THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
L'Université Paul Sabatier

Présentée et soutenue par :

Florent Lopez

Le 11 décembre 2015

Titre :

Task-based multifrontal QR solver for heterogeneous architectures

---

Solveur multifrontal QR à base de tâches pour architectures hétérogènes

---

Ecole doctorale et discipline ou spécialité :
ED MITT : Domaine STIC : Sûreté du logiciel et calcul haute performance

Unité de recherche :
IRIT - UMR 5505

Directeurs de Thèse :
Michel Daydé (ENSEEIHT-IRIT)
Alfredo Buttari (CNRS-IRIT)

Rapporteurs :
Timothy A. Davis (Texas A&M University)
Pierre Manneback (Université de Mons)

Autres membres du jury :
Frédéric Desprez (Inria Rhône-Alpes)
Iain S. Duff (Rutherford Appleton Laboratory)
Raymond Namyst (Inria Bordeaux Sud-Ouest)

Résumé

Afin de s'adapter aux architectures multicoeurs et aux machines de plus en plus complexes, les modèles de programmations basés sur un parallélisme de tâche ont gagné en popularité dans la communauté du calcul scientifique haute performance. Les moteurs d'exécution fournissent une interface de programmation qui correspond à ce paradigme ainsi que des outils pour l'ordonnancement des tâches qui définissent l'application.

Dans cette étude, nous explorons la conception de solveurs directes creux à base de tâches, qui représentent une charge de travail extrêmement irrégulière, avec des tâches de granularités et de caractéristiques différentes ainsi qu'une consommation mémoire variable, au-dessus d'un moteur d'exécution. Dans le cadre du solveur qr_mumps, nous montrons dans un premier temps la viabilité et l'efficacité de notre approche avec l'implémentation d'une méthode multifrontale pour la factorisation de matrices creuses, en se basant sur le modèle de programmation parallèle appelé "flux de tâches séquentielles" (Sequential Task Flow). Cette approche, nous a ensuite permis de développer des fonctionnalités telles que l'intégration de noyaux dense de factorisation de type "minimisation de communications" (Communication Avoiding) dans la méthode multifrontale, permettant d'améliorer considérablement la scalabilité du solveur par rapport a l'approche original utilisée dans qr_mumps. Nous introduisons également un algorithme d'ordonnancement sous contraintes mémoire au sein de notre solveur, exploitable dans le cas des architectures multicoeur, réduisant largement la consommation mémoire de la méthode multifrontale QR avec un impacte négligeable sur les performances.

En utilisant le modèle présenté ci-dessus, nous visons ensuite l'exploitation des architectures hétérogènes pour lesquelles la granularité des tâches ainsi les stratégies l'ordonnancement sont cruciales pour profiter de la puissance de ces architectures. Nous proposons, dans le cadre de la méthode multifrontale, un partitionnement hiérarchique des données ainsi qu'un algorithme d'ordonnancement capable d'exploiter l'hétérogénéité des ressources. Enfin, nous présentons une étude sur la reproductibilité de l'exécution parallèle de notre problème et nous montrons également l'utilisation d'un modèle de programmation alternatif pour l'implémentation de la méthode multifrontale.

L'ensemble des résultats expérimentaux présentés dans cette étude sont évalués avec une analyse détaillée des performance que nous proposons au début de cette étude. Cette analyse de performance permet de mesurer l'impacte de plusieurs effets identifiés sur la scalabilité et la performance de nos algorithmes et nous aide ainsi à comprendre pleinement les résultats obtenu lors des tests effectués avec notre solveur.

Mots-clés : méthodes directes de résolution de systèmes linéaires, méthode multifrontale, multicoeur, moteurs d'exécutions, ordonnancement, algorithmes d'ordonnancement sous contraintes mémoire, architectures hétérogènes, calcul haute performance, GPU

Abstract

To face the advent of multicore processors and the ever increasing complexity of hardware architectures, programming models based on DAG parallelism regained popularity in the high performance, scientific computing community. Modern runtime systems offer a programming interface that complies with this paradigm and powerful engines for scheduling the tasks into which the application is decomposed. These tools have already proved their effectiveness on a number of dense linear algebra applications.

In this study we investigate the design of task-based sparse direct solvers which constitute extremely irregular workloads, with tasks of different granularities and characteristics with variable memory consumption on top of runtime systems. In the context of the qr_mumps solver, we prove the usability and effectiveness of our approach with the implementation of a sparse matrix multifrontal factorization based on a Sequential Task Flow parallel programming model. Using this programming model, we developed features such as the integration of dense 2D Communication Avoiding algorithms in the multifrontal method allowing for better scalability compared to the original approach used in qr_mumps. In addition we introduced a memory-aware algorithm to control the memory behaviour of our solver and show, in the context of multicore architectures, an important reduction of the memory footprint for the multifrontal QR factorization with a small impact on performance.

Following this approach, we move to heterogeneous architectures where task granularity and scheduling strategies are critical to achieve performance. We present, for the multifrontal method, a hierarchical strategy for data partitioning and a scheduling algorithm capable of handling the heterogeneity of resources. Finally we present a study on the reproducibility of executions and the use of alternative programming models for the implementation of the multifrontal method.

All the experimental results presented in this study are evaluated with a detailed performance analysis measuring the impact of several identified effects on the performance and scalability. Thanks to this original analysis, presented in the first part of this study, we are capable of fully understanding the results obtained with our solver.

Keywords: sparse direct solvers, multifrontal method, multicores, runtime systems, scheduling, memory-aware algorithms, heterogeneous architectures, high-performance computing, GPU

Acknowledgements

I am extremely grateful to Alfredo Buttari for his excellent supervision and constant support throughout three years of intensive work. Without his efforts this thesis would not have been possible. Thanks also to my two co-advisors Emmanuel Agullo and Abdou Germouche for their fruitful collaborations and very interesting discussions. I also wish to thank Michel Daydé for his support during this thesis.

Thanks to Jack Dongarra and George Bosilca for their warm welcome at the University of Tennessee, Knoxville, during my six month scholar visit to the ICL Laboratory. Thanks to the Distributed Computing team for the great collaboration, valuable help and introducing me to the PaRSEC library.

I would like to thank all the members of the thesis committee for reviewing my dissertation and particularly the "Rapporteurs" Tim Davis and Pierre Manneback for the insightful reports they provided about this manuscript. Also a big thank you to Iain Duff for providing such a complete and detailed feedback on the dissertation.

Thanks to my colleagues and particularly my office mates who are now among my closest friends: François-Henry Rouet, Clement Weisbecker, Mohamed Zenadi, Clément Royet and Théo Mary. Thanks to them for the good moments we shared, and the advice and support I received from you.

Finally, I'm deeply grateful to my family for their love, support and encouragement during the production of this thesis.

Contents

Chapter 1

Introduction

1.1 Architectures

The world of computing, and particularly the world of High Performance Computing (HPC), have witnessed a substantial change at the beginning of the last decade as all the classic techniques used to improve the performance of microprocessors reached the point of diminishing returns [17]. These techniques, such as deep pipelining, speculative execution or superscalar execution, were mostly based on the use of Instruction Level Parallelism (ILP) and required higher and higher clock frequencies to the point where the processors power consumption, which grows as the cube of the clock frequency, became (or was about to become) unsustainable. This was not only true for large data or supercomputing centers but also, and even more so, for portable devices such as laptops, tablets and smartphones which have recently become very widespread. In order to address this issue, the microprocessor industry sharply turned towards a new design based on the use of Thread Level Parallelism (TLP) achieved by accommodating multiple processing units or *cores* on the same die. This led to the production of *multicore* processors that are nowadays ubiquitous. The main advantage over the previous design principles lies in the fact that the multicore design does not require an increase in the clock frequency but only implies an augmentation of the chip capacitance (i.e., the number of transistors) on which the power consumption only depends linearly. As a result, the multicore technology not only enables improvement in performance, but also reduces the power consumption: assuming a single-core processor with frequency f, a dual-core with frequency $0.75 * f$ is 50% faster and consumes 15% less energy.

Since their introduction, multicore processors have become increasingly popular and can be found, nowadays, in almost any device that requires some computing power. Because they allow for running multiple processes simultaneously, the introduction of multicore in high throughput computing or in desktop computing was transparent and immediately beneficial. In HPC, though, the switch to the multicore technology lead to a sharp discontinuity with the past as methods and algorithms had to be rethought and codes rewritten in order to take advantage of the added computing power through the use of TLP. Nonetheless, multicore processors have quickly become dominant and are currently used in basically all supercomputers. Figure 1.1 (*left*) shows the performance share of multicore processors in the Top500[1] list (a list of the 500 most powerful supercomputers in the world, which is updated twice per year); the figure shows that after their first appearance in the list (May 2005) multicore has quickly become the predominant technology in the Top500 list and ramped up to nearly 100% of the share in only 5 years. The figure also

[1] http://top500.org

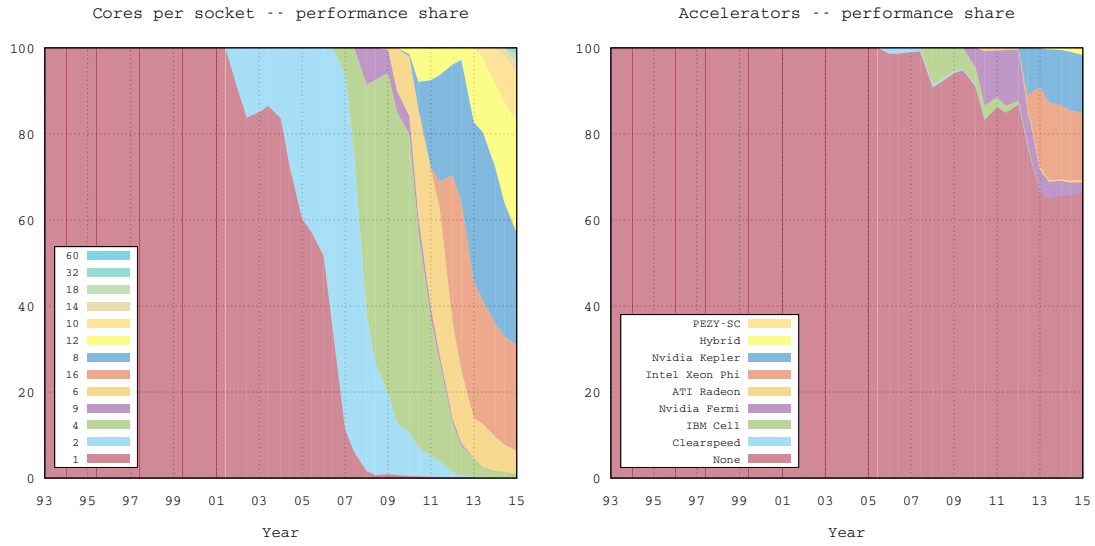


Figure 1.1: Performance share of multicore processors (on the left) and accelerators (on the right) in the Top500 list.

shows that the number of cores per socket has grown steadily over the years: a typical modern processor features between 8 and 16 cores. In Section 1.9.1 we present the systems used for the experiments reported in this document. The most recent and powerful among these processors is the Xeon E5-2680 which hosts 12 cores; each of these cores has 256 bit AVX vector units with Fused Multiply-Add (FMA) capability which, with a clock frequency of 2.5 GHz make a peak performance of 40 Gflop/s per core and 480 Gflop/s on the whole processor for double-precision floating point operations.

Around the same period as the introduction of multicore processors, the use of *accelerators* or *coprocessors* started gaining the interest of the HPC community. Although this idea was not new (for example FPGA boards were previously used as coprocessors), it was revamped thanks to the possibility of using extremely efficient commodity hardware for accelerating scientific applications. One such example is the Cell processor [65] produced by the STI consortium (formed by IBM, Toshiba and Sony) from 2005 to 2009. The Cell was used as an accelerator board on the Roadrunner supercomputer installed at the Los Alamos National Labrador (USA) which was ranked #1 in the Top500 list of November 2008. The use of accelerators for HPC scientific computing, however, gained a very widespread popularity with the advent of *General Purpose GPU* computing. Specifically designed for image processing operations, Graphical Processing Units (GPUs) offer a massive computing power which is easily accessible for highly data parallel applications (image processing often consists in repeating the same operation on a large number of pixels). This led researchers to think that these devices could be employed to accelerate scientific computing applications, especially those based on the use of operations with a very regular behaviour and data access pattern, such as dense linear algebra. In the last few years, GPGPU has become extremely popular in scientific computing and is employed in a very wide range of applications, not only dense linear algebra. This widespread use of GPU accelerators was also eased by the fact that GPUs, which were very limited in computing capabilities and difficult to program, have become, over the years, suited to a much wider range of applications and much easier to program thanks to the devel-

opment of specific high-level programming languages and development kits. Figure 1.1 (*right*) shows the performance share of supercomputers equipped with accelerators. Although some GPU accelerators are also produced by AMD, currently the most widely used ones are produced by Nvidia. Figure 1.2 shows a block diagram of the architecture of a recent Nvidia GPU device, the K40m of the Kepler family. This board is equipped with 15 Streaming Multiprocessors (SMX), each containing 192 single precision cores and 64 double precision ones for a peak performance of 1.43 (4.29) Tflop/s for double (single) precision computations. The K40m also has a memory of 12 GB capable of streaming data at a speed of 288 GB/s.

More recently Intel has also started producing accelerators, namely the Xeon Phi boards. The currently distributed models of the Xeon Phi devices, the Knights Corner family, can host up to 61 cores connected with a bi-directional ring interconnect. Each core has a 512 bit wide vector unit with FMA capability and the clock frequency can be as high as 1.238 GHz which makes an overall maximum peak performance of 1208 Gflop/s for double-precision, floating point computation. On-board memory can be as big as 16 GB and transfer data at a speed of 352 GB/s. One outstanding advantage of the Xeon Phi devices is that the instruction set is fully x86 compliant which allows for using "traditional" multithreading programming technologies such as OpenMP.

Figure 1.2: Block diagram of the Kepler K40m GPU.

Figure 1.3 shows a typical configuration of a modern HPC computing platform. This is formed by multiple (up to thousands) nodes connected through a high-speed network; each node may include multiple processors, each connected with a NUMA memory module. A node may also be equipped with one or more accelerators. Please note that the figure reports indicative values for performance, bandwidths and memory capacities and do not refer to any specific device. The figure shows that modern HPC platforms are based on extremely heterogeneous architectures as they employ processing units with different performance, memories with different capacities and interconnects with different latencies and bandwidths.

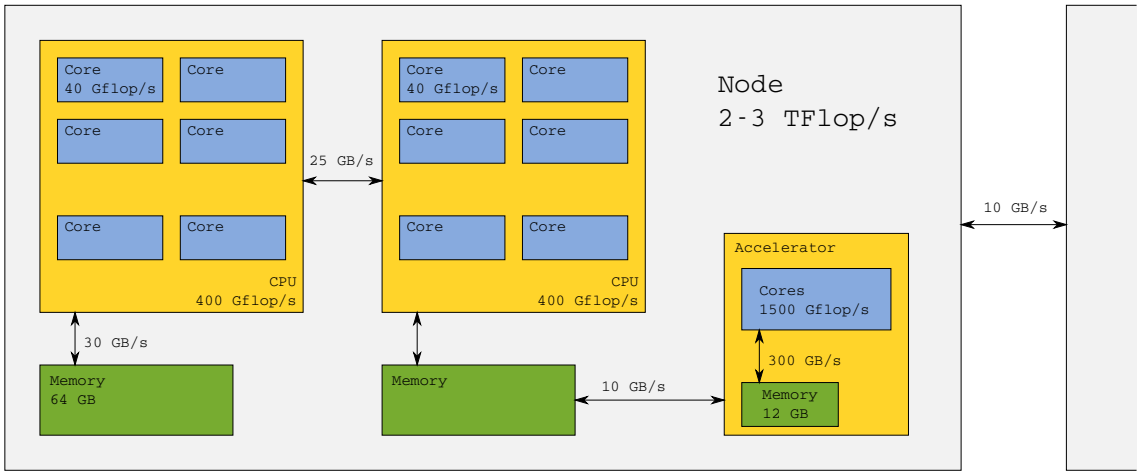


Figure 1.3: Illustration of a typical HPC computing platform architecture.

1.2 Linear systems and direct methods

1.2.1 Problems/applications

This thesis deals with the efficient and scalable implementation of direct solvers for large scale, sparse linear systems of equations on modern architectures equipped with multicore processors and accelerators, such as GPU devices. Specifically, this work focuses on sparse multifrontal solvers based on the QR factorization of the system matrix. This method decomposes the input matrix $A \in \mathbb{R}^{m \times n}$[2], assumed to be of full rank, into the product of a square, orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ and an upper triangular matrix $R \in \mathbb{R}^{n \times n}$.

Theorem 1.1 - Björck [23, Theorem 1.3.1].
Let $A \in \mathbb{R}^{m \times n}$, $m \geq n$. Then there is an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ such that

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{1.1}$$

where R is upper triangular with nonnegative diagonal elements. The decomposition (1.1) is called the QR decomposition of A, and the matrix R will be called the R-factor of A.

The columns of the Q matrix can be split in two groups

$$Q = [Q_1 Q_2] \tag{1.2}$$

where $Q_1 \in \mathbb{R}^{m \times n}$ is an orthogonal basis for the range of A, $\mathcal{R}(A)$ and $Q_2 \in \mathbb{R}^{m \times (m-n)}$ is an orthogonal basis for the kernel of A^T, $\mathcal{N}(A^T)$.

The QR decomposition can be used to solve square linear system of equations

$$Ax = b, \quad \text{with } A \in \mathbb{R}^{n \times n}, \tag{1.3}$$

[2]Without loss of generality here, and in the rest of this document, we will assume that the same algorithms and methods, programming techniques and experimental analysis can be generalized to the case of complex arithmetic.

as the solution x can be computed through the following three steps (where we use MATLAB notation)

$$\begin{cases} [QR] = qr(A) \\ z = Q^T b \\ x = R \backslash z \end{cases} \tag{1.4}$$

where, first, the QR decomposition is computed (e.g., using one of the methods described in the next section), an intermediate result is computed trough a simple matrix-vector product and, finally, solution x is computed through a triangular system solve. It must be noted that the second and the third steps are commonly much cheaper than the first and that the same QR decomposition can be used to solve matrix A against multiple right-hand sides b. If the right hand sides are available all together, then the second and third steps can each be applied at once to all of them. As we will explain the next two sections, the QR decomposition is commonly unattractive in practice for solving square systems mostly due to its excessive cost when compared to other available techniques, despite its desirable numerical properties.

The QR decomposition is instead much more commonly used for solving linear systems where A is overdetemined, i.e. where there are more equations than unknowns. In such cases, unless the right-hand side b is in the range of A, the system admits no solution; it is possible, though, to compute a vector x such that Ax is as close as possible to b, or, equivalently, such that the residual $\|Ax - b\|_2$ is minimized:

$$\min_x \|Ax - b\|_2. \tag{1.5}$$

Such a problem is called a *least-squares* problem and commonly arises in a large variety of applications such as statistics, photogrammetry, geodetics and signal processing. One typical example is given by linear regression where a linear model, say $f(x, y) = a + bx + cy$ has to be fit to a number of observations subject to errors $(f_i, x_i, y_i),\ i = 1, ..., m$. This leads to the overdetermined system

$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ \vdots & \vdots & \vdots \\ 1 & x_m & y_m \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}$$

Assuming the QR decomposition of a in Equation (1.1) has been computed and

$$Q^T b = \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} b = \begin{bmatrix} c \\ d \end{bmatrix}$$

we have

$$\|Ax - b\|_2^2 = \|Q^T Ax - Q^T b\|_2^2 = \|Rx - c\|_2^2 + \|d\|_2^2.$$

This quantity is minimized if $Rx = c$ where x can be found with a simple triangular system solve. This is equivalent to saying that $Ax = Q_1 Q_1^T b$ and thus solving Equation (1.5) amounts to finding the vector x such that Ax is the orthogonal projection of b over the range of A, as shown in Figure 1.4. Also note that $r = Q_2 Q_2^T b$ and thus r is the projection of b on the null space of A^T, $\mathcal{N}(A)$.

Another commonly used technique for the solution of the least-squares problem is the *Normal Equations* method. Because the residual r is in $\mathcal{N}(A^T)$

$$A^T(Ax - b) = 0$$

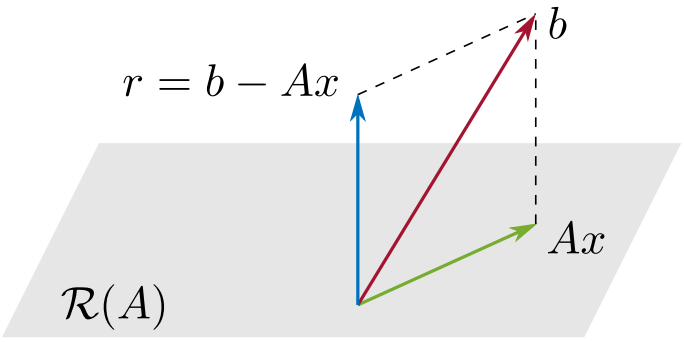


Figure 1.4: Solution of a least-squares problem.

and, thus, the solution x to Equation (1.5) can be found solving the linear system $A^TAx = A^Tb$. Because A^TA is Symmetric Positive Definite (assuming A has full rank), this can be achieved through the Cholesky factorization. Nonetheless, the method based on the QR factorization is often preferred because the conditioning of A^TA is equal to the square of the conditioning of A, which may lead to excessive error propagation.

The QR factorization is also commonly used to solve underdetermined systems, i.e., with more unknowns than equations, which admit infinite solutions. In such cases the desired solution is the one with minimum 2-norm:

$$min\|x\|_2, \quad Ax = b. \tag{1.6}$$

The solution of this problem can be achieved computing the QR factorization of A^T

$$[Q_1 Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix} = A^T$$

where $Q_1 \in \mathbb{R}n \times m$ and $Q_2 \in \mathbb{R}n \times (n - m)$. Then

$$Ax = R^TQ^Tx = [R^T 0] \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = b$$

and the minimum 2-norm solution follows by setting $z_2 = 0$. Note that Q_2 is an orthogonal basis for $\mathcal{N}(A)$ and, thus, the minimum 2-norm solution is computed by removing for any admissible solution $\tilde{x}$ all of its components in $\mathcal{N}(A)$.

For the sake of space and readability, we do not discuss here the use of iterative methods such as LSQR [91] or Craig [39] for the solution of problems (1.5) and (1.6) but we refer the reader to the book by Björck [23] which contains a thorough description of the classical methods used for solving least-squares and minimum 2-norm problems.

1.2.2 QR decomposition

The QR decomposition of a matrix can be computed in different ways; the use of Givens Rotations [57], Householder reflections [69] or the Gram-Schmidt orthogonalization [100] are among the most commonly used and best known ones. We will not cover here the use of Givens Rotations, Gram-Schmidt orthogonalization and their variants and refer, instead, the reader to classic linear algebra textbooks such as Golub et al. [59] or Björck [23] for an exhaustive discussion of such methods. We focus, instead, on the QR factorization based on Householder reflections which has become the most commonly used method especially

because of the availability of algorithms capable of achieving very high performance on processors equipped with memory hierarchies.

For a given a vector u, a *Householder Reflection* is defined as

$$H = I - 2\frac{uu^T}{u^T v} \tag{1.7}$$

and u is commonly referred to as *Householder vector*. It is easy to verify that H is symmetric and orthogonal. Because $P_u = \frac{uu^T}{u^T u}$ is a projector over the space of u, Hx can be regarded as the reflection of a vector x on the hyperplane that has normal vector u. This is depicted in Figure 1.5 (*left*).

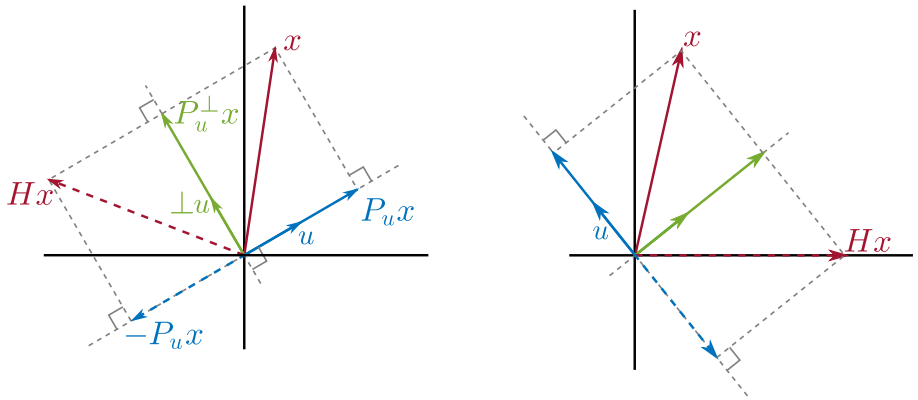


Figure 1.5: Householder reflection

The Householder reflection can be defined in such a way that Hx has all zero coefficients except the first, which means to say that $Hx = \pm\|x\|_2 e_1$ where e_1 is the first unit vector. This can be achieved if Hx is the reflection of x on the hyperplane that bisects the angle between x and $\pm\|x\|_2 e_1$. This hyperplane is orthogonal to the difference of these two vectors $x \mp \|x\|_2 e_1$ which can thus be used to construct the vector

$$u = x \mp \|x\|_2 e_1. \tag{1.8}$$

Note that, if, for example, x is close to a multiple of e_1, then $\|x\|_2 \approx x(1)$ which may lead to a dangerous cancellation in Equation (1.8); to avoid this problem u is commonly chosen as

$$u = u + sign(x(1))\|x\|_2 e_1. \tag{1.9}$$

In practice, it is very convenient to scale u in such a way that its first coefficient is equal to 1 (more on this will be said later). Assuming $v = u/u(1)$, through some simple manipulations, the Householder transformation H is defined as

$$H = I - \tau vv^T, \quad \text{where} \quad \tau = \frac{sign(x(1))(x(1) - \|x\|_2)}{\|x\|_2}. \tag{1.10}$$

Note that the matrix H is never explicitly built neither to store, nor to apply the Householder transformation; the storage is done implicitly by means of v and τ and the transformation can be applied to an entire matrix $A \in \mathbb{R}^{m\times n}$ in $4mn$ flops like this

$$HA = (I - \tau vv^T)A = A - \tau v(v^T A) \tag{1.11}$$

The Householder reflection can be computed and applied as described above using the `_larfg` and `_larf` routines in the LAPACK[15] library.

A sequence of Householder transformations can be used to zero-out all the coefficients below the diagonal of a dense matrix to compute its QR factorization:

$$H_n H_{n-1} ... H_2 H_1 A = R, \quad \text{where} \quad H_n H_{n-1} ... H_2 H_1 = Q^T.$$

Each transformation H_k annihilates all the coefficients below the diagonal of column k and modifies all the coefficients in the trailing submatrix $A(k : m, k + 1 : n)$. The total cost of this algorithm is $2n^2(m - n/3)$. The Q matrix is implicitly represented by means of the v_k vectors and the τ_k coefficients. One extra array has to be allocated to store the τ_k coefficients, whereas the v_k vectors can be stored inside matrix A in the same memory as the zeroed-out coefficients; this is possible because the v_k have been scaled as described above and thus the 1 coefficients along the diagonal must not be explicitly stored. The LAPACK `_geqr2` routine implements this algorithm.

Theorem 1.2 - Björck [23, Remark 2.4.2].
Let $\bar{R}$ denote the computed R. It can be shown that there exists an exactly orthogonal matrix $\bar{Q}$ (*not* the computed Q) such that

$$A + E = \bar{Q}\bar{R}, \quad \|E\|_F \leq c_1 u \|A\|_F,$$

where the error constant $c_1 = c_1(m, n)$ is a polynomial in m and n, $\|\cdot\|_F$ denotes the Frobenius norm and u the unit roundoff.

In other words, the Householder QR factorization is normwise backward stable.

The use of a QR factorization by mean of a sequence of orthogonal transformations to solve least-squares problems was introduced by Golub [58]; this method is also proven to be backward stable:

Theorem 1.3 - Björck [23, Remark 2.4.8].
Golub's method for solving the standard least squares problem is normwise backward stable. The computed solution $\hat{x}$ can be shown to be the exact solution of a slightly perturbed least squares problem

$$\min_x \|(A + \delta A)x - (b + \delta b)\|_2,$$

where the perturbations satisfy the bounds

$$\|\delta A\|_2 \leq cun^{1/2}\|A\|_2, \quad \|\delta b\|_2 \leq cu\|b\|_2$$

and $c = (6m - 3n + 41)n$.

Despite these very favorable numerical properties, the QR factorization is rarely preferred to the Gaussian Elimination (or LU factorization) with Partial Pivoting (GEPP) for the solution of square systems because its cost is twice the cost of GEPP and because partial pivoting is considered stable in most practical cases.

The above discussed algorithm for computing the QR factorization is basically never used in practice because it can only achieve a modest fraction of the peak performance of a modern processor. This is due to the fact that most of the computations are done in the application of the Householder transformation to the trailing submatrix as in Equation (1.11) which is based on Level-2 (i.e., matrix-vector) BLAS operations and thus limited by the

speed of the memory rather than the speed of the processor. In order to overcome this limitation and considerably improve the performance of the Householder QR factorization on modern computers equipped with memory hierarchies, Schreiber et al. [101] proposed a way of accumulating multiple Householder transformations and applying them at once by means of Level-3 BLAS operations.

Theorem 1.4 - Adapted form Schreiber et al. [101].
Let $Q = H_1 ... H_{k-1} H_k$, with $H_i \in \mathbb{R}^{m \times m}$ an Householder transformation defined as in Equation (1.10) and $k \leq m$. Then, there exist an upper triangular matrix $T \in \mathbb{R}^{k \times k}$ and a matrix $V \in \mathbb{R}^{m \times k}$ such that

$$Q = I - VTV^T.$$

Using this technique, matrix A can be logically split into $\lceil n/nb \rceil$ *panels* (block-columns) of size nb and the QR factorization achieved in the same number of steps where, at step k, panel k is factorized using the `_geqr2` routine, the corresponding T matrix is built using the `_larft` routine and then the set of nb transformations is applied at once to the trailing submatrix through the `_larfb` routine. This last operation/routine is responsible for most of the flops in the QR factorization: because it is based on matrix-matrix operations it can achieve a considerable fraction of the processor's peak performance. This method is implemented in the LAPACK `_geqrf` routine which uses an implicitly defined blocking size nb and discards the T matrices computed for each panel. More recently, the `_geqrt` routine has been introduced in LAPACK which employs the same algorithm but takes the block size nb as an additional argument and returns the computed T matrices.

1.2.3 Multifrontal QR methods

One of the most used and well known definitions of a sparse matrix is attributed to James Wilkinson:

> *"A sparse matrix is any matrix with enough zeros that it pays to take advantage of them".*

There are three main ways to take advantage of the fact that a matrix is mostly filled up with zeroes. First of all the memory needed to store the matrix is lesser than the memory needed for a dense matrix of the same size because the zeroes need not be stored explicitly. Second the complexity of numerous operations on a sparse matrix can be greatly reduced with respect to the same operation on a dense matrix of the same size because most of the zero coefficients in the original matrix can be skipped during the computation. Finally, parallelism can be much higher than in the dense case because some operations may affect distinct subsets of the matrix nonzeroes and can thus be applied concurrently. This clearly also applies to the QR factorization of a sparse matrix. We will make use of the example in Figure 1.6 to illustrate these advantages. The original matrix, shown on the left side of the figure, as well as the intermediate and final results can be stored in one of the many sparse storage formats such as Coordinate (COO) or Compressed Row Storage (CRS) which allow for storing a matrix with nz nonzero coefficients in a memory of order $O(nz)$. More specialized formats can be used for example depending of the specific algorithm that one has to implement (the multifrontal method described below is one of these) or on the particular architecture that has to be used. Now, imagine we apply the first step of a (unblocked) QR factorization to this matrix. As described in

the previous section, we want to annihilate all the coefficients below the diagonal in the first column: there are only two of them rather than $m-1$ and thus the Householder reflection can be computed at a much lower cost. The produced Householder vector has the same structure as the first column of A and, therefore, when the reflection is applied to the trailing submatrix, only part of its coefficients will be updated. Specifically, only coefficients along the rows that have a nonzero in the pivotal column (the first column in this case) and in the columns that have at least one nonzero in the same position as the pivotal column. As a result, the trailing submatrix update is also much cheaper than in the dense case because it only touches a restricted set of coefficients. Finally, it must be noted that the elimination of column one and of column two affect two disjoint sets of coefficients of A and can thus be computed in parallel.

Figure 1.6 also illustrate a very peculiar phenomenon of sparse matrix factorizations. Once the trailing submatrix is updated in an elimination step, all the concerned coefficients, identified as described above, are nonzero, even those that were zero in the original matrix. This is know as $fill-in$ and is represented by red underlined coefficients in Figure 1.6. It must be noted that fill-in coefficients generate, in turn, more fill-in in the subsequent steps of the factorization and thus the columns of the trailing submatrix become denser and denser as the factorization proceeds. The fill-in is such that the factors resulting from a sparse factorization are much denser than the original matrix even though they are still sparse, according to Wilkinson's definition.

It must be noted that when eliminating column k, the pivotal row is not necessarily row k as in the dense factorization, i.e., at the end of the factorization the coefficient of the R factor (the V matrix) are not necessarily above (below) the diagonal. In the example above, the R factor is distributed along rows $[1, 2, 3, 13, 7, 9, 15, 16, 8]$.

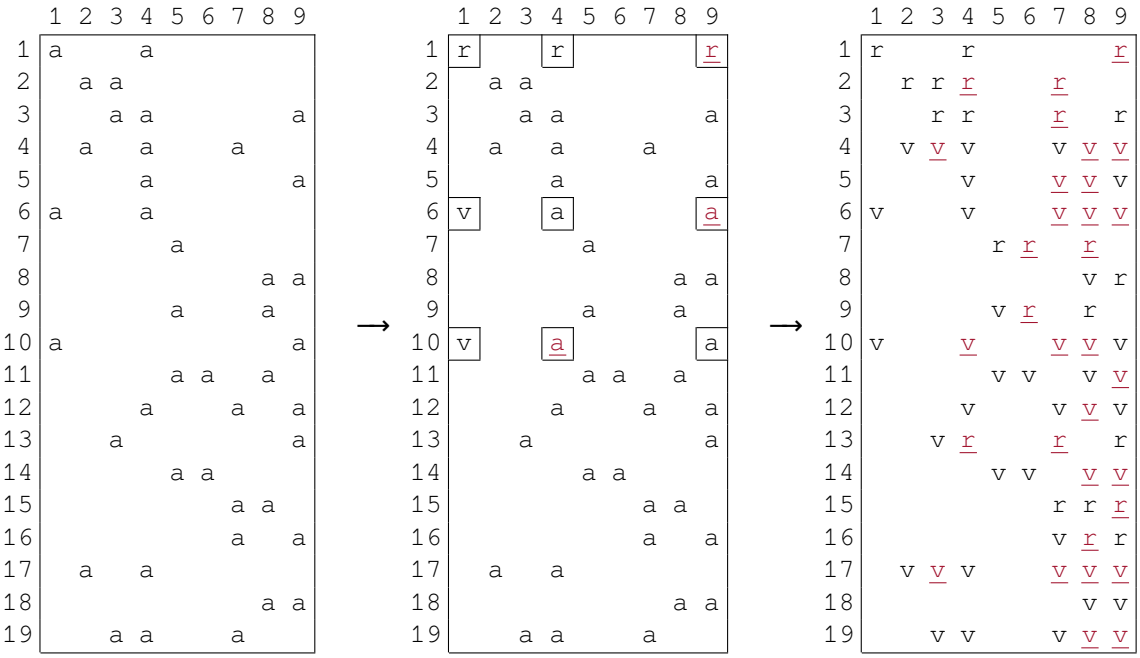


Figure 1.6: QR factorization of a sparse matrix: original matrix (*left*), first step (*middle*) and final result (*right*).

Taking advantage of the sparsity of a matrix in a sparse factorization is, however, very complex and requires a considerable amount of logic to retrieve nonzero coefficients, add

new ones, identify tasks that can be performed in parallel etc. Graph theory can assist in achieving these tasks as it allows for computing matrix permutations that reduce the fill-in, computing the number and position of fill-in coefficients, defining the dependencies between the elimination steps of the factorization. All this information is commonly computed in a preliminary phase, commonly referred to as *analysis* whose cost is $O(|A|+|R|)$ where $|A|$ and $|R|$ are, respectively, the number of nonzeroes in A and R. We refer the reader to classic textbooks on direct, sparse methods such as the ones by Duff et al. [48] or Davis [41] for a thorough description of the techniques used in the analysis phase. One this symbolic, structural information is available upon completion of the analysis, the actual factorization can take place using different techniques. For the sparse QR factorization, the most commonly employed technique is the multifrontal method which was first introduced by Duff and Reid [49] as a method for the factorization of sparse, symmetric linear systems. At the heart of this method is the concept of an *elimination tree*, introduced by Schreiber [102] and extensively studied and formalized later by Liu [84]. This tree graph, computed at the analysis phase, describes the dependencies among computational tasks in the multifrontal factorization. The multifrontal method can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix A^TA share the same structure under the hypothesis that the matrix A is *Strong Hall*:

Definition 1.1 - Strong Hall matrix [23, Definition 6.4.1].
A matrix $A \in \mathbb{R}^{m\times n}$, $m \geq n$ is said to have the **Strong Hall** property if every subset of k columns, $0 < k < n$, the corresponding submatrix has nonzeroes in at leas $k+1$ rows.

Based on this equivalence, the structure of the R factor of A is the same as that of the Cholesky factor of A^TA (excluding numerical cancellations) and the elimination tree for the QR factorization of A is the same as that for the Cholesky factorization of A^TA. In the case where the Strong Hall property does not hold, the elimination tree related to the Cholesky factorization of A^TA can still be used although the resulting QR factorization will perform more computations and consume more memory than what is really needed; alternatively, the matrix A can be permuted to a Block Triangular Form (BTF) where all the diagonal blocks are Strong Hall.

In a basic multifrontal method, the elimination tree has n nodes, where n is the number of columns in the input matrix A, each node representing one pivotal step of the QR factorization of A. This tree is defined by a parent relation where the parent of node i is equal to the index of the first off-diagonal coefficient in row i of the factor R. The elimination tree for the matrix in Figure 1.6 is depicted in Figure 1.7. Every node of the tree is associated with a dense *frontal matrix* (or, simply, *front*) that contains all the coefficients affected by the elimination of the corresponding pivot. The whole QR factorization consists in a topological order (i.e., bottom-up) traversal of the tree where, at each node, two operations are performed:

- **assembly**: a set of rows from the original matrix (all those which have a nonzero in the pivotal column) is assembled together with data produced by the processing of child nodes to form the frontal matrix. This operation simply consists in stacking these data one on top of the other: the rows of these data sets can be stacked in any order but the order of elements within rows has to be the same. The set of column indices associated with a node is, in general, a superset of those associated with its children;

- **factorization**: one Householder reflector is computed and applied to the whole frontal matrix in order to annihilate all the subdiagonal elements in the first column. This step produces one row of the R factor of the original matrix and a complement which corresponds to the data that will be later assembled into the parent node (commonly referred to as a *contribution block*). The Q factor is defined implicitly by means of the Householder vectors computed on each front; the matrix that stores the coefficients of the computed Householder vectors, will be referred to as the V matrix from now on.

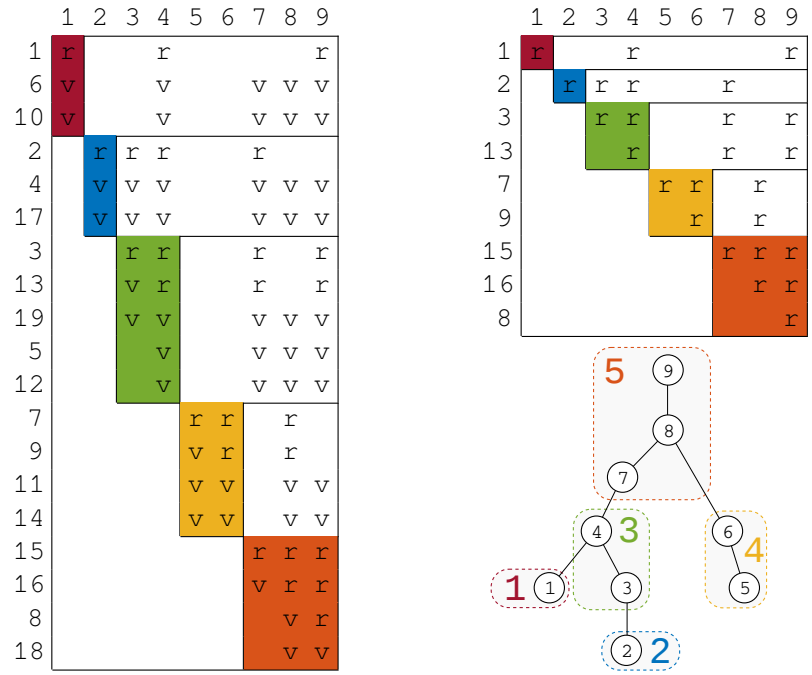


Figure 1.7: Example of multifrontal QR factorization. On the left side the factorized matrix (the same as in Figure 1.6 with a row permutation). On the upper-right part, the structure of the resulting R factor. On the right-bottom part the elimination tree; the dashed boxes show how the nodes are amalgamated into supernodes.

In practical implementations of the multifrontal QR factorization, nodes of the elimination tree are amalgamated to form *supernodes*. The amalgamated pivots correspond to rows of R that have the same structure and can be eliminated at once within the same frontal matrix without producing any additional fill-in in the R factor. The elimination of amalgamated pivots and the consequent update of the trailing frontal submatrix can thus be performed by means of efficient Level-3 BLAS routines through the WY representation [101]. Moreover, amalgamation reduces the number of assembly operations increasing the computations-to-communications ratio which results in better performance. The amalgamated elimination tree is also commonly referred to as *assembly tree*.

Figure 1.7 shows some details of a sparse QR factorization. The factorized matrix is shown on the left part of the figure. Note that this is the same matrix as in Figure 1.6 where the rows of A are sorted in order of increasing index of the leftmost nonzero in order to show more clearly the computational pattern of the method on the input data. Actually, the sparse QR factorization in insensitive to row permutations which means that any row permutation will always yield the same fill-in, as it can be verified comparing Figures 1.6 and 1.7. On the top-right part of the figure, the structure of the resulting R factor is shown.

The elimination/assembly tree is, instead reported in the bottom-right part: dashed boxes show how the nodes can be amalgamated into supernodes with the corresponding indices denoted by bigger size numbers. The amalgamated nodes have the same row structure in R modulo a full, diagonal block. It has to be noted that in practical implementations the amalgamation procedure is based only on information related to the R factor and, as such, it does not take into account fill-in that may eventually appear in the V matrix. The supernode indices are reported on the left part of the figure in order to show the pivotal columns eliminated within the corresponding supernode and on the top-right part to show the rows of R produced by the corresponding supernode factorization.

In order to reduce the operation count of the multifrontal QR factorization, two optimizations are commonly applied:

1. once a frontal matrix is assembled, its rows are sorted in order of increasing index of the leftmost nonzero. The number of operations can thus be reduced, as well as the fill-in in the V matrix, by ignoring the zeroes in the bottom-left part of the frontal matrix;

2. the frontal matrix is completely factorized. Despite the fact that more Householder vectors have to be computed for each frontal matrix, the overall number of floating point operations is lower since frontal matrices are smaller. This is due to the fact that contribution blocks resulting from the complete factorization of frontal matrices are smaller.

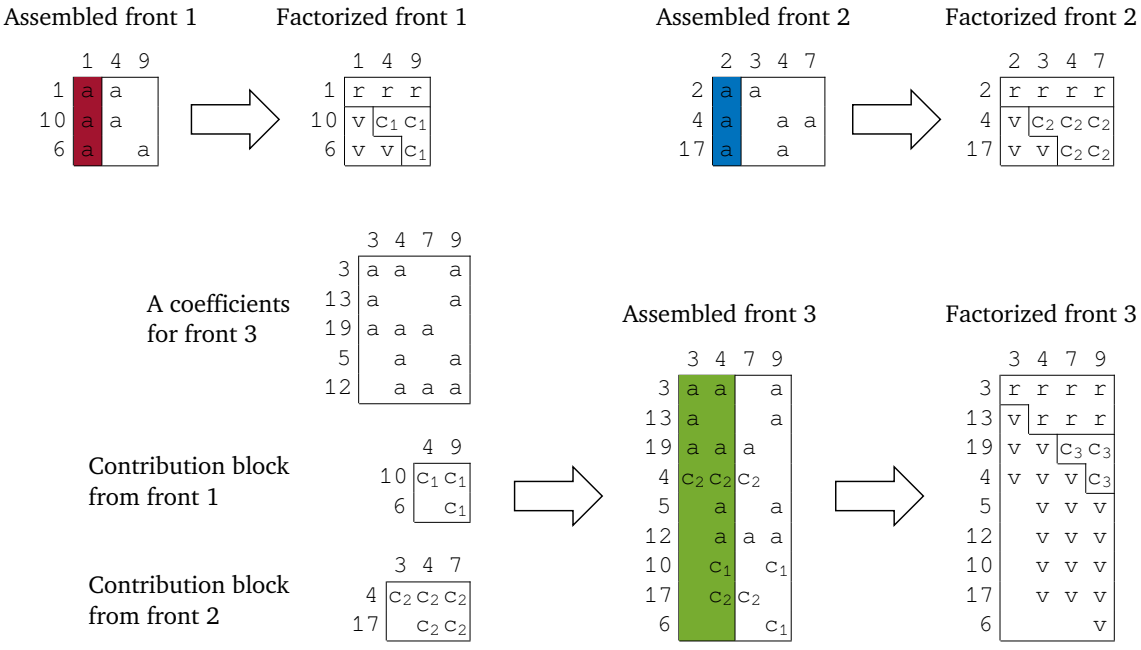


Figure 1.8: Assembly and factorization of the frontal matrices associated with supernodes 1, 2 and 3 in Figure 1.7. Coefficients from the original matrix are represented as `a`, those in the resulting R and V matrices as `r` and `v`, respectively, the fill-in coefficients as dots and the coefficients in the contribution blocks as `c` with a subscript to specify the supernode they belong to.

Figure 1.8 shows the assembly and factorization operations for the supernodes 1, 2 and 3 in Figure 1.7 when these optimization techniques (referred to as *Strategy 3* in

Amestoy et al. [14]) are applied. Note that, because supernodes 1 and 2 are leaves of the assembly tree, the corresponding assembled frontal matrices only include coefficients from the matrix A. The contribution blocks resulting from the factorization of supernodes 1 and 2 are appended to the rows of the input A matrix associated with supernode 3 in such a way that the resulting, assembled, frontal matrix has the *staircase* structure shown in Figure 1.8 (*bottom-middle*). Once the front is assembled, it is factorized as shown in Figure 1.8 (*bottom-right*).

A detailed presentation of the multifrontal QR method, including the optimization techniques described above, can be found in the work by Amestoy et al. [14] or Davis [41].

The multifrontal method can achieve very high efficiency on modern computing systems because all the computations are arranged as operations on dense matrices; this reduces the use of indirect addressing and allows the use of efficient Level-3 BLAS routines which can achieve a considerable fraction of the peak performance of modern computing systems.

1.3 Parallelization of the QR factorization

1.3.1 Dense QR factorization

Computing the QR factorization of a dense matrix is a relatively expensive task because as explained above, its cost is $O(n^2m)$ flops. Therefore, especially when dealing with large size problems, parallelization is a natural way of reducing the execution time. In addition, due to its high computational intensity (i.e., the number of operations divided by the amount of data manipulated), the QR factorization of a dense matrix can potentially achieve a good scaling under favorable circumstances (more on these below).

As explained in Section 1.2.2, the blocked QR factorization is a succession of panel reductions (`_geqrt`) and update (`_gemqrt`) operations. The panel reduction is essentially based on Level-2 BLAS operations. This means that this operation cannot be efficiently parallelized because its cost is dominated by data transfers, either from main memory to the processor, or among processors; moreover, the computation of a Householder reflector implies computing the norm of an entire column and therefore, at each internal step of the panel reduction, if the panel is distributed among multiple processors a heavily penalizing reduction operation has to take place. The trailing submatrix update, instead is very rich in Level-3 BLAS operations and is thus an ideal candidate for parallelization because the data transfers can easily overlapped with computations.

A very classic and straightforward way to parallelize the blocked QR factorization described in Section 1.2.2 is based on a fork-join model where sequential panel operations are alternated with parallel updates. This can be easily implemented if the trailing submatrix is split into block-columns and each block-column updated by a different process independently of the others, as in the interior loop of the following pseudocode:

```
do k=1, n/nb
   ! reduce panel a(k)
   call _geqrt(a(k))

   do j=k+1, n/nb
      ! update block-column a(j) w.r.t. panel a(k)
      call _gemqrt(a(k), a(j))
   end do
end do
```

The scalability of this approach can be improved noting that it is not necessary to wait for the completion of all the updates in step `k` to compute the reduction of panel `a(k+1)` but this can be started as soon as the update of block-column `a(k+1)` with respect to panel `a(k)` is finished. This technique is well known under the name of *lookahead* and essentially allows for pipelining two consecutive stages of the blocked QR factorization; the same idea can be pushed further in order to pipeline l stages, in which case we talk of `depth-l` lookahead. In the rest of this document we will refer to this approach as the 1D parallelization. Despite the improvements brought by lookahead, its scalability remains extremely poor; even if we had enough resources to perform concurrently all the updates in each stage of the blocked factorization, the execution time will be severely limited by the slow and sequential panel reductions. This is especially true for extremely overdetermined matrices where the cost of panel reductions becomes very high relative to the cost of update operations.

For this reason, when multicore architectures started to appear (around the beginning of the 2000s) and the core count of supercomputers started to ramp up, novel algorithms were introduced to overcome these limitations. These methods, known under the name of *tile* or *communication-avoiding* algorithms are based on a 2D decomposition of matrices into tiles (or blocks) of size `nb x nb`; this permits to break down the panel factorization and the related updates into smaller tasks which leads to a three-fold advantage:

1. the panel factorization and the related updates can be parallelized;
2. the updates related to a panel stage can be started before the panel is entirely reduced;
3. subsequent panel stages can be started before the panel is entirely reduced.

In these 2D (because of the decomposition into square blocks) algorithms, the topmost panel tile (the one lying on the diagonal) is used to annihilate the others; this can be achieved in different ways. For example, the topmost tile can be first reduced into a triangular tile with a general `_geqrt` LAPACK QR factorization which can then be used to annihilate the other tiles, one after the other, with `_tpqrt` LAPACK QR factorizations, where `t` and `p` stand for *triangular* and *pentagonal*, respectively. The panel factorization is thus achieved through a flat reduction tree as shown in Figure 1.10 (*left*). This approach allows for a better pipelining of the tasks (points 2 and 3 above); in the case of extremely overdetermined matrices, however, it provides only a moderate additional concurrency with respect to the 1D approach because the tasks within a panel stage cannot be executed concurrently. Another possible approach consists in first reducing all the tiles in the panel into triangular tiles using `_geqrt` operations; this first stage is embarrassingly parallel. Then, through a binary reduction tree, all the triangular tiles except the topmost one are annihilated using `_tpqrt` operations, as shown in Figure 1.10 (*middle*). This second method provides better concurrency but results in a higher number of tasks, some of which are of very small granularity. Also, in this case a worse pipelining of successive panel stages is obtained [47]. In practice a hybrid approach can be used where the panel is split into subsets of size `bh`: each subset is reduced into a triangular tile using a flat reduction tree and then, the remaining tiles are reduced using a binary tree. This is the technique used in the PLASMA library [9] and illustrated in Figure 1.10 (*right*) for the case of `bh`$= 2$. Other reduction trees can also be employed; we refer the reader to the work by Dongarra et al. [47] for an excellent survey of such techniques.

It must be noted that these algorithms require some extra flops with respect to the standard blocked QR factorization [34]; this overhead can be reduced to a negligible level by choosing and appropriate internal block size of the elementary kernels described above.

```
do k=1, n/nb
   ! for all the block-columns in the front
   do i = k, m/nb, bh
      call _geqrt(f(k,i))
      do j=k+1, n/nb
         call _gemqrt(f(k,i), f(i,j))
      end do
      ! intra-subdomain flat-tree reduction
      do l=i+1, min(i+bh-1,m/nb)
         call _tpqrt(f(i,k), f(l,k))
         do j=k+1, n/nb
            call _tpmqrt(f(l,k), f(i,j), f(l,j))
         end do
      end do
   end do
   do while (bh.le.m/nb-k+1)
      ! inter-subdomains binary-tree reduction
      do i = k, m/nb-bh, 2*bh
         l = i+bh
         if(l.le.m/nb) then
            call _tpqrt(f(i,k), f(l,k))
            do j=k+1, n
               call _tpmqrt(f(l,k), f(i,j), f(l,j))
            end do
         end if
      end do
      bh = bh*2
   end do
end do
```

Figure 1.9: Pseudo-code showing the implementation of the tiled QR.

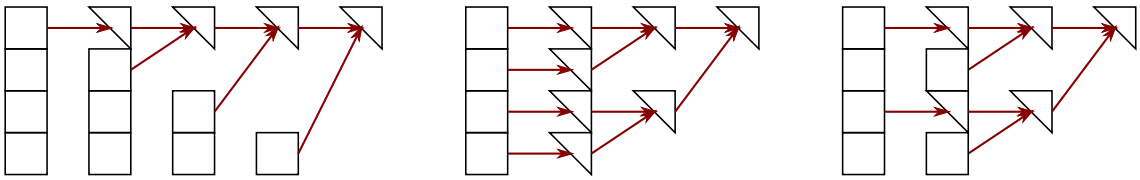


Figure 1.10: Possible panel reduction trees for the 2D front factorization. On the (*left*), the case of a flat tree, i.e., with bh$=\infty$. In the (*middle*), the case of a binary tree, i.e., with (bh$=1$). On the (*right*) the case of an hybrid tree with bh$=2$.

These communication avoiding algorithms have been the object of numerous theoretical studies [21, 22, 29] and have been used to accelerate the dense QR factorization on multicore systems either in single-node, shared memory settings [10, 61, 31, 34, 35], on distributed memory, multicore parallel machines [47, 44, 105, 116], GPU equipped systems [4, 16, 116] and even on Grid environments [8].

1.3.2 Sparse QR factorization

As explained in Section 1.2.3, sparsity implies that, when performing an operation on a matrix, it is often possible to identify independent tasks that can be executed concurrently, i.e., sparsity implies parallelism. In sparse, direct solvers and thus, being one of them, in the multifrontal QR method, this concurrency is expressed by the elimination tree: fronts that belong to different branches are independent and can thus be treated in any order and, possibly, in parallel. We refer to this type of parallelism as *tree parallelism*. The amount of concurrency available in the tree parallelism clearly depends on the shape of the tree and on the distribution of the computational load along its branches and, as a consequence, on the sparsity structure of the input matrix. As explained above, prior to its factorization a sparse matrix is permuted in order to reduce the fill-in; the shape of the elimination tree clearly depends on this permutation and, consequently, the amount of tree parallelism does too. Local methods such as Average Minimum Degree [12] (AMD) or its column variant COLAMD [40], although capable of reducing the fill-in, commonly lead to deep and rather unbalanced elimination trees where tree parallelism is difficult to exploit, especially in distributed memory systems where load balancing is harder to achieve. Nested-dissection [55] based methods instead, not only are very effective in reducing fill-in (much better than local methods especially for large scale problems), but also lead to wider (binary) and better balanced elimination trees because the load balancing can be taken into account by the dissector selection criterion.

Fronts, although much smaller in size than the input sparse matrix, are not small in general. Some of them may have hundred thousands rows or column and, consequently, their factorization may require a considerable amount of work. This provides a second source of concurrency which is commonly referred to as *front* or *node parallelism*. Any of the algorithms presented in the previous section can be used to factorize a frontal matrix using multiple processes.

The degree of concurrency in tree and node parallelism changes during the bottom-up traversal of the tree. Fronts are relatively small at the leaf nodes of the tree and grow bigger towards the root node; as a result, front parallelism is scarce at the bottom of the tree and abundant at the top. On the other hand, tree parallelism provides a high amount of concurrency at the bottom of the tree and only a little at the top part where the tree shrinks towards the root node. Because of this complementarity, using both types of parallelism is essential for achieving a good scalability. This is clearly much more difficult in the distributed memory parallel case where a careful mapping of processes to tree and node parallelism has to be done.

1.4 Programming models and runtime systems

Computing platform hardware has dramatically evolved ever since the computer science began, always striving to provide new convenient accelerating features. Each new accelerating hardware feature inevitably leaves programmers to decide whether to make their application dependent on that feature (and break compatibility) or not (and miss the potential benefit), or even to handle both cases (at the cost of extra management code in the application). This common problem is known as the performance portability issue.

The first purpose of runtime systems is thus to provide *abstraction*. Runtime systems offer a uniform programming interface for a specific subset of hardware (e.g., OpenGL or DirectX are well-established examples of runtime systems dedicated to hardware-accelerated graphics) or low-level software entities (e.g., POSIX-thread implementations). They are designed as thin user-level software layers that complement the basic, general purpose

functions provided by the operating system calls. Applications then target these uniform programming interfaces in a portable manner. Low-level, hardware dependent details are hidden inside runtime systems. The adaptation of runtime systems is commonly handled through drivers. The abstraction provided by runtime systems thus enables portability. Abstraction alone is however not enough to provide portability of performance, as it does nothing to leverage low-level-specific features to get increased performance.

Consequently, the second role of runtime systems is to *optimize* abstract application requests by dynamically mapping them onto low-level requests and resources as efficiently as possible. This mapping process makes use of scheduling algorithms and heuristics to decide the best actions to take for a given metric and the application state at a given point in its execution time. This allows applications to readily benefit from available underlying low-level capabilities to their full extent without breaking their portability. Thus, optimization together with abstraction allows runtime systems to offer portability of performance.

In the specific case of parallel work mapping, other approaches have occasionally been adopted instead of using runtimes. Many scientific applications and libraries, including linear system solvers, integrate their own, customized dynamic scheduling algorithms or even resort to static scheduling techniques, either for historical reasons, or to avoid the potential overhead of an extra runtime layer.

However, as multicore processors densify, as cache and memory hierarchies deepen, the resulting increase in complexity now makes the use of work-mapping runtime systems virtually unavoidable. Such work-mapping runtime systems take elementary task descriptions and dependencies as input and are responsible for dynamically scheduling the tasks on available computing units so as to minimize a given cost function (usually the execution time) under some pre-defined set of constraints.

Work-mapping runtime systems themselves are now facing new challenges with the recent move of the high performance community towards the use of specialized accelerating cores together with traditional general-purpose cores. They not only have to decide about the interest (or not) to use some specific hardware features, but also have to decide whether some entire application tasks should rather be performed on an accelerated core or is better left on a standard core.

In the case where specialized cores are located on an expansion card having its own memory (e.g., most existing GPUs), the input data of a task have to be copied from central memory to the card memory before the task can be run. The output results must also be copied back to the central memory once the task computation is complete. The cost of copying data between central memory and accelerator memory is not negligible. This cost, as well as data dependencies between tasks, must therefore also be taken into account by the scheduling algorithms when deciding whether to offload a given task, to avoid unnecessary data transfers. Transfers should also be done in advance and asynchronously so as to overlap communication with computation.

1.4.1 Programming models for task-based applications

Modern task-based runtime systems aim at abstracting the low-level details of the hardware achitechture and enhance the portability of the performance of the code designed on top of them. As it is the case in this thesis, in most cases, this abstraction relies on a *DAG of tasks*. In this DAG, vertices represent the tasks to be executed while edges represent the dependences between them.

While tasks are almost systematically explicitly encoded, runtime systems offer multiple ways to encode the dependencies of the DAG. Each runtime system usually comes

submit_task(fun_1, $\underline{x}$, y, id=id_1)

submit_task(fun_2, x, id=id_2)

declare_dependency($id_3 \leftarrow id_1$)

submit_task(fun_1, $\underline{z}$, w, id=id_3)

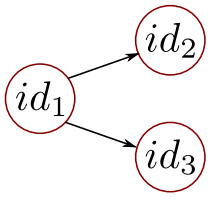


Figure 1.11: Pseudo-code (left) and associated DAG (right). Arguments corresponding to data that are modified by the function are underlined. The $id_1 \rightarrow id_3$ dependency is declared explicitly while the $id_1 \rightarrow id_2$ dependency is implicitly inferred with respect to the data hazard on x.

with its own API which includes one or multiple ways to encode the dependencies and their exhaustive listing would be out of the scope of this thesis. However, we may consider that there are two main modes for encoding dependencies. The most natural method consists in declaring *explicit dependencies* between tasks. In spite of the simplicity of the concept, this approach may have a limited productivity in practice as some algorithms may have dependencies that are difficult to express. Alternatively, dependencies may be implitcly computed by the runtime system thanks to the *sequential consistency*. In this latter approach, tasks are provided in sequence and the data they operate on are also declared.

We illustrate these two dominant modes of expression of dependencies with a simple example relying on a minimum number of pseudo-instructions. Assume we want to encode the DAG shown in Figure 1.4.1 (right) relying on an explicit dependency between tasks id_1 and id_3 and an implicit dependency between tasks id_1 and id_2. A task can be defined as an instance of a function working on a specific set of data, different tasks possibly being different instances of a same function. For instance, in our example we assume that tasks id_1 and id_3 are instances of function fun_1 while task id_2 is an instance of function fun_2. While tasks are instanciated with the `submit_task` pseudo-instruction (see Figure 1.4.1, left), the explicit dependency between tasks id_1 and id_3 can simply be encoded with a `declare_dependency` pseudo-instruction (see Figure 1.4.1, left). On the other hand, implicit dependencies aim letting the runtime system automatically infer dependencies thanks to so-called superscalar analysis [11] which aims at ensuring that the parallelization does not violate dependencies, following the sequential consistency. While CPUs implement such a superscalar analysis on chip at the instruction level [11], runtime systems implement it in a software layer on tasks. Superscalar analysis is performed on tasks and the associated input/output data they operate on. Assume that task id_1 operates on data x and y in read/write mode and read mode (calling $fun_1(\underline{x}, y)$ if the arguments corresponding to data which are modified by a function are underlined), respectively, while task id_2 operates on data x in read mode ($fun_2(x)$). Because of possible data hazards occurring on x between tasks id_1 and id_2, the superscalar analysis detects that a dependency is required to respect the sequential consistency.

Another important paradigm for handling dependencies consists of *recursive submission*. Indeed, it may be convenient for the programmer to let tasks trigger other tasks. Sometimes, one may furthermore need the task to be fully completed and cleaned up before triggering other tasks. Runtime systems often support this option through a so-called *call-back* mechanism consisting of a post-processing portion of code executed once the task is completed and cleaned up.

Depending on the context, the programmer affinity and portion of the algorithm to encode, different paradigms may be considered as natural and appropriate. For instance,

we propose and study a task-based multifrontal method relying a combination of these four types of dependencies (explicit, implicit, recursive, call-back) in Section 3.2.1 in order to reproduce as accurately as possible the behavior of the original `qr_mumps` code with a general purpose runtime system. Although relatively natural to write as it follows the trends of the original code, the resulting task-based code is complex and difficult to maintain because it relies on multiple paradigms for handling dependencies. Alternatively, one may rely on a well-defined and more simple programming model in order to design a relatively more simple code, easier to maintain and benefit from properties provided by the model. The *Sequential Task Flow* (STF) programming model consists on fully relying on `sequential consistency` using only implicit dependencies. The STF model, therefore, consists of submitting a sequence of tasks through a non blocking function call that delegates the execution of the task to the runtime system. Upon submission, the runtime system adds the task to the current DAG along with its dependencies which are automatically computed through data dependency analysis [11]. The actual execution of the task is then postponed to the moment when its dependencies are satisfied. As mentioned above, this paradigm is also sometimes referred to as *superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies. We propose and study the design of a multifrontal method based on this model in Chapter 4. We show that the simplicity of the model allows for designing more advanced numerical algorithms with an concise yet effective expression. We also rely on properties we can derive from the model in order to design a memory-aware mechanism in Chapter 5 and extend the scope of the method to heterogeneous platforms in Chapter 6.

One challenge in scaling to large scale many-core systems is how to represent extremely large DAGs of tasks in a compact fashion. Cosnard et al. [38] presented a model, namely the *Parameterized Task Graph* (PTG), which addresses this issue. In this model, tasks are not enumerated but parametrized and dependencies between tasks are explicit. For instance, in the DAG represented in Figure 1.4.1 (right), tasks id_1 and id_3 are two instances of the same type of task implementing fun_1. This property can be used to encode the DAG in a compact way inducing a lower memory footprint for its representation as well as ensuring limited complexity for parsing it while the problem size grows. For this reason the memory consumption overhead in the runtime system for representing the DAG can much lower for the PTG model than for the STF model. In addition with a STF model the DAG has to be completely unrolled whereas with a PTG the DAG is only partially unfolded during the execution following the task progression. From this point of view, the advantage of the PTG approach over the STF one can be crucial in a distributed memory context because the DAG is pruned on every nodes and only a portion of the DAG is represented on each nodes. This could considerably reduce the runtime overhead for the management of the DAG. On the other hand, knowing the entire DAG can be useful to compute the schedule of the DAG or give information to the dynamic scheduler by prepossessing the DAG. For these reasons we discuss the potential advantages of relying on such a model for designing a multifrontal method in Section 7.1.

1.4.2 Task-based runtime systems for modern architectures

Many initiatives have emerged in the past years to develop efficient runtime systems for modern heterogeneous platforms. Most of these runtime systems use a task-based paradigm to express concurrency and dependencies by employing a task dependency graph to represent the application to be executed. The main differences between all the ap-

proaches are related, to the programming model, to whether or not they manage data movements between computational resources and to which extent they focus on task scheduling.

Some runtime systems have been specifically designed for the development of parallel linear algebra applications. One of these if the TBLAS runtime system [106], which provides a simple interface to create dense linear algebra applications and automates data transfers. TBLAS assumes that programmers should statically map data on the different processing units but it supports heterogeneous data block sizes (i.e., different granularity of computations). The QUARK runtime system [79] was specifically designed for scheduling linear algebra kernels on multi-core architectures. It is characterized by a scheduling algorithm based on work-stealing and by its higher scalability in comparison with other dedicated runtime systems. Finally, the SuperMatrix runtime system [37], follows nearly the same idea as it represents the matrix hierarchically: the matrix is viewed as blocks that serve as units of data where operations over those blocks are treated as units of computation. The implementation transparently enqueues the required operations, internally tracking dependencies, and then executes the operations utilizing out-of-order execution techniques.

Most of the available runtime systems, however, do not target any specific type of applications and provide a general API. Qilin [87], for example, provides an interface to submit kernels that operate on arrays which are automatically dispatched between the different processing units of an heterogeneous machine. Moreover, Qilin dynamically compiles parallel code for both CPUs (by relying on the Intel TBB [95] technology) and for GPUs, using CUDA. Another relevant framework is Charm++ [75] which is a parallel variant of the C++ language that provides sophisticated load balancing and a large number of communication optimization mechanisms. Charm++ has been extended to provide support for accelerators such as the Cell processors as well as GPUs [78]. Many runtime systems propose a task-based programming paradigm. Runtime systems like KAAPI/XKAAPI [53] or APC+ [62], Legion [31], Realm [112] offer support for hybrid platforms mixing CPUs and GPUs. Their data management is based on a DSM-like mechanism: each data block is associated with a bitmap that permits to determine whether there is already a copy locally available to a specific processing unit or not. Moreover, task scheduling within KAAPI is based on work-stealing mechanisms or on graph partitioning. The StarSs project is actually an umbrella term that describes both the StarSs language extensions and a collection of runtime systems targeting different types of platforms [20, 19]. StarSs provides an annotation-based language which extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system. The PaRSEC [26, 25] (formerly DAGuE) runtime system dynamically schedules tasks within a node using a rather simple strategy based on a locality-aware work-stealing strategy. It was first introduced for linear algebra but was later extended to more generic applications. It takes advantage of the specific shape of the task graphs (in the sense that there are few types of tasks) to represent the task dependency graph in an algebraic fashion. More details on this tool are given in Section 1.4.2.2 The StarPU runtime system provides a generic interface for developing parallel, task-based applications. It supports multicore architectures equipped with accelerator as well as distributed memory systems. This runtime is capable of transparently handling data and provides a rich panel of features. The details of this runtime systems are given in Section 1.4.2.1. All the above mentioned efforts have contributed to proving the ease of use, the effectiveness and portability of general purpose runtime systems to the point where the OpenMP board has decided to include similar features in the latest OpenMP standard 4.0: the `task` construct was extended with the `depend` clause which enables the OpenMP runtime to automatically detect dependencies

among tasks and consequently schedule them. The same OpenMP standard also provides constructs for using accelerator devices.

Whereas task-based runtime systems were mainly research tools in the past years, their recent progress makes them now solid candidates for designing advanced scientific software as they provide programming paradigms that allow the programmer to express concurrency in a simple yet effective way and relieve him from the burden of dealing with low-level architectural details.

The work presented in this thesis relies on the StarPU runtime system. This is mostly due to its large set of features which include full control over the scheduling policy, support for accelerators and distributed memory parallelism and transparent handling of data. For this reason, this runtime is described in more details in Section 1.4.2.1. Part of our work is also focused on evaluating the use of the PaRSEC runtime system for the implementation of the multifrontal QR factorization. This runtime supports a radically different programming model with respect to StarPU; more details can be found in Section 1.4.2.2.

1.4.2.1 The StarPU runtime system

StarPU is a runtime system developed by the STORM (formerly RUNTIME) team at Inria Bordeaux specifically designed for the parallelization of algorithms on heterogeneous architectures. A complete description of StarPU can be found in the work by Augonnet et al. [18].

StarPU provides an interface which is extremely convenient for implementing and parallelizing applications or algorithms that can be described as a graph of tasks. Tasks have to be explicitly submitted to the runtime system along with the data they work on and the corresponding data access mode. Through data analysis, StarPU can automatically detect the dependencies among tasks and build the corresponding DAG. Once a task is submitted, the runtime tracks its dependencies and schedules its execution as soon as these are satisfied, taking care of gathering the data on the unit where the task is actually executed. In StarPU the execution is initiated by a *master thread*, running on a CPU, which is commonly in charge of submitting the tasks; the execution of the tasks, instead, is performed by *worker threads* (or, simply, *workers*) whose number and type (e.g., CPU or GPU) can be chosen by the programmer or by the user at run time. A CPU worker is bound to a CPU core whereas a GPU worker is bound to a GPU and a CPU core which is used to drive the work of the GPU. Note that nothing prevents worker threads from submitting tasks although this does not comply with the Sequential Task Flow model. Because StarPU has full control over a task and the associated data, these have to be declared to the runtime prior to the task submission. A task type can be declared through a *codelet* which specifies, among other things, the name of the task type, the units where it can be executed (e.g., CPU and/or GPU), the corresponding implementations (one for each type of unit) and the number of input data. Data, instead, is declared through a specific function call where the programmer informs StarPU about the location of the data (i.e., the data pointer) and about its properties such as the size, the rank, the leading dimension. Upon execution of this function call, StarPU returns a *handle*; once declared the data is not meant to be directly accessed by the programmer anymore but only through the handle and the associated methods. A task can roughly be defined as an instance of a task type coupled with a set of handles which represent the data used by the task itself.

We can more conveniently illustrate the functioning of StarPU using the simple sequential code in Figure 1.12 as an example. The purpose of the sequential example is to compute every elements of the array denoted `y`, using the input array `x`. The computation is done by applying the functions `f`, `g` as shown in the example. The right side of the figure

shows the dependencies between function calls in the various iterations of the for loop. Note that in this example we rely on a STF model for parallelization of the sequential code which is explained in Section 1.4.1.

```
1 for (i = 1; i < N; i++) {
    x[i] = f(x[i]);
3   y[i] = g(x[i], y[i-1]);
}
```

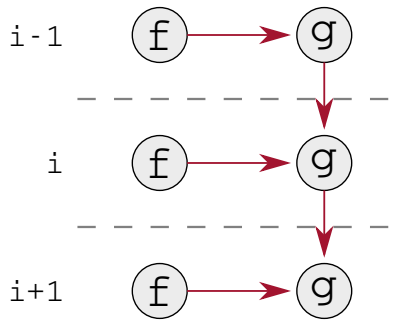


Figure 1.12: Simple example of a sequential code.

The parallel version of our simple example is proposed in Figure 1.13. In this code, executed by the master thread, we first declare the *codelet* structures. A codelet corresponds to the description of the code that is to be executed by a certain type of tasks which is defined by setting, at least, the following fields:

- `where` enumerates which computational units may execute the task type. The possibilities includes `STARPU_CPU` for CPU workers or `STARPU_CUDA` for GPU workers and can be composed when when a task type may be executed by several workers with the following notation `STARPU_CPU | STARPU_CUDA`.

- `cpu_funcs` and/or `cuda_funcs` are function pointers referring to the computational kernels (i.e., the actual implementations) to be executed by respectively CPU and CUDA workers.

- `nbuffers` indicates the numbers of data/handles manipulated by the tasks.

Assuming we want to execute `f` and `g` function calls in separate tasks, we need to declare two codelets, one for each type of task. Because a CPU and a GPU implementation exist for function `g`, the corresponding tasks of type `g_cl` can be executed on either type of units. Next, the data accessed by the tasks have to be registered; because we want each task to work on a single coefficient of the `x` and `y` arrays, we have to register each one of them to obtain the corresponding handle. This is done in the loop at line 23. Finally, the tasks are submitted in the loop at line 29 by specifying, for each task, the codelet and the corresponding data. The master thread finally reaches the barrier at line 43 where it sits waiting for all the tasks to be executed by the worker threads. When a task, say, of type `f_cl` is actually executed, the worker thread invokes the `f_cpu_func`; this retrieves the `x[i]` data using the dedicated method on the `x_handle[i]` handle and the calls function `f`.

As mentioned above, StarPU can transparently handle the data accessed by a task and move it where the task is actually executed. If, for example, for the iteration `i` of the loop in our example, function `f` is executed on the CPU and function `g` on the GPU, the data `x[i]` is automatically moved to the GPU memory. To avoid unnecessary data transfers, StarPU allows multiple copies of the same data to reside concurrently on several processing units and makes sure of their consistency. For instance, when a data is modified on a computational unit StarPU marks all the corresponding copies as invalid.

```
/* Codelet definition for kernel f */
struct starpu_codelet f_cl =
  {
    .where = STARPU_CPU,
    .cpu_funcs = { f_cpu_func },
    .nbuffers = 1
  };

/* Codelet definition for kernel g */
struct starpu_codelet g_cl =
  {
    .where = STARPU_CPU | STARPU_CUDA,
    .cpu_funcs = { g_cpu_func },
    .cuda_funcs = { g_cuda_func },
    .nbuffers = 3
  };

starpu_data_handle_t x_handle[N], y_handle[N];

starpu_init();

/* declaration of data handles */
for (i = 0; i < N; i++) {
  starpu_variable_data_register(&x_handle[i], STARPU_MAIN_RAM, (
   uintptr_t) &x[i], sizeof(double));
  starpu_variable_data_register(&y_handle[i], STARPU_MAIN_RAM, (
   uintptr_t) &y[i], sizeof(double));
 }

/* tasks submission */
for (i = 1; i < N; i++) {

  starpu_insert_task(&f_cl,
                     STARPU_RW, x_handle[i],
                     0);

  starpu_insert_task(&g_cl,
                     STARPU_R, x_handle[i],
                     STARPU_R, y_handle[i-1],
                     STARPU_W, y_handle[i],
                     0);

}

starpu_task_wait_for_all();

starpu_shutdown();
```

Figure 1.13: Simple example of a parallel version of the sequential code in Figure 1.12 using a STF model with StarPU.

Among the other features of the StarPU runtime system, in the work described in the following chapters, we will use the following:

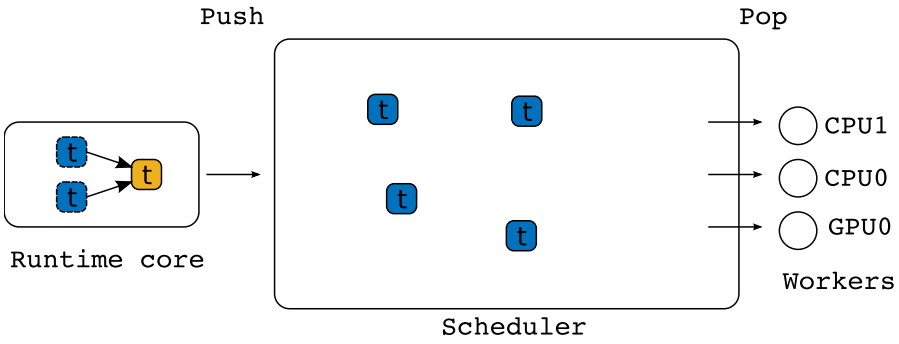


Figure 1.14: Two commonly used approaches to the multithreading of the multifrontal method.

- StarPU provides a framework for developing task scheduling policies in a portable way. The implementation of a scheduler consists in creating a task container and defining the code that will be triggered each time a new task gets ready to be executed (`push`) or each time a worker thread has to select the next task to be executed (`pop`) as illustrated in Figure 1.14. The implementation of each queue may follow various strategies (e.g. FIFO or LIFO) and sophisticated policies such as work-stealing may be implemented. StarPU comes with a number of predefined scheduling policies suited for different types of architectures and workloads. We will comment more on the development of schedulers in Sections 3.2.2, 4.4 and 6.2.
- If a task is assigned to a worker sufficiently ahead of its execution, the data it needs can be automatically prefetched. This allows for overlapping the data transfers with the execution of other tasks which results in better performance and scalability. We will describe the use of this feature in Section 6.2.
- In StarPU, it is possible to associate a *callback* function with a task. This is just a function which is executed upon termination of the task itself. We will use this feature in Section 3.2 to express some dependencies among tasks.
- StarPU has several facilities which allow for detailed performance profiling of an application. For example, it can generate execution traces containing accurate timings of all the executed tasks, of the runtime overhead, of data transfers etc. It can also automatically build performance models for the tasks. These can be used to implement complex scheduling policies (such as the HEFT-like policies described in Section 6.2) or to generate a linear program whose solution provides a lower bound for the performance achievable by an application. This feature as well as the execution traces, are used in our performance analysis approach presented in Chapter 2. StarPU can also dump the DAG of an application, which we use to compute the average degree of parallelism (or best achievable speedup) as described in Section 4.3.2.
- It is sometimes convenient to explicitly declare dependencies between tasks rather than letting the runtime system figure them out; StarPU provides methods to implement this.

1.4.2.2 The PaRSEC runtime system

PaRSEC is a runtime system based on a data-flow programming and execution model model which is developed at the ICL laboratory, University of Tennessee by Bosilca et al.

[25]. It provides a programming interface complying with the PTG model presented in Section 1.4.1 and is capable of handling applications, expressed as a DAG, on distributed, heterogeneous architectures.

In PaRSEC the DAG is represented with a compact format describing tasks and *data-flow*, i.e., how data flows from one task to another. During the execution, the DAG is dynamically unrolled upon events such as task completion. Thanks to the data-flow representation, communications are implicit and automatically handled by the runtime. The embedded scheduler is dynamic and designed to exploit the memory hierarchy of these architectures. In addition, because of the data-flow representation, the scheduler is able to maximize computation to communication overlap, exploit data locality and achieve load-balancing between the resources. As explained in Section 1.4.1, this runtime system, by exploiting the PTG model, is suited to target large scale systems.

```
1  N      [type = int]
2
3  task_f(i) /* Task name */
4
5  i = 1..N-1 /* Execution space declaration for parameter i */
6
7  : x(i) /* Task must be executed on the node where x(i) is stored */
8
9    /* Task reads x(i) from memory ... */
10   RW X <- x(i)
11   /* ... and sends it to task_g(i) */
12   -> X task_g(i)
13 BODY
14
15   X = f(X)    /* Code executed by the task */
16
17 END
18
19 task_g(i) /* Task name */
20
21 i = 1..N-1 /* Execution space declaration for parameter i */
22
23 : y(i) /* Task must be executed on the node where x(i) is stored */
24
25
26   /* Task reads x(i) fron task_f(i)... */
27   R X  <- X task_f(i)
28   /* ... y(i-1) from task_g(i-1)... */
29   R Y1 <- (i > 1) ? Y2 task_g(i-1) : y(i-1)
30   /* ... and sends y(i) to task_g(i+1) */
31   W Y2 -> (i < N-1) ? Y2 task_g(i+1)
32
33 BODY
34
35   Y2 = h(X, Y1) /* Code executed by the task */
36
37 END
```

Figure 1.15: Simple example of a parallel version of the sequential code in Figure 1.12 using a PTG model with PaRSEC.

PaRSEC provides a language called Job Data Flow (JDF) to express PTG parallel codes. During the compilation process, the files containing the JDF code are translated into C-code files by a specific compiler distributed with PaRSEC called *daguepp*. The DAG is defined by a set of *task types* that can be associated with several *parameters* defined on a given *range* of values. The tasks are associated with a list of *predecessors* and *successors* that define the dependencies in the DAG. These dependencies are generally based on data but may also represent precedence constraints. Tasks are associated with a code that will be executed for each task instances. A code specific to different types of resources, such as CPU or GPU, may be given.

We illustrate the features provided by PaRSEC using the same sequential example used in the previous section to introduce StarPU. A PTG parallel version of the sequential code in Figure 1.12 is shown in Figure 1.15, using the JDF language. For the sake of brevity we only show the PTG representation of our example and we do not present the declaration of data types or the instantiation of the DAG from the JDF.

In our JDF we define two different type of tasks, namely `task_f` and `task_g`, that are associated with the parameter `i`. This parameter is defined on the ranges `0..N-1` for `task_f` and `1..N-1` for `task_g`. The parameter `N` defined at the beginning of the JDF code, associated with a type, `int` in our example, and set when the DAG is instantiated. The data-flow associated with task `task_f` contains two edges. The first one reads data `x(i)` from memory in a variable referred to as `X`. The second edge send the modified data `X` to task `task_g(i)`. The data-flow associated with task `task_g` has three edges. The first incoming edge consists in retrieving the data `X` modified by task `task_f(i)`. The second incoming edges retrieved the data called `Y1` necessary to compute `Y2`. For this data we consider two cases. If `i > 1` then `Y1` is retrieved from the computed data `Y2` in task `task_g(i-1)` and otherwise it is read in memory. Finally `Y2` is send to tasks `task_g(i+1)` in the case where such task exists.

Note that the expression of the DAG presented above is independent from the problem size. The memory needed for its representation as well as the complexity for its analysis is constant when the problem grows. For this reason the memory consumption overhead in the runtime system for representing the DAG is much lower for the PTG model than for the STF model. In addition, with a STF model the DAG is entirely built and stored whereas with a PTG the DAG is only partially unfolded during the execution following the task progression. From this point of view, The advantage of the PTG approach over the STF one can be crucial in a distributed memory context because the DAG is pruned and only a portion of the DAG is represented on each node. This could considerably reduce the runtime overhead for the management of the DAG. On the other hand, knowing the entire DAG can be useful to compute the schedule of the DAG or give information to the dynamic scheduler by prepossessing the DAG.

1.5 Related work on dense linear algebra

The linear algebra computing community is spending a great deal of effort to tackle the new challenges raised by the drastic increase of the computational resources due to the relatively recent introduction of multicore processors. One commonly employed approach consists in reducing the granularity of computations and avoiding “fork-join” parallelism, as the scalability of this scheme suffers from an excessive amount of synchronizations. Most of the related work focuses on intra-node parallelization with a shared memory parallel programming paradigm. To be more precise, thread based parallelization is widely used to tackle the performance issues within a computing node. These concepts have been first

introduced in the field of dense linear algebra computations [33] where the main idea was to replace the commonly used data layout for dense matrices with one based on tiles/blocks and to write novel algorithms suited to this new data organization; by defining a task as the execution of an elementary operation on a tile and by expressing data dependencies among these tasks in a Directed Acyclic Graph (DAG), the number of synchronizations is drastically reduced in comparison with classical approaches (such as the LAPACK or ScaLAPACK libraries) thanks to a dynamic data-flow parallel execution model. This idea led to the design of new algorithms for various algebra operations [34, 94] now at the base of well known software packages like PLASMA [9] and FLAME [113].

Interestingly, to extract potential parallelism better, these software packages are already often designed with, to some extent, the concept of *task* before having in mind the goal of being executed on top of a runtime system. This possibility was considered when these software packages need to target modern heterogeneous systems. A lot of attention has recently been paid to the design of new algorithms able to fully exploit the huge potential of accelerators (mostly GPUs). The main challenges raised by these heterogeneous platforms are mostly related to task granularity and data management: although regular cores require fine granularity of data as well as computations, accelerators such as GPUs need coarse-grain tasks. This inevitably introduces the need for identifying the parts of the algorithm which are more suitable to be processed by accelerators. As for the multicore case described in the previous section, the exploitation of this kind of platform was first considered in the context of dense linear algebra algorithms. This has been accomplished in four main steps by

1. Designing and implementing efficient kernels [89, 90, 52, 114];
2. Designing algorithms for heterogeneous mono-accelerator platforms with an offloading approach [109];
3. Exploiting both CPUs and the accelerator in the context of mono-accelerator platforms [110];
4. Extending the designed approaches to the multi-accelerator case [93, 85, 10, 116].

While the first three steps do not require complex scheduling techniques, the latter relies on a dynamic scheduling approach to achieve the flexibility required by these heterogeneous platforms [3, 4]. Specifically, runtime systems are used to manage tasks dynamically, these runtime systems being either generic like StarPU [18] or PaRSEC [26, 25], or specific like QUARK [79] , or the one developed for the TBLAS [106] library. As a result, higher performance portability is also achieved thanks to the hardware abstraction layer introduced by runtime systems [5]. These efforts resulted in the design of the MAGMA library [9] on top of StarPU, the DPLASMA library [27] on top of PaRSEC and the adaptation of the existing FLAME library [73] to heterogeneous multicore systems using the SuperMatrix [37] runtime system.

1.6 Related work on sparse direct solvers

The literature on parallel, sparse direct methods is very abundant and many solver are currently available with different features and efficiency. Many of them are also based on task parallelism, to some extent, although very few (apart from PaStiX, described below, no other solver fully relies on a runtime system) rely on a general purpose runtime system for implementing parallelism. Some target multicore architectures, others are designed for

using GPUs and other for distributed memory systems; some can handle all these types of devices in a single package. A complete taxonomy of all the known solvers and methods is out of the scope of this document but we will make, in the rest of this section, a list of works that are more closely related to the subject of this thesis.

One of the first multifrontal QR solvers is the MA49 developed by Amestoy et al. [14] and still distributed in the HSL Mathematical Software Library. This solver was designed for shared memory multiprocessors and employed a rather simple approach to the use of tree and node parallelism where each type is exploited separately with a different technique. In this solver, tree parallelism is achieved trough a hand coded task queueing system, where a task is defined as the assembly and the factorization of a front; a task associated with a front is pushed in the task queue as soon as all the tasks associated with child fronts are completed. Node parallelism, instead, is entirely delegated to a multithreaded BLAS library.

More recently, in 2011, Tim Davis released a new multifrontal QR solver named SuiteSparseQR [42] (SPQR) which is also specifically designed for shared memory, multicore systems. In essence, SPQR uses the same approach as MA49 for implementing parallelism but relies on a runtime system, namely Intel Threading Building Blocks [95] for handling the tasks: tasks are submitted to the TBB runtime in a recursive fashion and barriers are used to ensure that a front is processed once the tasks associated to its children are done. As for node parallelism, SPQR relies on the use of multithreaded BLAS libraries.

Even more recently, Davis and his team developed a variant of the SPQR solver, called SPQR-GPU, that runs on a GPU; this is a GPU-only solver in the sense that the CPU is only used to drive the work of the GPU where all the computation takes place. SPQR-GPU employs 2D communication avoiding algorithms for the factorization of frontal matrices and uses a task-based approach for implementing parallelism. The elimination tree is split into pieces called "stages" such that each stage fits into the GPU memory (for smaller problems the whole elimination tree can be processed in a single stage). The handling of tasks is done trough the use of two nested schedulers which are hand-coded. The outer scheduler, called the "Sparse QR scheduler" handles the stages and as soon as a stage is ready for being processed (i.e., when its child stages are finished) spawns an instance of the inner scheduler, called the "Bucket scheduler", which is in charge of generating the numerical tasks related to the fronts in the stage and of submitting them to the GPU device for execution. The tracking of dependencies in both schedulers is done manually.

The UHM solver (which uses the LDL^T factorization) by Kim et al. [76] uses a similar approach to SPQR and SPQR-GPU but targets shared memory multicore machined (without GPU). It also employs a nested tasking mechanism. A recursive submission of OpenMP tasks (one task per front) is used to handle tree parallelism exactly in the same way as SPQR uses Intel TBB. When one such task is executed, it spawns an instance of an inner, hand-coded scheduler which generates OpenMP tasks for the operations related to the associated front. The tracking of the dependencies between these tasks is done in a rather complicated way where multiple tasks may be generated for the same operation although only one of them actually executes the operation. Another variant of the UHM solver can handle GPU devices [77]: tree parallelism is handled as in the base variant and node parallelism is achieved through a bulk synchronous execution of task lists with a stating mapping of tasks to processing units (CPUs or GPUs) based on a performance model.

In 2014 Hogg et al. [67] released SSIDS, a GPU-only multifrontal solver for sparse, symmetric indefinite systems. This solver shares some commonalities with SPQR-GPU: the factorization proceeds in successive steps where, at each step, a list of independent tasks is generated and submitted to the GPU for execution.

The same authors of SSID also developed the MA86 [66] and MA87 [68] solvers that target, respectively, the solution of symmetric indefinite and symmetric positive definite sparse linear systems on multicore machines. Both these solvers are based on a left-looking, supernodal factorization where supernodes are decomposed into tiles and use a task-based parallelization approach; the scheduling of tasks is achieved with a hand-coded task queueing system. The tracking of task dependencies is done associating a counter to each tile which describes its state and, consequently, the operations that can be executed on it; as soon as it is possible to execute an operation on a tile, the corresponding task is pushed in the task pool.

Among the other efforts to port sparse, direct methods on GPUs, we can cite the work by George et al. [56], Lucas et al. [86], and Yu et al. [119]. These approaches mainly target the multifrontal method for LU or Cholesly factorizations due to its very good data locality properties. The main idea is to treat some parts of the computations (mostly, trailing submatrix updates) entirely on the GPU. Therefore the main originality of these efforts is in the methods and algorithms used to decide whether or not a task can be processed on a GPU. In most cases this was achieved through a threshold based criterion on the size of the computational tasks. More complex approaches can be found in the work by [120, 99, 96]. These improve over previous efforts mostly by proposing techniques for aggregating fine grain operations to form large grain tasks which maximise the GPU occupancy (either by grouping basic BLAS operation or by treating a complete subtree as a single task) and pipelining to overlap communications with computations. In more recent work, Sao et al. [98] extend the SuperLU_Dist package to support Xeon Phi architectures using analogous techniques as in their previous effort [99].

The work which is most closely related to this thesis, is described in the PhD thesis of Xavier Lacoste [80]. Lacoste implemented two variants of the PaStiX [64] solver based, respectively, on STF and PTG task parallelism each relying on a different runtime system, namely StarPU and PaRSEC, respectively. Both variants implement a 1D, left-looking supernodal factorization and are capable of using multicore systems equipped with multiple GPUs. The variant based on StarPU can also run on distributed memory systems. These implementations do not take full advantage of the features of runtime systems. In the StarPU variant, task dependencies are declared explicitly through the use of tags, rather than inferred by the runtime through data analysis. In both variants the scheduling of tasks to GPU devices is static and relies on a performance model; moreover, no eviction policy is implemented: once the supernodes that are mapped on the GPU are processed, they are not brought back to the host device in order to free space for new ones. This means that the amount of computation that can be done on the GPU is limited by the size of the memory available on the device. Ultimately, Lacoste concluded that, in a shared memory context and up to a certain number of cores, it is possible to use runtime system to develop sparse direct solvers whose performance is on par with (or slightly lesser than) that of a finely tuned, hand coded package. Nonetheless he showed how runtime systems can ease the porting of such solvers on GPU equipped architectures thanks to the automatic dependency tracking and the transparent data handling capability.

1.7 The `qr_mumps` solver

1.7.1 Fine-grained parallelism in `qr_mumps`

As explained in the previous section, Other existing multithreaded, multifrontal QR solvers, such as MA49 or SPQR only explicitly handle tree parallelism, but rely on third party, multithreaded BLAS operations for exploiting node parallelism. This is illustrated

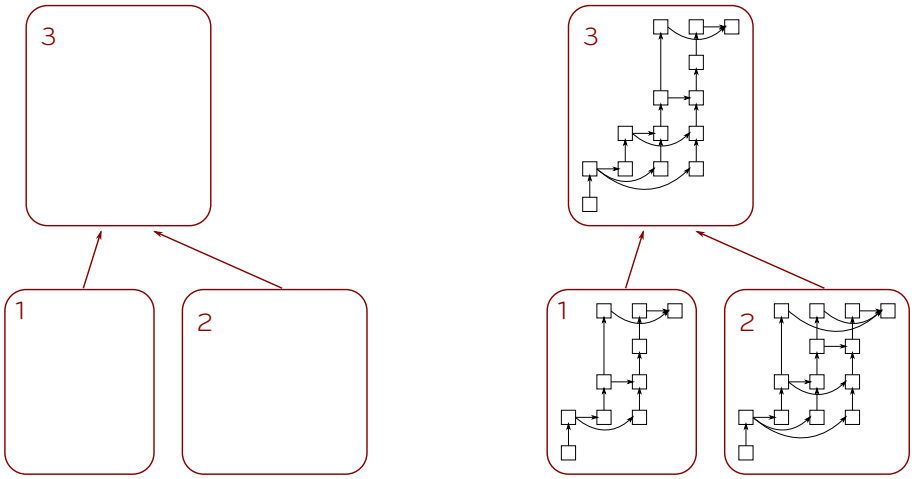


Figure 1.16: Two commonly used approaches to the multithreading of the multifrontal method.

in Figure 1.16 (*left*) Although this approach works reasonably well for a limited number of threads, its scalability suffers because of two main issues:

1. Because the node parallelism is delegated to an external multithreaded BLAS library, the number of threads dedicated to node parallelism and to tree parallelism has to be fixed before the execution of the factorization. Thus, a thread configuration that may be optimal for the bottom part of the tree will result in a poor parallelization of the top part and vice versa. If the BLAS library allows for, it could be possible to dynamically change the number of threads in each call to a BLAS routine but this is poorly portable and extremely difficult from an algorithmic point of view as it requires a very accurate performance model and sophisticated scheduling policies; to our knowledge, there is no existing solver that achieves this.

2. It is not possible to start working on a front until all of its children are completely factorized; this results from the fact that each front is entirely processed in a single task and may considerably limit the scalability of the code (we will provide experimental proof of this fact in Sections 4.2.3 and 4.3.2).

Other multifrontal solvers, such as UHM overcome the first of these two shortcomings by explicitly parallelizing the factorization of each front through a DAG based approach. This approach is depicted in Figure 1.16 (*right*). In this case each working thread can execute any factorization task no matter which front this comes from which ensures a good load balancing and does not require any performance modeling. Nonetheless, in this solver it is still not possible to start working on a front until all of of its children are completed.

The `qr_mumps` solver [32] pushes this approach even further and overcomes both the previously mentioned limitations by using a fine-grain partitioning and data flow model of execution which allows for handling both tree and node parallelism in a consistent way. The approach used in this solver uses a 1D partitioning of fronts into block-columns of size `nb` (a tunable parameter) as shown in Figure 1.17 (left) and defines five elementary kernels applied on block-columns or frontal matrices:

1. `activate`: this routine computes the structure of the front[3] (which essentially depends on the structure of the child fronts and the rows of the original matrix A associated with the front itself), allocates and initializes the front data structure and assembles the coefficients in the rows of the original matrix A associated with the front;

2. `assemble`: for a block-column in the child node, assembles the corresponding part of the contribution block into the parent node (if it exists). Note that, in practice, only a subset of the block-columns of `f`; moreover, because the assembly of a front simply consists in copying coefficients into distinct memory locations, the `assemble` operations are all independent;

3. `_geqrt`: computes the QR factorization of a block-column. This is the panel factorization in the LAPACK dense QR factorization; Figure 1.17 (middle) shows the data modified when the panel operation is executed on the first block-column;

4. `_gemqrt`: applies to a block-column the Householder reflectors computed in a previous `_geqrt` operation. This is the *update* operations in the LAPACK dense QR factorization; Figure 1.17 (right) shows the coefficients read and modified when the third block-column is update'd with respect to the first panel;

5. `deactivate`: stores the coefficients of the R and H factors aside and frees the memory containing the contribution block;

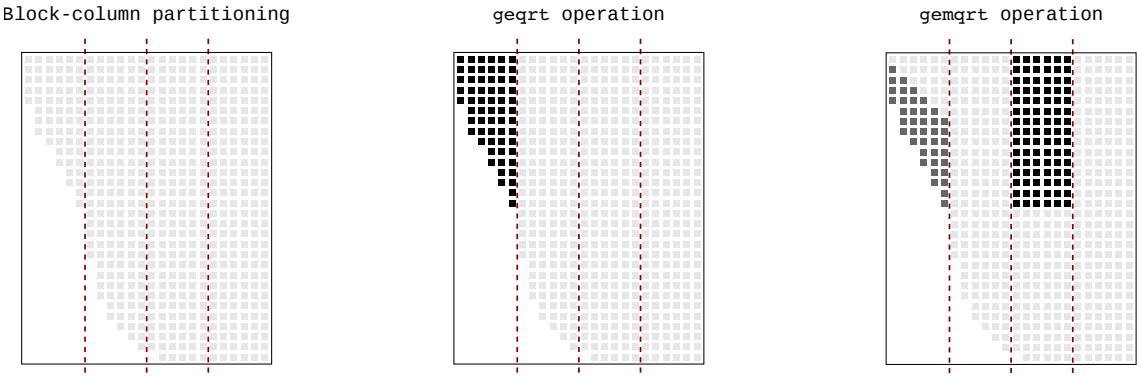


Figure 1.17: Block-column partitioning of a frontal matrix (*left*) and geqrt and gemqrt operations pattern (*middle* and *right*, respectively); dark gray coefficients represent data read by an operation while black coefficients represent written data.

The multifrontal factorization of a sparse matrix can thus be defined as a sequence of tasks, each task corresponding to the execution of an elementary operation of the type described above on a block-column or a front. The tasks are arranged in a Directed Acyclic Graph (DAG) such that the edges of the DAG define the dependencies among tasks and thus the relative order in which they have to be executed. In this approach assembly operations are parallelized and the status of each block-column is tracked individually; consequently, it is possible to start working on a block-column as soon as it is fully assembled regardless of the other block-columns in the same front. As a result, the factorization

[3]Note that this data could be computed at the analysis phase and stored but this would result in excessive memory consumption.

of a front can be pipelined with the processing of its children; we will refer to this extra source of concurrency as *inter-level parallelism* and we will provide more details on its advantages as well as a detailed analysis in the Sections 4.2.3 and 4.3.2. It must be noted that some supernodal solvers, such as HSL_MA87 by Hogg et al. [68] or PaStiX by Hénon et al. [64] can also exploit inter-level parallelism. Also, the multifrontal MUMPS solver by Amestoy et al. [13], can partially use this source of concurrency in the more challenging context of distributed memory parallelism: in MUMPS fronts are distributed among multiple MPI processes and each process can start working on its share of a front as soon as it is fully assembled regardless of the rest of the front.

Figure 1.18 shows the DAG associated with the subtree defined by supernodes one, two and three for the problem in Figure 1.7 for the case where the block-columns have size one[4]; the dashed boxes surround all the tasks that are related to a single front.

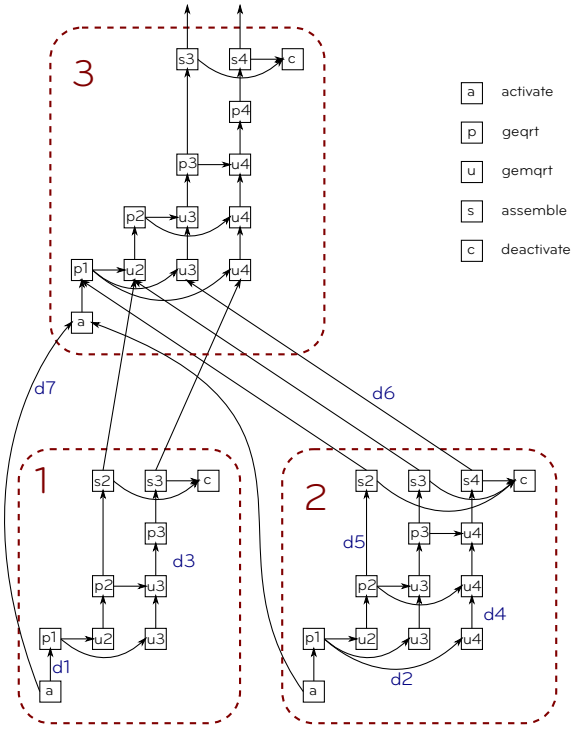


Figure 1.18: DAG associated with supernodes 1,2 and 3 in Figure 1.7. For the panel, update and assemble operations, the index of the block column is specified. For this example, the block-column size is chosen to be one.

The dependencies on the DAG represented in Figure 1.18 are defined as follows:

- `d1`: no other elementary operation can be executed on a front or on one of its block-columns until the front is not activated;
- `d2`: a block column can be updated with respect to a `_geqrt` operation only if the corresponding panel factorization is completed;
- `d3`: the `_geqrt` operation can be executed on block-column i only if it is up-to-date with respect to `_geqrt` $i-1$;

[4]Figure 1.18 actually shows the transitive reduction of the DAG, i.e., the direct dependency between two nodes is not shown in the case where it can be represented implicitly by a path of length greater than one connecting them.

- `d4`: a block-column can be updated with respect to a `_geqrt` i in its front only if it is up-to-date with respect to the previous panel $i-1$ in the same front;
- `d5`: a block-column can be assembled into the parent (if it exists) when it is up-to-date with respect to the last `_geqrt` factorization to be performed on the front it belongs to (in this case it is assumed that block-column i is up-to-date with respect to `_geqrt` i when the corresponding `_geqrt` operation is executed);
- `d6`: no other elementary operation can be executed on a block-column until all the corresponding portions of the contribution blocks from the child nodes have been assembled into it, in which case the block-column is said to be *assembled*;
- `d7`: since the structure of a frontal matrix depends on the structure of its children, a front can be activated only if all of its children are already active;

This DAG globally retains the structure of the assembly tree but expresses a higher degree of concurrency because tasks are defined on a block-column basis instead of a front basis. As explained, it implicitly represents both tree and node parallelism which allows the exploitation of both of them in a consistent way. Finally, it removes unnecessary dependencies making it possible, for example, to start working on the assembled block-columns of a front even if the rest of the front is not yet assembled and, most importantly, even if the children of the front have not yet been completely factorized.

The execution of the tasks in the DAG is driven by a data-flow model meaning that the tasks become ready for execution as soon as their input data are available. More details on how this is actually implemented will be provided in Section 3.1.

1.7.2 Tree pruning

Because the target of this work is a system with only a limited number of cores, the number of nodes in the elimination tree is commonly much larger than the number of working threads. It is, thus, unnecessary to partition every front of the tree and factorize it using a parallel algorithm. For this reason we employ a technique similar to that proposed by Geist and Ng [54]. As shown in Figure 1.19, we identify a layer in the elimination tree such that each subtree rooted at this layer is treated in a single task, which we called `do_subtree`, with a purely sequential code.

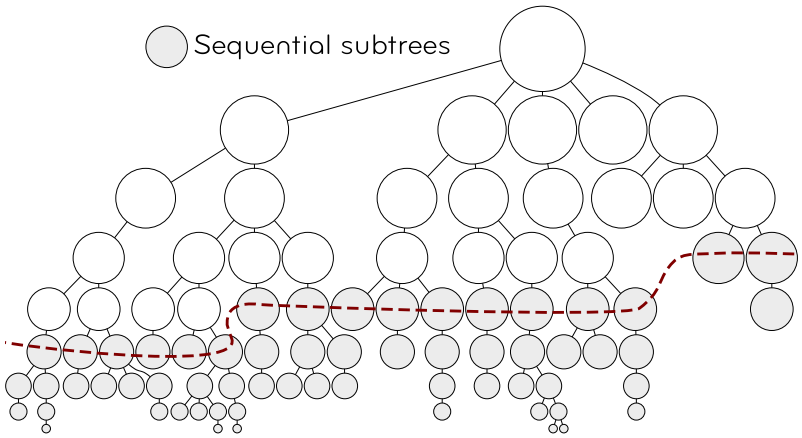


Figure 1.19: A graphical representation of how the logical amalgamation and logical pruning may be applied to an assembly tree.

This layer has to be as high as possible in order to reduce the number of potentially active nodes but low enough to provide a sufficient amount of tree-level parallelism on the top part of the tree.

This technique has a twofold advantage. First it reduces the number of active fronts during the factorization and, therefore, the cost of the scheduler as pointed out in Section 3.1. Second, it improves the efficiency of operations on those parts of the elimination tree that are mostly populated with small size fronts and, thus, less performance effective.

1.7.3 Blocking of dense matrix operations

The standard _geqrt and _gemqrt routines from LAPACK could be used for the corresponding tasks described above. This, however, would imply a (considerable) amount of extra-flops because these routines cannot benefit from the fact that, due to the fronts staircase structure, most block-columns are largely populated with zero coefficients. The above-mentioned LAPACK routines use an internal blocking of size ib . We modified these routines in such a way that the internal blocking is also used to reduce the flop count by skipping most of the operations related to the zeros in the bottom-left part of the tiles lying on the staircase.

It is obviously desirable to use blocked operations that rely on Level-3 BLAS routines in order to achieve a better use of the memory hierarchy and, thus, better performance. The use of blocked operations, however, introduces additional fill-in in the Householder vectors due to the fact that the staircase structure of the frontal matrices cannot be fully exploited. Therefore, even if the standard _geqrt and _gemqrt routines from LAPACK could be used for the corresponding tasks described above, this would imply a (considerable) amount of extra-flops. It can be safely said that it is always worth paying the extra cost of this additional fill-in because the overall performance will be drastically improved by the high efficiency of Level-3 BLAS routines; nonetheless it is important to keep this overhead under control and reduce it as much as possible. For this reason, qr_mumps uses slightly modified variants of the _geqrt and _gemrt routines where the size of the internal blocking ib can be explicitly set by the user and is then used to reduce the flop count by skipping most of the operations related to the zeros in the bottom-left part of the block-columns lying on the staircase.

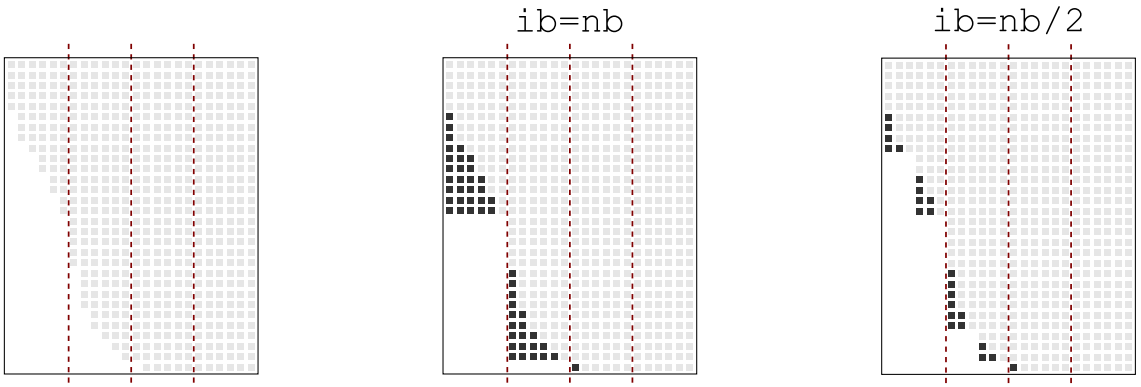


Figure 1.20: The effect of internal blocking on the generated fill-in. The light gray dots show the frontal matrix structure if no blocking of operations is applied whereas the dark gray dots show the additional fill-in introduced by blocked operations.

Figure 1.20 shows as dark gray dots the extra fill-in introduced by the blocking of operations `ib` with respect to the partitioning size `nb` on an example frontal matrix. As a result, these two parameters can be used, respectively, to control the granularity (and thus the efficiency) of BLAS operations and the granularity of tasks (and thus the amount of parallelism).

1.7.4 The `qr_mumps` scheduler

The scheduling and execution of tasks in `qr_mumps` is based on the use of a pool of tasks containing, throughout the execution of the factorization, a list of ready tasks, i.e., those tasks whose dependencies are all satisfied. This pool of tasks is handled through two other routines that are in charge of filling up the pool with tasks whenever they become ready for execution and picking up tasks for threads to execute them whenever they become idle; these are the `fill_queues` and `pick_task` routines, respectively.

The pseudocode in Figure 1.21 illustrates the main loop executed by all threads; at each iteration of this loop a thread:

1. checks whether the number of tasks globally available for execution has fallen below a certain value (which depends, e.g., on the number of threads) and, if it is the case, it calls the `fill_queues` routine, described below, which searches for ready tasks and pushes them into the tasks pool.

2. picks a task. This operation consists in popping a task from the tasks pool.

3. executes the selected task if the `pick_task` routine has succeeded.

Most of the complexity of this tasks scheduling mechanism is hidden in the `fill_queues` which is in charge of finding tasks that are ready for execution and pushing them into the pool of tasks. The pseudocode for this routine is shown in Figure 1.22. At every moment, during the factorization there exists a list of active fronts; the `fill_queues` routine goes through this list looking for ready tasks on each front. Whenever one such task is found, it is pushed in the pool of tasks. If no task is found related to any of the active fronts, a new ready front (if any) is scheduled for activation; the search for a front that can be activated follows a postorder traversal of the assembly tree, which provides a good memory consumption and temporal locality of data. Simultaneous access to the same front and the pool of tasks in the `fill_queues` and `pick_task` routines is prevented through the use of locks. As explained in Chapter 5, an efficient use of tree-level parallelism makes it hard, if not impossible, to follow a postorder traversal of the tree which results in increased memory consumption with respect to the sequential case. It is important to note that the proposed scheduling method tries to exploit node-level parallelism as much as possible and dynamically resorts to tree-level parallelism by activating a new node only when no more tasks are found on already active fronts. This keeps the tree traversal as close as possible to the one followed in the sequential execution and avoids the memory consumption to grow out of control.

This whole mechanism is implemented using an extremely limited subset of the OpenMP features, i.e., the `parallel` construct, the locks and the mutexes (`critical` sections).

1.7.4.1 Scheduling policy

In order to improve the efficiency of the code, `qr_mumps` implements a scheduling policy that aims at reducing the movement of data between the nodes of a NUMA systems and

```
mainloop: do
   if(n_ready_tasks < ntmin) then
      ! if the number of threads falls
      ! below a certain value, fill-up
      ! the queues
      call fill_queues()
   end if

   task = pick_task()

   select case(task%id)
   case(geqrt)
      call execute_geqrt()
   case(gemqrt)
      call execute_gemqrt()
   case(assemble)
      call execute_assemble()
   case(activate)
      call execute_activate()
   case(deactivate)
      call execute_deactivate()
   case(finish)
      exit mainloop
   end select
end do mainloop
```

Figure 1.21: main execution loop in qr_mumps, executed by every thread.

```
found = .false.
forall (front in active fronts)
   ! for each active front try to schedule
   ! ready tasks
   found = found .or. push_geqrt(front)
   found = found .or. push_gemqrt(front)
   found = found .or. push_assemble(front)
   found = found .or. push_deactivate(front)
end forall

if (found) then
   ! if tasks were pushed in the previous
   ! loop return
   return
else
   ! otherwise schedule the activation of
   ! the next ready front
   call push_activate(next ready front)
end if

if (factorization over) call push_finish()
```

Figure 1.22: fill_queue routine in qr_mumps, filling thread queues with tasks ready for execution

at reducing the time for traversing the DAG critical path which defines a lower bound on the execution time of the task graph.

In the multifrontal method, Level-3 BLAS routines such as `_gemqrt` operations represent the largest share of computational cost. However there remain a significant amount of Level-2 BLAS routines as well as symbolic operations such as `activate`, `assemble` and `_geqrt` tasks whose efficiency is limited by the speed of memory. In particular many tasks on the critical path are memory-bound and the impact of the time spent on this path becomes more and more important as the number of resources grows. In addition, some frontal matrices, especially at the bottom of the tree, are too small to achieve asymptotic performance on Level-3 routines. Therefore to achieve a good scalability of the multifrontal factorization, it is important to execute these memory-bound kernels as efficiently as possible. This can be done by executing the kernels on the threads which are the closest to the data they manipulate. For this purpose `qr_mumps` introduces a concept of ownership of a front: the thread that performs the activate operation on a front becomes its owner and, therefore, becomes the privileged thread to perform all the subsequent tasks related to that front. This leads to a better performance of tasks because of the first-touch rule which is implemented in most (if not all) modern operating systems. The task pool is then implemented as a set of queues, one for each thread. In the `fill_queue` routine, when a ready task is found, it is pushed in the queue associated with the thread that owns the related front. An `activate` task, instead, is pushed on the queue associated with the thread that executes the `fill_queues` routine because the ownership of the related front is not set yet. When a thread executes the `pick_task` routine, it first looks for a ready task in its own queue. In the case where no task is available on the local queue, an architecture aware work-stealing technique is employed, i.e., the thread will try to steal a task from queues associated with threads with which it shares some level of memory (caches or DRAM module on a NUMA machine) and if still no task is found it will attempt to steal a task from any other queue. The computer's architecture can be detected using tools such as hwloc [30]. Experimental results presented by Buttari [32] show that this technique provides some mild performance improvements.

In order to avoid any delay on the critical path that may increase the makespan and induce resource starvation, tasks along this path are set with higher priorities over other tasks. In the case of a dense QR factorization with a 1D block-column partitioning it is easy to see that panel operations lie on the critical path of the factorization task graph. In `qr_mumps` only two levels of priority can be used: although tasks are always popped from the head of each queue, they can be pushed either on the head or on the tail which allows the prioritisation of certain tasks. The panel operations for example are always pushed on the head to prioritise them over other tasks.

1.8 Positioning of the thesis

Direct methods, introduced in Section 1.2, constitute a popular approach to find the solution of large sparse linear systems of equations. They are often preferred for their robustness over other approaches like iterative methods whose efficiency largely depends on the numerical properties of the input problem. As presented in Section 1.3, in the cases of dense and sparse systems, recently developed factorization algorithms generate a high amount of concurrency and allow the exploitation of efficient kernels, yielding good performance on modern multicore architectures. The evolution of the hardware, however, with the increase in the number of cores per chips, the introduction of accelerators and the diminishing amount of memory per core, face researchers with new challenges that we

tackle in this study: how to implement complex and irregular algorithms in an efficient yet portable way? how to make algorithms evolve and improve without being limited or constrained by the complexity of their actual implementation? how to achieve the execution of complex workloads on heterogeneous architectures equipped with multiple execution units running at different speeds and with memories having different capacities and speeds? How to profile and analyze the performance of such codes on hybrid architectures?

This thesis attempts to address these issues through an approach which consists in relying on the use of modern runtime systems in order to achieve a performance and memory efficient yet portable implementation of the QR multifrontal method for heterogeneous architectures. Modern runtime systems, presented in Section 1.4.2, provide programming interfaces complying with DAG-based algorithms that have recently become more and more popular in the domain of linear algebra as explained for both the dense and sparse case in Sections 1.5 and 1.6, respectively. The use of runtime systems has been largely studied in the context of dense linear algebra but still represents a challenge for sparse algorithms such as the multifrontal method. The difficulty associated with the development of sparse methods on top of runtime systems lies in complexity of the DAG representing the application with a large amount of tasks, with a great variety of kernels, granularity and memory consumption.

The first issue we address in Chapter 3 is to validate the pertinence of this approach by porting `qr_mumps` to StarPU in a new version referred to as `qrm_starpu`. We show that we achieve good performance with `qrm_starpu` and provide a detailed performance analysis comparing our new version with the original solver. In Chapter 4 we redesign the previous implementation using a pure Sequential Task Flow model and improve our solver with the integration of 2D, communication avoiding front factorization algorithms. Furthermore we develop a memory-aware algorithm allowing for controlling the memory consumption of the parallel multifrontal method. As we mention in Section 1.6, few sparse solvers are based on runtime systems and none of them fully relies on these tools for handling parallelism and tasks execution, scheduling and data management. In our approach, we separate the expression of algorithms from low-level details such as dependency management, data transfer and data consistency that are delegated to the runtime system. In addition we take advantage of the expressiveness of the programming models that we use to develop new features.

Exploiting heterogeneous systems is extremely challenging due to the complexity of these architectures. Current approaches presented in Section 1.6 generally rely on simple static scheduling strategies and some of the state-of-the-art solvers only exploit the accelerator without taking advantage of the other resources on the architectures. In Chapter 6 we address the data partitioning and scheduling issues that are critical to achieve performance on these architectures. We extend the 1D block-column partitioning used in the `qr_mumps` solver (see Section 1.7) to a hierarchical block partitioning allowing to generate both fine and coarse granularity tasks adapted to the processing unit capabilities. We show that the simplicity of the STF model facilitates the implementation of this data partitioning with a complex dependency pattern and gives the ability to dynamically partition data. We develop a scheduling strategy capable of handling the task heterogeneity in the DAG and the diversity of resources on heterogeneous architectures. Thanks to the modular approach that we employ, the scheduler implemented in `qrm_starpu` is generic and totally usable in other StarPU based applications.

The efficiency of the approaches presented above are assessed with a detailed performance analysis presented in Chapter 2. This performance analysis approach allows for measuring and separately analysing several factors playing a role in performance and scalability such as locality issues and task pipelining. It should be noted this analysis requires

the ability to measure times spend in several part of the application. This information may be easily obtained via the runtime system. Nonetheless, the use of this method is not restricted to applications based on task parallelism but can be readily applied to any type of parallel code.

Finally, in Chapter 7 we address several issues related to the studies presented in the previous chapters. First we propose a PTG-based version of our solver implemented with PaRSEC. Our work clearly benefits from the features, the robustness and the efficiency of runtime systems but, at the same time, provides a very valuable feedback to the runtime developers community. In Sections 7.2 and 7.3 we discuss how `qrm_starpu` has been used to tune and validate a simulation engine for StarPU-based application and the use of scheduling contexts for improving the locality of reference to data.

1.9 Experimental settings

1.9.1 Machines

The following computers have been used for the experiments presented in the rest of this document:

- **Dude**: this is a shared-memory machine equipped with four AMD Opteron(tm) Processor 8431 (six cores) and 72 GB of memory. The cores are clocked an 2.4 GHz and have a peak performance of 9.6 Gflop/s each and thus a global peak performance of 230.4 Gflop/s for real, double precision computations. Figure 1.23 shows the architecture of one of the four sockets of this system. The compilers available on this system are the Intel `ifort` and `icc` 13.1.0 and the BLAS and LAPACK libraries are from Intel MKL 11.0.

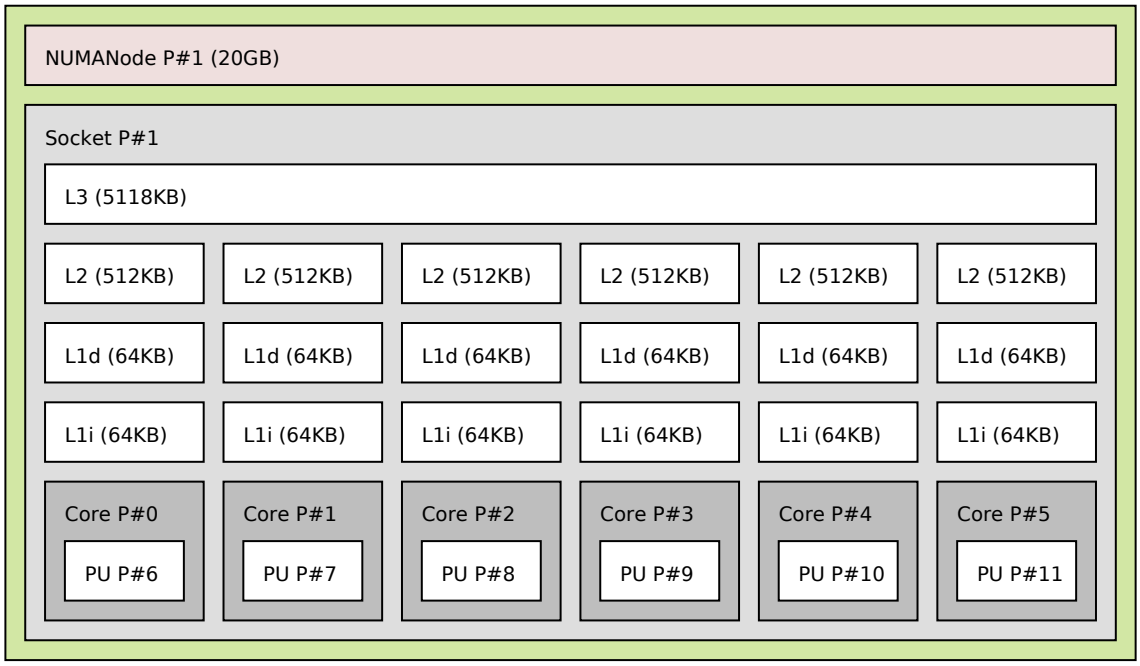


Figure 1.23: The architecture of one of the fours sockets on the Dude computer.

- **Ada**: supercomputer installed at the IDRIS French supercomputing center[5]. This is an IBM x3750-M4 system equipped with four Intel Sandy Bridge E5-4650 (eight cores) processors and 128 GB of memory per node. The cores are clocked at 2.7 GHz and are equipped with Intel AVX SIMD units; the peak performance is of 21.6 Gflop/s per core and thus 691.2 Gflop/s per node for real, double precision computations. Figure 1.24 shows the architecture of one of the four sockets of this system. The compilers available on this system are the Intel `ifort` and `icc` 15.0.2 and the BLAS and LAPACK libraries are from Intel MKL 11.2.

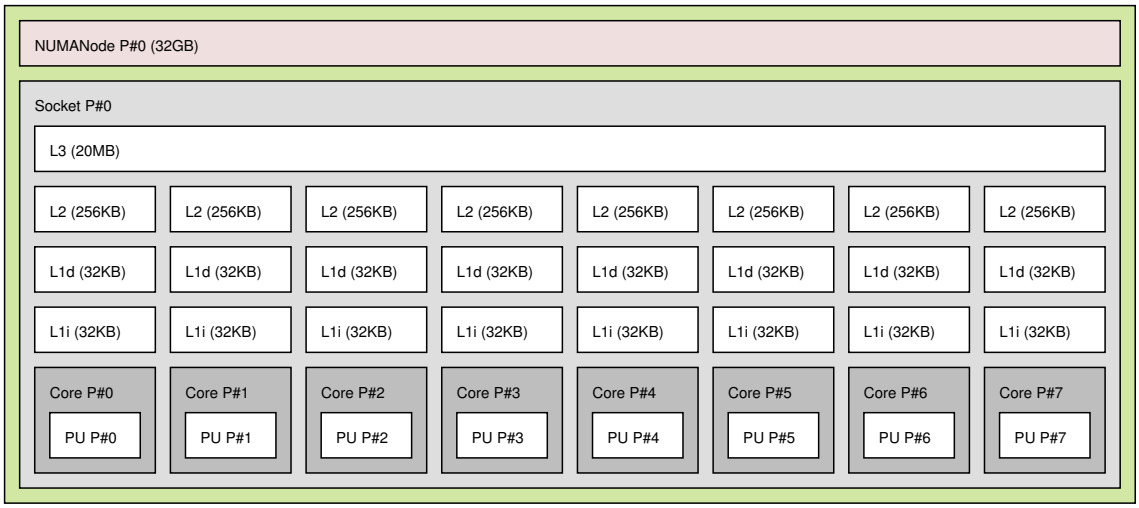


Figure 1.24: The architecture of one of the fours sockets on the Ada computer.

- **Sirocco**: five nodes cluster part of the PlaFRIM center [6]. Each nodes is equipped with two Haswell Intel Xeon E5-2680 (twelve cores) processors and 124 GB of memory per node. The cores are clocked at 2.5 GHz and are equipped with Intel AVX SIMD units. In addition, each node is accelerated with four Nvidia K40M GPUs; the peak performance is of 40.0 Gflop/s per core, 1.4 Tflop/s per GPU and thus 6.0 Tflop/s per node for real, double precision computations. Figure 1.25 shows the architecture of one of the two sockets of this system. The compiler available on this system are GNU `gcc` and `gfortran` 4.8.4 and the BLAS and LAPACK libraries are from Intel MKL 11.2.

1.9.2 Problems

The approaches and implementations presented in the following sections where tested and evaluated on a number of matrices from real life applications publicly available in the University of Florida Sparse Matrix Collection[43] plus one, the hirlam matrix, from the HIRLAM[7] research program. Table 1.1 shows the main characteristics of these matrices; the number of floating-point operations, which refers to the matrix factorization, is computed with an internal block size `ib` of 32 and for a fill-reducing ordering computed with the tool specified in column #3. Note that the same matrix is assigned a different id depending on the applied column permutation. The SCOTCH version used was 6.0.

[5] http://www.idris.fr
[6] https://plafrim.bordeaux.inria.fr
[7] http://hirlam.org

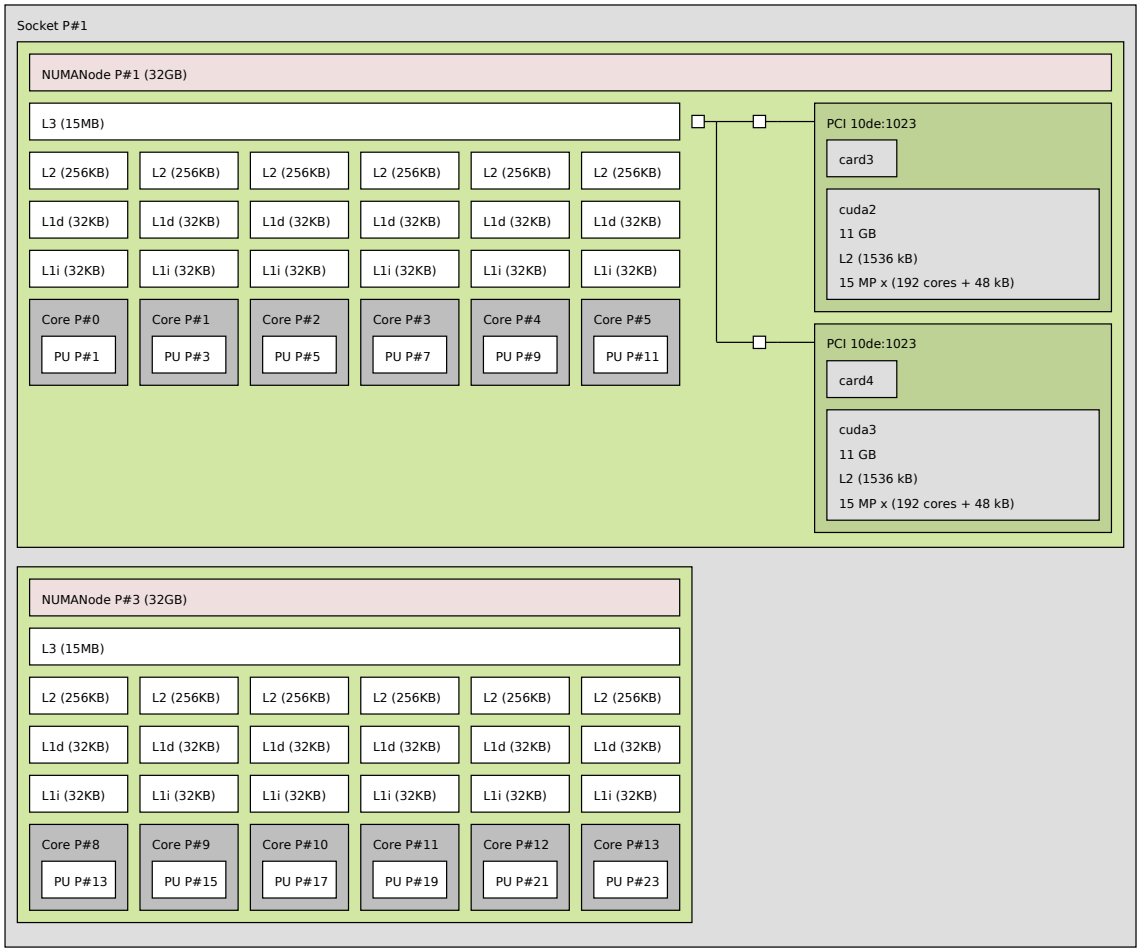


Figure 1.25: The architecture of one of the two sockets on the Sirocco computer.

id	Mat. name	Ordering	m	n	nz	op. count (Gflop)
1	tp-6	colamd	142752	1014301	11537419	277
2	karted	colamd	46502	133115	1770349	279
3	EternityII_E	colamd	11077	262144	1572792	566
4	degme	colamd	185501	659415	8127528	629
5	cat_ears_4_4	colamd	19020	44448	132888	786
6	hirlam	colamd	1385270	452200	2713200	2401
7	e18	colamd	24617	38602	156466	3399
8	flower_7_4	colamd	27693	67593	202218	4261
9	Rucci1	colamd	1977885	109900	7791168	12768
10	sls	colamd	1748122	62729	6804304	22716
11	TF17	colamd	38132	48630	586218	38209
12	hirlam	SCOTCH	1385270	452200	2713200	1384
13	flower_8_4	SCOTCH	55081	125361	375266	2851
14	Rucci1	SCOTCH	1977885	109900	7791168	5671
15	ch8-8-b3	SCOTCH	117600	18816	470400	10709
16	GL7d24	SCOTCH	21074	105054	593892	16467
17	neos2	SCOTCH	132568	134128	685087	20170
18	spal_004	SCOTCH	10203	321696	46168124	30335
19	n4c6-b6	SCOTCH	104115	51813	728805	62245
20	sls	SCOTCH	1748122	62729	6804304	65607
21	TF18	SCOTCH	95368	123867	1597545	194472
22	lp_nug30	SCOTCH	95368	123867	1597545	221644
23	mk13-b5	SCOTCH	135135	270270	810810	259751
24	TF16	colamd	15437	19321	216173	2884

Table 1.1: The set of matrices used for the experiments.

Chapter 2

Performance analysis approach

Performance profiling and analysis is one of the cornerstones of High Performance Computing. Nonetheless, it may be a challenging task to achieve, especially in the case of complex applications or algorithms and large or complex architectures. Therefore, it is no surprise that performance profiling has been the object of a very vast amount of literature.

In the case of sequential applications, a rather simple but effective approach consists in computing the efficiency of the code as a ratio of the attained speed and a reference performance which depends on the peak capability of the underlying architecture. Because processing units and memories work at different speeds, this reference performance varies depending on whether the application is compute bound (i.e., limited by the speed of the processing unit) or memory bound and to what extent. The *Roofline model* [115] is a popular method for computing this performance upper bound as a function of the *operational intensity*:

$$\begin{array}{c}\text{Attainable}\\ \text{Gflop/s}\end{array} = min \left\{ \begin{array}{c}\text{Peak Floating-point}\\ \text{Performance}\end{array} , \begin{array}{c}\text{Peak Memory}\\ \text{Bandwidth}\end{array} \times \begin{array}{c}\text{Operational}\\ \text{intensity}\end{array} \right\}.$$

The peak floating-point performance and peak memory bandwidth can be set equal to the theoretical values of the architecture; these values, however, are commonly unattainable and therefore these parameter values are computed using benchmarks like the BLAS `_gemm` (matrix-matrix multiply) operation, which is commonly considered as the fastest compute-bound operation, or the STREAM [88] benchmark, respectively. The roofline model can also be used for shared-memory, parallel applications but cannot be extended to the case where accelerators are used or to distributed-memory, parallel codes. Moreover, its use is difficult in the case of complex applications, such as the multifrontal method, which include both memory and compute-bound operations whose relative weight varies depending on the input problem.

The performance of a parallel code, either shared or distributed-memory, on a homogeneous platform (i.e., where all the processing units are the same) is commonly assessed measuring its *speedup*, i.e., the ratio between the sequential and the parallel execution times $t(1)/t(p)$, or, equivalently, the *parallel efficiency*

$$e(p) = \frac{t(1)}{t(p) \times p}$$

where p is the number of processing units used. The *scalability* measures the ability of a parallel code to reduce the execution time as more resources are provided. *Amadahl's law* can be used to define a bound on the achievable speedup (or parallel efficiency or

scaling) of a parallel code but, again, this is very hard to achieve for complex and irregular applications.

The emerging heterogeneous architectures represent a challenge for the performance evaluation of parallel algorithms compared to the uni-processor and parallel, homogeneous environments. Accelerators, not only process data at different speeds compared to the CPUs but also have different capabilities, i.e., are more or less suited to different types of operations, and are attached to their own memory which has different latency and bandwidth than that on the host. The performance analysis of codes that use accelerators is often limited to measuring the added performance brought by the accelerators, that is, a simple speed comparison with the CPU-only execution.

Although the above presented techniques can be used to achieve a rough evaluation of the performance of a parallel code, none of them provides any insight which can guide the HPC expert in reformulating or improving his algorithms or the programmer in optimizing his code in order to achieve better performance. Many factors play an important role in the performance and scalability of a code; among the others, we can mention the cost of data transfers and synchronizations, the granularity of operations, the properties of the algorithm and the amount of concurrency it can deliver. A quantitative evaluation of these factors can be extremely valuable.

In order to evaluate the effectiveness of the techniques proposed in Chapters 3, 4 and 6 as well as of the software that implements them, we developed a novel performance analysis approach [S1, C2]. First, we introduce a method for computing a relatively tight upper bound for the performance attainable by the parallel code, which is not merely a sum of the peak performance of the available processing units; this performance reference is computed by not only taking into account the features of the underlying architecture, but also the properties of the implemented algorithm and allows for evaluating the efficiency of a parallel code. Then we show how it is possible to factorize this efficiency measure into a product of terms that allow for assessing, singularly, the effect of several factors playing a role in the performance and scalability of a code. This analysis requires the ability to retrieve specific information from the execution that are easy to gather when using a runtime system.

2.1 General analysis

Consider the problem of evaluating the execution of a parallel application on a target computing environment composed of p heterogeneous processors such as CPUs and GPUs workers. Critical factors playing a role on parallel executions must be considered to compute realistic performance bounds and understand to what extent each of these factors may limit the performance. Idle times and data transfers (communications, in general) for example represent a major bottleneck for performance of parallel executions. Also we seek to quantify the cost of the runtime system, if any, compared to the workload in order to evaluate the effectiveness of these tools and estimate the overhead they induce.

In the proposed performance evaluation approach we perform a detailed analysis of the execution times by considering the cumulative times spent by all threads in the main phases of the execution:

- $t_t(p)$: The time spent in tasks which represent the workload of the application;
- $t_r(p)$: The time spent in the runtime for handling the execution of the application (in our case, this includes building the DAG and scheduling the tasks);

- $t_c(p)$: The time spent performing communications that are not overlapped by computations. This corresponds to the time spent by workers waiting for data to be transferred on their associated memory node before being able to execute a task;
- $t_i(p)$: The idle time spent waiting for dependencies between tasks to be satisfied.

The execution time of the factorization $t(p)$, may be expressed, using these cumulative times, as follows:

$$t(p) = \frac{t_t(p) + t_r(p) + t_c(p) + t_i(p)}{p}$$

The efficiency of a parallel code can be defined as

$$e(p) = \frac{t^{min}(p)}{t(p)}$$

where $t^{min}(p)$ is a lower bound on the execution time with p processes. A possible way of computing $t^{min}(p)$ for a task graph is to measure the execution time associated with the optimal schedule. However, given the complexity of the task scheduling problem in the general case it is not reasonable to compute this $t^{min}(p)$ for any input problem. Instead we choose to use a looser bound that consists in computing the optimal value of $t^{min}(p)$ for a relaxed version of the initial scheduling problem built on the following assumptions:

1. There are no dependencies between tasks which means that we consider an embarrassingly parallel problem, i.e., $t_i(p) = 0$;
2. The runtime does not induce any overhead on the execution time, i.e., $t_r(p) = 0$;
3. The cost of all data transfers is equal to zero, i.e., $t_c(p) = 0$;
4. Tasks are *moldable* meaning that they may be processed by multiple processors.

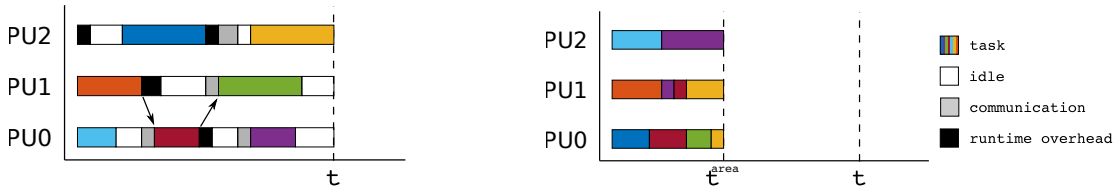
In order to compute $t^{min}(p)$ we introduce the following notation: for a set of tasks Ω running on a set of p resources denoted by R, we define α_r^ω as the share of work in task ω processed by resource r and t_r^ω as the time spent by resource r processing its share of task ω. Then $t^{min}(p)$ can be computed as the solution of the following linear program:

Linear Program 1
Minimize T such that, for all $r \in R$ and for all $\omega \in \Omega$:

$$\sum_{\omega \in \Omega} \alpha_r^\omega t_r^\omega = t_r \leq T \qquad \sum_{r=1}^{|R|} \alpha_r^\omega = 1$$

where the t_r^ω can be computed using a performance model. Note that the problem of finding $t^{min}(p)$ is equivalent to minimizing the area $\alpha_r^\omega t_r^\omega$ for all $\omega \in \Omega$ and $r \in R$. For this reason the optimal value $t^{min}(p)$ is replaced by $t^{area}(p)$ in the following. We illustrate how $t^{area}(p)$ is defined in Figure 2.1 on a simple execution with three resources. On the left of the figure is represented the trace of the real execution where $t(p)$ is measured. On the right is represented the optimal schedule for the relaxed version of the original scheduling problem where $t^{area}(p)$ is measured.

Note that parallelism is normally achieved by partitioning operations and data; this implies a smaller granularity of tasks and thus, likely, a poorer performance of the operations



(a) Actual schedule. (b) Area schedule.

Figure 2.1: Illustration on a simple Gantt chart of a parallel execution with three workers.

performed by them. Moreover, parallel algorithms often trade floating-point operations for concurrency and therefore may perform more operations than the corresponding sequential ones (this is, for example, the case of 2D communication avoiding QR factorizations, as explained in Section 1.3.1). Based on these observations, we can further refine the efficiency definition above replacing $t^{area}(p)$ with $\tilde{t}^{area}(p)$ computed as the solution of Linear Program 1 assuming $\omega \in \tilde{\Omega}$, where $\tilde{\Omega}$ is the set of tasks of the sequential algorithm. In other words, in $\tilde{t}^{area}(p)$ we assume that there is no performance loss when working on partitioned data and that the parallel algorithm has the same cost as the sequential one. Please note that this also models the fact that in some cases data are inherently of small granularity and, therefore, tasks that work on them have a poor performance regardless of the partitioning.

By replacing the term $t(p)$ in the expression of the parallel efficiency using cumulative times and noting that $\tilde{t}_t^{area}(p) = p \times \tilde{t}^{area}(p)$ from the definition of our lower bound, we may express the parallel efficiency as as

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}_t^{area}(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

$$= \overbrace{\frac{\tilde{t}_t^{area}(p)}{t_t^{area}(p)}}^{e_g} \cdot \overbrace{\frac{t_t^{area}(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

This expression allows us to decompose the parallel efficiency as the product of five well identified effects:

- e_g: the **granularity efficiency**, which measures how the overall efficiency is reduced by the data partitioning and the use of parallel algorithms. This loss of efficiency is mainly due to the fact that because of the partitioning of data into fine grained blocks, elementary operations do not run at the same speed as in the purely sequential code and also to the fact that the parallel algorithm may perform more flops than the sequential one;

- e_t: the **task efficiency**, measures how well the assignment of tasks to processing units matches the tasks properties to the units capabilities as well as the exploitation of data locality (more details are provided below);

- e_r: the **runtime efficiency**, which measures the cost of the runtime system with respect to the actual work done;

- e_c: the **communication efficiency**, which measures the cost of communications with respect to the actual work done due to data transfers between workers;

- e_p: the **pipeline efficiency**, which measures how well the tasks have been pipelined. This includes two effects. First, the quality of the scheduling because if the scheduling policy takes bad decisions (for example, it delays the execution of tasks along the critical path) many stalls can be introduced in the pipeline. Second, the shape of the DAG or, more generally, the amount of concurrency it delivers: for example, in the extreme case where the DAG is a chain of tasks, any scheduling policy will do as bad because all the workers except one will be idling at any time.

Analysis for homogeneous multicore systems In the case of an homogeneous architecture such as a multicore system with p cores, the solution of Linear Program 1 greatly simplifies and can be easily found.

$\tilde{t}^{area}(p)$ and $t^{area}(p)$ do not have to be computed but can be measured by timing, respectively, the sequential execution of the sequential algorithm and the sequential execution of the parallel algorithm (with the corresponding data partitioning), that is

$$\tilde{t}^{area}(p) = \frac{\tilde{t}(1)}{p}, \quad t^{area}(p) = \frac{t(1)}{p}. \tag{2.1}$$

Note that in a sequential execution the cost of the communications and of the runtime as well as the idle times are all identically equal to zero and therefore $t(1) = t_t(1)$ and $\tilde{t}(1) = \tilde{t}_t(1)$.

Replacing $t(p)$ and $t^{area}(p)$ in the expression of the parallel efficiency using the cumulative times we obtain:

$$\begin{aligned} e(p) &= \frac{\tilde{t}_t(1)}{t_t(p) + t_r(p) + t_i(p)} \\ &= \overbrace{\frac{\tilde{t}_t(1)}{t_t(1)}}^{e_g} \cdot \overbrace{\frac{t_t(1)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_i(p)}}^{e_p}. \end{aligned}$$

Here we assumed $e_c = 1$ because there are no explicit or measurable data transfers. These, however, happen implicitly when tasks access data which are remotely located in the NUMA memory system; this makes the tasks execution time $t_t(p)$ increase as the number of cores p increases. This effect is measured by e_t.

Analysis for heterogeneous systems Compared to the homogeneous case, computing $\tilde{t}^{area}(p)$ and $t^{area}(p)$ in the context of heterogeneous systems is more complex because task execution times depend on the type of resources where they are executed. Therefore we need need to solve the Linear Program in 1 to determine these values. The first step consists in gathering the execution times for every task on every possible computational unit. Then the linear system is built either statically if all tasks are known in advance or by registering it during an actual execution and finally solved using a linear program solver.

The tasks efficiency, in this case, still measures the effect of implicit communications but also, and more importantly, how well the heterogeneity of the processing units is exploited. In other words, e_t measures how well tasks have been mapped to computational units with respect to the optimal mapping computed by solving the linear program. Note that this efficiency is not necessarily lower than one and it is closely related to the pipeline efficiency:

- $e_t < 1$: tasks are globally being executed at a lower speed with respect to the optimal. This may happen because the tasks assigned to the different processing units are not of the good type; this is, for example, the case where very small granularity tasks are executed by GPUs;
- $e_t > 1$: note that a particularly naive scheduling policy can map all the tasks to faster units. This would obviously result in a small cumulative tasks execution time $t_t(p)$ but would inevitably lead to the starvation of the slower units and, as a consequence, to a poor pipeline efficiency e_p.

As a result, the quality of the scheduling policy can be measured by the product of the tasks and pipeline efficiencies $e_t \cdot e_p$ which is always less than one.

2.2 Discussion

As discussed in the previous section, the pipeline efficiency e_p may be lowered either because of a lack of parallelism in the DAG or as a result of bad scheduling decisions taken during the execution. The distinction between the two phenomena can be made qualitatively by looking at the execution traces but the analysis is basic and inaccurate.

Section 4.2.3. The analysis of the critical path defined as the longest path in the the DAG of tasks allows to decide what is the limiting factor for the pipeline efficiency. If we note CP the set of tasks on the critical path (longest path from an entry task to an exit task) of the DAG and L_ω the workload associated with the task ω, the maximum achievable speedup is the following:

$$S_\infty = \frac{\sum_{\omega \in \Omega} L_\omega}{\sum_{\omega \in CP} L_\omega}$$

This quantity gives the maximal amount of parallelism available in a DAG, and given a multicore architecture, one can expect to have at most the speedup given by its value compared to a sequential execution. In practice, the actual speedup would be much lower because of the aforementioned effects such as granularity efficiency and locality efficiency. Therefore, when targeting an architecture such as a multicore machine with p cores, a low value S_∞ (lower than p for example) indicates that the DAG is not suited and will not deliver enough parallelism to feed all the resources.

In a heterogeneous context the formulation of S_∞ becomes more complex as there is no longer equivalence between the length of a task and its execution time. For this reason the critical path may not be interpreted as the longest path in the DAG in terms of work load. In an effort to adapt the definition of S_∞ to heterogeneous architecture it is possible to reformulate it by replacing the length of the critical path with a lower bound on its length. For example we define CP_{min} as the critical path of a DAG, considering the minimum computational cost. Using the previous notations the minimal computational cost of a task ω on every resources may be expressed as $\min_{r \in R} t_r^\omega$. Then we use this notion to derive another definition of S_∞ as follows:

$$S_\infty = \frac{\sum_{\omega \in \Omega} \min_{r \in R} t_r^\omega}{\sum_{\omega \in CP_{min}} \min_{r \in R} t_r^\omega}$$

The previous definition of S_∞ for homogeneous architectures may be seen as a particular case of this definition. Note that the denominator in the latter formula does not equal the length of the critical path because it does not take into account the communication cost between processors. In addition as we will see later the definition of a critical path is

not trivial in an heterogeneous context as it is associated with a choice of metric for the computational cost of a task that can have several interpretations. For instance the critical path may be determined by considering the average computation cost of tasks denoted $\bar{t}^{\omega}$.

The second metric we use to evaluate the execution time $t(p)$, commonly referred to as *makespan*, is the ratio between this value and the length of the critical path whose expression is straightforward in a homogeneous case and can be interpreted as the efficiency of the makespan:

$$e_{makepan}(p) = \frac{t(p)}{\sum_{\omega \in CP} t_{\omega}}$$

As for the definition of our previous metric this formula may not be easily generalized to a heterogeneous context. Using the same methodology as previously we simply replace the expression of the critical path by a lower bound on the length of the critical path therefore corresponding to a lower bound on the makespan ensuring that this quantity is inferior to one.

Another idea to take into account the critical path in the case of a parallel heterogeneous execution, suggested by Agullo et al. [6], is to integrate a constraint in Linear Program 1 corresponding to the lower bound on the length of the critical path that we use in the previous metrics S_{∞} and $e_{makepan}(p)$. With this constraint, we express the fact that the makespan may not be larger than this lower bound:

$$\sum_{\omega \in CP_{min}} \min_{r \in R} t_r^{\omega} \leq T$$

However in the general case the number of variables is important (one variable per task) and increases with the size of the problem which makes the linear program hard to solve without constraints and consequently even harder when adding a constraint. Agullo et al. [6] show that given a problem (Cholesky factorization in their case) it is possible to reduce the linear program to another one of small size and whose size is independent from the problem. This is made possible by the regularity of dense linear algebra algorithm that they tackle. It may be noted that in our study it is not possible to make any assumptions on the regularity our problems.

To handle the complexity of complex DAGs arising in the case of sparse linear algebra it is possible to evaluate the impact of scheduling algorithm and communication using simulations. As presented by Agullo et al. [6] and Stanisic et al. [108] in the case of dense algorithms and as we show in Chapter 7.2 in the context of `qr_mumps` for sparse problems, with the help of tools that may be integrated in runtime systems such as StarPU-SimGrid it is possible to reproduce execution of a DAG on a parallel machine. In addition some parameters may be changed such as bandwidths and tasks processing speeds (in order to remove the effect of locality for example) in order to measure the influence of these parameters on the execution time and evaluate the behavior of scheduling strategies under some circumstances (for example when the number of processors grows) without performing the actual execution.

Chapter 3

Task-based multifrontal method: porting on a general purpose runtime system

The task-based multifrontal QR method implemented in `qr_mumps` and presented in Section 1.7 constitutes an extremely irregular workload, with tasks of different granularities and characteristics and with variable memory consumption. In the `qr_mumps` solver, the tasks that form this workload are scheduled through a hand-written code which relies on the knowledge of the algorithm. In this chapter we will first comment on the shortcomings of the scheduling method implemented in the `qr_mumps` solver described in Section 1.7. We will then focus on the porting of the `qr_mumps` algorithm on top of a general purpose runtime system in order to asses the usability of these tools on such large and complex workloads [C1]. As explained, modern runtime systems provide programming interfaces that comply with a DAG-based programming paradigm and powerful engines for scheduling the tasks into which the application is decomposed. For these reason, these tools are very well suited for `qr_mumps`; nonetheless, this porting is not straightforward and requires some care. The result of this work is a preliminary version of a new code which we refer to as `qrm_starpu`. This code will be used to run the experiments presented in Section 3.2.3 that show how this migration to a general purpose runtime system leads to a much more modular and portable code at the cost of a negligible loss of performance.

3.1 Efficiency and scalability of the `qr_mumps` scheduler

A consequence of the design of the `qr_mumps` scheduler presented in Section 1.7.4 is that the size of the search space for the `fill_queues` routine is proportional to the number of active fronts, at a given moment, during the factorization. The size of this search space, however, can grow excessively large and, consequently, the cost of the `fill_queues` routine may become unbearable. This is the case for example when the the elimination tree is composed of a large number of nodes. Moreover, when the number working threads grows we observe the following two effects: first the layer computed for the logical tree pruning (see Section 1.7.2) in the elimination tree tends to go down the tree which increases the number of frontal matrices to be activated during the factorization; Second the number of resources to feed the scheduler becomes higher and the activation of frontal matrices is triggered sooner during the factorization. In particular, nothing prevents the scheduler from activating all the frontal matrices as soon as they are ready which can make the search space arbitrarily large. Two different techniques were presented by Buttari [32] to

mitigate this problem. The first is the logical tree pruning discussed in Section 1.7.2 which essentially reduces the size of the tree although, as explained above, this reduction is less and less effective as the number of threads grows. The second is a tree reordering method that aims at computing a postorder traversal which reduces the number of simultaneous active nodes in a sequential execution; this is essentially a rewriting of the method proposed by Liu [83] for minimizing the memory consumption (see also Section 5.2.1) but with a different objective. It must be noted, though, that this method is just a heuristic and does not provide any mean of controlling the size of the search space; moreover, it prevents us from using other tree traversal orders that aim, for example, at optimizing performance of memory consumption.

In addition to this performance and scalability issue, it should be noted that the hand-coded scheduler in `qr_mumps` relies on the knowledge of the algorithm (specifically in the `fill_queues` routine) and may have to be modified each time the algorithm is updated.

In the next section we show that it is possible to overcome the limitations of the original `qr_mumps` scheduler presented above by replacing it with a modern and fully featured runtime system at basically no performance loss.

3.2 StarPU-based multifrontal method

3.2.1 DAG construction in the runtime system

In order to assess the usability of runtime systems for sparse, direct methods, we achieved, as a preliminary step, the implementation of the method described in Section 1.7 using one such tool. Namely, we used the StarPU runtime as a replacement for the hand-coded scheduler described in Section 1.7.4.1. In order to achieve this proof of concept, we tried to reproduce as accurately as possible the scheduling policy and the behaviour of the original `qr_mumps` code. This led to novel software, which we refer to as `qrm_starpu`. In our study we focus on the factorization phase that we express using the programming interface provided by StarPU while the analysis and solve phases remain unchanged from `qr_mumps`.

This novel implementation is structured around the pseudocode shown in Figure 3.1. This code is executed by the *master thread* which is not involved in the execution of tasks (see Section 1.4.2.1) and basically consists in a traversal of the elimination tree, where, at each node, the following operations are performed:

- Declare to the runtime system the dependencies between the activation of a node and the activation of its children nodes. In StarPU, this is done by assigning a identifier, called a `TAG`, to the corresponding tasks and then using these tags to add task dependencies through a dedicated routine. This dependency indicates that a node can only be activated once all of it children are activated;

- Submit the `activation` task to the runtime system containing the instructions for processing the current front. A description of this task is given in Figure 3.2.

The return from the blocking `wait_tasks_completion` routine in Figure 3.1 ensures that all the submitted tasks have been executed, i.e., that the factorization of the sparse matrix is completed.

The `activation` task has been enriched with respect to what presented in Section 1.7; not only does this task initialise the front data structure and allocate the corresponding memory, but it is also in charge of submitting the other tasks related to the activated front,

```
forall fronts f in topological order
   forall children c of f
      ! declare explicit dependency between node c and f
      call declare_dependency(id_f <- id_c)
   end do
   ! submit the activation of front f
   call submit(activation, f, id=id_f)
end do

call wait_tasks_completion()
```

Figure 3.1: Pseudo-code for the main code with submission of `activation` tasks

namely the `assemble`, `_geqrt` and `_gemqrt` tasks. Note that when the block-columns are allocated, they are also declared to the runtime system. This is done through the use of dedicated data structures called *handles*, as described in Section 1.4.2.1 and is a necessary step for the submission of the tasks that operate on block-columns and for the detection of their mutual dependencies.

Note that the `deactivate` task does not appear in the pseudocode given in Figure 3.1. This task instead, is executed in the *call-back* (see Section 1.4.2.1) associated with `assemble` tasks when assembly operation is complete. This completion is detected by means of a counter whose access is protected by a lock managed by the application and not visible by StarPU.

```
call activate(f)

forall children c of f
   forall blockcolumns j=1...n in c
      ! assemble column j of c into f
      call submit(assemble, c(j):R, f:RW)
   end do
end do

forall panels p=1...n in f
   ! panel reduction of column p
   call submit(_geqrt, f(p):RW)
   forall blockcolumns u=p+1...n in f
      ! update of column u with panel p
      call submit(_gemqrt, f(p):R, f(u):RW)
   end do
end do
```

Figure 3.2: Pseudo-code of the `activation` task.

Using the notation introduced in Section 1.7.1, Figure 1.18 we detail the declaration of dependencies encountered in our DAGs emerging from the multifrontal factorization:

- The dependency `d1` is ensured by the fact that the numerical operations are submitted in the `activation` tasks after the execution of the `activate` routine;
- The tasks corresponding to the numerical factorization of the front are submitted in the right order to express the dependencies `d2`, `d3` and `d4`. `_gemqrt` tasks related to a

panel reduction are submitted after the corresponding `_geqrt` task, which implicitly defines dependency `d2`. Similarly the `_geqrt` task on panel i is submitted after the submission of the `_gemqrt` task from step $i-1$ on the same block-column ensuring `d3` dependency. Finally `_gemqrt` operations with respect to panel reduction i are submitted after the one related to panel reduction $i-1$ creating the dependency `d4`;

- Using a topological order to traverse the elimination tree ensures that assembly tasks for a front are submitted after the submission of factorization tasks of children fronts which gives the dependency `d5` by inference;

- The dependency `d6` is expressed by submitting the `assemble` tasks before numerical tasks. This way no block-column is processed before it has been completely assembled;

- The dependency `d7` is explicitly expressed in the runtime system with the instruction `declare_dependency`.

In total we used three different ways to declare the dependencies in our DAGs and ensured the correctness of the numerical factorization: explicitly declared dependencies for `activation` tasks, inferred dependencies `_geqrt`, `_gemqrt` and `assemble` tasks and callback-based dependencies for the `deactivate` tasks. It must be noted, however, that these techniques only allow for expressing precedence relations between tasks but not a complete *data-flow*; this means that the runtime system is not fully aware of what data are needed for a task to execute. For example the dependencies between `activation` tasks are not associated with any data and dependencies between `assemble` and `deactivate` tasks are invisible to the runtime system. This is not important in a single-node, shared memory system because only one copy of each data exists and is never moved. On a system with multiple memory nodes (like a GPU equipped system or a distributed memory, parallel computer), however, the runtime would not be able to move the necessary data to the place where a task is actually executed and would not be able to handle the consistency between multiple copies of the same data. This is the main reason that led us to abandon this approach in favor of the STF compliant one described in Chapter 4.

The use of the DAG in the runtime system during the factorization is illustrated in Figure 3.3 using the DAG previously presented in Figure 1.18:

1. In phase 1, the `activate` tasks of supernodes 1, 2 and 3 are submitted and their mutual dependencies explicitly declared to StarPU. In this example the activation of supernode 3 depends on the activation tasks of supernodes 1 and 2 which are, therefore, the only two ready for execution;

2. In the case where multiple working threads are available, we can assume that supernodes 1 and 2 are activated at the same time in phase 2. Because these are leaf nodes their activation does not depend on any other node activations. As a result of the activation of these two supernodes, the related numerical tasks are submitted to StarPU;

3. When the execution of the numerical tasks associated with nodes 1 and 2 is terminated, supernode 3 is activated in phase 3; this is possible because the dependencies with the activations of nodes 1 and 2 are satisfied. Same as for the previous activations the numerical tasks are submitted in supernode 3, and in addition the assembly operations from child fronts 1 and 2 to front 3 are submitted;

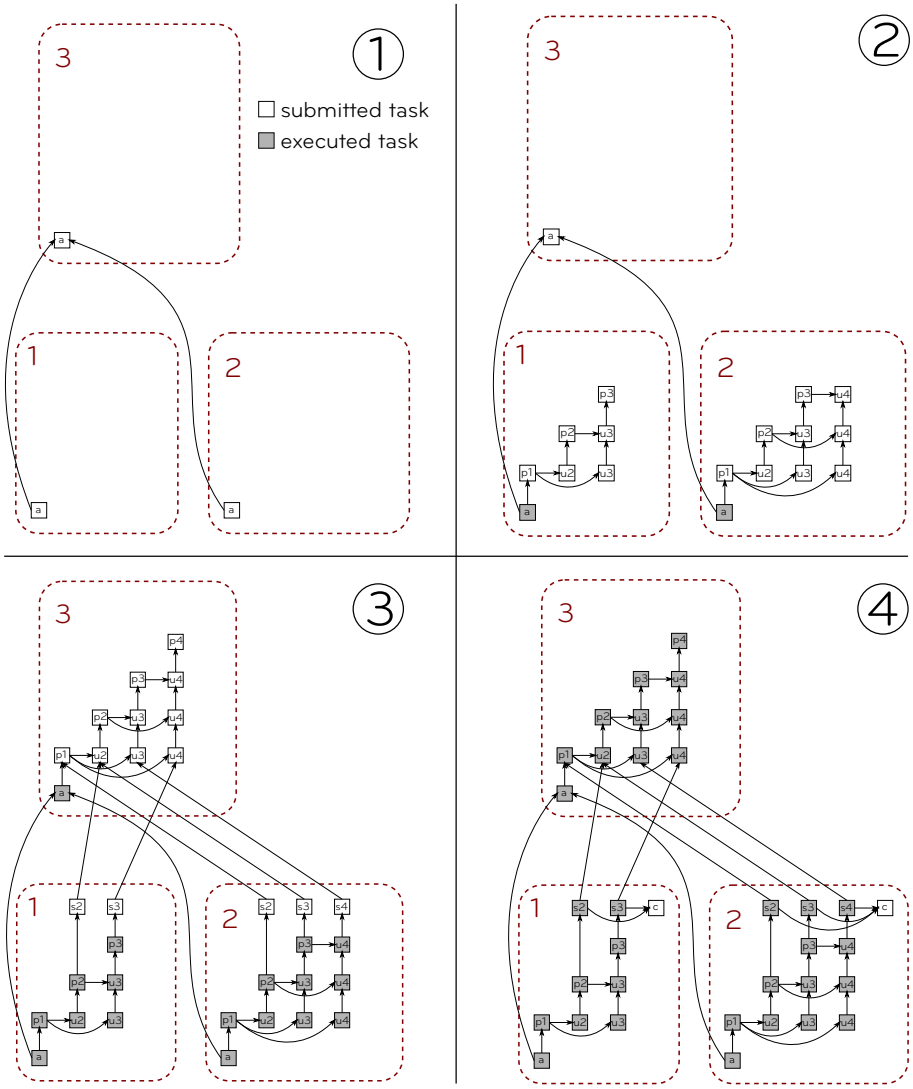


Figure 3.3: Dynamic construction of the DAG in `qrm_starpu`.

4. In phase 4 the `deactivate` tasks are submitted for front 1 and 2 at the end of assembly operations via callbacks of assembly tasks.

Please note that the activation of supernode 3 becomes ready at the end of step 2 and, therefore, nothing prevents the runtime system from executing it before the factorization tasks related to fronts 1 and 2. To some extent, the order of execution of tasks can be controlled with an appropriate tasks priority policy, as explained below.

In `qrm_starpu` the task submission is done progressively following the traversal of the elimination tree which depends on the execution of activation tasks. Similarly to the `qr_mumps` approach, only tasks corresponding to active frontal matrices are visible to the runtime system. This allows us to reduce the number of tasks in the runtime system and potentially reduces its memory consumption. We will provide experimental results in Section 3.2.3 to show how this technique effectively reduces the size of the DAG handled by the runtime system. The resolution of dependencies in StarPU is local to each task and is performed upon task completion: whenever a task is finished, the executing worker checks for potential ready tasks to push in the scheduler only among those which depend

on it. As a result, the tracking and updating of dependencies has a much smaller cost than the method implemented in `qr_mumps`; most importantly, this cost only depends on the number of outgoing edges of each task and is, therefore, relatively independent of the size of the input sparse problem.

There is one fundamental difference between `qr_mumps` and `qrm_starpu`. In the multifrontal QR method, the assembly of a front is an embarrassingly parallel operation because it simply consists in the copy of coefficients of the contribution blocks from the child nodes into different locations in the front. This means that any two assembly tasks can be done in any order and possibly in parallel even though they access the same block-column. `qr_mumps` can easily take advantage of this fact because the task dependencies are hard-coded in the `fill_queues` routine based on the knowledge of the algorithm. In `qrm_starpu`, instead, if two or more assembly tasks access the same block-column, a data-hazard is detected and thus the tasks are serialized according to the order of submission. In order to partially address this problem, the StarPU developers implemented a novel access mode `STARPU_COMMUTE` which makes it possible to instruct the runtime system about the possibility of executing two tasks in any order (but not in parallel).

This is an example of how our developments and evaluations provide valuable feedback to the community of runtime systems developers; we will show other examples of this fruitful collaboration in the next sections. This feature, however, was not yet available at the moment we implemented `qrm_starpu` and ran the experiments in Section 3.2.3; the commutativity among assembly tasks was instead used in the work described in Chapters 4, 5 and 6.

3.2.2 Dynamic task scheduling and memory consumption

The level of concurrency as well as the memory consumption of the multifrontal method based on StarPU depend on the path followed in the DAG and the scheduling policy used. As explained, the numerical tasks (i.e., panel reductions, updates and assemblies) are submitted by activation tasks which are, therefore, extremely important because they potentially increase the concurrency level and enhance the scalability. It must be noted, however, that executing an activation task too early is worthless because the numerical tasks it submits cannot be executed as they depend on many other, previously submitted tasks. On the other hand, the activation tasks in charge of allocating fronts increase the memory consumption. For this reason, during the dynamic construction of the DAG we need an efficient scheduling policy to have an amount of concurrency suited to the resources and to save memory from unnecessary allocations.

Maximizing concurrency while limiting the memory consumption is achieved by assigning to each type of tasks priority values as follows:

- `activate`: This tasks increases the memory consumption by allocating frontal matrix data structures and therefore we assign it a negative priority to prevent the scheduler from allocating fronts when there is enough parallelism to feed all resources and thus limit memory consumption;

- `assemble`: These tasks are critical for concurrency because all numerical tasks from a front depend on it. They are given the highest priority after deactivate operation which is 3;

- `_geqrt`: panel operations lie on the critical path of the DAG corresponding to the dense QR factorization when using a 1D block-column partitioning. They are assigned priority 2 which is higher than non-critical update tasks;

- _gemqrt: update operations are given priority 1 which is the lowest for numerical tasks;

- deactivate: The deactivation is responsible for deallocating the data structure of a frontal matrix and thus decreases the memory consumption. We therefore give this task the highest priority which is 4.

Similarly to the strategy employed in qr_mumps, activating a new front is only considered when no other tasks can be executed by a worker. As such, we exploit as much as possible the node-level parallelism instead of tree-level parallelism and thus take advantage of a better data locality and a lower memory consumption. In addition the native scheduler in qr_mumps can only handle two levels of task priority while in our implementation we consider an arbitrary number of priority.

The order of execution of activation tasks impacts the memory consumption for the factorization by allocating the front memory and allowing the deallocation of children nodes memory for contribution blocks as explained in [60]. In order to minimize as much as possible the memory footprint of the factorization, the front activation should follow as much as possible a postorder traversal of the elimination tree which allows us to minimize this memory consumption in the sequential case. To achieve a good efficiency of the memory usage it is important to prioritize the activation tasks according to this postorder traversal and this can be done by modifying the priority associated with activation tasks as follows: for a node numbered i using a postorder traversal we give the corresponding activation task a priority equal to $-i$. As shown in the experimental results below, this allows a conservative memory behaviour for the scheduling strategy compared to qr_mumps. This ordering of the elimination tree minimizes the number of active nodes during the factorization and thus maximizes the exploitation of node-level parallelism. However, for parallel execution, there is no guarantee on the actual execution of activation tasks and thus control over the memory usage of the factorization.

Although StarPU comes with some predefined scheduling policies, none of them supports arbitrary priorities. Therefore we choose to implement our own scheduler using the dedicated API (see Section 1.4.2.1). The implementation of our dynamic scheduler, illustrated in Figure 3.4, is based on a central sorted queue where tasks are ordered according to their priority and can be described by the following two routines:

- the push routine inserts a ready task to the central queue keeping the list sorted according to the task priorities. Upon termination of a task, the worker that has completed it checks the status of all the other tasks which depend on it and, if any of these has become ready for execution, it invokes the push method on it. This dependency check plus the push method may be seen as the equivalent of the fill_queues routine although it is much cheaper because the search space for ready tasks is much more restricted;

- the pop routine retrieves the highest priority task from the central queue. It is called by workers when they become idle and is equivalent to the pick_task from qr_mumps.

This scheduler is dynamic, generic and capable of taking into account task priorities. Moreover, it is compatible with every kind of worker supported by the runtime system including CPU and GPU workers although we do not recommend its use on such architectures where smarter scheduling policies are necessary to achieve acceptable performance (see Chapter 6). Unlike the qr_mumps scheduler, it does not exploit data locality in a NUMA system; the results presented by Buttari [32], however, show that that the locality

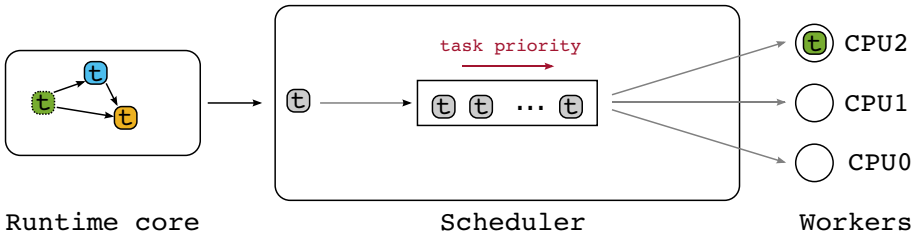


Figure 3.4: Sorted central queue scheduler in `qrm_starpu`.

aware scheduling proposed therein (see Section 1.7.4.1) only provided mild improvements and that an interleaved memory allocation policy does a much better job of reducing the penalty for distant memory accesses on a limited size NUMA machine. A much better method for dealing with the complexity of large NUMA machines is based on the concept of *contexts* which was developed and evaluated by Hugo et al. [71, 70] and which we briefly describe in Section 7.3. Another drawback of this scheduler lies in the fact that all the ready tasks are stored in a single queue: in the case were the number of ready tasks and of working threads is high, this may lead to a costly contention on the locks that prevent concurrent accesses to this data structure. In Section 4.4 we will describe the design and implementation of a novel scheduler that aims at overcoming these shortcomings but we believe that this scheduler provides the necessary features to conduct a fair comparison with the original `qr_mumps` code.

3.2.3 Experimental results

This section aims at evaluating the effectiveness of the proposed techniques as well as the performance of the resulting code. For this purpose, the behaviour of the `qrm_starpu` code will be compared to the original `qr_mumps` and also, briefly, to the SuiteSparseQR package (referred to as `spqr`) released by Tim Davis in 2009 [42]. The tests were done on a subset of the matrices in Table 1.1 on the Dude machine described in Section 1.9.1. Table 3.1 shows the factorization times (in seconds) using `qrm_starpu`, `qr_mumps` and `spqr` with different numbers of cores whereas Figure 3.5 presents the speedup achieved by each of these three solvers on 12 and 24 cores.

Both `qr_mumps` and `qrm_starpu` clearly outperform the `spqr` package by a factor greater than two thanks to the powerful programming and execution paradigm based on DAG parallelism. On the other hand, `qrm_starpu` is consistently but only marginally less efficient than `qr_mumps`, by a factor below 10% for eight out of eleven matrices and still only below 20% in the worst case. As a conclusion, the parallelization scheme impacts performance much more than the underlying low-level layer, validating the thesis that modular approaches based on runtime systems can compete with heavily hand-tuned codes.

For a better understanding of the behaviour of both `qr_mumps` and `qrm_starpu`, we conducted on both codes the analysis described in Section 2.1; the results of this analysis are reported on Figure 3.6. Please note that, for the purpose of this evaluation, the graph on the top-right corner of the figure reports the product of the granularity and the task efficiencies $e_g \times e_t$ as defined in Section 2.1 because these two metrics are expected to be equivalent for both codes as the results also prove. The granularity efficiency e_g , in fact, is the same since the same values for the inner and outer blocking parameters (see Figure 1.20) have been chosen for the two codes and thus exactly the same tasks are executed on both sides. As for the task efficiency e_t, it must be noted that all the experiments in

	Factorization time (sec.)										
Matrix	1	2	3	4	5	6	7	8	9	10	11
th.											
spqr 1	52.9	49.9	99.5	111.0	123.3	406.3	538.3	687.5	2081.0	4276.0	5361.0
spqr 2	34.5	32.1	63.4	69.0	74.1	238.1	290.1	379.1	1154.7	2870.4	2959.2
spqr 4	24.8	22.9	44.8	46.5	48.3	148.4	168.9	229.9	737.7	2001.0	1659.2
spqr 6	21.5	19.1	36.0	38.2	39.5	116.3	128.4	178.6	598.9	1845.9	1203.2
spqr 12	17.0	14.5	26.2	33.0	32.5	85.7	90.5	131.6	468.3	1644.0	769.9
spqr 18	15.7	12.7	22.5	28.8	29.1	73.4	78.7	119.1	404.9	1603.0	636.9
spqr 24	14.2	12.3	20.7	26.2	27.8	68.6	74.1	114.2	372.7	1389.3	588.6
qrm_starpu 1	51.8	49.0	97.5	104.8	137.5	417.6	496.1	733.6	1931.0	3572.0	5417.0
qrm_starpu 2	27.1	25.7	52.9	55.6	72.0	215.4	251.1	386.2	981.0	1813.0	2789.0
qrm_starpu 4	14.5	14.3	26.3	29.4	38.8	111.4	130.6	209.4	505.0	942.7	1410.0
qrm_starpu 6	10.6	10.2	18.9	20.8	27.2	77.9	91.6	153.7	349.9	676.0	979.4
qrm_starpu 12	6.9	6.2	10.9	12.4	16.2	43.4	50.4	92.4	190.3	439.3	525.8
qrm_starpu 18	6.0	5.0	9.0	9.8	13.1	32.0	38.1	69.1	146.3	363.7	371.1
qrm_starpu 24	5.7	4.4	8.0	8.5	12.4	28.1	32.9	58.0	122.7	336.3	305.9
qr_mumps 1	51.5	48.8	96.9	104.6	137.1	410.8	495.2	729.7	1928.0	3571.0	5420.0
qr_mumps 2	26.1	24.5	51.1	53.6	69.1	208.4	248.7	368.8	969.4	1793.0	2724.0
qr_mumps 4	13.7	12.7	25.0	27.0	35.6	105.1	125.6	189.8	505.0	903.9	1367.0
qr_mumps 6	9.6	8.7	17.1	18.4	24.5	71.9	85.8	127.9	333.0	617.6	919.1
qr_mumps 12	5.7	5.2	10.2	10.8	14.2	39.5	46.6	69.4	177.9	392.3	479.0
qr_mumps 18	4.9	4.3	8.3	8.6	11.7	29.6	34.0	53.3	137.2	342.3	339.0
qr_mumps 24	5.0	4.3	7.9	8.0	11.0	26.5	30.5	48.8	120.9	337.0	282.0

Table 3.1: Factorization times, in seconds, on an AMD Istanbul system for `qrm_starpu` (*top*), `qr_mumps` (*middle*) and `spqr` (*bottom*). The first row shows the matrix number.

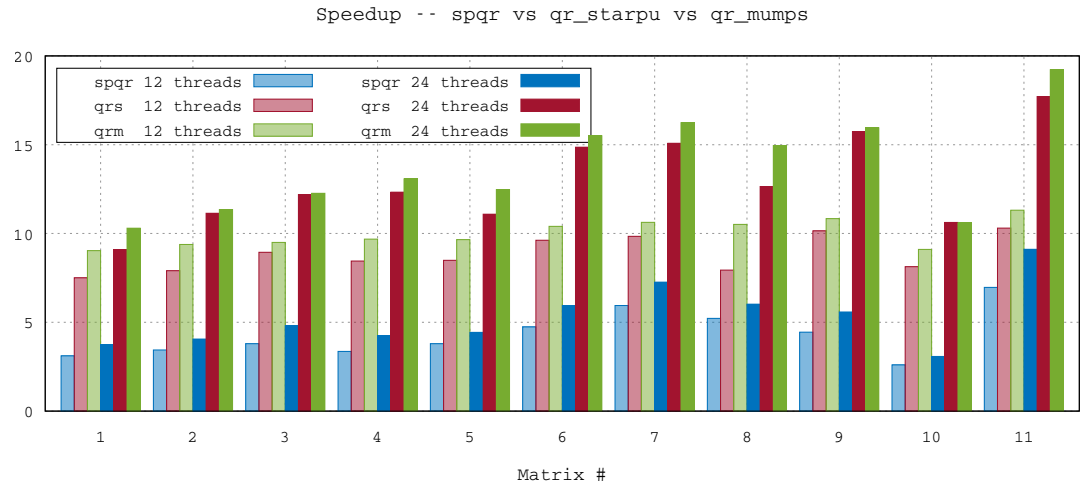


Figure 3.5: Speedup for the `spqr`, `qrm_starpu` and `qr_mumps` factorizations on 12 and 24 cores.

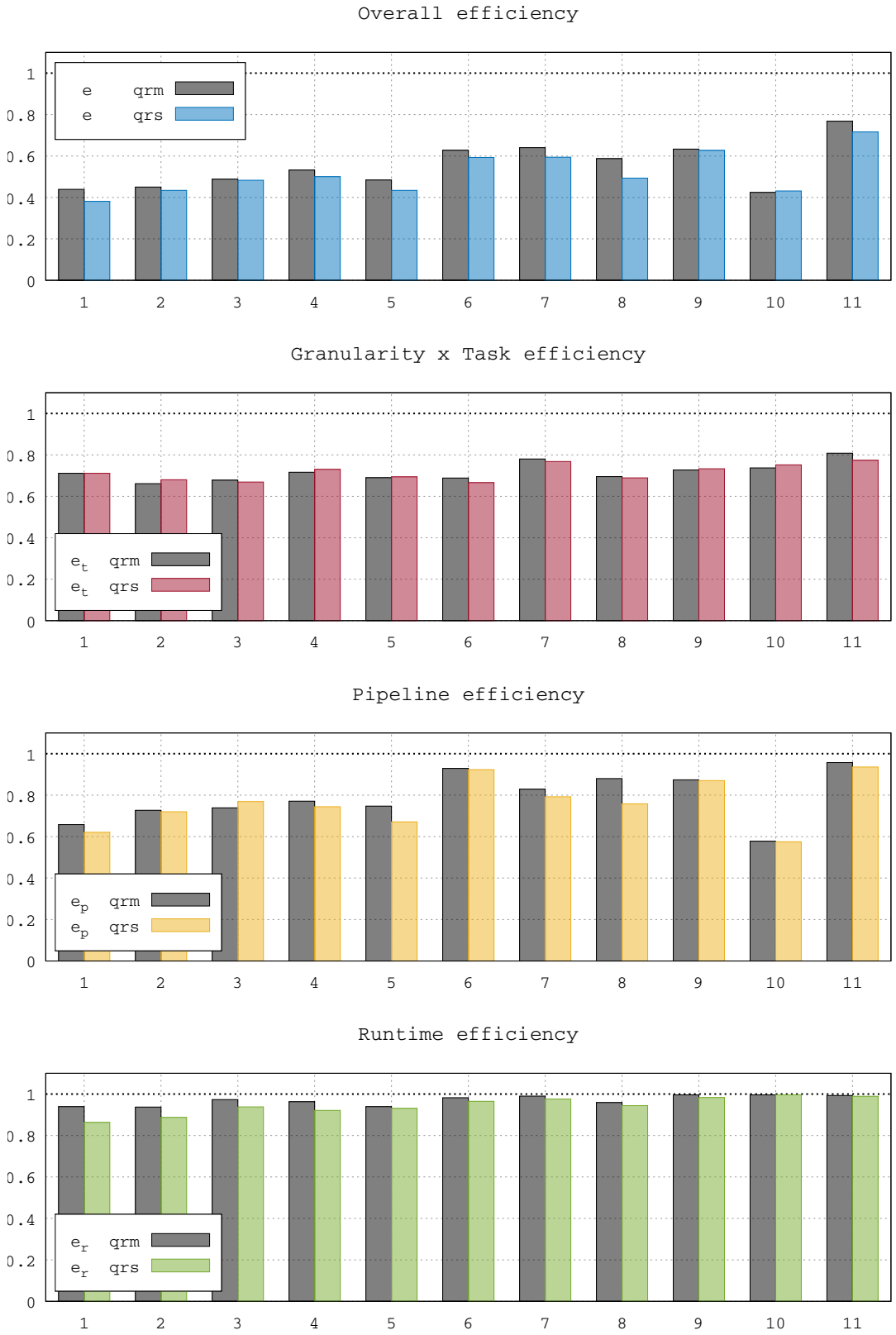


Figure 3.6: Efficiency measures for `qr_mumps` (qrm) and `qrm_starpu` (qrs).

this section, as well as in Chapter 4, have been run with an interleaved memory allocation policy which basically make the locality aware scheduling policy implemented in `qr_mumps` worthless; for this reason the two codes are also expected to have the same task efficiency. As the results in Figure 3.6 show, the performance difference between the two codes has to be found in the pipeline and and runtime efficiencies. The slightly worse pipeline efficiency is due to the different choice of priorities for the tasks that was made in `qrm_starpu` in order to contain the memory consumption (more comments on this below) and to the fact that assembly operations are not fully parallelized as explained above. The lower runtime efficiency, instead, can partly be explained by the fact that in `qrm_starpu` all the ready tasks are stored in a single, sorted central queue. This clearly incurs a relatively high cost due the contention on the locks used to prevent threads from accessing this data structure concurrently and due to the sorting of tasks depending on their priority; in Section 4.4 we will present a novel scheduler that overcomes most of this issues and consistently delivers better performance. As a result we can conclude that the performance difference between `qrm_starpu` and `qr_mumps` is merely due to minor, technical issues; the results presented in Chapter 4 with a much more refined and optimized implementation confirm this intuition.

Memory consumption is an extremely critical point to address when designing a sparse, direct solver. As the building blocks for designing a scheduling strategy on top of StarPU differ (and are more advanced) than what is available in `qr_mumps` (which relies on an *ad hoc* lightweight scheduler) we could not reproduce exactly the same scheduling strategy. Therefore we decided to give higher priority to reducing the memory consumption in `qrm_starpu`. This cannot easily be achieved in `qr_mumps` because its native scheduler can only handle two levels of task priority; as a result, fronts are activated earlier in `qr_mumps`, almost consistently leading to a higher memory footprint as shown in Figure 3.7. The figure also shows that both `qrm_starpu` and `qr_mumps` achieve on average the same memory consumption as `spqr`. On three cases out of eleven `spqr` achieves a significantly lower memory footprint; in Chapter 5 we will show that it is possible to reliably control and reduce the memory consumption of `qrm_starpu` and still achieve extremely high performance (roughly the same as the unconstrained case).

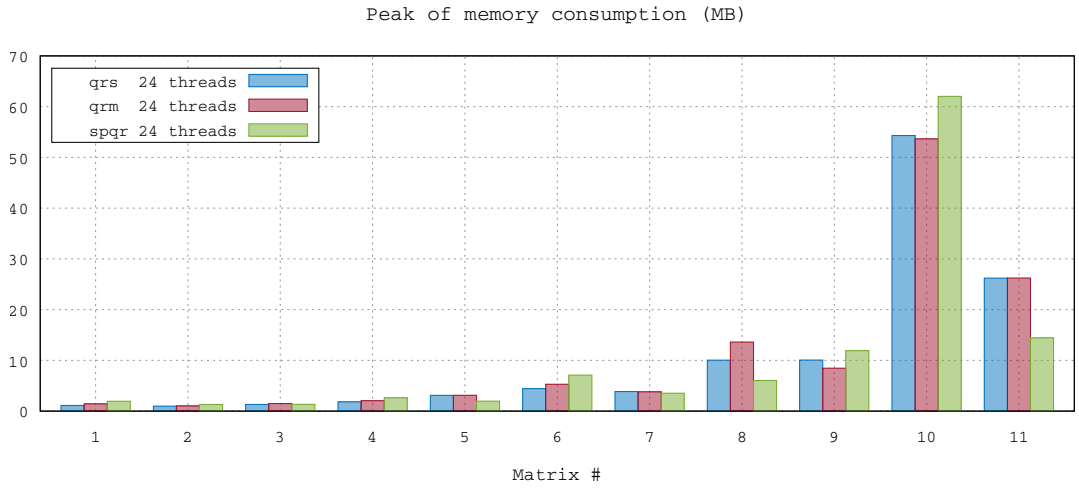


Figure 3.7: Memory peak for the factorization of the test matrices on 24 cores.

In the case where the whole DAG is submitted to the runtime system at the beginning of the execution, the memory needed for storing the task graph is proportional to the

problem size. Figure 3.8 shows the maximum number of tasks that the runtime system handles during the factorization versus the total number of tasks executed. The first being between 3 and 11 times smaller than the second, these data show that the technique proposed in Section 3.2 is effective in reducing the runtime system overhead and memory consumption.

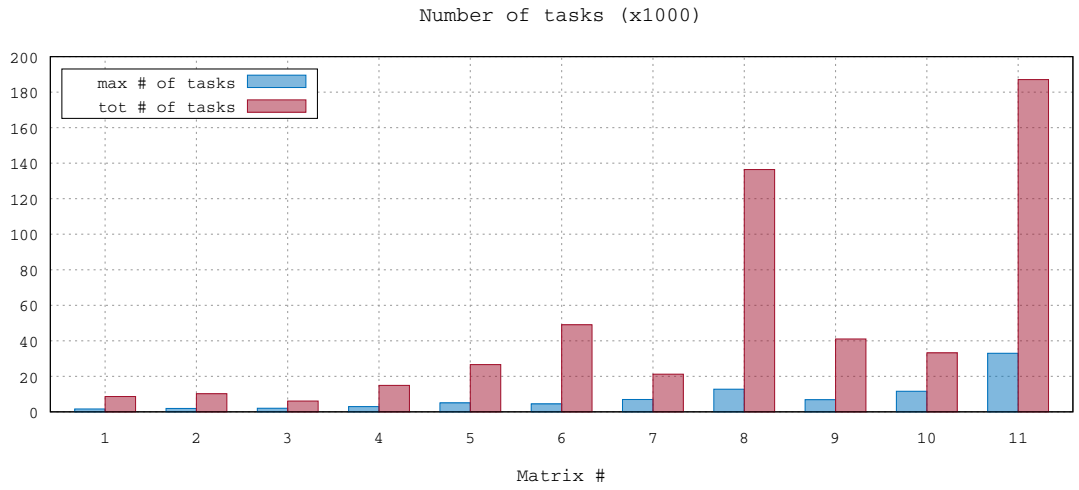


Figure 3.8: Maximum DAG size handled by StarPU during the factorization of the test matrices when using 24 threads.

Chapter 4

STF-parallel multifrontal QR method on multicore architecture

In the previous chapter we proposed a first approach for the implementation of the multifrontal QR method based on the StarPU runtime system which was mainly meant to asses the usability of a runtime system for implementing a sparse, direct method. For this reason we tried to reproduce as accurately as possible the behaviour of the original `qr_mumps` code. However, as explained at the end of Chapter 3, this implementation has a number of shortcomings, most importantly the complexity and difficult maintainability of the code. For this reason, in this chapter, we develop and evaluate experimentally a novel approach [S1] which implements the same algorithm (described in Section 1.7) in compliance with the Sequential Task Flow (STF) programming model presented in Section 1.4.1. We particularly emphasize the simplicity of the model for the parallelization of a complex algorithm such as the multifrontal factorization method and show the effectiveness of the approach.

We first illustrate the concept of the STF-parallelization of a sequential code with the implementation of a multifrontal method without dealing with the choice of a front partitioning scheme. Then, we propose a STF-parallel multifrontal method using a 1D block-column partitioning and show both the effectiveness of the approach and the limitations arising from the choice of partitioning. Finally, by leveraging the expressiveness of the programming model, we enhance the exploitation of the node-level parallelism by introducing dense 2D Communication Avoiding (CA) factorization algorithms of frontal matrices. We show a substantial performance gain using this strategy with a small impact on the runtime overhead. It should be noted that the use of such factorization schemes in the multifrontal method represents a big challenge as it makes the DAG much bigger with a more complex task dependency pattern. As previously discussed, we show that not only the model facilitates the implementation of such features but also incurs a negligible overhead in the runtime system.

4.1 STF-parallel multifrontal QR method

4.1.1 STF-parallelization

In this section we present the STF parallelization of the multifrontal method without using any specific frontal matrix partitioning. Figure 4.1 shows the main sequential pseudocode of the factorization using the four basic kernels namely `activate`, `assemble`, `deactivate` and `factorize`. This algorithms consists of a topological traversal of the elimination tree

where the following operations are done at each node: First the front is allocated in memory, then it is assembled with respect to its child nodes. As soon as this assembly is done the child nodes contribution blocks are deallocated. Finally the current front is factorized. The `factorize` routine in this pseudocode can simply replaced by a `_geqrt` routine if no front partitioning is applied. Applying a front partitioning consists in modifying `factorize` and `assemble` routines according to the chosen strategy.

```
forall fronts f in topological order
   ! allocate and initialize front
   call activate(f)

   ! front assembly
   forall children c of f
      call assemble(c, f)
      ! Deactivate child
      call deactivate(c)
   end do

   ! front factorization
   call factorize(f)
end do
```

Figure 4.1: Pseudo-code for the sequential multifrontal QR factorization.

Figure 4.2 shows the STF-parallel code for the multifrontal method corresponding to the sequential one in Figure 4.1. This code is obtained by replacing the routine calls with the submission of the corresponding tasks to the runtime system. The data referred to as `f` and `c` in the STF pseudo code correspond to symbolic representations of frontal matrices structures. These symbolic representations, called *handles* in StarPU terminology (see Section 1.4.2.1), must be registered before being used, either before the main factorization loop or in `activate` task. The memory allocation of actual data represented by a *handle* may be delegated to the runtime system via the dedicated memory allocation routine `starpu_malloc`. As explained in Section 1.4.2.1, once the data is registered in the runtime system, it may be moved to different memory locations depending on scheduling decisions, and the memory consistency across the architecture is ensured by the runtime system.

As in the previous case, in order to express task dependencies in the DAG presented in Section 1.7 it is necessary to specify which data each task uses and how it uses it, whether for read (`R`), write (`W`) or update (i.e., read and write `RW`). As already mentioned, prompted by the evaluation we conducted in the work presented in the previous chapter, the StarPU developers implemented a novel feature which allows the programmer to specify that two tasks can *commute*, i.e., can be executed in an order which is different from the submission order. This feature is useful for improving the concurrency of our implementation as it allows the runtime system to execute all the assembly operations related to a front in any order and not necessarily in a sequential fashion according to the submission order, as would otherwise happen due to the false detection of a data hazard on the assembled front. This feature can be used specifying the new `STARPU_COMMUTE` (for short, `C`, in our pseudocode) data access mode on the assembled front `f`. It must be noted that, this does not allow the concurrent execution of assembly operations. Nonetheless, we believe this is not too much of a penalty since these operations are extremely lightweight and can be overlapped with other tasks; additionally, it may even be detrimental to execute multiple assembly operations at the same time as they are particularly prone to false sharing issues.

```
forall fronts f in topological order
   ! allocate and initialize front
   call submit(activate, f:RW, children(f):R)

   ! front assembly
   forall children c of f
      call submit(assemble, c:R, f:RW|C)

      call submit(deactivate, c:RW)
   end do

   ! front factorization
   call submit(factorize, f:RW)
end do

call wait_tasks_completion()
```

Figure 4.2: Pseudo-code for the STF-parallel multifrontal QR factorization

If we use the notation presented in Figure 1.18 to identify the dependencies in the DAG we have:

- The dependency `d1` is guaranteed by submitting the `activate` task first when processing the front and taking as input the front data structure represented by `f` in a `RW` mode. Subsequent tasks working on this front take `f` in input with either a `R`, `W` or `RW`, and thus are not going to be executed until activation is done;
- Dependency `d5` is expressed by submitting assembly tasks after the submission of factorization tasks. This is the case because nodes are visited following a topological order in the elimination tree. The assembly operation consists in copying elements from children contribution blocks represented by `c` to the current node denoted by `f`. Therefore `c` is accessed in `R` mode and `f` in `RW` mode;
- The `factorize` task accesses `f` in a `RW` mode ensuring that the front is factorized only when assembly is done which expresses the dependency `d6`;
- To compute the structure of the current node, the `activate` task needs the structure of child fronts and thus takes as input their data structure in `R` mode. This ensures dependency `d7` because a front may be activated only when all child nodes have been activated.

Other dependencies namely `d1`, `d2` and `d3` in Figure 1.18 are specific to the 1D partitioning and may be modified depending on the choice of data partitioning for frontal matrices. The choice of data partitioning implies a modification to the assembly operations line 7 and factorization code line 13 in the pseudocode of Figure 4.2. The use of a 1D partitioning in this STF code is illustrated in Section 4.2 and is enhanced with the use of 2D partitioning in Section 4.3.

Note that the pseudocode of Figure 4.2 is exactly equivalent to the approach proposed by Davis [42] and Amestoy et al. [14]: tree parallelism is explicitly handled through the use of a tasking system whereas node parallelism can be exploited through the use of multithreaded BLAS routines within the `factorize` routine.

We would like to emphasize that the pseudocode in Figure 4.2 can be translated into fully functional code using the API of any other runtime system that complies with the STF programming model such as QUARK [117] and OpenMP [24]. As an example, the OpenMP-based parallel version of our algorithm is given in Figure 4.3. The tasks are instantiated using the OpenMP`TASK` construct and dependencies are declared thanks to the `DEPEND` clause used to declare data accesses. It must be noted that with OpenMP it is not necessary to declare the data to the runtime, as is done in StarPU through the use of handles; this is because OpenMP does not handle the data (e.g., it does not move it between different memory nodes nor does it handle the coherency between multiple copies) but only uses it to infer dependencies between tasks.

```
!$omp parallel

!$omp master
forall fronts f in topological order
   ! allocate and initialize front
   !$omp task depend(inout:f) depend(in:children(f))
   call activate(f)
   !$omp end task

   ! front assembly
   forall children c of f
      !$omp task depend(out:f) depend(in:c)
      call assemble(c, f)
      !$omp end task
      ! Deactivate child
      !$omp task depend(inout:c)
      call deactivate(c)
      !$omp end task
   end do

   ! front factorization
   !$omp task depend(inout:f)
   call factorize(f)
   !$omp end task
end do

!$omp end master

!$omp end parallel
```

Figure 4.3: Pseudo-code for STF-parallel multifrontal QR factorization implemented with OpenMP.

Despite the wide availability of the OpenMP technology, as explained in Sections 1.4.2 and 1.4.2.1 we have chosen to rely of the StarPU runtime system because of its large panel of features, most importantly, the possibility of controlling the scheduling, the support for accelerators and the transparent handling of data.

4.2 STF multifrontal QR method 1D

4.2.1 STF parallelization with block-column partitioning

In this section we extend the approach presented in the previous section by integrating a 1D partitioning of frontal matrices. The sequential factorization, when using this partitioning scheme, is given in Figure 4.4. This code is obtained by replacing the `assemble` and `factorize` routines in the original sequential code of Figure 4.1 respectively by the block-column based assembly operation and factorization of a node.

```
forall fronts f in topological order
   ! allocate and initialize front
   call activate(f)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call assemble(c(j), f)
      end do
      ! Deactivate child
      call deactivate(c)
   end do

   forall panels p=1...n in f
      ! panel reduction of column p
      call _geqrt(f(p))
      forall blockcolumns u=p+1...n in f
         ! update of column u with panel p
         call _gemqrt(f(p), f(u))
      end do
   end do
end do
```

Figure 4.4: Pseudo-code for the sequential multifrontal QR factorization with 1D partitioned frontal matrices.

Similarly to the previous STF algorithm, the STF parallel code, presented in Figure 4.5, is obtained by replacing the routine calls in the sequential code with the submission of corresponding tasks. In this case, however, special attention must be given to the fact that the submission of the assembly and factorization tasks of a front can be done only after its structure is known (most importantly, for the each assembly task we must know which block-columns of the parent front it touches) and thus, in the pseudocode of Figure 4.4, upon completion of the `activate` routine. This prevents us from computing the structure of the front in a task because the submission of the subsequent tasks would be suspended until the actual execution of the activation task is achieved. For this reason the `activate` routine was redefined into a routine which only computes the structure of the front and registers the block-column handles to the runtime system; this routine is executed synchronously by the master thread which allows for continuing the task submission without interruption. The most time consuming operations of the old `activate` routine, i.e., the data allocation and initialization and the assembly of the sparse matrix coefficients, are moved into a new routine called `init` which can be executed within a specific task. Note that the `activate` task is very lightweight and does not induce any relevant delay in the submission of tasks. Alternatively, the structure of all the fronts could be pre-computed

during the analysis phase and stored but, as explained, this would result in excessive memory consumption.

```
forall fronts f in topological order
   ! compute structure and register handles
   call activate(f)

   ! allocate and initialize front
   call submit(init, f:RW, children(f):R)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call submit(assemble, c(j):R, f:RW|C)
      end do
      ! Deactivate child
      call submit(deactivate, c:RW)
   end do

   forall panels p=1...n in f
      ! panel reduction of column p
      call submit(_geqrt, f(p):RW)
      forall blockcolumns u=p+1...n in f
         ! update of column u with panel p
         call submit(_gemqrt, f(p):R, f(u):RW)
      end do
   end do
end do

call wait_tasks_completion()
```

Figure 4.5: Pseudo-code for the STF-parallel multifrontal QR factorization with 1D partitioned frontal matrices.

Using the data access modes and the order of task submission, the runtime system automatically infers dependencies between tasks and thus builds the DAG, specifically:

- the activation of a node `f` depends on the activation of its children;
- all the other tasks related to a node `f` depend on the node activation;
- the `assembly` of a block-column `j` of a front `c` into its parent `f` depends on all the `_geqrt` and `_gemqrt` tasks on `c(j)` and on the activation of `f`;
- the `_geqrt` task on a block-column `p` depends on all the assembly and all the previous update tasks concerning `p`;
- the `_gemqrt` task on a block-column `u` with respect to `_geqrt p` depends on all the `assembly` and all the previous `_gemqrt` tasks concerning `u` and the related `_geqrt` task on block-column `p`;
- the deactivation of a front `c` can only be executed once all the related `_geqrt`, `_gemqrt` and `assemble` tasks are completed.

This STF compliant implementation has a number of obvious advantages over that presented in the previous chapter. First of all, in this code all the dependencies between tasks are automatically inferred by the runtime system through data analysis, whereas in the previous implementation they were either automatically inferred or explicitly defined through different mechanisms (see Section 3.2). This means that, here, the dataflow is complete, i.e., the runtime system knows exactly which data is needed by each task and can take the necessary actions to transfer the data where the task is actually being executed. This property is necessary to achieve the porting of our solver on GPU-equipped architectures, as described in Chapter 6. Second, the master thread is the only one in charge of submitting the tasks, and it does it in exactly the same order as in a sequential execution whereas, in the previous code, the submission of some tasks was done, asynchronously, within other tasks. This property give us full control over the submission of tasks which allows us to develop the efficient technique described in Chapter 5 for controlling the memory consumption. Last but not least, this code is much simpler, easier to read, to develop and maintain because, in essence, it is a sequential code that runs in parallel; nevertheless, this code is extremely efficient, as the experimental results in the next section show. Thanks to the simplicity of this programming model and to the modularity of this code that we could effectively improve its performance and scalability by implementing more complex parallelization schemes with a relatively contained programming effort, as we will show in Section 4.3.1.

A minor, but profitable improvement over the original `qr_mumps` solver and the one described in Chapter 3, is the use of a blocked storage format. In the previous versions the frontal matrices are allocated as a whole memory area and therefore the partitioning is logical. In this implementation, instead, each block column is allocated individually; although this does not bring any improvement to the performance (because Fortran uses column-major storage), it saves some memory due to the staircase structure of the fronts, as shown in Figure 4.10.

4.2.2 Experimental results

The implementation presented above was tested and evaluated on a subset of the test matrices in Table 1.1; the experiments were done on one node of the Ada supercomputer presented in Section 1.9.1. The purpose of the present experimental study is to assess the usability and efficiency of runtime systems using a STF parallel programming model for complex, irregular workloads such as the QR factorization of a sparse matrix. The reference sequential execution times are obtained with a purely sequential code (no potential runtime overhead) with no frontal matrix partitioning which ensures that all the LAPACK and BLAS routines execute at the maximum possible speed (no granularity trade-off).

The performance of the parallel 1D factorization depends on the choice of the values for a number of different parameters. These are the block-column `nb` on which the amount of concurrency depends and the internal block size `ib` on which the efficiency of elementary BLAS operations and the global amount of flop depend. As explained in Section 1.7.3, this parameter defines how well the staircase structure of each front is exploited. The choice of these values depends on a number of factors, such as the number of working threads, the size and structure of the matrix, the shape of the elimination tree and of frontal matrices and the features of the underlying architecture. It has to be noted that these parameters may be set to different values for each frontal matrix; moreover, it would be possible to let the software automatically choose values for these parameters. Both these tasks are very difficult and challenging and are out of the scope of this study. Therefore, for our experiments we performed a large number of runs with varying values for all these

parameters, using the same values for all the fronts in the elimination tree, and selected the best results (shortest running time) among those. For the sequential runs internal block sizes `ib={32, 40, 64, 80, 128}` were used for a total of five runs per matrix. For the 1D parallel STF case, the used values were `(nb,ib)={(128,32), (128,64), (128,128), (160,40), (160,80)}` for a total of five runs per matrix.

All of the results presented in this section were produced without storing the factors in order to extend the tests to the largest matrices in our experimental set that could not otherwise be factorized (even in sequential) on the target platform. This was achieved by simply deallocating the block-columns containing the factor coefficients at each `deactivate` task rather than keeping them in memory. As confirmed by experiments that we do not report here for the sake of space and readability, this does not have a relevant impact on the following performance analysis.

	Sequential reference			Parallel 1D STF			
Mat.	ib	Time (s.)	Gflop/s	nb	ib	Time (s.)	Gflop/s
12	40	1.00E+02	14.4	128	64	5.337E+00	272.4
13	32	1.73E+02	17.0	128	128	9.809E+00	312.0
14	80	3.40E+02	17.1	128	128	1.922E+01	309.8
15	128	5.76E+02	19.0	128	128	3.116E+01	352.0
16	80	8.71E+02	19.2	128	128	4.646E+01	362.0
17	80	1.18E+03	17.7	128	32	4.945E+01	407.8
18	128	1.58E+03	19.3	128	128	8.383E+01	365.9
19	128	3.25E+03	19.5	128	128	1.501E+02	422.4
20	128	3.99E+03	16.7	128	64	6.432E+02	102.7
21	128	9.93E+03	19.7	128	128	4.402E+02	446.8
22	128	1.13E+04	19.7	128	128	5.207E+02	430.6
23	128	1.37E+04	19.2	128	128	6.233E+02	422.7

Table 4.1: Sequential reference execution time and optimum performance for the STF 1D factorization on Ada (32 cores).

Table 4.1 shows for the STF 1D algorithm the parameter values delivering the shortest execution time along with the corresponding attained factorization time and Gflop rate. The Gflop rates reported in the table are related to the operation count achieved with the internal block size `ib`. The smaller block-column size of 128 always delivers better performance because it offers a better compromise between concurrency and efficiency of BLAS operations, whereas a large internal block size is more desirable because it leads to better BLAS speed despite a worse exploitation of the fronts staircase structure.

Figure 4.6, generated with the timing data in Table 4.1, shows the speedup achieved by the 1D parallel code with respect to the sequential one when using all the 32 cores available on the system. This figure shows that the speedup increases with the problem size and may be extremely low on some problems such as for matrix #20 whose speedup is less than 7. As we will show below, through a detailed analysis of performance results, the 1D partitioning limits the concurrency in the multifrontal factorization especially in the case of over-determined frontal matrices which is the common case for our test problems.

Using the performance analysis approach presented in Chapter 2, we evaluated the performance of our implementation by means of efficiency measures. Here, $\tilde{t}_t(1)$ and $t_t(1)$ in Equation (2.1), were computed, respectively, by timing the purely sequential code (with no block-column partitioning) and the 1D code in a sequential fashion. All other timings,

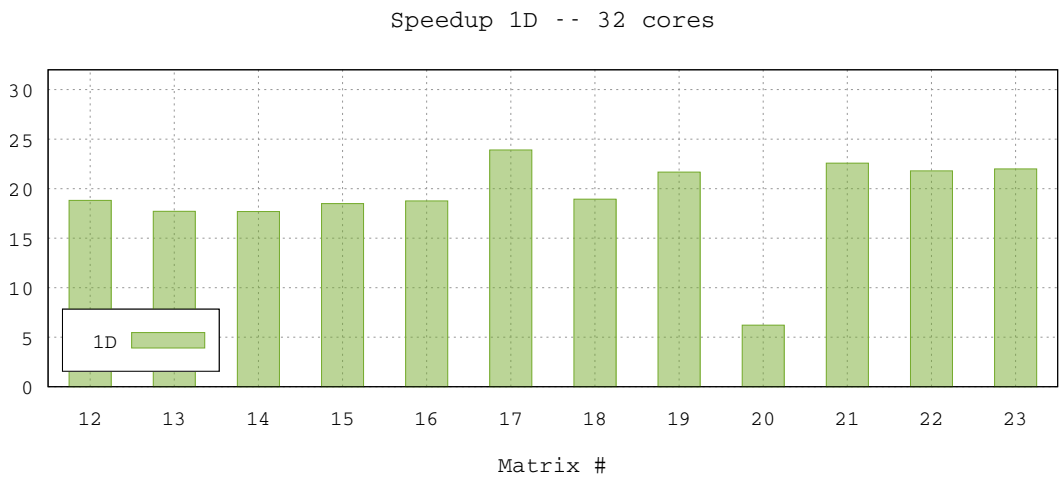


Figure 4.6: Speedup of the STF 1D algorithm with respect to the sequential case on Ada (32 cores).

i.e. $t_r(p)$, $t_c(p)$ and $t_i(p)$ are returned by StarPU upon execution of the parallel code with 1D block-column partitioning on $p = 32$ cores. All these measurements are related to the parameter settings in Table 4.1.

The 1D partitioning provides a coarse granularity of tasks resulting in a good granularity efficiency with a value greater than 0.8 on the tested problems. However it suffers from poor cache behaviour in the case of extremely overdetermined frontal matrices because of the tall-and-skinny shape of block-columns. This phenomenon is accentuated when the problem grows which explains the decrease of the granularity efficiency when the matrix size increases. As shown in Figure 4.7, the lowest granularity efficiency is obtained on the matrix #20 which represents an extreme case for the impact of overdetermined matrices because most of the flops are done in one front with roughly 1.3 M rows and only 7 K columns. The performance loss induced by the runtime is extremely small in our case with an overhead lower than 2% on average showing the efficiency of our scheduling strategy.

The efficiency results in Figure 4.7 show that the most limiting factors for the scalability of the factorization are the impact of data locality and the task pipelining. As explained for the efficiency of granularity, data locality may not be efficiently exploited because of the 1D partitioning which does not allows for a good exploitation of low-level memory resulting in performance loss during parallel execution due to memory contention and inter-level communications. The relatively poor pipeline efficiency obtained on smaller matrices is due to the limited amount of concurrency offered by the 1D partitioning especially for overdetermined matrices. This lack of parallelism is compensated by the tree-level parallelism and is thus less evident on large problems. As expected the matrix #20 gives extremely low pipeline efficiency resulting from the low concurrency level provided by the 1D partitioning on a tall-and-skinny matrix.

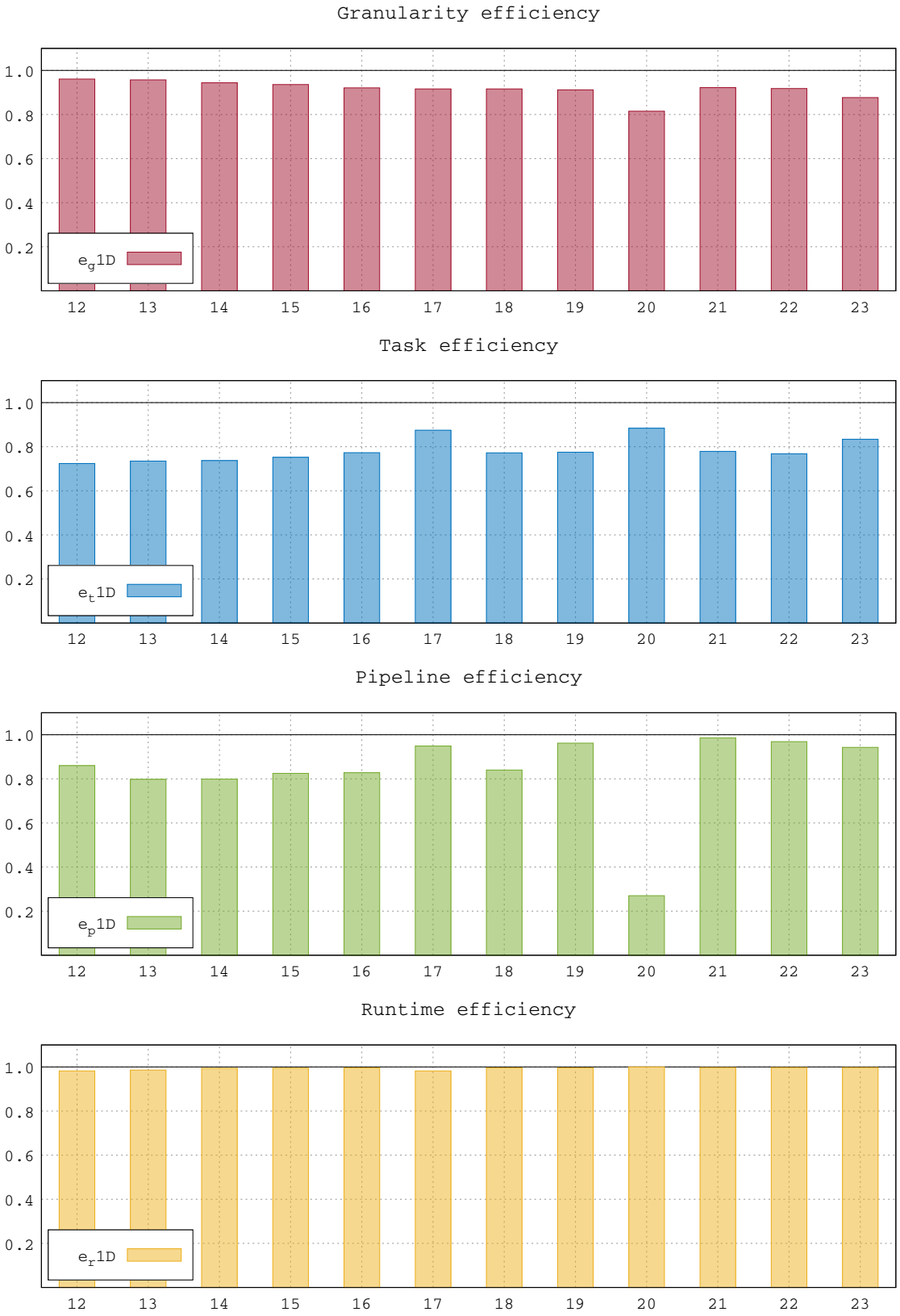


Figure 4.7: Efficiency measures for the STF 1D algorithm on Ada (32 cores).

4.2.3 The effect of inter-level parallelism

As explained in Section 1.7, `qr_mumps` and `qrm_starpu` can take advantage of what we called *inter-level parallelism* which allows for working on a block-column of a front as soon as it becomes fully assembled regardless of the status of the other block-columns within the same front or within child fronts. As a result, the factorization of a front can be pipelined with those of the child nodes for an extra source of parallelism. This technique is rather complex to implement as it requires tracking the assembly status of each column individually; in `qrm_starpu` this is implicitly and very efficiently done by the runtime system through the tracking of dependencies between tasks.

Although the elimination tree can be traversed in any topological order, a postorder is commonly chosen because it has favorable properties in terms of memory consumption. In a parallel execution the traversal deviates from this postorder in order to use tree parallelism and achieve better concurrency but, if possible, it is better to stay as close as possible to the sequential postorder in order to avoid excessive memory consumption. This can be achieved in a relatively easy way in a share memory context using techniques like the one presented in Section 3.2.2 but it is much more complex in a distributed memory parallel setting (see, for example, the work by Agullo et al. [2]). As a result, in a shared memory parallel multifrontal solver, the factorization can be imagined as a wavefront which starts from one corner of the elimination tree, say, the bottom left corner, and sweeps the entire tree moving towards the opposite corner, the top right one. When this wavefront reaches the last, rightmost branch, all the fronts therein are treated sequentially, one after the other because no tree parallelism is available anymore. This lack of concurrency can incur a severe penalty if the inter-level parallelism is not used because quite a considerable fraction of the total volume of operations is done on these last fronts. Figure 4.8 shows, on the top part, the execution trace for the factorization of matrix #5 on the Dude machine when inter-level parallelism is not implemented. This trace was produced by adding "fake" dependencies to the `qrm_starpu` solver in order to reproduce the behaviour shown in Figure 1.16 (*right*). A different color has been used for all the tasks related to each front. It is seen visible how the lack of tree parallelism on the last few fronts introduces a heavy penalty stall in the execution traces. On the contrary, when inter-level parallelism is implemented, the execution trace (see the bottom trace in Figure 4.8) is relatively densely populated until the end of the execution leading to a gain in time of around 30%.

Because the runtime controls all the components of the parallel execution of a code, from the execution of tasks to the handling of their mutual dependencies, it can provide accurate measurement and data that allow for detailed performance analysis. Specifically, the StarPU runtime system can produce the DAG that includes all the tasks along with their dependencies where the tasks can be weighted with either their execution time or other useful information provided by the programmer. Using these data, we conducted a *critical-path analysis* for the factorization of our test matrices using the STF parallel code. Assuming that the critical path (the longest path in the DAG) is a lower bound on the execution time, the maximum achievable speedup or average degree of concurrency can be defined as the ratio between the sum of the weights of all the tasks along the critical path (i.e., length of the critical path) and the sum of the weights in the whole DAG:

$$\text{max\_speedup} = \text{avg\_concurrency} = \frac{\sum_{i \in DAG} w_i}{\sum_{i \in CP} w_i}$$

where w_i is the weight of task i. For each matrix, the DAG used to conduct this analysis is the one related to the case where 32 working threads are used (remember that

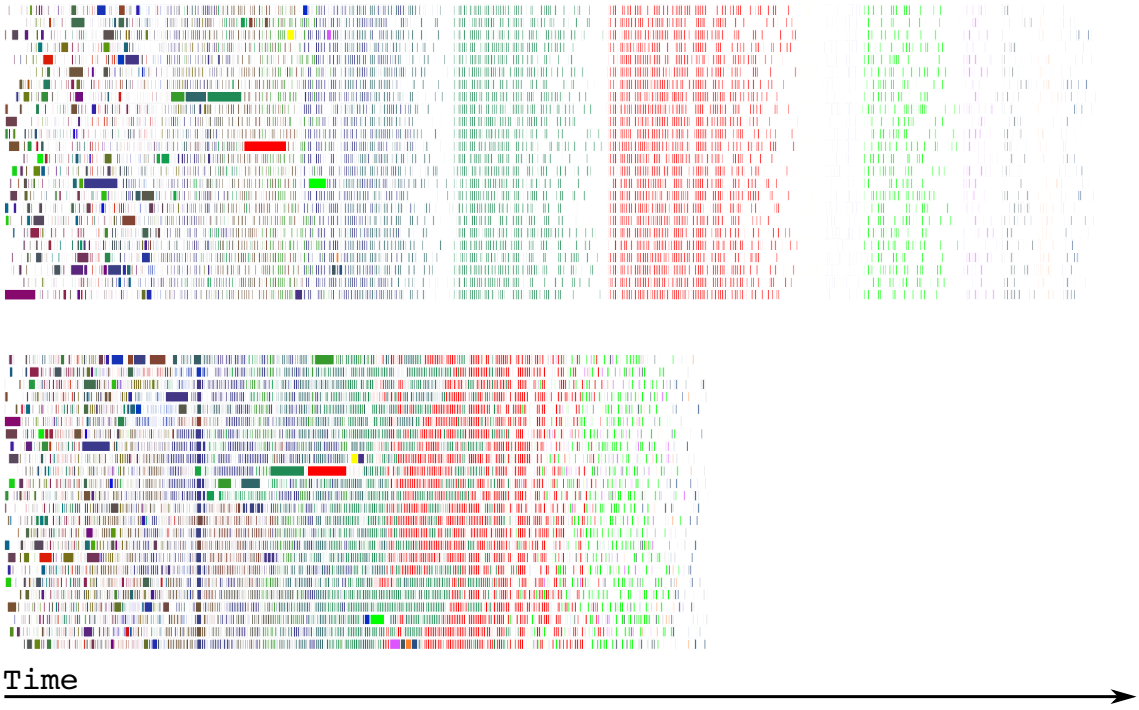


Figure 4.8: Execution traces for matrix #5 on the Dude architectures using 24 cores without (*top*) and with (*bottom*) inter-level parallelism.

the DAG changes with the number of threads because of a different choice of the tree-pruning level as described in Section 1.7.2). The weight of tasks is chosen to be equal to the execution time measured in an execution with only one working thread. This precise choice was made because we believe that granularity plays an important role in the critical path analysis and thus the execution time is a more realistic measure than, for example, the number of flops (or the amount of data movement for memory-bound tasks); at the same time, the single-threaded execution was chosen to exclude from the analysis the effects of data locality (the reader can refer to Figure 4.7 for a measure of those). The critical path in the DAG is computed through a Depth-First Search. Figure 4.9 shows the maximum achievable speedup for the factorization of the test matrices using the method presented above, with or without inter-level parallelism. The figure shows that the inter-level parallelism is beneficial (considerably, for some problems such as #19 or #22) in all cases except for matrices #17 and #20. On these two problems the inter-level parallelism does not bring any benefit for different reasons. In matrix #17 all the largest fronts, except the root, are largely underdetermined which means that concurrency is abundant within each front and thus the lack of pipelining between fronts along the last branch is not penalizing. In matrix #20, instead, most of the flops are performed in a single, huge front and thus the benefit of inter-level parallelism is only marginal.

4.3 STF multifrontal QR method 2D

With a 1D partitioning of frontal matrices, the node parallelism is achieved because all the updates related to a panel can be executed concurrently and because panel operations can be executed at the same time as updates related to previous panels (this technique is well

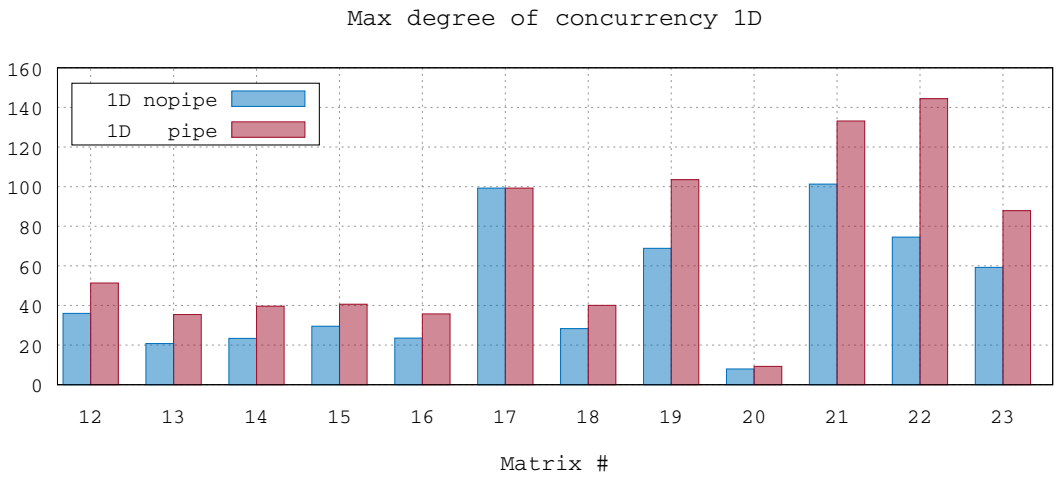


Figure 4.9: Maximum degree of concurrency on Ada (32 cores) emerging from the DAG with and without inter-level parallelism.

known under the name of *lookahead*). It is clear that when frontal matrices are strongly overdetermined (i.e., they have many more rows than columns, which is the most common case in the multifrontal QR method) this approach does not provide much concurrency. In the multifrontal method this problem is mitigated by the fact that multiple frontal matrices are factorized at the same time. However, considering that in the multifrontal factorization most of the computational weight is related to the topmost nodes where tree parallelism is scarce, a 1D front factorization approach can still seriously limit the scalability as shown in the previous section.

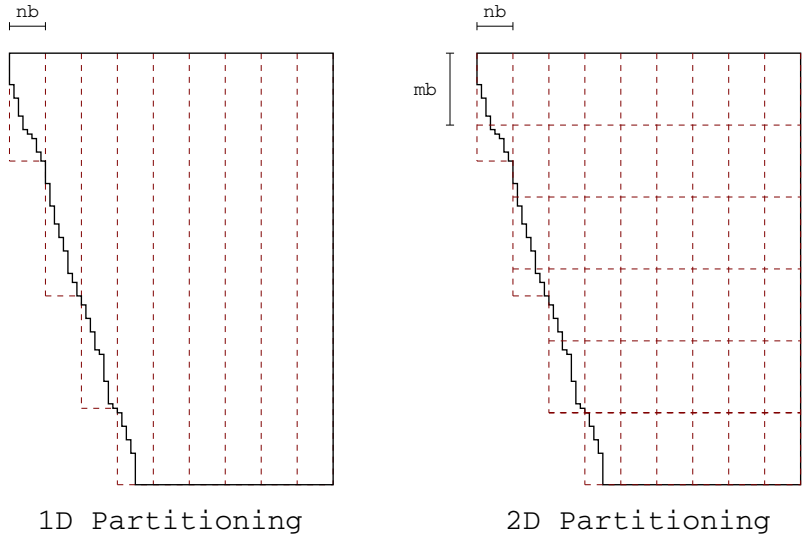


Figure 4.10: 1D partitioning of a frontal matrix into block-columns (*left*), 2D partitioning into tiles (*right*) with blocked storage.

This severe limitation can be overcome by employing communication-avoiding algorithms for the factorization of frontal matrices. These methods, discussed in Section 1.3.1,

are based on a 2D decomposition of the fronts into blocks of size `mb x nb` as shown in Figure 4.10 and allow for increasing the concurrency in both front and inter-level parallelism. We integrated the flat/binary hybrid approach (see Section 1.3.1 for the details) in our multifrontal STF parallel code. Lines 17-24 in Figure 4.5 were replaced by the pseudocode in Figure 4.11 which implements the described 2D factorization algorithm; note that this code ignores the tiles that lie entirely below the staircase structure of the front represented by the `stair` array.

```
1  do k=1, n
       ! for all the block-columns in the front
3      do i = k, m, bh
           call submit(_geqrt, f(k,i):RW)
5          do j=k+1, n
               call submit(_gemqrt, f(k,i):R, f(i,j):RW)
7          end do
           ! intra-subdomain flat-tree reduction
9          do l=i+1, min(i+bh-1,stair(k))
               call submit(_tpqrt, f(i,k):RW, f(l,k):RW)
11             do j=k+1, n
                   call submit(_tpmqrt, f(l,k):R, f(i,j):RW, f(l,j):RW)
13             end do
           end do
15     end do
       do while (bh.le.stair(k)-k+1)
17         ! inter-subdomains binary-tree reduction
           do i = k, stair(k)-bh, 2*bh
19             l = i+bh
               if(l.le.stair(k)) then
21                 call submit(_tpqrt, f(i,k):RW, f(l,k):RW)
                   do j=k+1, n
23                     call submit(_tpmqrt, f(l,k):R, f(i,j):RW, f(l,j):RW)
                   end do
25             end if
           end do
27         bh = bh*2
       end do
29 end do
```

Figure 4.11: Pseudo-code showing the implementation of the tiled QR.

The assembly operations have also been parallelized according to the 2D frontal matrix blocking: lines 9-11 in Figure 4.5 were replaced with a double, nested loop to span all the tiles lying in the contribution block: each assembly operation reads one tile of a node `c` and assembles its coefficients into a subset of the tiles of its parent `f`. As a consequence, some tiles of a front can be fully assembled and ready to be processed before others and before the child nodes are completely factorized. This finer granularity (with respect to the 1D approach presented in the previous section) leads to more concurrency since a better pipelining between a front and its children is now enabled.

The development of this version also included a number of other, minor improvements:

- In our implementation tiles do not have to be square but can be rectangular with more rows than columns. This is only a minor detail from an algorithmic point of

view but, as far as we know, it has never been discussed in the literature and, as described in Section 4.3.1, provides considerable performance benefits for our case;

- As in the 1D case presented in the previous section, block storage is also used in this case, as shown in Figure 4.10 (`right`). In addition to the memory savings, which are the same as in the 1D case, here the block storage also benefits the performance because of the lower leading dimension of the blocks;
- The `_tpqrt` and `_tpmqrt` LAPACK routines were modified in order to cope as efficiently as possible with the fronts staircase structure. This is done using the approach described in Section 1.7.3 for the `_geqrt` and `_gemqrt` routines.

This parallelization leads to very large DAGs with tasks that are very heterogeneous, both in nature and granularity; moreover, not only are intra-fronts task dependencies more complex because of the 2D front factorization, but also inter-fronts task dependencies due to the parallelization of the assembly operations. The use of an STF-based runtime system relieves the developer from the burden of explicitly representing the DAG and achieving the execution of the included tasks on a parallel machine.

4.3.1 Experimental results

In this section we evaluate the STF parallelization of the multifrontal QR method based on a 2D partitioning of frontal matrices and compare this approach with the original 1D partitioning presented in Section 4.2. Using the same settings presented in Section 4.2.2 we experiment with our code on a subset of the matrices presented in Table 1.1 on the Ada computer equipped with 32 cores.

	Parallel 2D STF					
Mat.	mb	nb	ib	bh	Time (s.)	Gflop/s
12	576	192	32	4	4.303E+00	321.6
13	480	160	40	8	7.217E+00	397.5
14	480	160	40	12	1.426E+01	399.6
15	480	160	40	20	2.427E+01	442.1
16	480	160	32	16	3.781E+01	435.4
17	640	160	40	4	4.784E+01	424.4
18	480	160	40	24	6.922E+01	439.0
19	480	160	32	∞	1.408E+02	441.9
20	576	192	32	24	1.728E+02	379.5
21	576	192	32	∞	4.286E+02	453.7
22	576	192	64	∞	4.807E+02	462.8
23	576	192	64	20	5.642E+02	462.6

Table 4.2: Optimum performance for the STF 2D factorization on Ada (32 cores).

Table 4.2 parameter values delivering the shortest execution time along with the corresponding attained factorization time and Gflop/s rate. As explained in Section 4.2.2 for the 1D case, the performance of the factorization depends on a combination of several parameters. For the parallel 2D STF case these parameters are, the size of the tiles `(mb,nb)`, the type of panel reduction algorithm set by the `bh` parameter described in Section 1.3.1 and the internal block size `ib`. Since the optimum values for parameters depends on a large number of factors it is extremely difficult to choose automatically the best values for

every given problems. Moreover we increased the complexity of this problem compared to the 1D case because the number of parameters is greater for 2D algorithms. Therefore, using the same experimental protocol as for the previous strategy we performed a large number of runs for each problem with varying values for all the input parameters, using the same values for all the fronts in the elimination tree, and selected the best results (shortest running time) among those. We tested the following values `(nb,ib)={(160,32), (160,40), (192, 32), (192,64)}`, `mb={nb, nb*2, nb*3, nb*4}` and `bh={4, 8, 12, 16, 20, 24, ∞}` for a total of 112 runs per matrix ($bh = \infty$ means that a flat reduction tree was used). Note that compared to the 1D case, concurrency is abundant when using the 2D partitioning. For this reason we choose bigger values for `nb` for two reasons: it increases the granularity of tasks in order to achieve a better BLAS efficiency and it limits the number of tasks in the DAG which contributes to keeping the runtime overhead small. The internal block size `ib`, however, has to be relatively small to keep the flop overhead (see Section 1.3.1) under control. Finally, as for the experiments in Section 4.2.2, factors were discarded during the factorization in order to test the the largest matrices from our problem set.

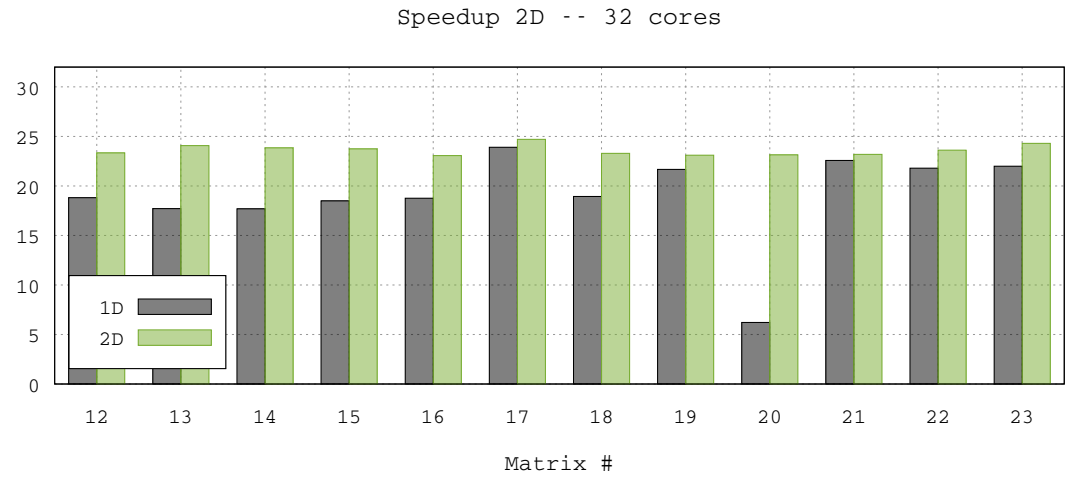


Figure 4.12: Speedup of the 1D and 2D algorithms with respect to the sequential case on Ada (32 cores).

The speedup achieved by the STF 2D implementation is shown in Figure 4.12 along with the speedup obtained with the STF 1D implementation previously presented in Section 4.2.2. Results show that the 2D algorithm provides better efficiency on all the tested matrices especially for the smaller ones and those where frontal matrices are extremely overdetermined, such as matrix #20, where the 1D method does not provide enough concurrency (as explained in Section 4.2.2, in this problem most of the operations are done on a frontal matrix which has over a million rows and only a few thousands columns). The average speedup achieved by the 2D code is 23.61 with a standard deviation of 0.53, reaching a maximum of 24.71 for matrix #17. For the 1D case, instead, the average is 19.04 with a standard deviation of 4.55. In conclusion, the 2D code achieves better and more consistent scalability over our set of matrices.

Figure 4.13 shows the efficiency analysis obtained with both the 1D and 2D algorithms. The 2D algorithms obviously have a lower task efficiency because of the smaller granularity of tasks and because of the extra flops. Note that the 1D code may suffer from a

poor cache behaviour in the case of extremely overdetermined frontal matrices because of the extremely tall-and-skinny shape of block-columns. This explains why the granularity efficiency for the 2D code on matrix #20 is better than for the 1D code unlike for the other matrices. 2D algorithms, however, achieve better locality efficiency than 1D most likely due to the 2D partitioning of frontal matrices into tiles which have a more cache-friendly shape and size than the extremely tall and skinny block-columns used in the 1D algorithm. Not surprisingly, the 2D code achieves much better scheduling efficiency (i.e., less idle time) than the 1D code on all matrices: this results from a much higher concurrency, which is the purpose of the 2D code. As for the runtime efficiency, it is in favor of the 1D implementation due to a much smaller number of tasks with bigger granularity and simpler dependencies. However the performance loss induced by the runtime is extremely small in both cases: less than 2% on average and never higher than 4% for the 2D implementation; this is also due to the efficiency of the scheduler specifically implemented and used for these experiments (see Section 4.4).

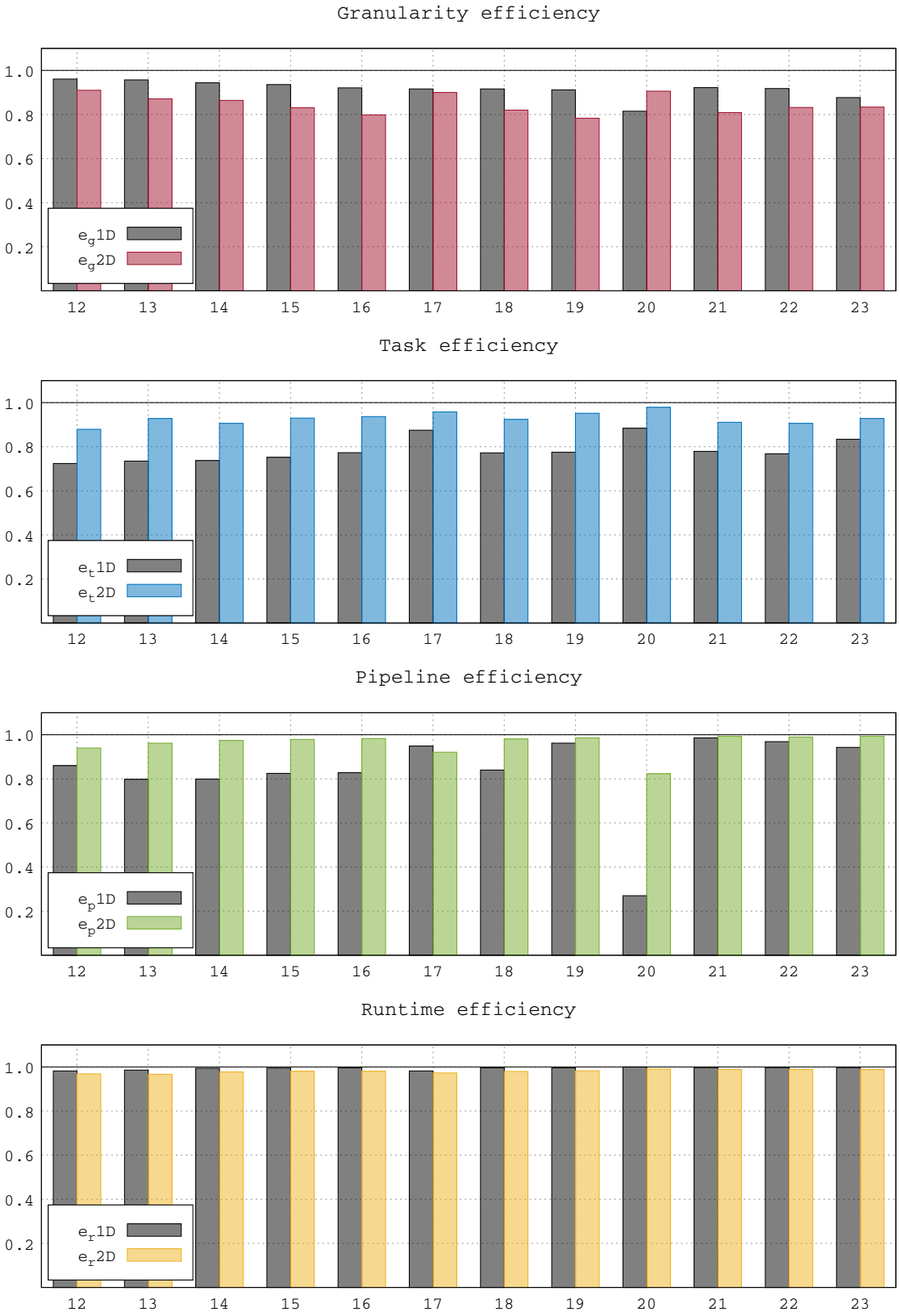


Figure 4.13: Efficiency measures for the STF 1D algorithm on Ada (32 cores).

4.3.2 Inter-level parallelism in 1D vs 2D algorithms

First, we computed the average degree of concurrency for the dense QR factorization using the 1D factorization and the 2D with a flat or binary panel reduction tree with the following settings

- tiles are square of size `nbxnb`;
- the weight of tasks is given by their flop count: in units of $nb^3/3$ this is equal to 4 for `_geqrt`, 6 for `_gemqrt`, 6 and 2 for `_geqrt` with, respectively, a square or triangular bottom tile, 12 and 6 for `_gemqrt` with, respectively, a square or triangular bottom tile (see Dongarra et al. [47]);
- matrices are of size $m * nb \times n * nb$ with $5 \leq m \leq 100$ and $5 \leq n \leq 50$.

These results are plotted in Figure 4.14: the leftmost plot shows the average degree of concurrency for the 1D algorithm, the center plot shows the benefit brought by the 2D algorithm with a flat tree over the 1D method and the rightmost shows the added benefit of the binary panel reduction tree over the flat. Note that the y-axis corresponds to the number of row tiles and the x-axis represents the number of column tiles in the dense matrices. A number of interesting conclusions may be drawn from this figure. First, the 1D algorithm provides poor concurrency and its scalability degrades as the number of rows increases with respect to the number of columns; in the best case the 1D algorithm reaches an ideal speedup of 22.22. The 2D algorithm with a flat panel reduction tree can clearly generate much more parallelism and reaches an ideal speedup of 604 although it suffers from poor scalability when matrices are extremely overdetermined like in the bottom-left corner of the middle plot. Finally, the 2D algorithm with binary panel reduction tree improves the scalability of the factorization but only when the matrix is extremely overdetermined otherwise it is either comparable to the flat tree algorithm or less efficient; this is due to a worse pipelining of successive panel stages as explained by Dongarra et al. [47].

Similarly to the 1D case, we computed the maximum degree of parallelism in the DAG produced by the algorithm. The results compared with the values presented in Section 4.2.3 for the 1D method are given in Figure 4.15.

These experimental results show that, as expected, the 2D front factorization algorithms combined with the assembly operations by blocks provide much higher concurrency than the 1D version and that the inter-level parallelism can push the available parallelism to even higher levels.

Figure 4.12 shows that the most significant gains are obtained, through the use of the 2D method, on matrices #12-#16, #18 and #20. Looking at Figure 4.15, it is possible to observe that, for these matrices, the 1D algorithm provides an average degree of concurrency which is barely higher that the number of available threads (or much smaller in the case of matrix #20); the use of the 2D method combined with the inter-level parallelism, instead, provides enough concurrency to feed all the working cores. This is confirmed by the scheduling efficiency measures reported in Figure 4.13. For the remaining matrices the 1D method already provides enough parallelism and thus the benefit of the 2D method is only marginal and mostly due to a better locality efficiency.

For different reasons, the results related to matrices #17 and #20 are odd, when compared to the others. As explained in Section 4.2.3, in problem #17 all the largest fronts, except the root, are underdetermined. In this case 2D algorithms do not bring any improvement, also because of a longer critical path due to a worse tasks efficiency. The speedup observed in Figure 4.12 is therefore essentially due to a better locality efficiency.

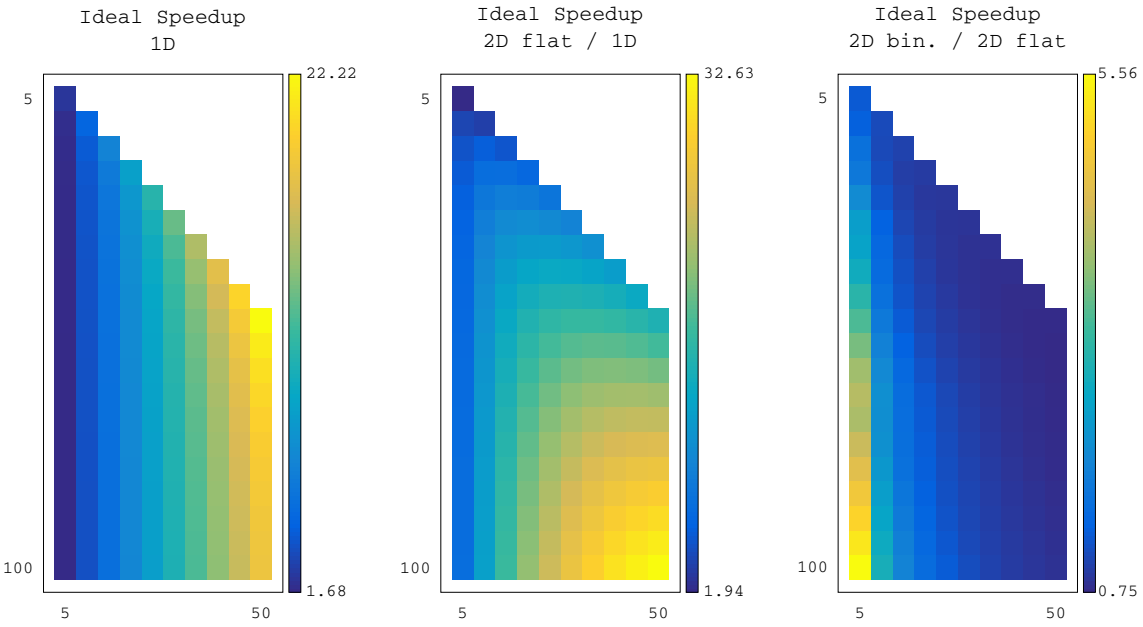


Figure 4.14:

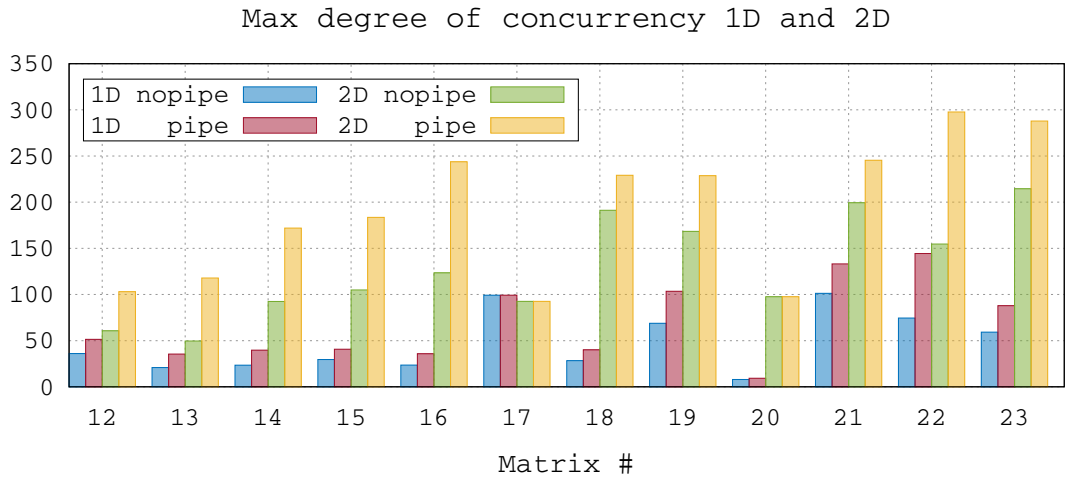


Figure 4.15: Maximum degree of concurrency on Ada (32 cores) coming from the DAG with and without inter-level parallelism for both 1D and 2D partitioning.

As for matrix #20, it has been already explained that the cost of the factorization is dominated by the factorization of a single extremely over-determined front: in this case the use of a 2D algorithm with a hybrid panel reduction tree (see Table 4.2) allows a considerable reduction in the length of the critical path.

4.4 Task scheduling with LWS

As explained in Section 1.4, the scheduler is one component of a runtime system and is in charge of handling the ready tasks and deciding where and when they have to be executed. Although in a heterogeneous memory context the scheduler does not have to take difficult decisions, its efficiency and the policy it implements play an important role in the performance and scalability of the code. In such a context, the scheduler has to

- be efficient: every time a task is completed the scheduler is used twice. First, to push all the tasks that have just become ready upon completion of the previous task and second to pop a task among all the ready ones in order to feed the worker that has just become idle. The relative weight of these operations becomes more and more heavy as the number of workers increases, as the number of tasks increases and as the average duration of tasks decreases. Therefore, the scheduler has to execute in an extremely efficient way in order not to slow down the working threads.
- handle priorities: it can be extremely beneficial for reducing the overall execution time to prioritize tasks according to a prescribed policy. For example, giving higher priority to tasks along the critical path can help reducing the makespan or giving higher priority to tasks that have higher fan-out may improve the total amount of available concurrency.
- improve data locality: modern multicore architectures are equipped with very deep memory hierarchies. Although data transfers among memory levels and among the memories associated with different cores (caches on NUMA modules) are implicit they can be quite expensive and be a major limiting factor for the scalability of a code. It is therefore important to schedule tasks in such a way that both spatial and temporal locality are improved.

The scheduler presented in Section 3.2.2 aimed mainly at providing the possibility of assigning arbitrary priorities to tasks. This scheduler, however, has evident flaws, especially from the efficiency point of view because it led to an excessive contention over the locks used to prevent simultaneous access to the single queue of tasks. Other predefined schedulers were available in the StarPU runtime system but each of them was weak in one of the points mentioned above. For this reason we developed a novel scheduler which aims at addressing all these issues. This scheduler is based on the use of multiple queues, namely, one per working thread. Upon termination of a task, a worker gathers from the runtime all the tasks that have moved into a ready state because of the completion of that task and pushes them to its own queue. Then, the worker tries to pop one ready task from its own queue. If it succeeds, it executes the tasks, otherwise a work-stealing mechanism is put in place where the worker thread tries to pick a ready task from the queues associated with other threads. The stealing is done following a precise order defined by the system architecture: a worker will first attempt to steal a task from workers that are closer in the memory hierarchy and then from those that are further away. Because the worker threads are bound to cores, this order can be established statically, for each worker, at the moment when the scheduler is instantiated using a tool such as `hwloc` (which is already internally used by StarPU). Within each queue, tasks are sorted according to an arbitrary priority value assigned by the user. Figure 4.16 depicts the structure of this scheduler which has been eventually integrated in StarPU under the name of `lws` (for Locality Work-Stealing) and is now available to the whole community of users.

The `lws` scheduler does not suffer from excessive contention because, as long as ready tasks are numerous and work stealing does not happen, each worker accesses its own queue

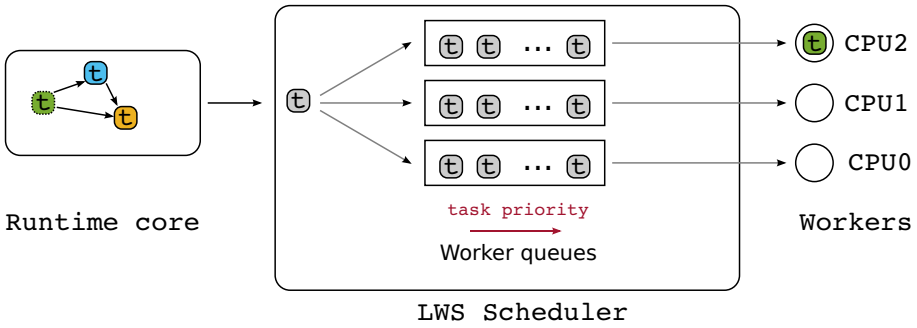


Figure 4.16: LWS (Locality Work Stealing) scheduler.

independently from the others. Data locality is better exploited because each ready task is pushed to the same queue where one of its predecessors was stored and because the work-stealing is guided by the system architecture. Finally the possibility of assigning an arbitrary priority to each task, can help making the tree traversal in a parallel execution closer to the one followed in a sequential execution.

The `lws` scheduler was compared to some of the predefined schedulers in StarPU (i.e., all those designed for shared-memory, homogeneous architectures) namely:

- `prio`: this scheduler allows assigning tasks a priority between -5 and 5 and uses one queue per priority value. Therefore it has limited support for handling priorities and may be subject to scaling problems due to the fixed and limited number of queues. It does not deal with data locality.

- `ws`: uses one queue per worker and implements a work-stealing mechanism. Contention is reduced due to the use of multiple queues and data locality is partially exploited because tasks are pushed to the same queue as their predecessors. The victims in the work-stealing method are chosen in a round-robin fashion. This scheduler has no support for priorities.

- `eager`: this is based on a single, shared pool of tasks with no support for priorities.

Figure 4.17 shows the execution times obtained using the three above schedulers, relative to the execution time obtained with `lws`. Although the gains are only marginal, the figure shows that `lws` is consistently better that the others. The benefit obtained is likely to increase with the number of cores or when the granularity of tasks decreases.

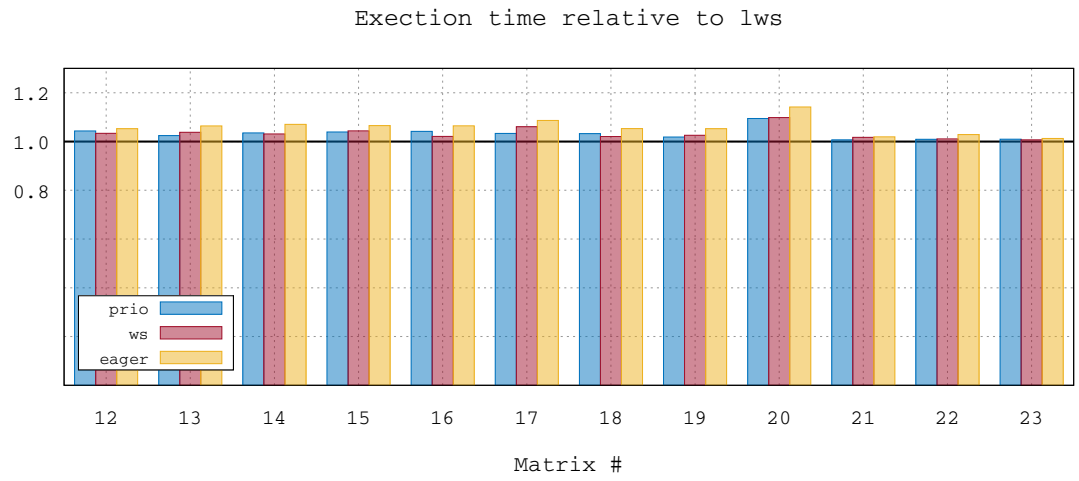


Figure 4.17: Factorization times obtained with pre-existing schedulers normalized to the one obtained with lws.

Chapter 5

Memory-aware multifrontal method

In this chapter we tackle the problem of controlling the memory consumption of the multifrontal QR method and aim at imposing a constraint on the maximum memory usage during the execution. The algorithm, based on the STF model and presented in the previous chapter may lead to a large memory consumption as all the frontal matrices could be allocated immediately after the beginning of the execution. The strategy employed to reduce the memory consumption consists in trying to keep the traversal of the elimination tree as close as possible to a postorder that minimizes the memory footprint in a sequential execution. This approach, however, does not provide any means of controlling or predicting, in the parallel factorization, the memory consumption which can grow arbitrarily. After a brief description of the memory behaviour for the multifrontal method, and reviewing the existing techniques for controlling its memory footprint, we propose a modification of the STF algorithm capable of reliably controlling the memory consumption [S1] of a parallel factorization. Although, in principle, this technique may reduce the amount of available concurrency by limiting the use of tree parallelism, the experimental results reported below show that the actual reduction of performance is basically negligible.

5.1 Memory behavior of the multifrontal method

The memory needed to perform the multifrontal factorization (QR as well as LU or other types) is not statically allocated at once before the factorization begins but is allocated and deallocated as the frontal matrices are activated and deactivated, as described in Section 1.2.3. Specifically, each activation task allocates all the memory needed to process a front; this memory can be split into two parts:

1. a persistent memory: once the frontal matrix is factorized, this part contains the factor coefficients and, therefore, once allocated it is never freed, unless in an out-of-core execution (where factors are written on disk)[1];

2. a temporary memory: this part contains the contribution block and is freed by the `deactivate` task once the coefficients it contains have been assembled into the parent front.

[1] In some other cases the factors can also be discarded as, for example, when the factorization is done for computing the determinant of the matrix.

As a result, the memory footprint of the multifrontal method in a sequential execution varies greatly throughout the factorization. Starting at zero, it grows fast at the beginning as the first fronts are activated, it then goes up and down as fronts are activated and deactivated until it reaches a maximum value (we refer to this value as the *sequential peak*) and eventually goes down towards the end of the factorization to the point where the only thing left in memory is the factors. The memory consumption varies depending on the particular topological order followed for traversing the elimination tree and techniques exist to determine the memory minimizing traversal for sequential executions as explained in Section 5.2.

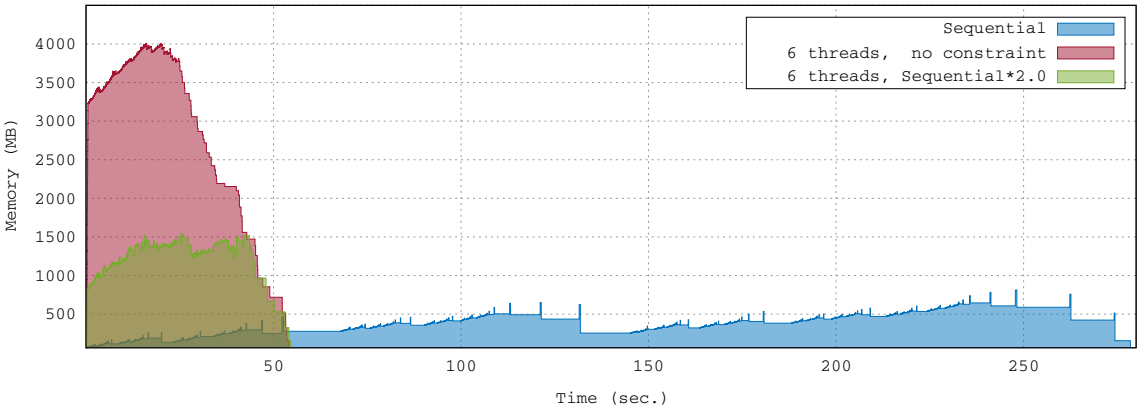


Figure 5.1: The memory profiles for matrix #12 for a sequential (in blue) and 6-threaded parallel factorization without memory constraint (in red) and with a constraint equal to 2.0 times the sequential peak (in green).

Figure 5.1 shows, in the blue curve, the typical memory consumption profile of a sequential multifrontal factorization; the data in this figure results from the factorization of matrix #12. The memory footprint varies considerably throughout the factorization and presents spikes immediately followed by sharp decreases; each spike corresponds to the activation of a front whereas the following decrease corresponds to the deactivation of its children once its assembly is completed.

It has to be noted that the use of tree parallelism normally increases the memory consumption of the multifrontal method simply because, in order to feed the working processes, more work has to be generated and thus more fronts have to be activated concurrently. The memory consumption of the STF parallel code discussed so far can be considerably higher than the sequential peak (up to 3 times or more). This is due to the fact that the runtime system tries to execute tasks as soon as they are available and to the fact that activation tasks are extremely fast and only depend upon each other; as a result, all the fronts in the elimination tree are almost instantly allocated at the beginning of the factorization. This behaviour, moreover, is totally unpredictable because of the very dynamic execution model of the runtime system. This is depicted in Figure 5.1 with the red curve which shows the memory consumption for the case where our runtime based STF factorization is run with six threads; in this case the memory consumption is roughly five times higher than the sequential peak. Figure 5.2 shows the memory consumption of the STF parallel code relative to the sequential code on a number of matrices when executed on the Ada computer using 32 threads. As shown, in several cases the memory increase can be considerable, especially in the case where the factors are discarded due to the fact that, in this case, the relative weight of the temporary memory is less.

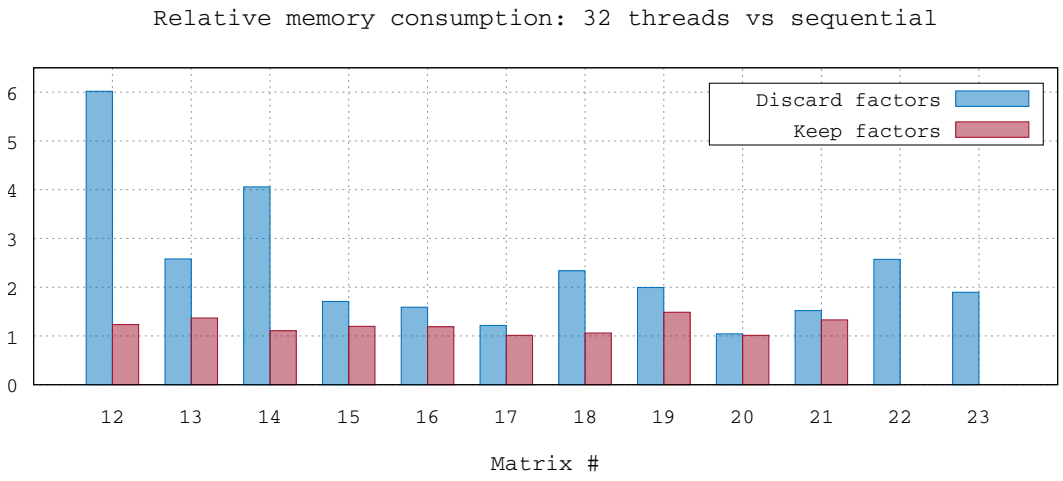


Figure 5.2: Memory consumption of the parallel STF code with 32 threads relative to the sequential memory peak on the Ada computer either discarding or keeping the factors in memory. Matrices #22 and #23 could only be factorized discarding the factors due to limited memory availability on the machine.

5.2 Task scheduling under memory constraint

5.2.1 The sequential case

Task scheduling problems generally focus on minimizing the total completion time although in some cases it may be interesting to consider other or additional objectives such as memory usage. Such problems were originally studied by Sethi et al. [104] for the evaluation of arithmetic expressions whose computation consists in the traversal of the tree representing them. The objective of the aforementioned work is to compute these arithmetic expressions using the minimal number of registers. The problem may be formulated as a *tree pebble game* which has a polynomial complexity for tree-shaped graphs [104] and is shown to be NP-hard by Sethi [103] for general DAGs if nodes cannot be pebbled more than once.

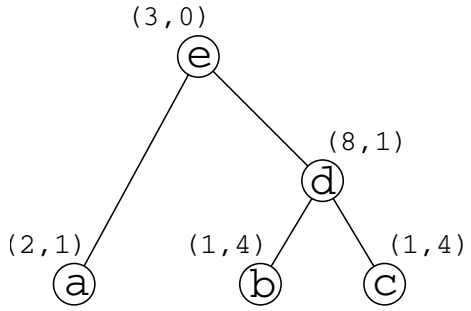
The memory-minimizing postorder traversal of the multifrontal tree is given by Liu [83] where the author notes the relation between the *tree pebble game* and the memory usage of the multifrontal method. The proposed algorithm, consists in a bottom-up traversal of the tree where the traversal is determined as follow:

1. If node k is a leaf, then compute its memory peak $P_k = mfront_k$ where $mfront_k$ is the memory associated with node k;

2. Otherwise, if node k is not a leaf, the idea is to reorder the child sequence $1, .., nc_i$ such that it minimizes the quantity $\max_{i=1,..,nc_i}(x_i + \sum_{j=1}^{i-1} y_j)$ where x_i represent the memory usage to process node i and $\sum_{j=1}^{k-1} y_j$ the memory remaining after processing the first $i-1$ nodes. Liu proves [83] that this quantity can be minimized by ordering the child nodes in descending order of $x_i - y_i$. The value associated with "x_i" and "y_i" depends on the memory management and assembly scheme as extensively presented and discussed by L'Excellent [81]. Finally the memory peak associated with node k may be computed using the new child sequence. In our case we have

$x_i = max(P_i, mfront_k + F_k)$ and $y_j = cb_j + F_j$ where cb_j represents the memory size of the contribution block for node j and F_j the size of factors in the subtree rooted at node j. In the case of an out-of-core execution the previous expressions are changed into $x_i = max(P_i, mfront_k)$ and $y_j = cb_j$.

The optimal memory ordering is obtained using a depth-first traversal on the tree using the rearranged child sequences. The minimal memory usage for the sequential traversal of the elimination tree is given by the memory peak computed at the root node. Figure 5.3 shows the variation of the memory usage during the factorization. The table on the right side of the figure shows the memory consumption for a sequential execution where the tree is traversed in natural order which is a-b-c-d-e; the corresponding sequential peak is equal to 22 memory units. If we apply the above algorithm to the case of this figure then we find that the optimal memory traversal is b-c-d-a-e and the corresponding sequential memory peak is equals to 19 memory units.



Task	Memory
activate(a)	3
activate(b)	8
activate(c)	13
activate(d)	**22**
deactivate(b)	18
deactivate(c)	14
activate(e)	17
deactivate(a)	16
deactivate(d)	15

Figure 5.3: The memory consumption (*right*) for a 5-nodes elimination tree (*left*) assuming a sequential traversal in natural order. Next to each node of the tree the two values corresponding to the factors (permanent) and the contribution block (temporary) sizes in memory units.

Liu [82] observed that the postorder may not give the best memory usage among all topological orders and proposes an algorithm to find the optimal memory topological order for a given tree. Motivated by tree structures emerging from the multifrontal factorization, Jacquelin et al. [74] give an alternative algorithm less costly for computing this memory optimal topological order. In practice, only postorders are considered because they provide good locality properties and, as pointed out by Jacquelin et al. [74], they often give optimal traversal on trees coming from real applications.

5.2.2 The parallel case

The memory-minimization problem, extensively studied in sequential, has received little attention in the parallel case. In recent work, Eyraud-Dubois et al. [50] show that the parallel variant of the *pebble game* is NP-complete and prove that there is no approximation algorithm that can be designed to tackle the problem. They propose, instead, several heuristics for scheduling task trees which aim at reducing the memory consumption of a multifrontal factorization[2] in a shared memory, parallel setting. One of these heuristics, MemBookingInnerFirst is such that the parallel processing of the elimination tree

[2]Note that the problem is presented in a much more general form in their paper and not necessarily related to the multifrontal factorization.

can be achieved while respecting a prescribed memory bound. This is achieved through a memory reservation system; roughly speaking, when a tree node is activated not only is the memory needed for its processing allocated but the memory needed to process its parent is reserved. This reservation ensures that, when the parent node becomes ready, enough memory is available to process it; this prevents memory deadlocks (see the next section for an explanation of this issue). The authors validated experimentally this heuristic, as well as the others, by simulating a rather simplistic shared memory parallelization scheme.

In the distributed memory parallel case, the memory issue is much harder to tackle because, not only has the memory consumption to be minimized but also it must be balanced among the various nodes participating in the factorization. Sparse direct solvers commonly rely on the *Proportional Mapping* [92] method for distributing the work and memory loads across the computer nodes. In this method all the resources assigned to a node of the elimination tree are partitioned into subsets whose size is relative to the weight (either in terms of memory or flops) of the child subtrees and each subset assigned to the root of the corresponding child subtree. All the available resources are first mapped on the root node and then the method works recursively until all the nodes of the tree have been assigned some resources. The proportional mapping method mostly aims at balancing the (memory or computational) load of the processes but does not offer any guarantee on the overall memory consumption. Indeed Rouet [97] proved that this method leads to a poor memory efficiency and scalability. A radically different approach, sometimes referred to as *tree sequential* mapping, consists in mapping all the available resources on every single node of the eliminations tree: as a result, the nodes of the tree are traversed sequentially and each node is processed in parallel using all the available resources. This approach achieves exactly the same memory consumption as in a sequential execution but suffers from very poor performance because tree parallelism is not used and because an excessive amount of resources is allocated to fronts of potentially small size. In his PhD thesis, Rouet [97] refines a memory aware method that was first introduced by Agullo [1] and which maps the computational resources aiming at a good load balance under a prescribed memory constraint. The method attempts to apply the proportional mapping method but every time a proportional mapping step is applied to a set of siblings it also checks whether the imposed memory constraint is respected within the corresponding subtrees. If it is not the case, the method serializes the processing of the sibling subtrees by enforcing precedence constraints. Roughly speaking, the method tries to apply proportional mapping as much as possible but whenever this leads to a violation of the memory constraints it locally reverts to tree sequential mapping.

5.3 STF Memory-aware multifrontal method

This section proposes a method for limiting the memory consumption of parallel executions of our STF code by forcing it to respect a prescribed memory constraint which has to be equal to or bigger than the sequential peak. This technique shares commonalities with the MemBookingInnerFirst heuristic proposed by Eyraud-Dubois et al. [50]. On the other hand, whereas Eyraud-Dubois et al. [51] only consider the theoretical problem, the present study proposes a new and robust algorithm to ensure that the imposed memory constraint is guaranteed while allowing a maximum amount of concurrency on shared-memory multicore architectures. In the remainder of this section we assume that the tree traversal order is fixed and we do not tackle the problem of finding a different traversal that minimizes the memory footprint.

We rely on the STF model to achieve this objective with a relatively simple algorithm.

In essence, the proposed technique amounts to subordinating the submission of tasks to the availability of memory. This is done by suspending the execution of the outer loop in Figure 4.2 if not enough memory is available to activate a new front until the required memory amount is freed by already submitted `deactivate` tasks. Special attention has to be devoted to avoiding memory deadlocks, though. A memory deadlock may happen because the execution of a front deactivation task depends (indirectly, through the assembly tasks) on the activation of its parent front; therefore the execution may end up in a situation where no more fronts can be activated due to the unavailability of memory and no more deactivation tasks can be executed because they depend on activation tasks that cannot be submitted. An example of memory deadlock may be shown using Figure 5.3. We remember that, as explained in Section 5.2.1, the memory-minimizing traversal for the tree in this figure is b-c-d-a-e, which leads to a memory consumption of 19 units. Assume a parallel execution with a memory constraint equal to the sequential peak. If no particular care is taken, nothing prevents the runtime system from activating nodes a, b and c at once thus consuming 13 memory units; this would result in a deadlock because no other front can be activated nor deactivated without violating the constraint.

This problem can be addressed by ensuring that the fronts are allocated in exactly the same order as in a sequential execution: this condition guarantees that, if the tasks submission is suspended due to low memory, it will be possible to execute the deactivation tasks to free the memory required to resume the execution. Note that this only imposes an order in the allocation operations and that all the submitted tasks related to activated fronts can still be executed in any order provided that their mutual dependencies are satisfied. This strategy is related to the Banker's Algorithm proposed by Dijkstra in the early 60's [45, 46].

```
1  forall fronts f in topological order
       do while (size(f) > avail_mem) wait
3      ! allocate and initialize front: avail_mem -= size(f)
       call activate(f)
5
       ! initialize the front structure
7      call submit(init, f:RW, children(f):R)

9      ! front assembly
       forall children c of f
11         ...
           ! Deactivate child: avail_mem += size(cb(f))
13         call submit(deactivate, c:RW)
       end do
15
       ! front factorization
17     ...

19 end do
   call wait_tasks_completion()
```

Figure 5.4: Pseudo-code showing the implementation of the memory-aware task submission.

In our implementation this was achieved as shown in Figure 5.4. Before performing a front activation (line 4), the master thread, in charge of the submission of tasks, checks if enough memory is available to perform the corresponding allocations (line 2);

if this is the case, the allocation of the frontal matrix (and the other associated data) is performed within the `activate` routine. This activation is a very lightweight operation which consists in simple memory bookkeeping (due to the first-touch rule) and therefore does not substantially slow down the task submission. The front initialization is done in the `init` task (line 7) submitted to the runtime system which can potentially execute it on any worker thread, as described in Section 4.2. If the memory is not available, the master thread suspends the submission of tasks until enough memory is freed to continue. In order not to waste resources, the master thread is actually put to sleep rather than leaving it to sit on active wait. This was manually implemented through the use of POSIX thread locks and condition variables: the master thread goes to sleep waiting for a condition which is signaled by any worker thread that frees memory by executing a `deactivate` task. When woken up, the master checks again for the availability of memory. This work has prompted the StarPU developers to extend the runtime API with routines (the `starpu_memory_allocate/deallocate()` and `starpu_memory_wait_available()`) that implement this mechanism and easily allow for implementing memory-aware algorithms.

5.4 Experimental results

This section describes and analyses experiments that aim at assessing the effectiveness of the memory aware scheduling presented in Section 5.3. Here we are interested in two scenarios:

1. In-Core (IC) execution: this is the most common case where the computed factors are kept in memory. In this case matrices #22 and #23 could not be used because of the excessive memory consumption;

2. Out-Of-Core (OOC) execution: in this scenario the factors are written to disk as they are computed in order to save memory. In this case the memory consumption is more irregular and more considerably increased by parallelism. We simulate this scenario by discarding the factors as we did in Sections 4.2.2 and 4.3.1; note that by doing so we are assuming that the overhead of writing data to disk has a negligible effect on the experimental analysis reported here.

These experiments measure the performance of both the 1D and 2D factorization (with the parameter values in Table 4.1 and Table 4.2) within an imposed memory footprint. Experiments were performed using 32 cores with memory constraints equal to $\{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\} \times sequential\_peak$ both when the factors are kept in memory and when they are discarded. The parameter settings used for these experiments are those reported in Tables 4.1 and 4.2.

For almost all 1D and 2D IC tests as well as all 2D OOC tests, a performance as high as the non constrained case (presented in Section 4.2.2 and Section 4.3.1) could be achieved with a memory exactly equal to the sequential peak, which is the lower bound that a parallel execution can achieve. This shows the extreme efficiency of the memory-aware mechanism for achieving high-performance within a limited memory footprint. Combined with the 2D numerical scheme, which delivers abundant node parallelism, the memory-aware algorithm is thus extremely robust since it could process all considered matrices at maximum speed with the minimum possible memory consumption. In a few cases a slight increase (always lower than 20%) in the factorization time can be observed (especially when the constraint is set equal to the sequential peak). In only three cases it is possible

to observe a smooth decrease of the factorization time as the constraint on the memory consumption is relaxed: these are the OOC, 1D factorization of matrices #12, #14 and #18.

To explain this extreme efficiency, we performed the following analysis. As explained in Section 5.3, prior to activating a front, the master thread checks whether enough memory is available to achieve this operation. If it is not the case, the master thread is put to sleep and later woken up as soon as one `deactivate` task is executed; at this time the master thread checks again for the availability of memory. The master thread stays in this loop until enough deactivation tasks have been executed to free up the memory needed to proceed with the next front activation. Every time the master thread was suspended or resumed we recorded the time stamp and the number of ready tasks (i.e., those whose dependencies were all satisfied).

Figure 5.6 shows the collected data for matrix #12 with an imposed memory consumption equal to the sequential peak, in the OOC case using both the 1D (*left*) and 2D (*right*) methods. In this figure, each (x, y) point means that at time x the master thread was suspended or resumed and that, at that time, y tasks where ready for execution or being executed. The width of each graph shows the execution time of the memory constrained factorization whereas the vertical dashed line shows the execution time when no limit on the memory consumption is imposed. The figure leads to the following observations:

- in both the 1D and 2D factorizations, the number of ready tasks falls, at some point, below the number of available cores (the horizontal, solid line); this lack of tasks is responsible for a longer execution time with respect to the unconstrained case.

- in the 1D factorization this lack of tasks is more evident; this can be explained by the fact that the 1D method delivers much lower concurrency than the 2D one and therefore, suspending the submission of tasks may lead more quickly to thread starvation. As a result, the difference in the execution times of the constrained and unconstrained executions is more evident in the 1D factorization.

For all other tests, either the number of tasks is always (much) higher than the number of workers or the tasks submission is never (or almost never) interrupted due to the lack of memory; as a result, no relevant performance degradation was observed with respect to the case where no memory constraint is imposed. This behavior mainly results from two properties of the multifrontal QR factorization:

1. the size of the contribution blocks is normally very small compared to the size of factors, especially in the case where frontal matrices are overdetermined;

2. the size of a front is always greater than or equal to the sum of the sizes of all the contribution blocks associated with its children (because in the assembly operation, contribution blocks are not summed to each other but stacked).

As a result, in the sequential multifrontal QR factorization, the memory consumption grows almost monotonically and in most cases the sequential peak is achieved on the root node or very close to it. For this reason, when the tasks submission is interrupted in a memory-constrained execution, a large portion of the elimination tree has already been submitted and the number of available tasks is considerably larger than the number of working threads. Other types of multifrontal factorizations (LU, for instance) are likely to be more sensitive to the memory constraint because they do not possess the two properties described above. By the same token, it is reasonable to expect that imposing a memory

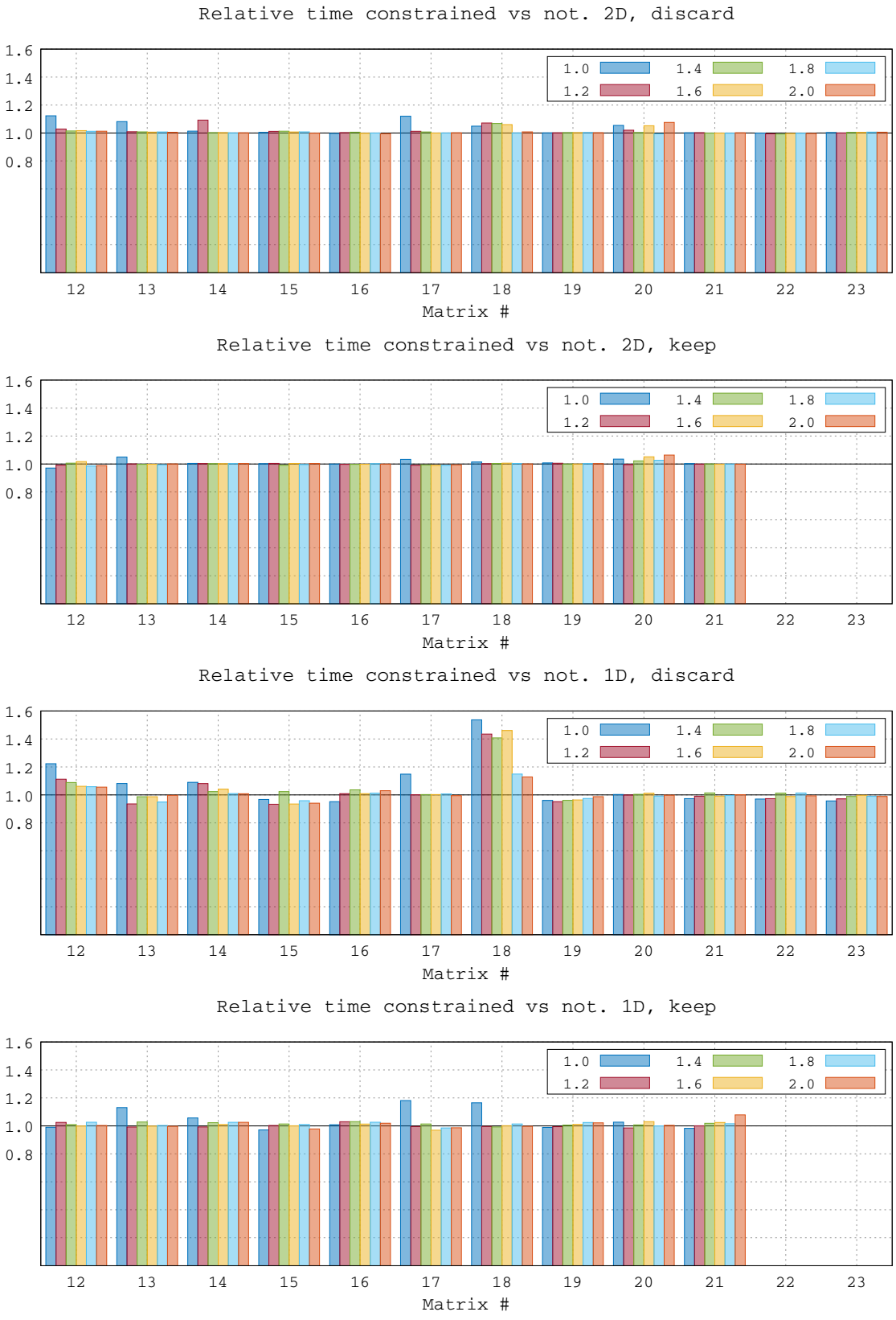


Figure 5.5: Memory-constrained factorization times relative to the unconstrained execution for the 2D and 1D methods either discarding (OOC) or keeping (IC) the factors in memory. Matrices #22 and #23 could only be factorized discarding the factors (OOC) due to limited memory availability on the machine.

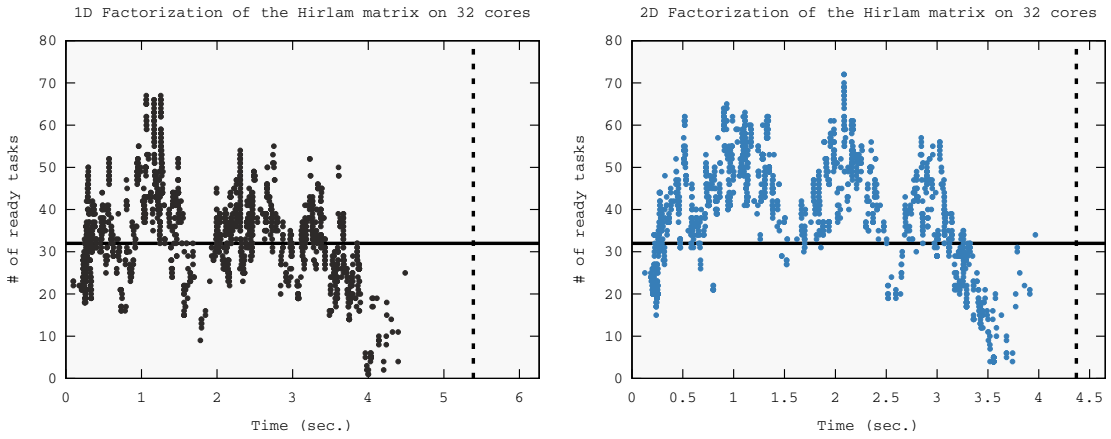


Figure 5.6: Concurrency under a memory constraint for the Hirlam matrix on Ada (32 cores).

constraint could more adversely affect performance when larger numbers of threads are used.

Chapter 6

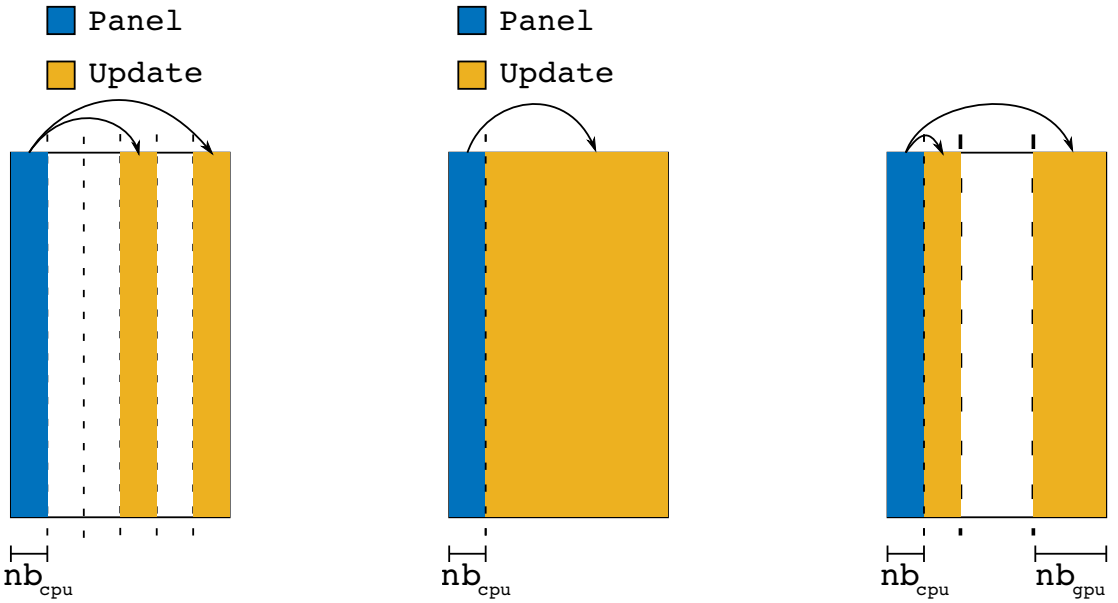
STF-parallel multifronatal QR method on heterogeneous architecture

In this chapter we present the implementation of a multifrontal method for heterogeneous architectures such as GPU-accelerated multicore systems. The implementation presented in Chapter 4 constitutes the basis for this work and as we showed in the multicore case, the runtime system facilitates the development of such algorithms in the context of heterogeneous systems. First of all, the runtime system is capable of handling the data transfers and managing the data consistency across the architecture then, by providing kernels specific to accelerators, it is able to execute tasks on accelerators following a given scheduling strategy. We show in this chapter that solely providing the kernels for accelerator is not sufficient to efficiently exploit the potential performance of accelerator based systems and we propose frontal matrix partitioning and scheduling strategies to attain good performance on such machines. In addition using the performance analysis approach presented in Chapter 2 we are capable of evaluating how well we are exploiting the accelerated architectures on which we performed our experiments.

6.1 Frontal matrices partitioning schemes

Finding frontal matrix partitioning strategies that allows an efficient exploitation of both CPU and GPU resources constitutes one of the main challenges in implementing a multifrontal method for GPU-accelerated multicore systems. This results from the fact that GPUs, which are potentially able to deliver much higher performance than CPUs, require coarse granularity operations to achieve peak performance while a CPU reaches its peak with relatively small granularity tasks. The approach presented in Section 4.2, where frontal matrices are uniformly partitioned into small size block-columns, could be readily ported to GPU-accelerated platforms by simply providing GPU implementations for the various tasks to the StarPU runtime system. This, however, would result in an unsatisfactory performance because, due to the fine granularity of tasks, only a small fraction of the GPU performance could be used. This front partitioning strategy, which we refer to as *fine-grain* partitioning, shown in Figure 6.1(a), is not suited to heterogeneous architectures despite being able to deliver sufficient concurrency to feed both the CPU cores and the GPU and reduce idle times on all the resources. A radically different approach is what we refer to as *coarse-grain* partitioning (Figure 6.1(b)), where fine-grained panel tasks are executed on CPU and large-grain (as large as possible) update tasks are performed on

GPU. This corresponds to the algorithm used in the MAGMA package [9] and aims at obtaining the best acceleration of computationally intensive tasks on the GPU. In order to keep the GPU constantly busy, static scheduling is used that allows the overlapping of GPU and CPU computation thanks to a depth-1 lookahead technique; this is achieved by splitting the trailing submatrix update into two separate tasks of, respectively, fine and coarse granularity. This second approach clearly incurs the opposite problem than the one we face with the fine-grain partitioning: despite being able to maximize the efficiency of GPU operations with respect to the problem size, it severely limits the amount of node parallelism as well as of inter-level parallelism (see Section 1.7.1 and 4.2.3) and therefore leads to resource (especially CPUs) starvation.



(a) Fine-grain partitioning.

(b) Coarse-grain partitioning.

(c) Hierarchical-grain partitioning.

Figure 6.1: Partitioning schemes.

In order to take advantage of the fine and coarse-grain approaches and to overcome the limitations of both, we developed a *hierarchical* partitioning of fronts (Figure 6.1(c)) which is similar to the approach proposed by Wu et al. [116] and corresponds to a trade-off between parallelism and GPU kernel efficiency with task granularity suited for both types of resources. The front is first partitioned into coarse grain block-columns, referred to as outer block-columns, of width $\texttt{nb}_{\texttt{GPU}}$ suitable for GPU computation (this happens at the moment when the front is activated) and then each outer block-column is dynamically re-partitioned into inner block-columns of width $\texttt{nb}_{\texttt{CPU}}$ appropriate for the CPU only immediately before being factorized. This is achieved through dedicated partitioning tasks which are subject to dependencies with respect to the other, previously submitted, tasks that operate on the same data. When these dependencies are satisfied, StarPU ensures that the block being re-partitioned is in a consistent state, in case there are multiple copies of it. Furthermore, StarPU ensures that the partitioning is performed in a logical fashion: no actual copy is performed and there is no extra data allocated. The partitioning is done using two tasks: `partition` and `unpartition`. In order to partition a data i represented by the handle `f(i)` into n pieces, it is necessary to declare the handles associated with the sub-data `f(i,1)..f(i,n)`. The `partition` task takes as input the data to be partitioned

```
forall outer panels o_p=1...o_n in f
   ! partition (outer) block column f(o_p) into
   ! i_n inner block columns f(o_p,1) .. f(o_p,i_n)
   call submit(partition, f(o_p):R, f(o_p,1):W... f(o_p,i_n):W)

   forall inner panels i_p=1..i_n
      ! panel reduction of inner block column i_p
      call submit(_geqrt, f(i_p):RW)
      forall inner blockcolumns i_u=i_p+1..i_n in f(o_p)
         ! update (inner) column in_u with panel i_p
         call submit(_gemqrt, f(i_p):R, f(i_u):RW)
      end do

      forall outer blockcolumns o_u=o_p+1..o_n
         ! update outer block column o_u with panel i_p
         call submit(_gemqrt, f(i_p):R,f(o_u):RW)
      end do
   end do

   ! unpartition (outer) block column
   call submit(unpartition, f(o_p,1):R...f(o_p,i_n):R, f(o_p):W)
end do
```

Figure 6.2: STF code for the hierarchical QR factorization of frontal matrices.

```
forall panels p=1...n in f

   ! partition trailing submatrix
   call submit(partition, f(p,tr):R, f(p,bc):W, f(p+1,tr):W)

   ! panel reduction of block column p
   call submit(_geqrt, f(p,bc):RW)

   !  update trailing submatrix p+1 with panel p
   call submit(_gemqrt, f(p+1,tr):RW)

   ! unpartition trailing submatrix
   call submit(unpartition, f(p,bc):R, f(p+1,tr):R, f(p,tr):W)

end do
```

Figure 6.3: STF code for the coarse QR factorization of frontal matrices.

with a `Read` access mode and the resulting sub-data with a `Write`. The `unpartition` tasks take as input the sub-data with a `Read` access mode and the original data with a `Write` access mode. In the STF code, as long as all tasks working on sub-data are submitted between the `partition` and `unpartition` tasks and no tasks working on the partitioned data are submitted, the data consistency between data and sub-data is ensured. It should be noted that in order to avoid memory copy, both `partition` and `unpartition` tasks should be executed on the node where the data is allocated and in this case these tasks are associated with an empty function.

In order to use the hierarchical-grain partitioning in the multifrontal factorization, the initial STF code corresponding to the QR factorization of a front using a fine-grain partitioning (lines 14-21 in Figure 4.5) is turned into the one proposed in Figure 6.2 for hierarchically partitioned fronts. We define inner and outer tasks depending on whether these tasks are executed on inner or outer block-columns. In order to ease the understanding we use different names for inner and outer updates although both types of tasks perform exactly the same operation and thus employ the same code.

Note that the `partition` and `unpartition` tasks allow us to easily implement of the coarse-grain partitioning approach as shown in Figure 6.3. For this partitioning we need to define two handles per panel `p`: `f(p,bc)` corresponding to the block-column whose size is defined by the parameter $\texttt{nb}_{\texttt{CPU}}$ and `f(p,tr)` corresponding to the trailing submatrix including the current block-column `p`. This scheme will only be used for evaluation purposes in Section 6.4.1 for the computation of performance bounds.

The work described above has prompted the StarPU developers to implement the dynamic partition and unpartition capability in the runtime system. As a result, the StarPU API now includes the `starpu_data_partition_plan`, `starpu_data_partition_submit` and `starpu_data_unpartition_submit` which allow for, associating a partitioning scheme with some data handle and submitting a partition and an unpartition task respectively.

6.2 Scheduling strategies

Along with the fronts partitioning strategies discussed in the previous section, task scheduling is a key factor for archiving reasonable performance on heterogeneous systems. One strategy to schedule the tasks resulting from the partitioning is to statically assign the coarse granularity tasks to GPUs and fine granularity tasks to CPU cores. This is the strategy adopted in the work by Lacoste [80] presented in Section 1.6 where the GPU kernels are statically mapped onto the devices. However, in our problem, the variety of front shapes and staircase structures combined with this hierarchical partitioning induces an important workload heterogeneity, making load balancing extremely hard to anticipate. For this reason, we chose to rely on a dynamic scheduling strategy.

In the context of a heterogeneous architecture, the scheduler should be able to handle the workload heterogeneity and distribute the tasks taking into account a number of factors including resource capabilities or memory transfers while ensuring a good load balance between the workers. Dynamic scheduling allows for dealing with the complexity of the workload and limits load imbalance between resources.

Algorithms based on the Heterogeneous Earliest Finish Time (HEFT) scheduling strategy by Topcuouglu et al. [111] represent a commonly used and well known solution to scheduling task graphs on heterogeneous systems. Figure 6.4 illustrates, in the context of runtime based applications, an implementation of a HEFT-like scheduling strategy. These methods consist in first ranking tasks (typically according to their position with respect to the critical path) and then assigning them to resources using a minimum completion

time criterion. Despite the fact that GPUs can accelerate the execution of most (if not all) tasks, not all tasks are accelerated by the same amount, depending on their type (e.g., compute or memory bound, regular or irregular memory access pattern) or granularity: therefore we say that some tasks have a better acceleration factor on the GPU than others. The main drawback of HEFT-like methods lies in the fact that the acceleration factor of tasks is ignored during the worker selection phase, i.e., these methods do not attempt to schedule a task on the unit which is best suited for its execution. In addition, the centralized decision during the worker selection potentially imposes a significant runtime overhead during the execution. A performance analysis conducted with the so-called `dmdas` StarPU built-in implementation of HEFT showed that these drawbacks are too severe for designing a high-performance multifrontal method.

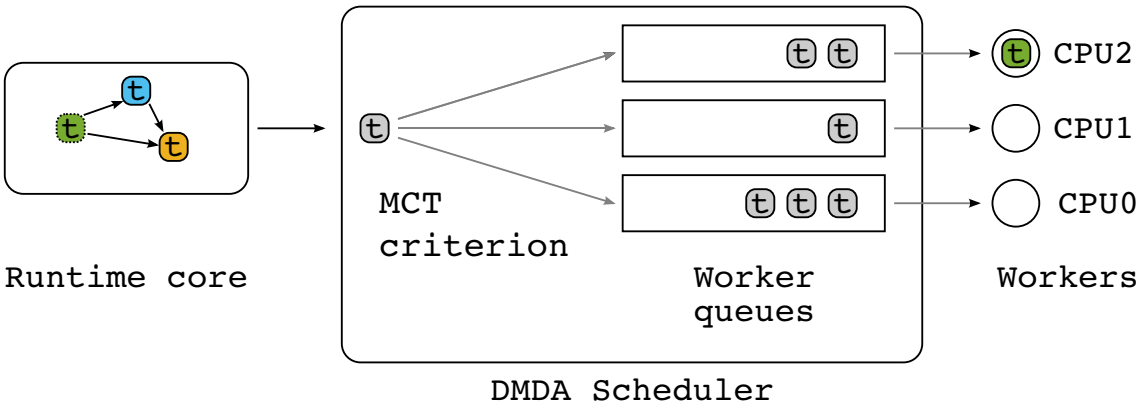


Figure 6.4: DMDA (Deque Model Data Aware) scheduler.

Instead, we implemented and extended a scheduling technique known as HETEROPRIO, first introduced by Agullo et al. [7] in the context of Fast Multipole Methods (FMM). This technique is inspired by the observation that a DAG of tasks may be extremely irregular with some part where concurrency is abundant and others where it is scarce as shown in Figure 6.5. In the first case we can perform tasks on the units where they are executed the most effectively without any risk of incurring resource starvation because parallelism is plentiful. In the second case, however, what counts most is to prioritize tasks which lie along the critical path because delaying their execution would result in penalizing stalls in the execution pipeline. As a result, in the HETEROPRIO scheduler the execution is characterized by two states: a *steady-state* when the number of tasks is large compared to the number of resources and a *critical-state* in the opposite case. The scheduler can automatically switch from one state to another depending on a configurable criterion which mostly depends on the amount of ready tasks and of computational resources.

A complex, irregular workload, such as a sparse factorization, is typically a succession of steady and critical state phases, where the steady-state corresponds to rich concurrency regions in the DAG whereas the critical-state corresponds to scarce concurrency regions. During a steady-state phase, tasks are pushed to different scheduling queues depending on their expected acceleration factor (see Figure 6.6). In our current implementation, we have defined one scheduling queue per type of tasks (eight in total as listed in column 1 of Table 6.1). When they pop tasks, CPU and GPU workers poll the scheduling queues in different orders. The GPU worker first polls scheduling queues corresponding to coarse-grain tasks such as outer updates (priority 0 on GPU in Table 6.1) because their acceleration factor is higher. On the contrary, CPU workers first poll scheduling queues of

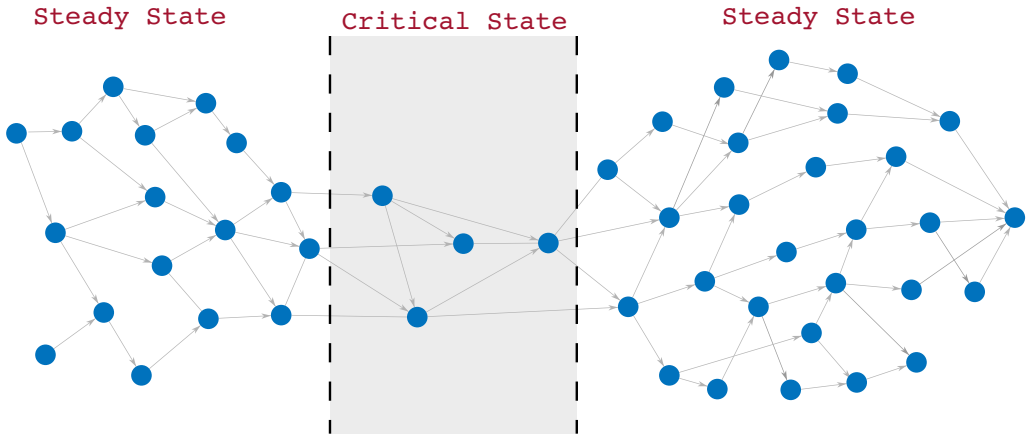


Figure 6.5: Example of a DAG with a succession of steady and critical states corresponding to rich and poor concurrency regions respectively as defined in the HETEROPRIO scheduler.

small granularity such as subtree factorizations or inner panels (as well as tasks performing symbolic work such as activation that are critical to ensure progress). Consequently, during a steady-state, workers process tasks that are best suited for their capabilities. The detailed polling orders are provided in Table 6.1. Furthermore, to ensure fairness in the progress of the different paths of the elimination tree, tasks within each scheduling queue are sorted according to the distance (in terms of flop) between the corresponding front and root node of the elimination tree.

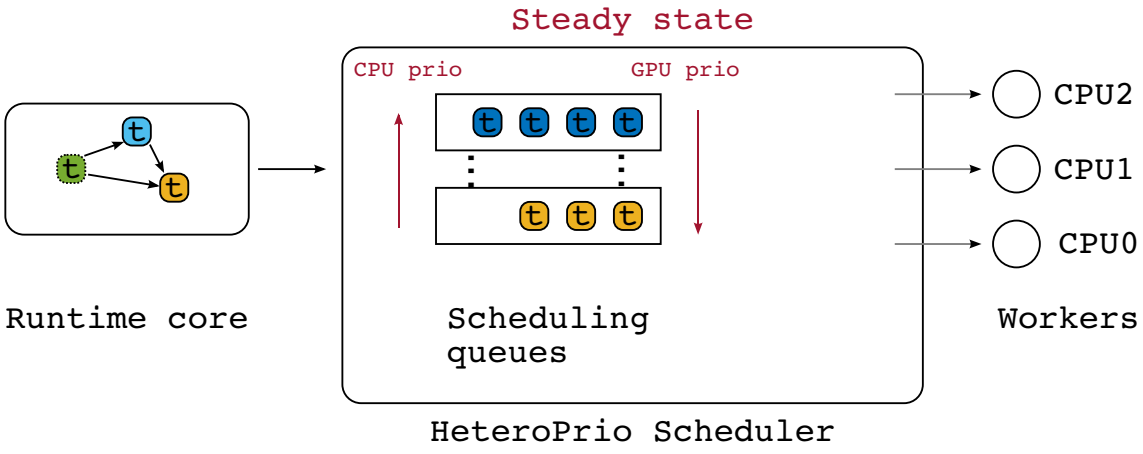


Figure 6.6: HeteroPrio steady-state policy.

In the original HETEROPRIO scheduler [7], the worker selection is performed right before popping the task in a scheduling queue following the previously presented rules. If data associated to the task are not present on the memory node corresponding to the selected worker then the task completion time is increased by the memory transfers. While the associated penalty is usually limited in the FMM case [7], preliminary experiments (not reported here for a matter of conciseness) showed that it may be a severe drawback for the multifrontal method. For the purpose of the present study, we have therefore extended the original scheduler by adding worker queues (one queue per worker) along with the scheduling queues as shown in Figure 6.6. When it becomes idle, a worker pops a task

from its worker queue and then fills it up again by picking a new task from the scheduling queues through the polling procedure described above. The data associated with tasks in a worker queue can be automatically prefetched on the corresponding memory node while the corresponding worker is executing other tasks. If the size of the worker queues is too high, a task may be assigned to a worker much earlier than its actual execution, which may result in a sub-optimal choice. Therefore, because no additional benefit was observed beyond this value, we set this size to two in our experiments.

Scheduling queues	Steady-state		Critical-state	
	CPU	GPU	CPU	GPU
activate	0	-	0	-
assemble	7	-	5	-
deactivate	1	-	1	-
do_subtree	2	2	2	0
part./unpart.	3	-	3	-
inner_panel	4	-	4	-
inner_update	5	1	6	1
outer_update	6	0	7	2

Table 6.1: Scheduling queues and polling orders in HeteroPrio.

When the number of tasks becomes low (with respect to a fixed threshold which is set depending on the amount of computational power of the platform), the scheduling algorithm switches to critical-state. CPU and GPU workers cooperate to process critical tasks as early as possible in order to produce new ready tasks quickly. For instance, because outer updates are less likely to be on the critical path, the GPU worker will select them last in spite of their high acceleration factor. The last two columns of Table 6.1 provide the corresponding polling order. Additionally, in this state, CPU workers are allowed to select a task only if its expected completion time does not exceed the total completion time of the tasks remaining in the GPU worker queue. This extra rule prevents CPU workers from selecting all the few available tasks and leaving the GPU idle whereas it could have finished processing them all quickly.

6.3 Implementation details

In the proposed implementation we use, in addition to the CPU kernels, two GPU kernels: the `_gemqrt` kernel corresponding to the update operation and the `do_subtree` kernel for the factorization of subtrees whose root node is on the layer resulting from the logical tree pruning as explained in section 1.7.2. The `_gemqrt` on GPU is performed using the `magma_{s,d,z,c}larfb_gpu` routine from the MAGMA package [9] which is the equivalent to the `_larfb` kernel for CPUs. In order to exploit computationally intensive kernels and avoid irregularities in the memory access pattern of the factors V that are applied to a block-column, this kernel performs extra computations on the elements on the upper-right corner of the matrix V. This strategy allows us to obtain better kernel performance at the cost of additional flops for this operation. Note, also, that this routine is not capable of exploiting, internally, the staircase structure of the panel. The `do_subtree` on a GPU consists in a sequential tree traversal where fronts are initialized and assembled on the CPU and factorized on the GPU. For this factorization operation, a coarse-grain partitioning is used similarly to the `magma_{s,d,c,z}geqrf` routine from MAGMA. This

routine however cannot be used is our case because of the staircase structure of frontal matrices. Instead, we developed a routine called {s,d,c,z}geqrf_stair_gpu performing coarse-grain factorization of a front using the same mapping of tasks as the MAGMA routine magma_{s,d,z,c}geqrf and capable of exploiting the frontal matrices staircase structure.

As explained in Section 1.4.2.1, when tasks are scheduled on a GPU worker StarPU automatically handles the memory transfers of associated data on the GPU memory node. For the gemqrt tasks with respect to a panel operation, the V and T matrix resulting from the panel and the C block-column to be updated are associated with the tasks and thus will be automatically transferred to the GPU if necessary. In the case of a do_subtree the CPU-GPU memory transfers are performed by {s,d,c,z}geqrf_stair_gpu kernels. The data manipulated in the task are not associated with the task in order to prevent the runtime system from transferring the whole subtree to the GPU before the execution of a task. Instead, the data manipulated in a do_subtree task are represented by a symbolic handle allowing a declaration of the dependencies with the tasks depending on the do_subtree tasks.

6.4 Experimental results

6.4.1 Performance and analysis

We tested the previously presented implementation on a subset of the test matrices in Table 1.1; the experiments were done on one node of the Sirocco cluster presented in Section 1.9.1. In this experimental study we evaluate the performance of the STF parallel multifrontal QR method on heterogeneous architectures using the frontal matrices partitioning and scheduling strategies described above. We first tested the parallel code using CPU workers only and then we added one GPU worker. For the CPU only experiments we use the fine-grain (i.e. block-column) partitioning presented above and in Section 4.2. As explained in Section 4.2.2 the performance of the parallel factorization using a block-column partitioning depends on several parameters. In the fine-grain case these parameters are nb which impacts the concurrency generated in the DAG and the inner block size ib which affects the efficiency of elementary BLAS kernels and determines the global amount of flop performed during the factorization. Finding the optimal parameters for this is extremely difficult because it depends on a great number of factors such as the number of workers, size and structure of the matrix. For this reason we performed a large test with several combinations for ib and nb and selected the best results in terms of factorization time. The values used for the experiments were (nb,ib)={(128,64), (128,128), (192,64), (192,192), (256,64), (256,128), (256,256)}. Similarly to the experimental setting presented in Section 4.2.2 the factors are discarded during the execution. The scheduling used for the experiments in a multicore context is lws presented in Section 4.4.

The performance of the code in the multicore case using a fine-grain partitioning is reported in Table 6.2 with two configurations: the first using the twelve cores of a E5-2680 processor and the second using the twenty-four cores of two E5-2680 processors available on the machine. The table shows the shortest execution time along with the corresponding Gflop/s rates obtained for the factorization of the tested matrices and the optimal parameters ib and nb for which this performance was attained.

For the heterogeneous experiments, we use the hierarchical-grain partitioning presented in section 6.1. The parameters defining a hierarchical block column partitioning are the size of the outer block column nb_{gpu} and the size of the inner block column nb_{gpu}.

	12 CPUs (1×E5-2680)				24 CPUs (2×E5-2680)			
Mat.	nb	ib	Time (s.)	Gflop/s	nb	ib	Time (s.)	Gflop/s
12	192	64	8.892E+00	138.667	128	128	4.922E+00	275.329
13	128	128	1.188E+01	226.780	128	128	8.525E+00	316.029
14	192	192	2.364E+01	221.062	128	128	1.444E+01	351.824
15	128	128	4.151E+01	252.123	128	128	2.468E+01	424.054
16	192	192	5.849E+01	272.335	128	128	3.734E+01	421.033
17	128	64	7.181E+01	271.779	128	64	4.270E+01	457.060
18	128	128	1.104E+02	248.881	128	128	6.209E+01	442.527
19	192	192	2.212E+02	284.040	128	128	1.269E+02	489.390
21	192	192	6.318E+02	294.604	192	192	3.352E+02	555.283

Table 6.2: Optimum performance for the STF fine-grain 1D factorization on Sirocco in homogeneous case with both configurations 12 CPUs (1×E5-2680) and 24 CPUs (2×E5-2680).

The value ib is fixed such that $\texttt{ib} = \texttt{nb}_{\texttt{cpu}}$ because, as explained above the _gemqrt operation on the GPU cannot take advantage of the staircase structure and because, as seen in Section 4.2.2, this is commonly the choice which yields the best performance. As for the multicore case, we performed a large set of tests with several combinations for nb_{cpu} and nb_{gpu} and selected the best results in terms of factorization time. The values used for the experiments were $(\texttt{nb}_{\texttt{gpu}},\texttt{nb}_{\texttt{cpu}})=\{$(256,128), (256,256), (384,128), (384,384), (512,128), (512,256), (512,512) (768,128), (768,256), (768,384), (896,128), (1024,128), (1024,256), (1024,512) $\}$. The scheduler used for the experiments in an heterogeneous context is HETEROPRIO presented in Section 6.2.

The performance of the code in the heterogeneous case using a hierarchical partitioning is reported in Table 6.2 with two configurations: first using the twelve cores of a E5-2680 processor plus one GPU K40M and second using the twenty-four cores of two E5-2680 processors available on the machine plus one GPU K40M. The table shows the shortest execution time along with the corresponding Gflop/s rates obtained for the factorization of the tested matrices and the optimal parameters nb_{gpu} and nb_{cpu} for which these performance were attained. Note that in the case were $\texttt{nb}_{\texttt{gpu}} = \texttt{nb}_{\texttt{cpu}}$, then the hierarchical-grain partitioning is equivalent to the fine-grain partitioning. Figure 6.7 plots the data in Tables 6.2 and 6.3 along with the performance obtained with a coarse-grain partitioning approach; note that matrices #22 and #23 cannot be factorized with this approach because it requires the entire trailing submatrix to be on the GPU for an update operation.

While the results presented in this section show the interest of the hierarchical scheme with respect to the other proposed partitioning strategies, one may wonder how this scheme behaves in terms of absolute performance. The most straightforward reference would be the cumulative peak performance over all computational units. However, as explained in Chapter 2, this choice does not take into account the fact that, due to their nature (because of their granularity or the nature of operations they perform), tasks cannot be executed at the peak speed and may result in an excessively loose bound on achievable performance. For this reason we perform the performance analysis proposed in Chapter 2 to evaluate the obtained results. We compute $\tilde{t}^{area}(p)$ as an upper bound on the performance of our application and the efficiency measures e_g, e_t, e_p, e_c and e_r to identify the main factors limiting the performance of the execution with respect to this

	12 CPUs (1×E5-2680) + 1 GPU (1×K40M)				24 CPUs (2×E5-2680) + 1 GPU (1×K40M)			
Mat.	nb_{gpu}	nb_{cpu}	Time (s.)	Gflop/s	nb_{gpu}	nb_{cpu}	Time (s.)	Gflop/s
12	384	384	8.302E+00	214.172	384	384	6.960E+00	255.468
13	384	384	1.014E+01	309.913	256	256	9.731E+00	304.155
14	384	384	1.803E+01	311.883	256	256	1.404E+01	382.014
15	768	256	2.051E+01	525.130	512	256	2.026E+01	531.609
16	896	128	2.810E+01	559.479	896	128	2.866E+01	548.547
17	192	192	8.700E+01	246.755	256	256	6.797E+01	329.905
18	512	256	4.918E+01	560.600	768	256	4.882E+01	564.733
19	768	256	1.039E+02	611.656	796	384	8.763E+01	741.399
21	768	256	2.792E+02	671.463	1024	256	2.403E+02	780.160

Table 6.3: Optimum performance for the STF hierarchical-grain 1D factorization on Sirocco in heterogeneous case with both configurations 12 CPUs (1×E5-2680) + 1 GPU (1×K40M) and 24 CPUs (2×E5-2680) + 1 GPU (1×K40M).

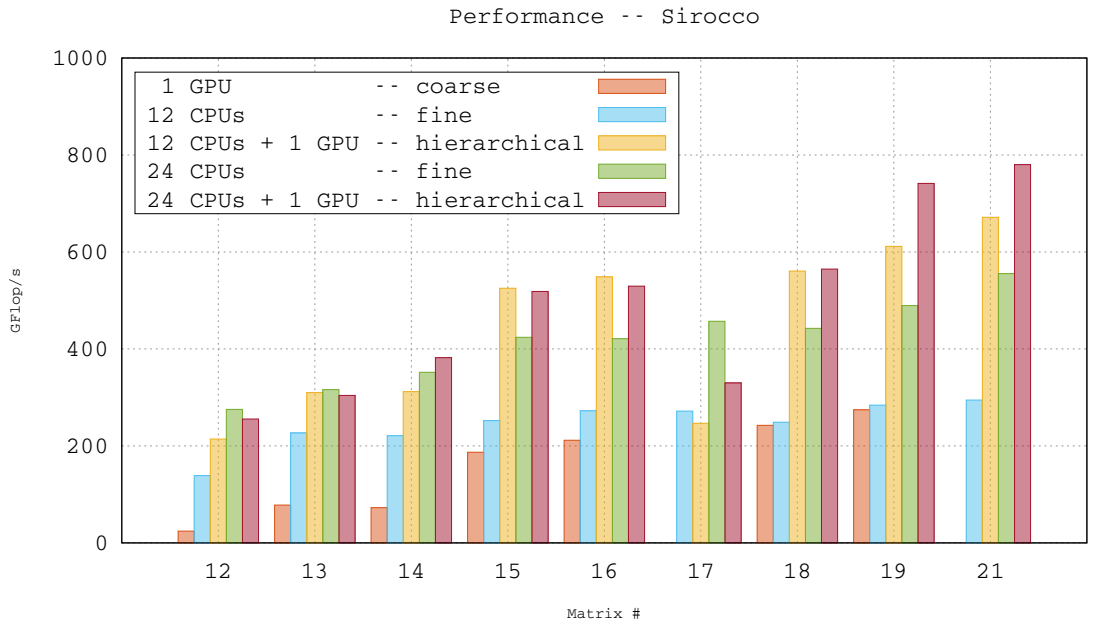


Figure 6.7: Performance with heterogeneous vs multicore algorithms on Sirocco platform.

upper bound. Technically, $\tilde{t}^{area}(p)$ and $t^{area}(p)$ are computed by running two instances of the code, respectively, one with coarse grain partitioning and the other with hierarchical partitioning and solving, in both cases, the Linear Program 1 defined in Chapter 2. Note that in order to generate the linear problem, it is necessary to run the code multiple times so that StarPU can build accurate performance profiles.

Figure 6.8 shows the efficiency analysis for our code on the test matrices. With a runtime efficiency e_r greater than 0.9 for the tested matrices we see that the cost of the runtime system is negligible compared to the workload. In addition, the runtime overhead becomes relatively smaller and smaller as the size of the problems increases. These results

Figure 6.8: Efficiency measures for the STF heterogeneous algorithm on Sirocco with both configurations 12 CPUs (1×E5-2680) + 1 GPU (1×K40M) and 24 CPUs (2×E5-2680) + 1 GPU (1×K40M). Note that due to technical issues in StarPU, we are currently unable to obtain the efficiency measures for matrix # 17.

also show that our scheduling policy makes a good job in assigning tasks to the units where they can be executed more efficiently. The task efficiency e_t, lies between 0.8 and 1.2 for all tested matrices except for matrix #15, denotes a good load balancing of the workload between the CPUs and the GPU. We observe in our experiments that the task efficiency may be greater than one as for matrix #15, #16 and #18. As explained in Chapter 2, this is simply due to the fact that too many tasks are affected to faster units, e.g., GPUs; this implies starvation of the slower units which translates into a weaker pipeline efficiency,

as shown in Figure 6.8. The most penalizing effect on the global efficiency is the pipeline efficiency e_p. In addition, for all tested matrices except matrix #21 the pipeline efficiency decreases when the number of cores goes from twelve to twenty-four. This is mainly due to a lack of concurrency resulting from the choice of partitioning. This choice aims at achieving the best compromise between the efficiency of kernels and the amount of concurrency; it must be noted that e_p could certainly be improved by using a finer grain partitioning but this would imply a worse efficiency of the tasks and thus, as a consequence, higher values for both $t_t(p)$ and $t^{area}(p)$. In addition, smaller matrices do not deliver enough parallelism to feed all the resources which explains that the pipeline efficiency is greater on the biggest problems. Similarly to the runtime efficiency, the communication efficiency is rather good with values greater than 0.85. This shows that the scheduler is capable of efficiently overlapping task execution with communications thanks to the data prefetching capability enabled by the use of worker threads. All in all, we can observe that the parallelization efficiency is satisfactory especially on the biggest problems.

6.4.2 Multi-streaming

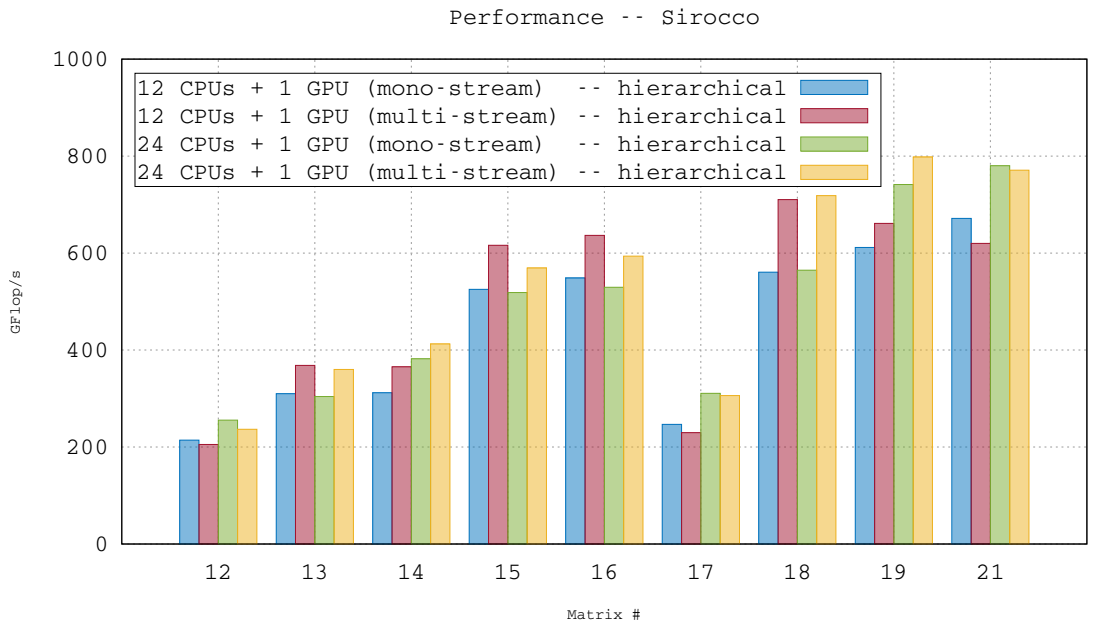


Figure 6.9: Performance with heterogeneous multi-stream vs mono-stream configurations on Sirocco platform.

On a GPU, a stream is defined as a sequence of commands that execute in order and is a feature available on relatively recent GPUs [1]. The use of multiple streams allows the execution of concurrent kernels on the device thus increasing the occupancy of the GPU when executing small grain tasks that are unable to feed all the available resources. This allows us to use smaller values for the parameters $\texttt{nb}_{\texttt{cpu}}$ and $\texttt{nb}_{\texttt{gpu}}$ leading to a greater concurrency in the DAG and thus potentially better performance. In addition, it allows us

[1] Available on compute capability 2.x and higher devices but exploitable since compute capability 3.5 with the introduction of Hyper-Q technology.

to exploit tree parallelism on the GPU with the possible execution of tasks from different frontal matrices in the elimination tree. However, it should be noted that in some cases the parallel execution of several kernels in different streams cannot be completely overlapped if these kernels achieve a high occupancy of GPU resources. In StarPU we recall that each stream is viewed as a CUDA worker and is unable to detect whether the GPU is fully occupied or not. For this reason, the use of multiple streams can limit the pipelining of tasks and lower the performance. In Table 6.4 we report on the performance results of our solver when using multiple streams along with the corresponding parameters used to obtain these values. For the experiments, we performed several runs using the same combinations for the parameters $\texttt{nb}_{\texttt{cpu}}$ and $\texttt{nb}_{\texttt{gpu}}$ as in the previously presented mono-stream case. We tested {1,2,4} streams on the GPU referred to as #s in the Table. Finally we selected the values resulting in the lowest times for the factorization.

	12 CPUs (1×E5-2680) + 1 GPU (1×K40M) multistream				
Mat.	nb_{gpu}	nb_{cpu}	# s	Time (s.)	Gflop/s
12	256	256	4	7.683E+00	205.349
13	512	256	4	7.935E+00	368.321
14	384	384	4	1.539E+01	365.383
15	768	256	4	1.748E+01	616.156
16	896	128	2	2.470E+01	636.492
17	256	256	2	9.766E+01	229.609
18	512	256	4	3.882E+01	710.208
19	768	256	4	9.610E+01	661.302
21	512	256	2	3.024E+02	619.949
	24 CPUs (2×E5-2680) + 1 GPU (1×K40M) multistream				
Mat.	nb_{gpu}	nb_{cpu}	# s	Time (s.)	Gflop/s
12	192	192	4	6.206E+00	236.649
13	256	256	4	8.117E+00	360.062
14	512	256	2	1.434E+01	374.022
15	768	128	2	1.839E+01	569.094
16	896	128	4	2.558E+01	614.596
17	256	256	2	7.326E+01	306.083
18	512	256	4	3.838E+01	718.350
19	768	384	2	8.142E+01	797.947
21	768	384	2	2.466E+02	770.882

Table 6.4: Optimum performance for the STF hierarchical-grain 1D factorization on Sirocco in heterogeneous case with both configurations 12 CPUs (1×E5-2680) + 1 GPU (1×K40M) and 24 CPUs (2×E5-2680) + 1 GPU (1×K40M).

A comparison of the optimal parameters in both mono-stream and multi-stream cases shows that when using multiple streams, the best results are generally obtained for smaller values of $\texttt{nb}_{\texttt{cpu}}$ and $\texttt{nb}_{\texttt{gpu}}$ than in the mono-stream case. For example for matrix #12 with a 24 CPUs and 1 GPU, the optimal parameters $\texttt{nb}_{\texttt{gpu}} = \texttt{nb}_{\texttt{cpu}} = 384$ become $\texttt{nb}_{\texttt{gpu}} = \texttt{nb}_{\texttt{cpu}} = 192$ when using multiple stream on the GPU. Similarly for matrix #14 the optimal parameters $(\texttt{nb}_{\texttt{gpu}}, \texttt{nb}_{\texttt{cpu}})$ = (768,256) become $(\texttt{nb}_{\texttt{gpu}}, \texttt{nb}_{\texttt{cpu}})$ = (512,256)

when using multiple stream on the GPU. This happens because the smaller block sizes deliver better concurrency and, despite the smaller granularity of tasks, a good GPU occupancy is achieved thanks to the use of multiple streams. Using the Gflop/s rates presented in Table 6.4 for the and in Table 6.3, we compare in Figure 6.9 performance obtained between the mono-stream and multi-stream configurations.

6.4.3 Comparison with a state-of-the-art GPU solver

In this section we provide a comparison of our solver with the GPU-enabled version of the `spqr` solver [118] briefly discussed in Section 1.6. This solver is specifically designed for a GPU-only execution where all the operations (linear algebra kernels as well as assemblies) are executed on the GPU and one core is used to drive the activity of the GPU through a technique referred to as *bucket scheduling*.

#	Factorize time (s)	
	`qr_mumps`	`spqr`
2	1.661E+00	3.210E+00
5	4.164E+00	5.469E+00
6	9.084E+00	*
24	8.893E+00	1.493E+01
7	8.585E+00	1.874E+01
8	1.652E+01	2.254E+01
9	2.834E+01	*
11	6.001E+01	**

Table 6.5: Factorization time for the test matrices with `qr_mumps` (24 CPU cores and 1 GPU) using multiple streams. On the last column, the factorization times for the `spqr` solver. $*$ means that the solver returned an erroneous solution and $**$ means that the memory requirement for these matrices exceeded the GPU memory size.

A comparison of the execution time of `qr_mumps` and `spqr` is shown in Table 6.5. In both cases a COLAMD ordering is applied resulting in the same amount of flop during factorization. The performance results show that, despite the additional logic needed to exploit both the GPU and the CPUs and to processes problems that require higher memory than available on the GPU, our solver achieves a better performance than `spqr` on tested problems. This is essentially due to the extra power provided by the CPUs that `qrm_starpu` is capable of using. It must be noted that `spqr` could not factorize matrix #11 because of memory consumption issues. As explained in Section 1.6, `spqr`-GPU is a GPU-only code and, in its current implementation, all the data is allocated on the GPU memory. Because the memory needed to process the entire elimination tree may greatly exceed the memory available on the GPU, the elimination tree is statically split into stages so that each stage fits in the GPU memory. The minimum size of a stage, however, is a *family*, i.e., a front and the contribution blocks from its children. In the tree of large size problems, there may be one or more families whose size exceeds the GPU memory; the current implementation of `spqr`-GPU is thus not capable of solving these problems. Our solver, instead, always runs to completion as long as enough memory is available on the host and will be capable of using the GPU for all tasks whose memory footprint does not exceed the GPU memory size. Note that the memory footprint for inner updates is two inner block-columns and for outer updates is one inner and one outer block-columns and therefore we expect that

most of the tasks will fit in the GPU memory unless the size of the frontal matrices is extremely high.

In two cases (matrices #6 and #9) the `spqr` solver returned an erroneous solution.

6.5 Possible minor, technical improvements

The code we implemented and that was used to produce the experimental results presented above was principally developed with the objective of validating and assessing the effectiveness of the front partitioning and tasks scheduling methods discussed in the previous sections. There are, nonetheless, a number of minor technical improvements that can be applied to the code and that are likely to bring substantial benefits to its performance and scalability. Although we plan to pursue these developments in future work, we briefly discuss them in this section.

- **GPU memory allocation**. In our current implementation, frontal matrices are always allocated on the host memory, which implies that the data has to be transferred to the GPU before being processed by GPU tasks. In some specific cases it should be possible to allocate the memory directly on the device which allows for saving communications. This is, for example, the case when `do_subtree` tasks are executed on the GPU.

- **Memory pinning**. Memory pinning enables faster data transfers between the host and the device. Although pinned memory allocations have a higher latency, this is proven to be extremely beneficial for the factorization of a large dense matrix. However, in the case of our code, which uses block storage, many, small size allocations are executed for each frontal matrix, which renders the pinning overhead unbearable. Moreover, because all the allocations are done by the master thread, this overhead slows down the submission of tasks. It should be possible to perform pinned allocations selectively only on those fronts which are more likely to be processed on the GPU, or to defer the allocation to the `init` task in order to prevent the overhead from slowing down the submission of tasks.

- **Staircase-aware GPU kernels**. As explained in Section 6.3, `qrm_starpu` currently relies on MAGMA routines for performing inner and outer updates. As a result, we are not fully capable of taking advantage of the fronts staircase structure. Implementing kernels that can exploit the staircase would readily result in a performance improvement.

- **GPU panel factorization and assemblies**. At the moment we do not have a GPU implementation for the `_geqrt` and `assembly` tasks. Although these operations are likely to have small acceleration factors relative to the updates, they can still be run faster on the GPU and, moreover, being capable of computing them on the device would allow us to save some data transfers.

- **Efficient subtrees**. The performance of GPU `do_subtree` tasks can be improved by employing approaches like the one used for `spqr`-GPU or SSIDS (see Section 1.6).

Chapter 7

Associated work

7.1 PTG multifrontal QR method

In the previous chapters we presented the parallelization of a multifrontal method based on an STF model and implemented with the StarPU runtime system. In this section we propose an alternative approach using a PTG model presented in Section 1.4.1. This parallel version of the multifrontal method is implemented with the PaRSEC runtime system introduced in Section 1.4 and referred to as `qrm_parsec`. As explained in Section 1.4.1, contrary to the STF model where task dependencies are inferred by the runtime system, the PTG model requires the explicit declaration of all the dependencies in the DAG to the runtime system. In the PaRSEC runtime system, this is done through a specific language called JDF. This property allows great flexibility in the expression of dependencies and prevents the occurrence of unnecessary dependencies in the computed DAG as is the case for the assembly tasks in the implementation described in Chapters 3, 4, 5 and 6. On the other hand, the expression of the DAG for the multifrontal QR factorization represents a challenge due to the complexity of its dependency pattern and because it is only known dynamically at runtime.

A simple approach for the implementation of the PTG-parallel factorization consists in expressing the whole DAG in a single JDF. This solution however raises several problems. First the dependency pattern associated with the factorization of the frontal matrices and the assembly operations is dynamically known during the tree traversal and PaRSEC does not currently support this property. This approach is therefore not suited and as a consequence prevents us from using inter-level parallelism as we show below. It should be noted that this is only a technical limitation of the runtime system and not of the programming model. Regarding the PTG model, it is conceivable to represent a DAG which is dynamically built as it is unrolled during execution. The second reason for choosing another approach is that the complexity of the dependency pattern increases with the amount of tasks in the DAG. In the proposed implementation of `qrm_parsec` we use a simpler approach based on hierarchical DAGs. We consider a two-level hierarchy with an outer DAG and multiple inner DAGs spawned by the tasks of the higher level DAG: the outer DAG contains tasks related to the activation, deactivation, initialization and assembly of the nodes of the elimination tree whereas each inner DAG contains all the tasks for factorizing the related front. This approach is illustrated in Figure 7.1 where three different DAGs denoted by 1, 2 and 3 are spawned by tasks in the outer DAG. Note that this approach is equivalent to the one used in the UHM solvers presented in Section 1.6.

The PaRSEC implementation of the multifrontal QR factorization in our solver is split into several JDF files:

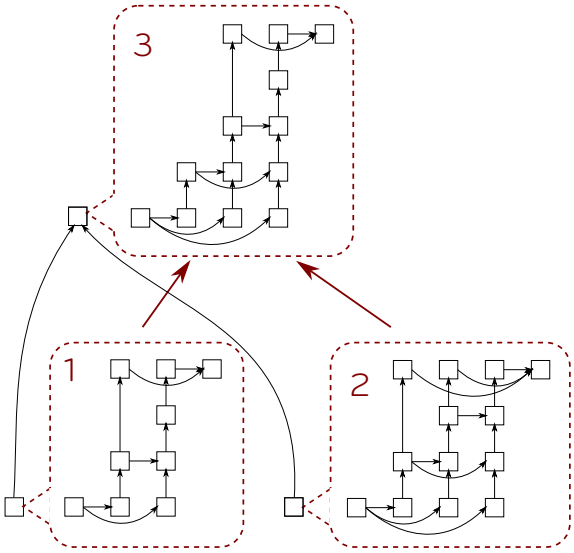


Figure 7.1: Two levels hierarchical DAGs implemented in PaRSEC. The inner DAGs are spawned by tasks contained in the top level DAG.

- `factorization.jdf`: the JDF representing the factorization DAG operating at the elimination tree level. This file regroups the tasks applied on the nodes of the elimination tree which are `do_subtree`, `activate`, `init`, `deactivate` and `assemble`. The `init` task is responsible for the initialization of a node and for instantiating the DAG for the factorization of this node whose definition is contained in another JDF. Similarly the `assembly` task is only responsible for the creation of the DAG for the assembly operations. The completion of these two tasks is achieved only when all the tasks in the DAG they instantiated have been executed;

- `qr_1d.jdf, qr_2d.jdf`: the JDF files for the frontal matrix factorization with 1D, block-column and 2D, tile partitioning, respectively. These implementations are based on the one found in the DPLASMA library [28] which provide dense linear algebra kernels routine for distributed systems built on top of the PaRSEC runtime systems. We adapted these kernels to the specific staircase structure of frontal matrices presented in Section 1.2.3;

- `assembly.jdf`: the JDF for the assembly operations. In the DAG instantiated by this JDF, each task corresponds to the assembly of a block from all the blocks in children frontal matrices contributing to it. Note that in order to express the dataflow for these assemblies, we need to compute, for every block in a frontal matrix, a list of contributing blocks in children node. This mapping is computed upon front activation and is not required when using a STF model.

Note that this approach does not allows us to finely express the dependencies between the tasks contained in inner DAGs. As shown in Figure 7.1 with dependencies between DAGs 1 and 3, and, 2 and 3, this approach limits the task pipelining because tasks in DAG 3 wait for the completion of all tasks in DAGs 1 and 2. Therefore, it is not possible to exploit *inter-level parallelism* with this strategy. As highlighted in Section 1.7.1 and in the context of `qrm_starpu` in Section 4.2.3 and Section 4.3.2 the exploitation of inter-level parallelism concurrency plays an important role in the scalability of the multifrontal method.

The JDF representation of a QR factorization with a 1D block-column partitioning is presented in Figure 7.2. This JDF is similar to the DPLASMA implementation except that we used the _geqrt and _gemqrt kernels described in Section 1.7, respectively for the panel and update operations, capable of exploiting the staircase structure of block-columns. The task types for the panel and update operations are respectively GEQRT and GEMQRT. The GEQRT tasks are associated with the parameter p corresponding to the panel index. This parameter has values in the range 0..NP-1 where NP represents the number of panel operations in the front. The GEMQRT task is defined by two parameters. The first represents the panel operations and the second represents the subsequent update operations depending on each panel operation. For each panel operation p we perform update operations on block-columns p+1..NC-1. The GEQRT tasks produce the data V and T which are matrices resulting from the panel factorization. These data are sent to the corresponding update tasks which are GEMQRT(p, p+1..NC-1) for a given panel p. Concerning the GEMQRT tasks, for a given a block-column u, it retrieves the V and T matrices of the corresponding panel p along with the block-column issued by the update with respect to the previous panel task denoted GEMQRT(p-1,u). Once the update operation has been executed, the block-column is sent either to the next update operation denoted GEMQRT(p+1,u) or to the panel operation denoted GEQRT(u) if the block-column is up-to-date.

Figure 7.3 shows a portion of the JDF code for the frontal matrix factorization with 2D partitioning with the description of task type GEQRT. For the sake of clarity, the description of other task types is not presented. Similarly to the 1D case, the proposed implementation is based on the JDFs implemented in DPLASMA. However, because of the staircase structure of frontal matrices and the 2D partitioning, the data-flow is significantly modified compared to the original implementation. As opposed to the 1D case, the data-flow associated with GEQRT tasks depend on the frontal matrix staircase structures. In addition, as pointed out in Section 4.3, the blocks we consider are not necessarily square and this flexibility that we allow increases the number of cases to take into account in the expression of the data-flow. As we already discussed in Section 1.3.1, in the 2D QR factorization, the upper triangular part of the diagonal tile is used to annihilate the other tiles in the same column. Therefore after the kth GEQRT, the tile is sent to a TPQRT task associated with the sub-diagonal task in row i. However it is possible that this tile does not exist because of the staircase structure of the frontal matrix. For this reason the data-flow edge associated with the upper triangular part of the tile denoted R_kk is conditioned with (i < lct) ? R_kk TPQRT(i+1,k).

The experimental results for the PaRSEC version of our solver are presented in Figure 7.4. These results show the scalability of qr_parsec and qr_starpu on the Dude system equipped with 24 cores using both the 1D and 2D factorization algorithms. These results illustrate the efficiency of the 2D algorithm compared to the 1D algorithm already observed in Chapter 4. Although qr_parsec offers good scalability, for every tested matrices it is lower than the scalability obtained with qrm_starpu. As we mentioned above this difference lies in the fact that we are unable to exploit inter-level parallelism in qr_parsec. The importance of this on the amount of concurrency in the factorization is detailed in Section 4.3.2.

```
GEQRT(p)

p = 0 .. (NP-1)

  : A(0,p)

  RW A_p <- (p==0) ?  A(0,p) : C_u GEMQRT(p-1, p)
         -> (p < NC-1) ? V_p GEMQRT(p, (p+1)..(NC-1))
         -> A(0,p)

  RW T_p <- T(0, p)                                          [type = LITTLE_T]
         -> (p < NC-1) ? T_p GEMQRT(p, (p+1)..(NC-1))     [type = LITTLE_T]
         -> T(0,p)

  ;NC

BODY
    {

      _geqrt_stair(&m, &n, &ib,
                   &stair[off], &off,
                   A_p + off, &lda,
                   T_p, &ldt,
                   work, &info);

    }
END

GEMQRT(p, u)

  p = 0..(NP-1)
  u = (p+1)..(NC-1)

  : A(0,u)

    READ  V_p <- A_p GEQRT(p)
    READ  T_p <- T_p GEQRT(p)                                [type = LITTLE_T]
    RW    C_u <- (p==0) ? A(0,u) : C_u GEMQRT(p-1, u)
              -> ((u == p+1) && (u <= (NP-1))) ? A_p GEQRT(u)
              -> ((u > p+1) && (p < (NP-1))) ? C_u GEMQRT(p+1, u)

  ; (NC-u)

BODY
    {

      _gemqrt_stair("l", "t",
                    &m, &j, &k, &ib,
                    &stair[off], &off,
                    V_p + off, &ldv,
                    T_p, &ldt,
                    C_u  + off, &ldc,
                    work, &info);

    }

END
```

Figure 7.2: Code for the 1D block-column dense QR factorization with PaRSEC.

```
GEQRT(k)

  lt = inline_c %{ return ( ( MIN(M,N)-1 ) / NB ); %}

  k = 0 .. lt

  /* row index of the last tile in the previous column */
  lctp = inline_c %{ lct_f90(front, k-1) ; %}
  /* row index of the last tile in the current column */
  lct  = inline_c %{ lct_f90(front, k) ; %}

  /* row index of the first tile in the current column */
  ft = inline_c %{ return ( k*NB ) / MB; %}
  ftp = inline_c %{ return ( ( k-1 )*NB ) / MB; %}

  i = ft

  : A(i, k, front, qrm_mat)

  RW A_kk <- ( k == 0 ) ? A(i, k, front, qrm_mat)
          <- ( (k > 0) & ( ft == ftp) & ( i == lctp) ) ? A_kj GEMQRT(k-1, k)
          <- ( (k > 0) & ( ft == ftp) & ( i < lctp ) ) ? A_kj GEMQRT(k-1, k)
          <- ( (k > 0) & ( ft > ftp) & ( i <= lctp ) ) ? A_ij TPMQRT(k-1, i, k)
          <- ( (k > 0) & ( ft > ftp ) & ( i > lctp ) ) ? A(i, k, front, qrm_mat)
          -> ( i < lct ) ? R_kk TPQRT(i+1, k) : A(i, k, front, qrm_mat)
          -> V_kk pnl_kk_typechange(k)
          -> ( i == lct ) ? A(i, k, front, qrm_mat)

  RW T_kk <- T(i, k, front, qrm_mat)                          [type = LITTLE_T]
          -> T(i, k, front, qrm_mat)                          [type = LITTLE_T]
          -> ( k < (NC-1) ) ? T_kk upd_kj(k, (k+1)..(NC-1))   [type = LITTLE_T]

    ;NC*NC*NC

BODY
    {

      _geqrt_stair(&m, &n, &ib,
                   &stair[offa], &ofsa,
                   A_kk + ofs, &lda,
                   T_kk, &ldt,
                   work, &info);
    }
END
```

Figure 7.3: Code for the GEQRT task in the 2D dense QR factorization with PaRSEC.

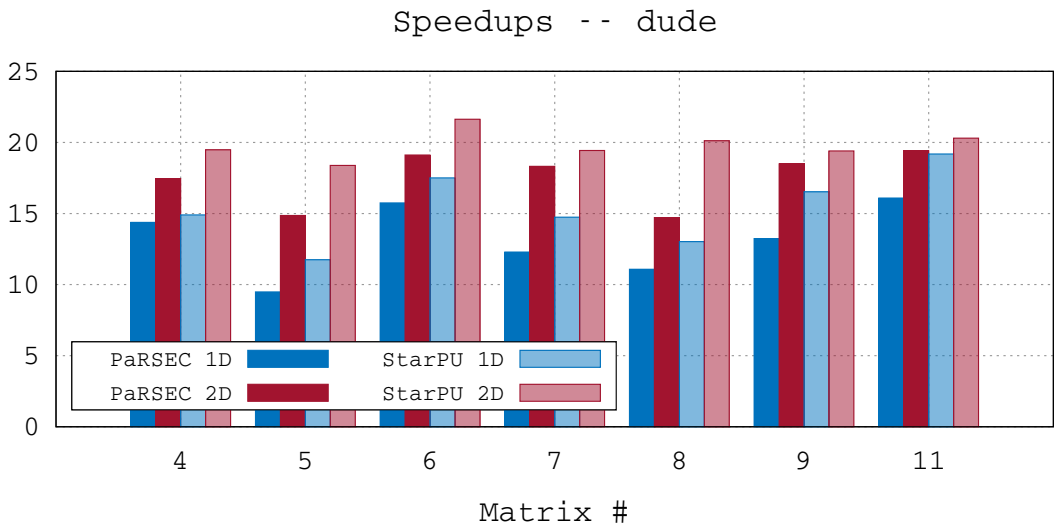


Figure 7.4: Speedup for the qr_starpu and qr_parsec on dude system (24 cores).

7.2 Simulation of qrm_starpu with StarPU SimGrid

The development of scientific software that aims to be portable across different architectures requires a considerable amount of performance analysis and tuning. This is a very challenging task to achieve for large and complex codes which feature an irregular workload and which are designed for modern computing platforms equipped with heterogeneous processing units. This is even more difficult if one considers that access to computing platforms with the desired characteristics may be limited if at all possible. In the case of runtime-based software, this issue can be addressed by taking advantage of the fact that the runtime has full control over the execution of the tasks in which the workload is decomposed: the runtime system can be instructed not to actually execute the tasks, but instead to simulate their execution reproducing their resource consumption (be it PU time or memory) on the target architecture. Because the tasks are not actually executed, and therefore the resources are not actually consumed, this technique allows the simulation of the execution of very large workloads on large and complex architectures even on systems with a relatively limited amount of resources like a laptop computer.

Researchers from the MESCAL project at the LIG Laboratory of Grenoble have realized a coarse-grain hybrid simulation/emulation of StarPU applications [107] on top of SimGrid[36], a simulation toolkit specifically designed for distributed system simulation. In this approach the application (i.e., the work of the master thread) and runtime are *emulated* as their actual code is executed and the tasks submission and scheduling are therefore done as in a real execution; tasks, however, are not actually executed but *simulated*. An initial calibration of the target machine is run first to derive performance profiles and models (interconnect topology, data transfers, computation kernels) that are given as input to SimGrid. Subsequent simulations can then be performed in a reproducible way on personal commodity laptops without requiring any further access to the target machines.

This approach was proved capable of achieving accurate performance prediction of two dense linear algebra algorithms on a number of heterogeneous systems [108], yet its effectiveness on more irregular and heterogeneous workloads had to be assessed. For this purpose, we collaborated [0] with the authors of the StarPU-SimGrid platform to evaluate the accuracy of this simulation engine on the `qrm_starpu` solver, precisely on the 1D STF variant presented in Section 4.2. Compared to the abovementioned study on the simulation of dense linear algebra applications with StarPU and SimGrid, the main difficulty arises from the application structure and from the larger variety of computation tasks or kernels called with a wide variety of input parameters. When working with dense matrices, it is common to use a global fixed block size and a given kernel type (e.g., `_gemm`) is therefore always called with the same parameters throughout the execution, which makes its duration on the same processing unit very stable and easy to model. In the `qrm_starpu` factorization, the amount of work that has to be done by a given kernel greatly depends on its input parameters. These parameters may or may not be explicitly given to the StarPU runtime and we thus had to rework the `qrm_starpu` task submission model to ensure StarPU can propagate this information to SimGrid. The modelling of the different kernels used in `qrm_starpu` was done using the following assumptions:

- `_geqrt`: this kernel is a simple wrapper around the corresponding LAPACK routine and thus it can be completely characterized by the size $m \times n$ of the block it operates on. This information is readily available to StarPU through the task parameters `nb` (the column width), `mb` (the block-column height) and `bk` (the front factorization stage or panel number). Given that $m = mb - (bk - 1) \times nb$ and $n = nb$, the

complexity of this kernel is

$$T_{geqrt} = a + 2b(nb^2 \times mb) - 2c(nb^3 \times bk) + \frac{4d}{3}nb^3,$$

where a, b, c and d are machine and memory hierarchy dependent constant coefficients.

- _gemqrt: The duration of the _gemqrt kernel also depends on the geometry of the data it operates upon, defined by the same mb, nb, and bk parameters. This kernels simply wraps the LAPACK _gemqrt routine which applies k Householder reflections of size $m, m-1, ..., m-k+1$ on a matrix of size $m \times n$ where m, n and k are equal to $mb - (bk-1) \times nb$, nb and nb, respectively. Therefore, its *a priori* complexity is defined as:

$$T_{gemqrt} = a' + 4b'(nb^2 \times mb) - 4c'(nb^3 \times bk) + 3d'nb^3$$

- init: the execution time of this kernel depends on the number of coefficients in the fronts (because they have to be initialized to zero) #Zeros and the number #Aasm of coefficients from the input sparse matrix that have to be assembled in the front. These parameters had to be added to the signature of the task submission routine in order to make their value available to StarPU.

- assemble: the execution time of tihs kernel solely depends on the number #Casm of coefficients from the block-column that have to be assembled into the parent front. This parameter had to be added to the signature of the task submission routine in order to make its value available to StarPU.

- do_subtree: the execution time of this kernel mostly depends on the number of flops performed therein and on the number of fronts in the subtree (this gives a measure of the granularity of operations and thus their efficiency). These parameters had to be added to the signature of the task submission routine in order to make their value available to StarPU. Note that this data is computed during the analysis phase and thus already available.

- deactivate: this kernel is considered negligible as it only acounts for less than 1% of the total execution time.

To evaluate the quality of our approach, we used two different kinds of nodes from the Plafrim[1] platform. The *fourmi* nodes feature 2 Quad-core Nehalem Intel Xeon X5550 with a frequency of 2.66 GHz and 24 GB of RAM memory and the *riri* nodes comprise 4 Deca-core Intel Xeon E7-4870 with a frequency of 2.40 GHz and 1 TB of RAM memory. The *fourmi* nodes proved to be easier to model as their CPU architecture is well balanced with 4 cores sharing L3 cache on each of the 2 NUMA nodes. Such a configuration leads to little cache contention. However, the RAM of these nodes is limited and thereby limits the matrices that can be factorized to a certain size. Although the huge memory of the *riri* machine puts almost no restriction on the matrix choice, its memory hierarchy with 10 cores sharing the same L3 cache lead to cache contention that can be tricky to model.

The execution time of a single kernel on a certain machine greatly depends on the machine characteristics (namely CPU frequency, memory hierarchy, compiler optimization, etc.). Obtaining accurate timing is thus a critical step of the modelling. To predict the

[1] https://plafrim.bordeaux.inria.fr/

performance of the factorization of a set of matrices on a given experimental platform, we first benchmark the kernels identified in the previous section.

For the kernels that have clear dependency on the matrix geometry (`_geqrt` and `_gemqrt`), we wrote simple sequential benchmarking scripts, that pseudo-randomly choose different parameter values, allocate the corresponding matrix and finally run the kernel, capturing its execution time. However, for kernels (`do_sutree`, `init` and `assemble`) whose code is much more complex and depends on many factors, including even dependencies on previously executed tasks, creating a simplistic artificial program that would mimic such a sophisticated code is very difficult. Since each sparse matrix has a unique structure, the corresponding DAG is very different and the kernel parameters (such as height and width) greatly vary from one matrix factorization to another. Consequently, it is very hard to construct a single linear model that is appropriate for every use cases. The inaccuracies caused by such model imperfection can produce either underestimation or overestimation of the kernel duration and thus of the whole application makespan as well. Therefore, to benchmark such kernels we rely on traces generated by a real `qrm_starpu` execution (possibly on different matrices than those that need to be studied) instead of a careful experiment design.

The result of this benchmark is analyzed with R to obtain linear models that are then provided to the simulation. Table 7.1 presents a summary of the prediction quality for each kernel as well as the minimal number of parameters that have to be taken into account for all the tasks in the factorization of matrix #7 on the fourmi system. For all of them, the adjusted R^2 value[2] is close to 1, which indicates an excellent predictive power.

	`_geqrt`	`_gemqrt`	`do_subtree`	`init`	`assemble`
1.	nb	nb	#Flops	#Zeros	#Casm
2.	mb	mb	#Nodes	#Aasm	/
3.	bk	bk	/	/	/
R^2	0.99	0.99	0.99	0.99	0.86

Table 7.1: Summary of the modeling of each kernel based on matrix #7 on fourmi.

At each step of the regression, we check that the models are adequate through a careful inspection of the regression summaries and of the residual plots. These models are then linked with the simulator and the experimental platform is then no longer of use as `qrm_starpu` can be run in simulation mode on a commodity laptop. Using a recent and more powerful machine only improves the simulation speed and possibly allows for running several simulations in parallel. In such simulations, we recall that the code of `qrm_starpu` and of StarPU is run for real (the application and the runtime are emulated) but all computation intensive and memory consuming operations are faked and converted into simple simulation delays. SimGrid is used for managing the simulated time and the synchronization between the different threads.

Figure 7.5 tracks the execution of the `_geqrt` kernel and indicates the duration of the tasks at each time. To ease the correspondence between the real execution and the simulation, the colour of each point is related to the job id of the task. Both colours and the pattern of the points suggest that the traces match quite well. Even though the scheduling is not exactly the same, it is still very close. Similar analysis have been performed for all the other kernels as well and the results were very much alike.

[2]The coefficient of determination, denoted R^2, indicates how well data fit a statistical model and ranges from 0 to 1. An R^2 of 0 indicates that the model explains none of the variability of the response data around its mean while an R^2 of 1 indicates that the regression line perfectly fits the data.

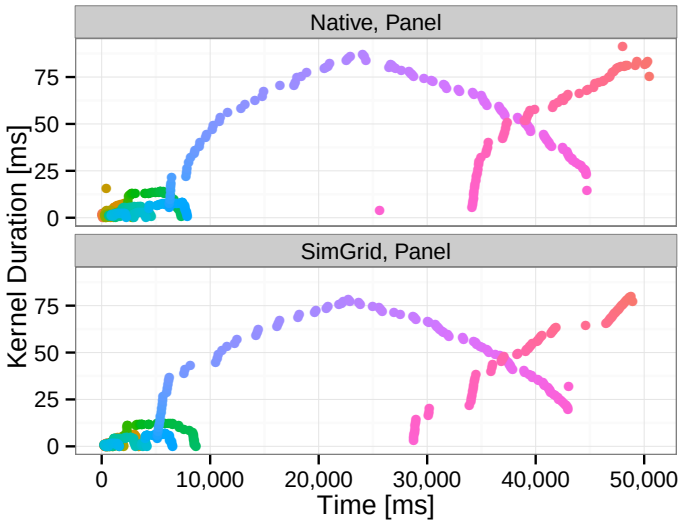


Figure 7.5: Native vs simulated execution time of _geqrt tasks in the factorization of matrix #7 as a time sequence on the Fourmi system. Colour is related to the task id.

Figure 7.6 shows the ratio between the simulated makespan and the makespan of an actual execution on the Fourmi and Riri systems. On the first system the simulation is extremely accurate since the error is, on average, around only 3%; on this machine the actual execution time for matrices #9, #10 and #11 could not be measured because of the limited available memory. This obviously reveals one advantage of the simulation platform which allows for evaluating the performance of a code or an algorithm on a hypothetical architecture with larger memory than is actually available. On the Riri system the simulation is not as accurate as the error is, on average, around 8.5% when ten cores are used. This is mostly due to the pressure on the L3 cache which is shared by all the ten cores on a socket; the simulation engine is not currently capable of adjusting the performance model to this effect. On 40 cores the error increases again but the results can still be considered accurate and useful for the purpose of performance profiling and evaluation. In all cases the simulated makespan is systematically a slight underestimation of the actual execution time as our approach ignores the runtime overhead and a few cache effects.

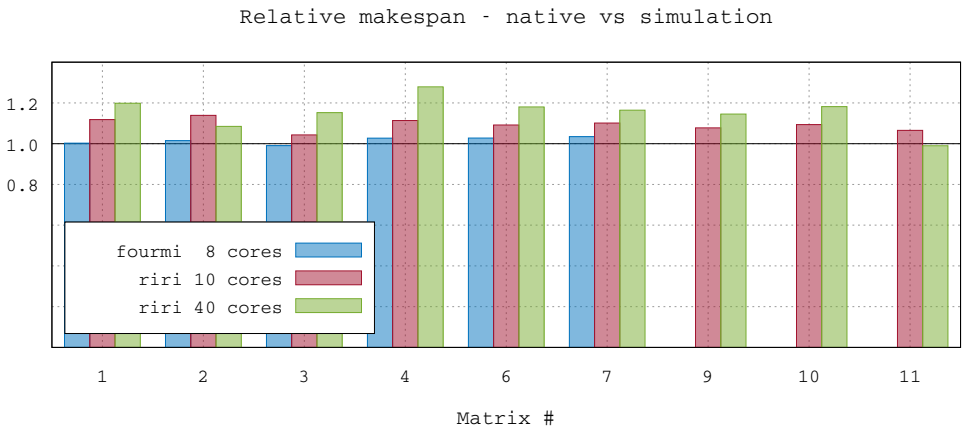


Figure 7.6: Native vs simulated makespan on the Fourmi system

By the same token, the StarPU-SimGrid simulation engine can be used to reproduce the memory consumption of a StarPU based code. Because in the simulation the tasks are not actually executed, the simulation engine cannot directly track the memory consumption (for example, by intercepting the calls to allocation and free routines). For this reason it was necessary to provide StarPU with the memory consumption associated to the different tasks. In `qrm_starpu` the only three tasks that allocate or free memory are the `init`, `do_subtree` and `deactivate`. A parameter whose value corresponds to the memory consumption of the task was added to the submission routine for these three types. Figure 7.7 presents a comparison between the simulated memory consumption against the actual memory consumption in three consecutive runs for matrix #6; yellow and blue show the memory allocated within the `do_subtree` and `init` tasks, respectively. The figure clearly shows that the simulation is quite accurate in estimating the peak memory consumption as well as the memory consumption throughout the whole factorization. It must be noted that, because the scheduling of tasks is completely dynamic, even two actual runs may differ slightly in their memory profile, as it can be seen in Figure 7.7.

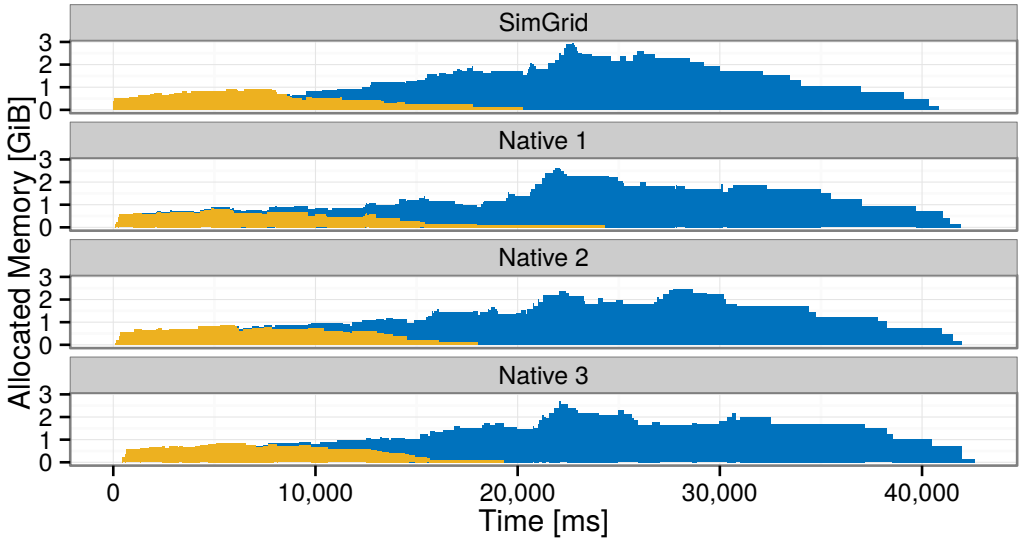


Figure 7.7: Native and simulated memory consumption for matrix #6 on the Fourmi system.

7.3 StarPU contexts in `qrm_starpu`

The memory system of modern multicore systems is typically arranged in a Non Uniform Memory Access (NUMA) configuration: the memory is physically split into several modules and each module is commonly associated with a socket or processor. All the modules share the same address space which means that every core in every processor can access transparently any data in any module, yet the speed of access to data may vary considerably depending on where the data is actually located. Access to remote data implies transfers from one socket to another whose costs are far from being negligible. Figure 4.13 measures this effect on task efficiency. The cost of these implicit transfers can be more or less expensive depending on the connectivity between the memory modules and in some cases can seriously harm the scalability of a multithreaded code.

The StarPU team has recently developed a feature that aims at addressing this issues through the use of *scheduling contexts* [71]. A scheduling context can be

> *defined as a structure able to encapsulate a parallel code and restrict its execution on a section of the machine. By means of the scheduling contexts, a parallel code runs on top of an abstract machine, whose set of resources can be dynamically changed during its execution. This allows the programmers to control the distribution of computing resources (i.e. CPUs and GPUs) over co-executing parallel kernels* (Hugo [72])

Scheduling contexts can be used for efficiently composing StarPU based applications. Imagine a scenario where two applications have to be executed concurrently on the same machine. Two contexts can be created, one for each application, and each of them can be assigned an amount of resources (e.g., processing units) proportional to the relative weight of the associated application; all the tasks related to one application will be submitted to, and handled by the corresponding scheduling context and, as a consequence, executed by the associated resources. This will prevent the two applications from interfering with each other (for example polluting each others caches) and from competing for resources.

Scheduling contexts were later used to improve the scalability of `qrm_starpu` on NUMA architectures by Hugo et al. [70]. This idea stems from the observation that separate branches of the elimination tree are independent and can thus be assigned to different contexts including disjoint subsets of resources whose size is established depending on their relative weight. Resources within each context share some level of memory (e.g., cache or NUMA module), which allows the reduction the cache miss rate or the transfer of data among NUMA modules and thus improves performance.

The analysis phase of the `qrm_starpu` solver has been extended with a proportional mapping [92] step. This algorithm consists in a top-down traversal: all the available resources are mapped on the root node of the elimination tree and then they are partitioned into *bundles* and each bundle assigned to a set of child nodes. The amount of resources in each bundle depends on the relative weight of the subtrees rooted at the child nodes. This process is applied recursively until all the bundles contain a minimum, non divisible amount of resources (for instance, one core). The weight of each subtree can be computed as the time needed to process the entire subtree, according to a given performance model. Each bundle is then mapped to context. This results in a tree of contexts which matches the memory hierarchy as shown in Figure 7.8: all the resources in a context/bundle share some level of memory (say, L2 cache) and all those in the higher level context, obtained by merging all the contexts in a set of siblings, share a lower level memory (L3 cache).

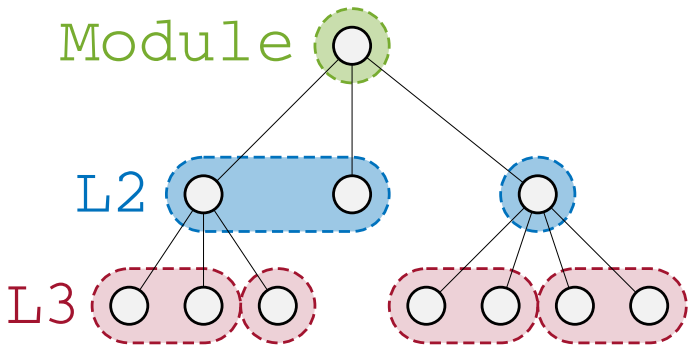


Figure 7.8: Mapping of tree nodes to contexts.

Note that the initial mapping of resources may be inaccurate due to inaccuracies of the performance model or because it cannot take into account conditions that are related to the load of the machine. In order to address this issue a *hypervisor* process constantly monitors

the execution of the factorization collecting information about its progress and updating the performance model. Whenever the hypervisor detects that the actual execution has excessively (according to a tolerance parameter) deviated from the ideal processing speed, it triggers a resource reallocation: the resources assigned to a set of sibling contexts are redistributed as established by the solution of a linear program that aims at minimizing the time to process the subtrees within each of these contexts. This reallocation is applied to all the yet unprocessed levels of the tree. Note that the fact that the resources are only redistributed within a set of siblings allows for maintaining a good data locality.

The use of contexts has been integrated in the `qrm_starpu` variant described in Section 3 and evaluated on the test matrices listed in Table 7.2. These matrices were column-permuted with the METIS v5.0.2 ordering tool and the tests were run on a ccNUMA platform equipped with 8 Intel E7-8837 processors having 8 cores clocked at 2.67 GHz and having 24 MB of L3 cache for a total of 64 cores. The platform is equipped with 300 GB of memory organized in groups of 100 GB each interconnected with a relatively slow memory bus.

#	Mat. name	m	n	nz	op. count (Gflops)
1	TF15	7742	6334	80057	93.90
2	tp-6	142752	1014301	11537419	381.82
3	pre2	659033	659033	5834044	777.67
4	esoc	37830	327062	6019939	891.58
5	Rucci1	1977885	109900	7791168	5316.94
6	ultrasound80	130228	130228	2032536	64777.40
7	conv3d64	836550	836550	12548250	108491.50

Table 7.2: Matrices test set. The operation count is related to the matrix factorization with METIS column permutation. All the matrices are from the University of Florida collection except the conv3d64 which is provided by CEA-CESTA and the ultrasound80 (Propagation of 3D ultrasound waves) provided by M. Sosonkina.

Figure 7.9 plots the execution times for the contexts-based variant relative to the standard implementation. The figure shows that the use of contexts brings a considerable benefit to the scaling of the `qrm_starpu` factorization due to a much better locality of reference to data. This was also assessed by explicitly measuring the number of local accesses to data. Figure 7.10 plots, for matrices #5 and #7 on 32 cores, the fraction of fronts with respect to the percentage of related tasks that are executed locally. For example, for the Rucci1 matrix, the figure shows that for roughly 65% of the fronts, 90 to 100% of the related tasks have been executed on the NUMA module where the fronts are actually stored. For both matrices, this figure shows that the use of contexts is effective in reducing the amount of data transfers between NUMA modules and, therefore, in improving performance.

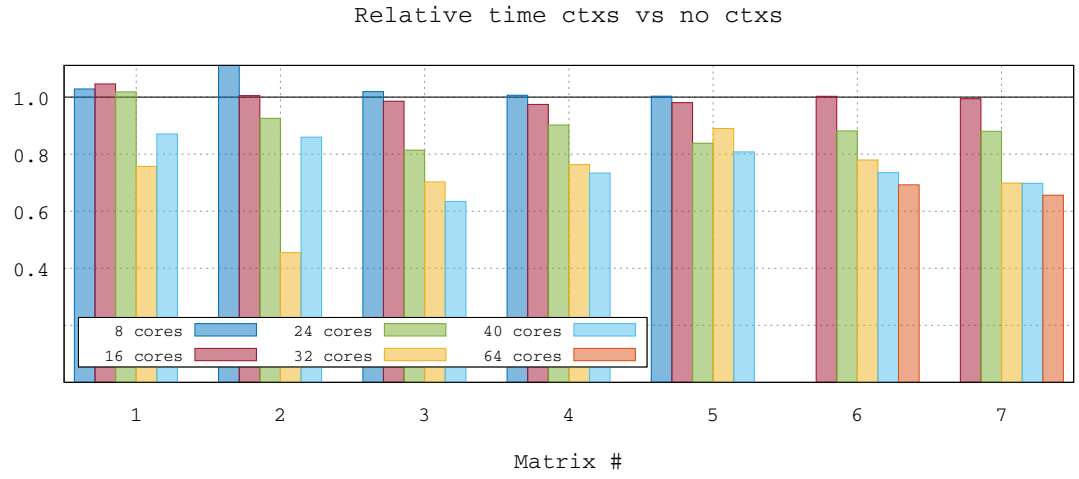


Figure 7.9: Relative execution times for the version with contexts.

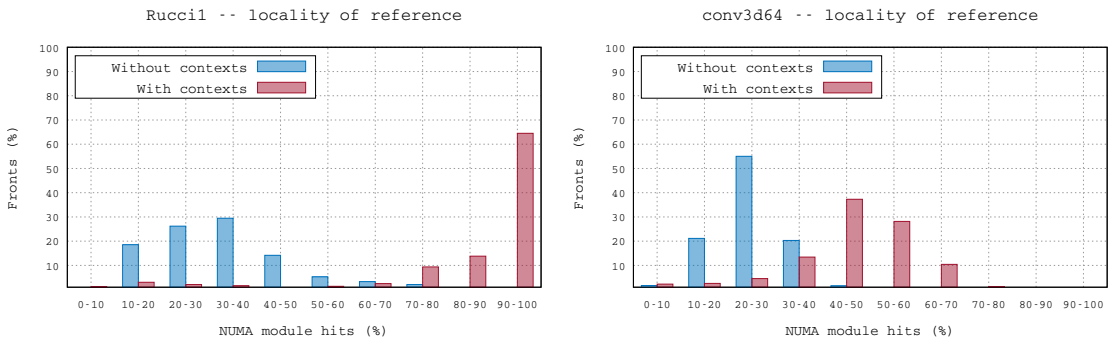


Figure 7.10: Locality of references with and without contexts.

Chapter 8

Conclusion

8.1 General conclusion

In this thesis we have addressed the design of a sparse direct solver, capable of efficiently exploiting multicore and heterogeneous architectures, using modern runtime systems.

The first issue we addressed in Chapter 3 is to prove the relevance of our approach by porting the multifrontal method implemented in `qr_mumps` on top of StarPU. In this chapter the StarPU based version of `qr_mumps`, referred to as `qrm_starpu`, is experimentally validated by performing a comparative study with the original solver which, essentially, assesses the validity of the approach and the usability of runtime systems for the implementation of sparse, direct methods.

Although capable of achieving good performance this implementation relies on a mixture of parallel programming models (see Section 1.4.1) which constitutes a hindrance to the development of new features. For this reason, we redesigned the implementation `qrm_starpu` using a pure Sequential Task Flow model. In Chapter 4, we first presented a STF-parallelization of the multifrontal method using the original 1D block-column partitioning scheme (the same as in the `qr_mumps` solver and in the approach described in Chapter 3), showed the expressiveness and simplicity of use of this parallel programming model and proved the efficiency of the resulting code. Based on this, we improved the solver by integrating 2D communication avoiding front factorization algorithms and show how this is relatively easy to implement, thanks to simplicity of the programming model. The superior performance and scalability of the resulting solver is assessed through a rich set of experimental results and a finely detailed performance analysis. Furthermore we developed and present in Chapter 5 a memory-aware algorithm which allows us to control the memory consumption of the parallel multifrontal method. We showed on a set of test matrices that we were able to obtain the same performance results as the non constrained case when enforcing a memory limit equal to the minimal memory consumption associated with the sequential algorithm.

In compliance with the STF parallel programming model, we moved forward to heterogeneous architectures in Chapter 6 and addressed the data partitioning and scheduling issues that are critical to achieve performance on these architectures. We extended the 1D block-column partitioning into a hierarchical block partitioning allowing to generate both fine and coarse granularity tasks suited to the processing unit capabilities. We develop a scheduling strategy capable of handling the task heterogeneity in the DAG and the diversity of resources on heterogeneous architectures. A performance analysis on a set of test matrices indicates that we are capable of efficiently exploiting heterogeneous architectures.

The validity and efficiency of all the methods presented in this thesis are assessed through experimental results. Considering the complexity of algorithms and computer architectures, this challenged us with the difficulty of profiling and analysing the performance of our implementations as well as the difficulty of identifying and quantifying potential sources of inefficiency. For this purpose, we developed a performance analysis approach, presented in Chapter 2, capable of measuring and separately analysing several factors that play a role in performance and scalability such as locality issues and task pipelining. Additionally, this analysis allows us to measure the cost of the runtime system and thus provides a further proof of the efficiency and usability of these tools.

Finally we presented in Chapter 7 some collaborative work done in the context of `qrm_starpu`. First we implemented another version of `qr_mumps` using an alternative programming model, namely a *parametrized task graph*. Then we focused on reproducing and simulating parallel executions in `qrm_starpu`. Lastly, we briefly presented the use of scheduling contexts with StarPU that enhances the exploitation of data locality in the multifrontal method.

Although our work clearly benefited from the features, the performance and the reliability of modern runtime systems, it must be noted that it also provided a valuable feedback to the developers of these tools. The results achieved by our work led to the development and implementation of novel features within the StarPU runtime system such as commutable tasks, dynamic partitioning/unpartitioning tasks and methods for guiding the submission of tasks based on the memory consumption.

8.2 Perspectives and future work

The work achieved in this thesis opens up a large number of opportunities for future developments both from an algorithmic (novel methods and algorithms for improving the performance and reliability of the solver) and from the software performance optimization points of view.

In our study on heterogeneous architectures, we presented experimental results on GPU-accelerated architectures. We believe that the strategies that we have developed to exploit these architectures can be used to exploit systems equipped with other types of accelerators such as the Intel Xeon Phi coprocessors. The techniques that we proposed are designed for heterogeneous architectures in general and thus are independent from GPUs. The algorithms that we presented are implemented using a high-level programming model while the runtime system handles the underlying architecture. Provided that we give the specific kernels to the runtime system, we should be able to exploit Xeon Phi-accelerated architectures.

Thanks to the runtime system approach, the current version of `qrm_starpu` is capable of exploiting multiple GPUs. However it is unlikely to have good performance because it does not address potential problems coming from GPU-to-GPU data transfers. Therefore, we need to address data locality issues by developing an appropriate scheduling policy.

In this thesis we focused on single node, multicore and heterogeneous architectures although modern runtime systems like StarPU and PaRSEC are capable of using distributed-memory architectures. Implementing the multifrontal QR factorization on such systems represents a hard challenge and requires special care in developing appropriate data distribution and mapping as well as tasks scheduling methods in order to minimize the cost of data transfers through the network that connects the nodes of the system.

The `qrm_starpu` solver developed in this thesis provides a reliable and efficient platform for algorithmic developments and opens up opportunities for developing novel numerical

methods with a relatively limited implementation effort.

The performance of the methods presented in this thesis, depends on the value of a number of parameters, most notably the block-sizes for either the 1D, 2D or hierarchical fronts partitioning. In the experimental results presented in the previous chapters, we simply tested a range of values for each of these parameters and only reported performance for the best cases. Nonetheless, it should be possible to automatically select the values that yield optimal or close to optimal performance depending on the properties of the workload (shape of the elimination tree and/or of the DAG, granularity of tasks, shape and size of the frontal matrices) and the machine (number and type of processing units, speed of memory, performance of BLAS routines). This is an extremely challenging task to achieve but may lead to a much better ease of use of the `qrm_starpu` solver. Moreover, it should be noted that the value of these blocking sizes does not have to be the same for all the frontal matrices in the elimination tree. For example, at the bottom of the tree, where much tree parallelism is available, node parallelism can be reduced by choosing larger block sizes or even a 1D partitioning in order to improve the efficiency of single tasks. By the same token, the panel reduction method in 2D factorization algorithms, may be chosen depending on the shape of every single front. This is clearly a very difficult challenge to tackle but may lead to considerable performance improvements.

One critical task we plan to pursue in future work is the extension of the `qrm_starpu` solver to support rank-deficient systems. Techniques like the Heath [63] method (also used in the `spqr` solver) can be employed to handle this type of problem and possibly be improved through the use of restricted pivoting techniques. It must be noted, though, that these techniques require special care in the task submission and scheduling. The detection of rank deficiency within a frontal matrix implies modifications of the structure of its ancestors (potentially up to the root of the elimination tree). Because rank deficiencies can only be detected during the matrix factorization, it is not possible to submit the whole DAG solely relying on the information from the symbolic analysis phase. The submission of some tasks is subordinated to the result of previous ones and therefore the DAG dynamically changes at runtime. Special strategies must be designed and implemented to cope with this issue.

Regarding the extremely good performance results obtained with our memory-aware algorithm in `qrm_starpu`, it should be noted that our approach is very conservative in terms of memory and may limit the exploitation of tree parallelism which could severely reduce the scalability of our solver on larger scale platforms. In addition, the proposed algorithm is especially suited to the multifrontal QR factorization for the reasons that we enumerated in our study and might not be as efficient in the context of other types of multifrontal methods. We could thus refine our algorithm with strategies such as the *memory booking* heuristics proposed by Eyraud-Dubois et al. [50].

This thesis essentially focused on the factorization of a sparse matrix. As explained in the introduction, this is only one of the three main phases of a sparse direct solver. The factorization is commonly followed by one or more solve operations with one or more right-hand sides. Although all the techniques presented in this thesis can readily be applied to the solve phase, it must be noted that in the case of a single right-hand side the solve phase becomes memory-bound. This means that the relative weight of tasks is very small (much smaller than in the factorization phase) and therefore the runtime overhead is likely to increase. A different parallelization scheme may have to be employed in this case to overcome this issue. As for the analysis, no floating-point computations are performed in the various steps of this analysis which are all, inherently, memory bound. Although this phase is relatively light with respect to the factorization and the solve, its weight becomes more significant as these are accelerated by exploiting parallelism. The use of runtime

systems for parallelizing the analysis phase may be challenging due to the nature of the operations performed therein.

Publications

Submitted articles

[S1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. *Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems*. Tech. rep. IRI/RT–2014-03–FR. Submitted to ACM Transactions On Mathematical Software. IRIT, Nov. 2014. URL: http://buttari.perso.enseeiht.fr/stuff/IRI-RT--2014-03--FR.pdf.

Conference proceedings

[C1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Multifrontal QR Factorization for Multicore Architectures over Runtime Systems". In: *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg, 2013, pp. 521–532. ISBN: 978-3-642-40046-9. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_53.

[C2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. *Task-based multifrontal QR solver for GPU-accelerated multicore architectures*. Tech. rep. IRI/RT–2015–02–FR. Accepted at the HiPC 2015 conference. IRIT, June 2015. URL: https://hal.archives-ouvertes.fr/hal-01166312v2.

Posters

[P1] E. Agullo et al. *Matrices Over Runtime Systems at Exascale*. Poster at the SuperComputing 2015 conference. 2015.

Conference talks

[T1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. *Multifrontal QR Factorization for Multicore Architectures over Runtime Systems*. Presentation at the Euro-Par 2013 international conference, Aachen August 26-30 2013. 2013.

[T2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. *Sparse direct solvers on top of a runtime system*. Presentation at the PMAA 2014 international conference, Lugano July 2-4 2014. 2014.

[T3] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. *Sparse direct solvers on top of a runtime system*. Presentation at the SIAM Computational Science and Engineering international conference, Salt Lake City, March 14-18 2015. 2015.

[T4] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. *Task-based multifrontal QR solver for GPU-accelerated multicore architectures*. Presentation at the Sparse Days in St Girons international conference, St Girons, June 29 - July 2 2015. 2015.

[T5] A. Decollas and F. Lopez. *Direct methods on GPU-based systems, preliminary work towards a functioning code*. Presentation at the Sparse Days workshop, Toulouse, June 2012. 2012.

References

[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, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet. *Robust memory-aware mappings for parallel multifrontal factorizations*. Submitted to SIAM SISC. 2012.

[3] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. "LU factorization for accelerator-based systems". In: *AICCSA*. 2011, pp. 217–224. DOI: http://dx.doi.org/10.1109/AICCSA.2011.6126599.

[4] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators". In: *Proceedings of 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. 2011, pp. 932–943. DOI: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2011.90.

[5] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. "A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs". In: *in GPU Computing Gems, Jade Edition* 2 (2011), pp. 473–484.

[6] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault. "Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms". In: *Heterogeneity in Computing Workshop 2015*. Hyderabad, India, May 2015. URL: https://hal.inria.fr/hal-01120507.

[7] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. *Task-based FMM for heterogeneous architectures*. Research Report RR-8513. Inria, Apr. 2014, p. 29. URL: https://hal.inria.fr/hal-00974674.

[8] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langou. "QR factorization of tall and skinny matrices in a grid computing environment". In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. Apr. 2010, pp. 1–11. DOI: 10.1109/IPDPS.2010.5470475.

[9] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012037. URL: http://stacks.iop.org/1742-6596/180/i=1/a=012037.

[10] E. Agullo, J. Dongarra, R. Nath, and S. Tomov. "Fully Empirical Autotuned QR Factorization For Multicore Architectures". In: *CoRR* abs/1102.5328 (2011).

[11] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[12] P. R. Amestoy, T. A. Davis, and I. S. Duff. "Algorithm 837: AMD, an approximate minimum degree ordering algorithm". In: *ACM Transactions On Mathematical Software* 33(3) (2004), pp. 381–388.

[13] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. "MUMPS: a general purpose distributed memory sparse solver". In: *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21*. Ed. by A. H. Gebremedhin, F. Manne, R. Moe, and T. Sørevik. Lecture Notes in Computer Science 1947. Springer-Verlag, 2000, pp. 122–131.

[14] P. R. Amestoy, I. S. Duff, and C. Puglisi. "Multifrontal QR factorization in a multiprocessor environment". In: *Int. Journal of Num. Linear Alg. and Appl.* 3(4) (1996), pp. 275–300.

[15] E. Anderson et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[16] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. "Communication-Avoiding QR Decomposition for GPUs". In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 48–58. ISBN: 978-0-7695-4385-7. DOI: 10.1109/IPDPS.2011.15. URL: http://dx.doi.org/10.1109/IPDPS.2011.15.

[17] K. Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. TECHNICAL REPORT, UC BERKELEY, 2006.

[18] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631. URL: http://hal.inria.fr/inria-00550877.

[19] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs". In: *Euro-Par*. 2009, pp. 851–862. DOI: 10.1007/978-3-642-03869-3_79.

[20] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. "Parallelizing dense and banded linear algebra libraries using SMPSs". In: *Concurrency and Computation: Practice and Experience* 21.18 (2009), pp. 2438–2456.

[21] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. "Communication lower bounds and optimal algorithms for numerical linear algebra". In: *Acta Numerica* 23 (May 2014), pp. 1–155. ISSN: 1474-0508. DOI: 10.1017/S0962492914000038. URL: http://journals.cambridge.org/article_S0962492914000038.

[22] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. "Minimizing Communication in Numerical Linear Algebra". In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), pp. 866–901. DOI: 10.1137/090769156. eprint: http://dx.doi.org/10.1137/090769156. URL: http://dx.doi.org/10.1137/090769156.

[23] Å. Björck. *Numerical methods for Least Squares Problems*. Philadelphia: SIAM, 1996.

[24] T. O. architecture review board. *OpenMP 4.0 Complete specifications*. 2013.

[25] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. "PaRSEC: Exploiting Heterogeneity to Enhance Scalability". In: *Computing in Science and Engineering* 15.6 (2013), pp. 36–45. DOI: 10.1109/MCSE.2013.98. URL: http://dx.doi.org/10.1109/MCSE.2013.98.

[26] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra. "DAGuE: A generic distributed DAG engine for High Performance Computing". In: *Parallel Computing* 38.1-2 (2012), pp. 37–51. DOI: 10.1016/j.parco.2011.10.003.

[27] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra. "Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach". In: *Scalable Computing and Communications: Theory and Practice* (2013), pp. 699–733.

[28] G. Bosilca et al. "Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA". In: *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*. Anchorage, United States, May 2011, pp. 1432–1441. URL: https://hal.inria.fr/hal-00809680.

[29] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert. "Tiled QR Factorization Algorithms". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 7:1–7:11. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063393. URL: http://doi.acm.org/10.1145/2063384.2063393.

[30] 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 Application". In: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference*. Feb. 2010, pp. 180–186. DOI: 10.1109/PDP.2010.67.

[31] A. Buttari. "Fine granularity sparse QR factorization for multicore based systems". In: *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*. PARA'10. Reykjavik, Iceland: Springer-Verlag, 2012, pp. 226–236. ISBN: 978-3-642-28144-0. URL: http://dx.doi.org/10.1007/978-3-642-28145-7_23.

[32] A. Buttari. "Fine-Grained Multithreading for the Multifrontal QR Factorization of Sparse Matrices". In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C323–C345. eprint: http://epubs.siam.org/doi/pdf/10.1137/110846427. URL: http://epubs.siam.org/doi/abs/10.1137/110846427.

[33] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. "The impact of multicore on math software". In: *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*. PARA'06. Umeå;, Sweden: Springer-Verlag, 2007, pp. 1–10. ISBN: 3-540-75754-6, 978-3-540-75754-2. URL: http://dl.acm.org/citation.cfm?id=1775059.1775061.

[34] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "A class of parallel tiled linear algebra algorithms for multicore architectures". In: *Parallel Comput.* 35 (1 Jan. 2009), pp. 38–53. ISSN: 0167-8191. DOI: 10.1016/j.parco.2008.10.002. URL: http://dl.acm.org/citation.cfm?id=1486274.1486415.

[35] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "Parallel tiled QR factorization for multicore architectures". In: *Concurr. Comput. : Pract. Exper.* 20.13 (2008), pp. 1573–1590. ISSN: 1532-0626. DOI: 10.1002/cpe.v20:13.

[36] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". Anglais. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. DOI: 10.1016/j.jpdc.2014.06.008.

[37] E. Chan, F. G. V. Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. A. van de Geijn. "SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks". In: *PPOPP*. 2008, pp. 123–132.

[38] M. Cosnard and M. Loi. "Automatic task graph generation techniques". In: *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*. Vol. 2. Jan. 1995, 113–122 vol.2. DOI: 10.1109/HICSS.1995.375471.

[39] E. J. Craig. "The N-step iteration procedures". In: *Journal of Mathematics and Physics* 34 (1955), pp. 64–73.

[40] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. "A column approximate minimum degree ordering algorithm". In: *ACM Trans. Math. Softw.* 30.3 (Sept. 2004), pp. 353–376. ISSN: 0098-3500. DOI: 10.1145/1024074.1024079. URL: http://doi.acm.org/10.1145/1024074.1024079.

[41] T. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006. DOI: 10.1137/1.9780898718881. eprint: http://epubs.siam.org/doi/pdf/10.1137/1.9780898718881. URL: http://epubs.siam.org/doi/abs/10.1137/1.9780898718881.

[42] T. A. Davis. "Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 8:1–8:22. ISSN: 0098-3500. DOI: 10.1145/2049662.2049670. URL: http://doi.acm.org/10.1145/2049662.2049670.

[43] T. A. Davis and Y. Hu. "The university of Florida sparse matrix collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. URL: http://doi.acm.org/10.1145/2049662.2049663.

[44] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. "Communication-optimal Parallel and Sequential QR and LU Factorizations". In: *SIAM J. Sci. Comput.* 34.1 (Feb. 2012), pp. 206–239. ISSN: 1064-8275. URL: http://dx.doi.org/10.1137/080731992.

[45] E. W. Dijkstra. "Een algorithme ter voorkoming van de dodelijke omarming". circulated privately. 1965. URL: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF.

[46] E. W. Dijkstra. "The Mathematics Behind the Banker's Algorithm". English. In: *Selected Writings on Computing: A personal Perspective*. Texts and Monographs in Computer Science. Springer New York, 1982, pp. 308–312. ISBN: 978-1-4612-5697-7. DOI: 10.1007/978-1-4612-5695-3_54. URL: http://dx.doi.org/10.1007/978-1-4612-5695-3_54.

[47] J. Dongarra, M. Faverge, T. Hérault, M. Jacquelin, J. Langou, and Y. Robert. "Hierarchical QR factorization algorithms for multi-core clusters". In: *Parallel Computing* 39.4-5 (2013), pp. 212–232. URL: http://hal.inria.fr/hal-00809770.

[48] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. New York, NY, USA: Oxford University Press, Inc., 1986. ISBN: 0-198-53408-6.

[49] I. S. Duff and J. K. Reid. "The multifrontal solution of indefinite sparse symmetric linear systems". In: *ACM Transactions On Mathematical Software* 9 (1983), pp. 302–325.

[50] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien. "Parallel Scheduling of Task Trees with Limited Memory". In: *ACM Trans. Parallel Comput.* 2.2 (June 2015), 13:1–13:37. ISSN: 2329-4949. DOI: 10.1145/2779052. URL: http://doi.acm.org/10.1145/2779052.

[51] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien. "Parallel scheduling of task trees with limited memory". In: *CoRR* abs/1410.0329 (2014). URL: http://arxiv.org/abs/1410.0329.

[52] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. "Parallel Computing Experiences with CUDA". In: *IEEE Micro* 28.4 (2008), pp. 13–27.

[53] T. Gautier, F. Le Mentec, V. Faucher, and B. Raffin. "X-kaapi: A Multi Paradigm Runtime for Multicore Architectures". In: *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*. 2013, pp. 728–735. DOI: 10.1109/ICPP.2013.86. URL: http://dx.doi.org/10.1109/ICPP.2013.86.

[54] A. Geist and E. G. Ng. "Task scheduling for parallel sparse Cholesky factorization". In: *Int J. Parallel Programming* 18 (1989), pp. 291–314.

[55] A. J. George. "Nested dissection of a regular finite-element mesh". In: *SIAM J. Numer. Anal.* 10.2 (1973), pp. 345–363.

[56] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury. "Multifrontal Factorization of Sparse SPD Matrices on GPUs". In: *Proceedings of 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. 2011, pp. 372–383.

[57] W. Givens. "Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form". English. In: *Journal of the Society for Industrial and Applied Mathematics* 6.1 (1958), pp. 26–50. ISSN: 03684245. URL: http://www.jstor.org/stable/2098861.

[58] G. Golub. "Numerical methods for solving linear least squares problems". English. In: *Numerische Mathematik* 7.3 (1965), pp. 206–216. ISSN: 0029-599X. DOI: 10.1007/BF01436075. URL: http://dx.doi.org/10.1007/BF01436075.

[59] G. H. Golub and C. F. Van Loan. *Matrix Computations. 4th ed.* Baltimore, MD.: Johns Hopkins Press, 2012.

[60] A. Guermouche, J.-Y. L'Excellent, and G. Utard. "Impact of Reordering on the Memory of a Multifrontal Solver". In: *Parallel Computing* 29.9 (2003), pp. 1191–1218.

[61] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. "Tile QR factorization with parallel panel processing for multicore architectures". In: *IPDPS*. IEEE, 2010, pp. 1–10. URL: http://dx.doi.org/10.1109/IPDPS.2010.5470443.

[62] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek. "Improving performance of adaptive component-based dataflow middleware". In: *Parallel Computing* 38.6-7 (2012), pp. 289–309.

[63] M. T. Heath. “Some Extensions of an Algorithm for Sparse Linear Least Squares Problems”. In: *SIAM Journal on Scientific and Statistical Computing* 3.2 (1982), pp. 223–237. DOI: 10.1137/0903014. eprint: http://dx.doi.org/10.1137/0903014. URL: http://dx.doi.org/10.1137/0903014.

[64] P. Hénon, P. Ramet, and J. Roman. “PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems”. In: *Parallel Computing* 28.2 (Jan. 2002), pp. 301–321.

[65] H. P. Hofstee. “Power Efficient Processor Architecture and The Cell Processor”. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. HPCA ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 258–262. ISBN: 0-7695-2275-0. DOI: 10.1109/HPCA.2005.26. URL: http://dx.doi.org/10.1109/HPCA.2005.26.

[66] J. D. Hogg and J. A. Scott. *An indefinite sparse direct solver for large problems on multicore machines*. Tech. rep. RAL-TR-2010-011. Rutherford Appleton Laboratory, 2010.

[67] J. Hogg, E. Ovtchinnikov, and J. Scott. *A sparse symmetric indefinite direct solver for GPU architectures*. Tech. rep. RAL-P-2014-006. STFC Rutherford Appleton Lab., 2014. URL: https://epubs.stfc.ac.uk/work/12189719.

[68] J. Hogg, J. K. Reid, and J. A. Scott. “Design of a Multicore Sparse Cholesky Factorization Using DAGs”. In: *SIAM J. Scientific Computing* 32.6 (2010), pp. 3627–3649.

[69] A. S. Householder. “Unitary Triangularization of a Nonsymmetric Matrix”. In: *J. ACM* 5.4 (Oct. 1958), pp. 339–342. ISSN: 0004-5411. DOI: 10.1145/320941.320947. URL: http://doi.acm.org/10.1145/320941.320947.

[70] A. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. “A Runtime Approach to Dynamic Resource Allocation for Sparse Direct Solvers”. In: *Parallel Processing (ICPP), 2014 43rd International Conference on*. Sept. 2014, pp. 481–490. DOI: 10.1109/ICPP.2014.57.

[71] A. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. “Composing multiple StarPU applications over heterogeneous machines: A supervised approach”. In: *International Journal of High Performance Computing Applications* 28.3 (2014), pp. 285–300. DOI: 10.1177/1094342014527575. eprint: http://hpc.sagepub.com/content/28/3/285.full.pdf+html. URL: http://hpc.sagepub.com/content/28/3/285.abstract.

[72] A.-E. Hugo. “Composability of parallel codes on heterogeneous architectures”. Theses. Université de Bordeaux, Dec. 2014. URL: https://tel.archives-ouvertes.fr/tel-01162975.

[73] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. V. Zee. “The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations”. In: *J. Parallel Distrib. Comput.* 72.9 (2012), pp. 1134–1143.

[74] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. “On Optimal Tree Traversals for Sparse Matrix Factorization”. In: *Proceedings of 25th International Parallel and Distributed Processing Symposium (IPDPS’11)*. IEEE Computer Society, 2011.

[75] L. V. Kalé and S. Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based On C++”. In: *OOPSLA*. 1993, pp. 91–108.

[76] K. Kim and V. Eijkhout. “A Parallel Sparse Direct Solver via Hierarchical DAG Scheduling”. In: *ACM Trans. Math. Softw.* 41.1 (Oct. 2014), 3:1–3:27. ISSN: 0098-3500.

[77] K. Kim and V. Eijkhout. “Scheduling a Parallel Sparse Direct Solver to Multiple GPUs”. In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*. May 2013, pp. 1401–1408.

[78] D. M. Kunzman and L. V. Kalé. “Programming heterogeneous clusters with accelerators using object-based programming”. In: *Scientific Programming* 19.1 (2011), pp. 47–62.

[79] J. Kurzak and J. Dongarra. “Fully Dynamic Scheduler for Numerical Computing on Multicore Processors”. In: *LAPACK working note* lawn220 (2009).

[80] X. Lacoste. “Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems”. PhD thesis. Talence, France: LaBRI, Université Bordeaux, Feb. 2015. URL: http://www.labri.fr/~ramet/restricted/these_lacoste_preprint.pdf.

[81] J.-Y. L’Excellent. “Multifrontal methods for large sparse systems of linear equations: parallelism, memory usage, performance optimization and numerical issues”. Habilitation. École Normale Supérieure de Lyon, 2012.

[82] J. W. H. Liu. “An Application of Generalized Tree Pebbling to Sparse Matrix Factorization”. In: *SIAM J. Algebraic Discrete Methods* 8.3 (July 1987), pp. 375–395. ISSN: 0196-5212. DOI: 10.1137/0608031. URL: http://dx.doi.org/10.1137/0608031.

[83] J. W. H. Liu. “On the storage requirement in the out-of-core multifrontal method for sparse factorization”. In: *ACM Transactions On Mathematical Software* 12 (1986), pp. 127–148.

[84] J. W. H. Liu. “The Role of Elimination Trees in Sparse Factorization”. In: *SIAM Journal on Matrix Analysis and Applications* 11 (1990), pp. 134–172.

[85] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. “A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators”. In: *VECPAR*. 2010, pp. 93–101.

[86] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes. “Multifrontal computations on GPUs and their multi-core hosts”. In: *Proceedings of the 9th international conference on High performance computing for computational science*. VECPAR’10. Berkeley, CA: Springer-Verlag, 2011, pp. 71–82. ISBN: 978-3-642-19327-9. URL: http://dl.acm.org/citation.cfm?id=1964238.1964249.

[87] C.-K. Luk, S. Hong, and H. Kim. “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping”. In: *MICRO*. 2009, pp. 45–55.

[88] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. Charlottesville, Virginia: University of Virginia, 1991-2007. URL: http://www.cs.virginia.edu/stream/.

[89] R. Nath, S. Tomov, and J. Dongarra. “Accelerating GPU Kernels for Dense Linear Algebra”. In: *VECPAR*. 2010, pp. 83–92.

[90] R. Nath, S. Tomov, and J. Dongarra. “An Improved MAGMA GEMM For Fermi Graphics Processing Units”. In: *IJHPCA* 24.4 (2010), pp. 511–515.

[91] C. C. Paige and M. A. Saunders. "LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares". In: *ACM Trans. Math. Softw.* 8.1 (Mar. 1982), pp. 43–71. ISSN: 0098-3500. DOI: 10.1145/355984.355989. URL: http://doi.acm.org/10.1145/355984.355989.

[92] A. Pothen and C. Sun. "A mapping algorithm for parallel sparse Cholesky factorization". In: *SIAM Journal on Scientific Computing* 14 (1993), pp. 1253–1253.

[93] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. "Solving dense linear systems on platforms with multiple hardware accelerators". In: *PPOPP*. 2009, pp. 121–130.

[94] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. V. D. Geijn, F. G. V. Zee, and E. Chan. "Programming matrix algorithms-by-blocks for thread-level parallelism". In: *ACM Trans. Math. Softw.* 36.3 (2009).

[95] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[96] S. C. Rennich, D. Stosic, and T. A. Davis. "Accelerating Sparse Cholesky Factorization on GPUs". In: *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*. IA3 '14. New Orleans, Louisiana: IEEE Press, 2014, pp. 9–16. ISBN: 978-1-4799-7056-8. DOI: 10.1109/IA3.2014.6. URL: http://dx.doi.org/10.1109/IA3.2014.6.

[97] F.-H. Rouet. "Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides". anglais. Thèse de doctorat. Toulouse, France: Institut National Polytechnique de Toulouse, Oct. 2012. URL: http://tel.archives-ouvertes.fr/tel-00785748.

[98] P. Sao, X. Liu, R. Vuduc, and X. Li. "A Sparse Direct Solver for Distributed Memory Xeon Phi-Accelerated Systems". In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. May 2015, pp. 71–81. DOI: 10.1109/IPDPS.2015.104.

[99] P. Sao, R. W. Vuduc, and X. S. Li. "A Distributed CPU-GPU Sparse Direct Solver". In: *Euro-Par 2014 Parallel Processing*. 2014, pp. 487–498.

[100] E. Schmidt. "Über die Auflösung linearer Gleichungen mit Unendlich vielen unbekannten". German. In: *Rendiconti del Circolo Matematico di Palermo (1884-1940)* 25.1 (1908), pp. 53–77. DOI: 10.1007/BF03029116. URL: http://dx.doi.org/10.1007/BF03029116.

[101] R. Schreiber and C. Van Loan. "A storage-efficient WY representation for products of Householder transformations". In: *SIAM J. Sci. Stat. Comput.* 10 (1989), pp. 52–57.

[102] R. Schreiber. "A new implementation of sparse Gaussian elimination". In: *ACM Transactions On Mathematical Software* 8 (1982), pp. 256–276.

[103] R. Sethi. "Complete Register Allocation Problems". In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC '73. Austin, Texas, USA: ACM, 1973, pp. 182–195. DOI: 10.1145/800125.804049. URL: http://doi.acm.org/10.1145/800125.804049.

[104] R. Sethi and J. D. Ullman. "The Generation of Optimal Code for Arithmetic Expressions". In: *J. ACM* 17.4 (Oct. 1970), pp. 715–728. ISSN: 0004-5411. DOI: 10.1145/321607.321620. URL: http://doi.acm.org/10.1145/321607.321620.

[105] F. Song, H. Ltaief, B. Hadri, and J. Dongarra. "Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.48. URL: http://dx.doi.org/10.1109/SC.2010.48.

[106] F. Song, A. YarKhan, and J. Dongarra. "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems". In: *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC'09*. 2009.

[107] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut. "Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures". In: *Concurrency and Computation: Practice and Experience* (May 2015). DOI: 10.1002/cpe.

[108] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J. Méhaut. "Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-core Architectures". In: *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. 2014, pp. 50–62. DOI: 10.1007/978-3-319-09873-9_5. URL: http://dx.doi.org/10.1007/978-3-319-09873-9_5.

[109] S. Tomov, J. Dongarra, and M. Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems". In: *Parallel Computing* 36.5-6 (2010), pp. 232–240.

[110] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. "Dense linear algebra solvers for multicore with GPU accelerators". In: *IPDPS Workshops*. 2010, pp. 1–8.

[111] H. Topcuouglu, S. Hariri, and M.-Y. Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219. DOI: 10.1109/71.993206.

[112] S. Treichler, M. Bauer, and A. Aiken. "Realm: an event-based low-level runtime for distributed memory architectures". In: *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*. 2014, pp. 263–276. DOI: 10.1145/2628071.2628084. URL: http://doi.acm.org/10.1145/2628071.2628084.

[113] F. Van Zee, E. Chan, R. van de Geijn, E. Quintana, and G. Quintana-Orti. "Introducing: The Libflame Library for Dense Matrix Computations". In: *Computing in Science Engineering* PP.99 (2009), p. 1. ISSN: 1521-9615. DOI: 10.1109/MCSE.2009.154.

[114] V. Volkov and J. Demmel. "Benchmarking GPUs to tune dense linear algebra". In: *SC08*. 2008, p. 31.

[115] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[116] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. "Hierarchical DAG scheduling for Hybrid Distributed Systems". In: *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Hyderabad, India, May 2015, pp. 156–165.

[117] A. YarKhan, J. Kurzak, and J. Dongarra. *QUARK Users' Guide: QUeueing And Runtime for Kernels*. Tech. rep. Innovative Computing Laboratory, University of Tennessee, 2011.

[118] S. Yeralan, T. Davis, and S. Ranka. *Sparse QR factorization on the GPU*. Tech. rep. University of Florida, 2015.

[119] C. D. Yu, W. Wang, and D. Pierce. "A CPU-GPU hybrid approach for the unsymmetric multifrontal method". In: *Parallel Comput.* 37 (12 Dec. 2011), pp. 759–770. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2011.09.002. URL: http://dx.doi.org/10.1016/j.parco.2011.09.002.

[120] D. Zou, Y. Dou, S. Guo, R. Li, and L. Deng. "Supernodal sparse Cholesky factorization on graphics processing units". In: *Concurrency and Computation: Practice and Experience* 26.16 (2014), pp. 2713–2726. ISSN: 1532-0634. DOI: 10.1002/cpe.3158. URL: http://dx.doi.org/10.1002/cpe.3158.