition solvers. *Parallel Computing*, 36(5-6):285–296, 2010.

[56] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[57] A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.

[58] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.

[59] A. Gupta. WSMP: Watson Sparse Matrix Package part i - direct solution of symmetric sparse systems version 1.0.0. Technical Report TR RC-21886, IBM research division, T.J. Watson Research Center, Yorktown Heights, 2000.

[60] A. Gupta. WSMP: Watson Sparse Matrix Package part ii - direct solution of general sparse systems version 1.0.0. Technical Report TR RC-21888, IBM research division, T.J. Watson Research Center, Yorktown Heights, 2000.

[61] A. Gupta. A shared- and distributed-memory parallel general sparse direct solver. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):263–277, 2007.

[62] A. N. Habermann. Prevention of system deadlocks. *Commun. ACM*, 12(7):373–ff., July 1969.

[63] M. T. Heath, E. G. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[64] P. Hénon, P. Ramet, and J. Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, Jan. 2002.

[65] T. Hérault, J. Herrmann, L. Marchal, and Y. Robert. Determining the optimal redistribution. Rapport de recherche RR-8499, INRIA, Mar. 2014.

[66] T. Hoefler and A. Lumsdaine. Message progression in parallel computing - to thread or not to thread? In *IEEE International Conference on Cluster Computing*, pages 213–222, 2008.

[67] J. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM Journal on Scientific Computing*, 6(32):3627–3649, 2010.

[68] J. D. Hogg and J. A. Scott. Achieving bit compatibility in sparse direct solvers. Technical Report RAL-P-2012-005, RAL, Oct. 2012.

[69] A. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing multiple StarPU applications over heterogeneous machines: A supervised approach. *International Journal of High Performance Applications*, 28:285–300, 2104.

[70] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal sparse Cholesky. *Future Generation Computer Systems*, 20(3):425–440, 2004.

[71] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. On optimal tree traversals for sparse matrix factorization. In *IPDPS'2011, the 25th IEEE International Parallel and Distributed Processing Symposium*, pages 556–567, Washington, DC, USA, 2011. IEEE Computer Society Press.

[72] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. PFunc: modern task parallelism for modern high performance computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 43:1–43:11, New York, NY, USA, 2009. ACM.

[73] G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, Sept. 1998.

[74] G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, Sept. 1998.

[75] H. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[76] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 147–156. Springer Berlin Heidelberg, 2007.

[77] X. Lacoste, P. Ramet, M. Faverge, Y. Ichitaro, and J. Dongarra. Sparse direct solvers with accelerators over DAG runtimes. Research report RR-7972, INRIA, 2012.

[78] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[79] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[80] J.-Y. L'Excellent and M. W. Sid-Lakhdar. A study of shared-memory parallelism in a multifrontal solver. *Parallel Computing*, 40(3-4):34–46, 2014.

[81] X. S. Li. Evaluation of SuperLU on multicore architectures. *Journal of Physics: Conference Series*, 125:012079, 2008.

[82] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, 2003.

[83] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.

[84] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.

[85] J. W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM J. Algebraic Discrete Methods*, 8:375–395, 1987.

[86] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[87] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.

[88] D. Manivannan and M. Singhal. An efficient distributed algorithm for detection of knots and cycles in a distributed graph. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):961–972, 2003.

[89] E. G. Ng and B. W. Peyton. Fast implementation of the minimum local fill ordering heuristic, 2014. SIAM Workshop on Combinatorial Scientific Computing (CSC14), Lyon, France, 21-23 July 2014.

[90] E. G. Ng and P. Raghavan. Performance of greedy heuristics for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20:902–914, 1999.

[91] S. Operto, J. Virieux, P. R. Amestoy, J.-Y. L'Excellent, L. Giraud, and H. Ben Hadj Ali. 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics*, 72(5):195–211, 2007.

[92] F. Pellegrini. SCOTCH and LIBSCOTCH 5.0 User's guide. Technical Report, LaBRI, Université Bordeaux I, 2007.

[93] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proceedings of HPCN'97, Vienna, LNCS 1225*, pages 370–378, Apr. 1997.

[94] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.

[95] G. Prasanna and B. Musicus. Generalized multiprocessor scheduling and applications to matrix computations. *Parallel and Distributed Systems, IEEE Transactions on*, 7(6):650–664, June 1996.

[96] M. Ramachandran and M. Singhal. Decentralized semaphore support in a virtual shared-memory system. *The Journal of Supercomputing*, 9(1-2):51–70, 1995.

[97] E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, 19(3):682–695, 1998.

[98] F.-H. Rouet. *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides.* PhD thesis, Institut National Polytechnique de Toulouse, Oct. 2012.

[99] P. Sao, R. Vuduc, and X. Li. A distributed cpu-gpu sparse direct solver. In F. Silva, I. Dutra, and V. Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 487–498. Springer International Publishing, 2014.

[100] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report CS-2004-004, CS Department, University of Basel, August 2004.

[101] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.

[102] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.

[103] Tz. Slavova. *Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems.* Ph.D. dissertation, Institut National Polytechnique de Toulouse, Apr. 2009. Available as CERFACS Report TH/PA/09/59.

[104] E. Solomonik, A. Bhatele, and J. Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 77:1–77:11, New York, NY, USA, Nov. 2011. ACM.

[105] F. Sourbier, S. Operto, J. Virieux, P. R. Amestoy, and J.-Y. L'Excellent. FWT2D: a massively parallel program for frequency-domain full-waveform tomography of wide-aperture seismic data – part 2: numerical examples and scalability analysis. *Computer and Geosciences*, 35(3):496–514, 2009.

[106] S. D. Stoller. Leader election in asynchronous distributed systems. *IEEE Transactions on Computers*, 49(3):283–284, Mar. 2000.

[107] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–159, 1972.

[108] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[109] C. Vömel. *Contributions to research in high performance scientific computing for sparse matrices.* Ph.D. dissertation, INPT, March 2003. TH/PA/03/26.

[110] C. Vuchener and A. Esnard. Graph repartitioning with both dynamic load and dynamic processor allocation. In *International Conference on Parallel Computing-ParCo2013*, 2013.

[111] D. M. Wadsworht and Z. Chen. Performance of MPI broadcast algorithms. In *Proceedings of 22nd International Parallel and Distributed Processing Symposium (IPDPS'08)*, pages 1–7, 2008.

[112] C. Weisbecker. *Improving multifrontal solvers by means of algebraic block low-rank representations.* PhD thesis, Institut National Polytechnique de Toulouse, http://ethesis.inp-toulouse.fr/archive/00002471/, Oct. 2013.

[113] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[114] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.