HABILITATION A DIRIGER DES RECHERCHES

présentée à l'Université Paris-Sud

Spécialité: Informatique

par

Marc Baboulin

Maître de conférences, Université Paris-Sud
Chaire Inria Saclay - Île-de-France

Résolutions rapides et fiables pour les solveurs d'algèbre linéaire numérique en calcul haute performance.

Fast and reliable solutions for numerical linear algebra solvers in high-performance computing.

Jean-Claude Bajard	Professeur, Université Pierre et Marie Curie	France	*Examinateur*
Philippe Dague	Professeur, Université Paris-Sud	France	*Président du Jury*
Frédéric Desprez	Directeur de Recherche, Inria/Ens Lyon	France	*Rapporteur*
Jack Dongarra	Professeur, University of Tennessee	USA	*Examinateur*
Serge Gratton	Professeur, Enseeiht Toulouse	France	*Membre invité*
Philippe Langlois	Professeur, Université de Perpignan	France	*Rapporteur*
Jose Roman	Professeur, Universitat Politècnica de València	Espagne	*Rapporteur*
Brigitte Rozoy	Professeur, Université Paris-Sud	France	*Examinatrice*

Résumé

Dans cette Habilitation à Diriger des Recherches (HDR), nous présentons notre recherche effectuée au cours de ces dernières années dans le domaine du calcul haute-performance. Notre travail a porté essentiellement sur les algorithmes parallèles pour les solveurs d'algèbre linéaire numérique et leur implémentation parallèle dans les bibliothèques logicielles du domaine public. Nous illustrons dans ce manuscrit comment ces calculs peuvent être accelérés en utilisant des algorithmes innovants et être rendus fiables en utilisant des quantités spécifiques de l'analyse d'erreur.

Nous expliquons tout d'abord comment les solveurs d'algèbre linéaire numérique peuvent être conçus de façon à exploiter les capacités des calculateurs hétérogènes actuels comprenant des processeurs multicœurs et des GPUs. Nous considérons des algorithmes de factorisation dense pour lesquels nous décrivons la répartition des tâches entre les différentes unités de calcul et son influence en terme de coût des communications. Ces calculs peuvent être également rendus plus performants grâce à des algorithmes en précision mixte qui utilisent une précision moindre pour les tâches les plus coûteuses tout en calculant la solution en précision supérieure.

Puis nous décrivons notre travail de recherche dans le développement de solveurs d'algèbre linéaire rapides qui utilisent des algorithmes randomisés. La randomisation représente une approche innovante pour accélérer les calculs d'algèbre linéaire et la classe d'algorithmes que nous proposons a l'avantage de réduire la volume de communications dans les factorisations en supprimant complètement la phase de pivotage dans les systèmes linéaires. Les logiciels correspondants on été développés pour architectures multicœurs éventuellement accélérées par des GPUs.

Enfin nous proposons des outils qui nous permettent de garantir la qualité de la solution calculée pour les problèmes de moindres carrés sur-déterminés, incluant les moindres carrés totaux. Notre méthode repose sur la dérivation de formules exactes ou d'estimateurs pour le conditionnement de ces problèmes. Nous décrivons les algorithmes et les logiciels qui permettent de calculer ces quantités avec les bibliothèques logicielles parallèles standards.

Des pistes de recherche pour les années à venir sont données dans un chapitre de conclusion.

Mots clés: Calcul haute-performance, solveurs d'algèbre linéaire dense, systèmes linéaires, moindres carrés linéaires, processeurs multicœurs, Graphics Processing Units (GPU), algorithmes randomisés, analyse d'erreur inverse, estimation de conditionnement, LAPACK, ScaLAPACK, PLASMA, MAGMA.

Abstract

In this "Habilitation à Diriger des Recherches" (HDR), we present our research in high-performance scientific computing over the recent years. Our work has been mainly related to parallel algorithms for numerical linear algebra solvers and their parallel implementation in public domain software libraries. We illustrate in this manuscript how these calculations can be accelerated using innovative algorithms and be made reliable using specific quantities in error analysis.

First we explain how numerical linear algebra solvers can be designed to exploit the capabilities of current heterogeneous multicore+GPU systems. We consider dense factorization algorithms for which we describe the work splitting between the architectural components and its influence in terms of communication cost. These computations can also be significantly enhanced thanks to mixed precision algorithms that use lower precision for the most expensive tasks while achieving higher precision accuracy for the results.

Then we present our research in developing fast linear agebra solvers using randomized algorithms. Randomization represents a very promising approach to accelerate linear algebra computations and the class of algorithms that we developed has the advantage of reducing the amount of communication in dense factorizations by removing completely the pivoting phase in linear system solutions. The resulting software has been developed for multicore machines possibly accelerated by GPUs.

Finally we propose numerical tools that enable us to assess the quality of the computed solution of overdetermined linear least squares, including the total least squares approach. Our method is based on deriving exact values or estimates for the condition number of these problems. We describe algorithms and software to compute these quantities using standard parallel libraries.

Research tracks for the coming years are given in a concluding chapter.

Keywords: High-performance computing, dense linear algebra solvers, linear systems, linear least squares, multicore processors, Graphics Processing Units (GPU), randomized algorithms, backward error analysis, condition number estimation, LAPACK, ScaLAPACK, PLASMA, MAGMA.

Contents

Introduction

A major challenge for the high-performance computing (HPC) community is to develop efficient algorithms that take advantage of current parallel architectures such as multicore processors or Graphics Processing Units (GPU). In particular, the transition from traditionally based MPI-only parallelism to MPI associated with many-core requires the rewriting of many algorithms in existing software [30], including for instance the public domain libraries LAPACK [2] and ScaLAPACK [21]. Also with the increase in parallelism and heterogeneity, the data-communication costs become more than ever a major bottleneck. This requires to investigate new approaches, sometimes non conventional in numerical libraries (*e.g.* randomized algorithms), in order to reduce communication at its minimum. At the same time, there is an increasing demand for high-resolution simulations that require extremely accurate solutions. Our research focuses on these two fronts: developing innovative algorithms that take advantage of recent advances in hardware and also propose new results in the area of error analysis for HPC applications so that reliable information can be provided regarding the numerical quality of the computed solution.

This document is the manuscript of "Habilitation à Diriger des Recherches" (HDR) defended by Marc Baboulin at University Paris-Sud. It describes the research accomplished by Marc Baboulin over the recent years mainly in the area of parallel algorithms for numerical linear algebra solvers, which are at the heart of many scientific computing applications. These algorithms and software aim to be applied to large linear systems or large linear least squares problems arising in engineering applications and to be implemented on current heterogeneous parallel computers. In this work, our goal is to be able to solve challenging problems coming from scientific and real industrial applications but with the constant motivation of making these new functionalities available in open source scientific libraries. In this manuscript, we will develop three main ideas that have motivated our research during the recent years.

The first one is related to designing algorithms that fully **exploit the inherent heterogeneity of current parallel systems**. Indeed heterogeneous computing using accelerated manycore systems is becoming a *de facto* standard in HPC, as observed in the latest trends from the Top 500 list[1]. The targetted architectures in our research are multicore systems possibly accelerated by GPUs. A first issue in developing algorithms for such heterogeneous systems is related to the way we split the work between the multicore and the GPU with the objective of getting the best from each computational unit. For instance we would preferably execute small and nonparallelizable tasks on CPU while the large and parallelizable tasks would be executed by the GPU. A major concern is also

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

to reduce or mask the (slow) communication speed inside or between these architectural components and to properly schedule the execution of the algorithm. Indeed, a major concern for the developers of numerical algorithms comes from the exponentially growing gap between communication and computation efficiency. Performance can also be significantly improved by using mixed precision algorithms that achieve a given accuracy outcome with lower precision computations for the most expensive tasks. This technique has the advantage of improving runtime and requiring lower data movement (note that it also reduces energy consumption [64]). Our research on heterogeneous computing has been mainly applied to dense factorizations in linear algebra.

The second idea that we develop in this document concerns **the use of statistical techniques to accelerate dense linear algebra solvers**. Randomized algorithms are becoming very attractive in HPC due to the significant acceleration they can provide for large size simulations. Recent advances in the field include for instance random sampling algorithms [7], low-rank matrix approximation [73] or general matrix decompositions [56]. Our research in this area concerns the solution of linear systems and the assesment of numerical quality via statistical condition estimates. The randomized algorithms that we recently developed have the advantage of reducing the amount of communication in dense factorizations by removing the expensive phase of pivoting and to reduce significantly the number of arithmetic operations in estimating condition numbers. The randomized solvers are being progressively integrated into standard linear algebra libraries and take advantage of current multicore+GPU architectures.

The third idea concerns the **numerical quality in HPC applications** and more specifically linear algebra calculations. It is essential, when developing new solvers, to propose also, preferably in the same libraries, numerical tools that assess the quality of the computed solution. Of course, this should be obtained at a reasonable cost and ideally the by-products of the solution process should be reused for the numerical verification phase. Numerical quality in scientific computing is a wide research area that proposes various approaches (see *e.g.* [18, 68, 70, 71, 87]). In our research we focus developing methods and tools around the concept of problem conditioning. In a preliminary work we derived analytical formulas for the quantities of interest. We specifically chosen the overdetermined linear least squares problem (linear systems being a special case of the latter). Here again, we wanted to investigate statistical methods to accelerate computation. Using our theoretical work, we developed very recently routines for public domain libraries LAPACK and MAGMA [2]

This HDR manuscript is structured as follows. In Chapter 1 we explain how we can exploit specificities of current heterogeneous many-core architectures to design faster algorithms in the context of linear algebra factorizations. In Chapter 2 we describe a class of innovative algorithms based on randomization that enable us to avoid pivoting and then to minimize communication in linear algebra solvers. In Chapter 3 we present numerical tools to assess quality of solutions obtained for HPC applications. Finally, we give some conclusions and research perspectives for the coming years.

[2]Matrix Algebra on GPU and Multicore Architectures, `http://icl.cs.utk.edu/magma/`

Chapter 1

Taking advantage of parallel multicore-GPU architectures

1.1 Introduction to heterogeneous computing

Graphics Processing Units (GPU), originally designed for graphics applications, have substantially evolved over the years to include more functionality and programmability turning them into General Purpose GPUs (GPGPUs). Their impressive bandwidth combined with their floating-point performance has enabled and motivated their use in applications well beyond graphics. The latest trends show that they are continuously gaining ground in High-Performance Computing (HPC). For instance, we observe that systems using accelerators/co-processors represent 57 systems in the June 2012 release of the Top 500 list[1] (vs 39 systems 6 months ago). Also 3 systems accelerated with NVIDIA Fermi GPUs are among the 10 fastest supercomputers.

The current generation of GPUs Fermi can achieve more than 500 Gflop/s of IEEE standard double-precision floating-point arithmetic and the upcoming Kepler GK110 already announces 1 Tflop/s of double precision throughput with more than 80% DGEMM efficiency (vs 60-65% on the prior Fermi architecture). GPUs also have a programming model (see *e.g.* [76]) that does not require software developers to know about graphics in order to use GPUs for general purpose computing. These features have cemented even further the important role of GPUs in todays HPC. Even power consumption that was pointed out in a recent past as a major drawback of graphics devices is also significantly improving with for instance a performance per watt for Kepler which is 3 times that of Fermi. Looking again at the Top 500 list, it is also interesting to observe that 2 GPU-accelerated systems are among the 6 most energy efficient supercomputers.

As a result GPUs have moved "closer" to CPUs in terms of functionality and programmability. At the same time, CPUs have also acquired functionalities similar to that of GPUs (see *e.g.* Intel SSE2 or PowerPC AltiVec). Currently, major chip manufacturers are developing next-generation of microprocessors that integrate multicore CPU and GPU components (AMD Fusion or Intel MIC). These trends make it more evident that next generation of supercomputers will be hybrid and will rely on the integration (in varying proportions) of homogeneous multicore and GPU type of components.

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

Due to the high ratio of floating-point calculations to data required, many Dense Linear Algebra (DLA) algorithms have been of very high performance (*e.g.* close to the machine peak) on standard CPU architectures. Older generation of GPUs did not have memory hierarchy and their performance exclusively relied on high bandwidth. But this lack of acceleration has recently changed due to a combination of factors. First, since 2008 GPUs have clearly outperformed standard CPUs in single precision but also in double precision arithmetics [98]. Second, GPUs have significantly outpaced CPUs in bandwidth (about an order of magnitude higher than current multi-socket multicore systems). Finally, by having memory hierarchy, GPUs can be programmed for memory reuse and hence not rely exclusively on high bandwidth in order to achieve a high percentage of their theoretical performance peak. These new architectural trends result in an increase of parallelism and heterogeneity as well as ever increasing data-communication costs. This has motivated our research to develop efficient linear algebra algorithms for hybrid architectures in order to integrate them into public domain libraries. The basic ideas for developing efficient algorithms and software have been initially presented in [11, 14] and then developed in more details in the framework of the MAGMA project [75, 94, 95]. MAGMA, similarly to LAPACK [2] and ScaLAPACK [21], is being build as a community effort, incorporating the newest developments in hybrid algorithms and scheduling, and aiming at minimizing synchronizations and communication in these algorithms. The goal of these efforts is to redesign the DLA algorithms in LAPACK to fully exploit the power of current heterogeneous systems of multi/manycore CPUs and accelerators, and deliver the fastest possible time to an accurate solution. Indeed, the algorithms included so far in MAGMA 1.1 manage to overcome bottlenecks associated with just multicore or GPUs, to significantly outperform corresponding packages for any of these homogeneous components taken separately. MAGMA's one-sided factorizations for example (and linear solvers) on a single Fermi GPU (and a basic CPU host) can outperform state-of-the-art CPU libraries on high-end multi-socket, multicore nodes (*e.g.*, using up to 48 modern cores). In the context of hybrid algorithms, there have been also a number of new developments related to minimizing the communication in one-sided factorizations. For instance communication-avoiding techniques have been recently applied in [10] to the solution of general dense linear systems via LU factorization or in [3] to the QR factorization. Such improvements have become essential due to the increasing gap between communication and computation costs.

In designing DLA algorithms for multicore or GPU, the requirements for efficient execution are high parallelism and reduced communication to mask slow memory speeds. But when we combine CPU and GPU architectures, algorithms should also be properly hybridized. This means that the work load should be balanced throughout the execution, and the work scheduling/mapping should ensure matching of architectural strengths to algorithmic requirements. In Section 1.2 we explain how linear algebra solvers can be implemented in order to reduce communication and exploit the architectural strengths of each component. We present specific examples based on the Cholesky and LU factorizations, which further illustrate the concept of hybrid multicore+GPU computing for DLA.

Another observation is that on modern architectures, the performance of 32-bit operations is often at least twice as fast as the performance of 64-bit operations (due to the faster speed of 32-bit floating point arithmetic but also to the fact that the amount

of bytes moved through the memory system is halved). This is the case for multicore processors but also for technologies such as Field Programmable Gate Arrays (FPGA) and GPUs. Then by using a combination of 32-bit and 64-bit floating point arithmetic, the performance of many dense and sparse linear algebra algorithms can be significantly enhanced while maintaining the 64-bit accuracy of the resulting solution. The relevance of this method for modern architectures was shown in [72] and then more details and applications were given in [9]. In Section 1.3 we present this methodology in the context of solving a system of linear equations and we provide performance results on current multicore+GPU architectures.

1.2 Hybrid multicore-GPU solvers

1.2.1 Designing hybrid algorithms in dense linear algebra factorizations

We present in this section the underlying method for combining multicore and GPU in DLA algorithms. It consists of representing algorithms as a collection of BLAS-based tasks [1] that are executed over the multicore and the GPU. This abstracts us from the specificities in programming a GPU. This approach relies on high performance BLAS implementations, which are available for current multicore and GPU systems. The three main ideas that define our approach are the following:

1. Use BLAS level parallelism, where the matrix resides on the GPU, and the CPU is running, for example, LAPACK style code represented as a sequence of CUBLAS kernels,

2. Offload to the CPU small kernels that are inefficient for the GPU,

3. Use asynchronicity between CPU and GPU whenever possible in the offload/load process.

We can illustrate this idea in Figure 1.1 by considering the Cholesky factorization (in its so called left-looking variant [79]). The matrix to be factored is allocated on the GPU memory and the code is as in LAPACK with BLAS calls replaced by CUBLAS, which represents the first idea in the list above. Since steps 2 and 3 of the algorithm are independent, and the Cholesky factorization of matrix B (notations as in Figure 1.1) would have been inefficient for the GPU (small problem of size 128 x 128 for example, *i.e.* cannot have enough parallelism to utilize all the cores of the GPU), B is offloaded and factorized on the CPU, which illustrates the second idea. Finally, steps 2 and 3 of the algorithm are executed in parallel as calls to CUDA are asynchronous, and SPOTRF is executed without waiting for the completion of cublasSgemm, which illustrates the third idea. In addition to overlapping just the computation, for cards that support it, sending B to the CPU and moving the result back could be overlapped with the GPU computation (of cublasSgemm in this case) when asynchronous copy calls are used.

This principle of "hybridization" can be generalized to dense factorization algorithms for which the block algorithms involve factoring block columns (called panels) using Level-2 BLAS algorithms from LAPACK-style routines. This panel factorization would then be performed on the CPU while the trailing matrix updates (Level-3 BLAS) are done on the GPU. Usually we try to overlap the work on the CPU and the GPU. Ideally, as the CPU is

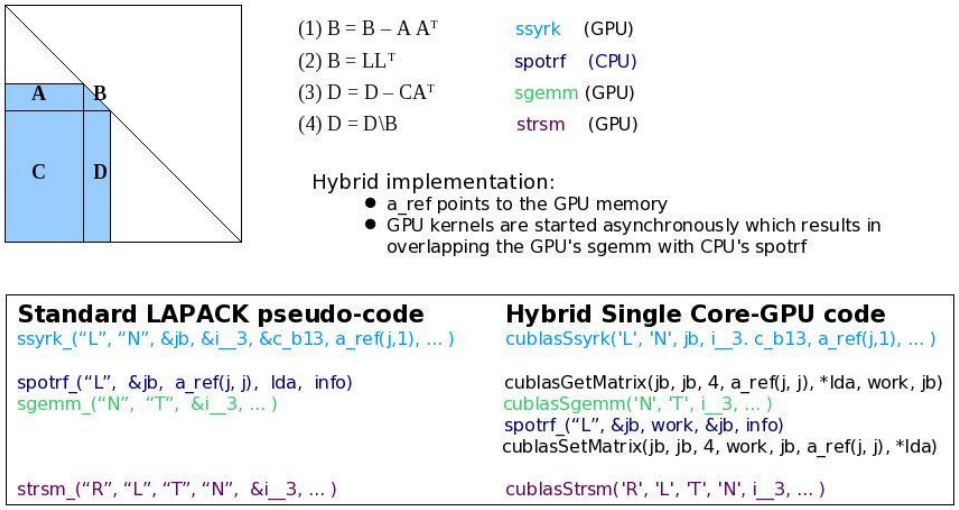


Figure 1.1: Hybridization of left-looking Cholesky factorization.

working on the critical path, we want the CPU never to work by itself. Indeed we achieve this for certain matrix sizes as shown in Figure 1.2 for the QR factorization (CPU alone disappears from the graph for matrix size about 6000), the overhead representing here mostly communication. More details about the task splitting between CPU and GPU will be given in Section 1.2.2 by considering the example of the widely used LU factorization.

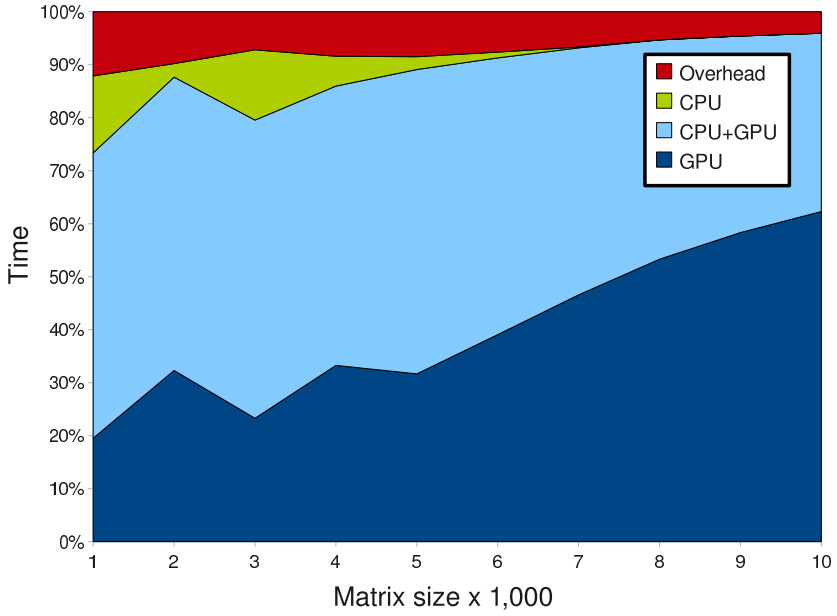


Figure 1.2: Time breakdown for hybrid QR factorization (single precision)
Intel Core2 Q9300 (1 x 4 cores @ 2.50 GHz) - GPU C2050 (448 CUDA cores @ 1.15 GHz).

1.2.2 Application to the LU factorization

Let us explain how the principles of hybridization described in Section 1.2.1 can be applied to the LU factorization (note that the LU algorithm is used in the LINPACK benchmark for the Top 500 list). We described a hybrid multicore+GPU version of this algorithm in [94] as it is implemented in the MAGMA library (release 1.1). Current libraries like LAPACK implement LU factorization using a block algorithm, which factors the input matrix by iterating over blocks of columns (panels). At each iteration, the LU factorization of the current panel is computed, and then the trailing submatrix is updated. When using hybrid architectures the computation can be split as shown in Figure 1.3 that represents a matrix factored via a right-looking block LU factorization [38, p. 85], where the dark part has been already factored. The initial matrix has been downloaded to the GPU. We describe in Algorithm 1.2.1 a current iteration for the factorization of the matrix depicted in Figure 1.3.

Algorithm 1.2.1 Iteration for LU factorization using MAGMA

1: The current panel (1) is downloaded to the CPU.
2: (1) is factored by the CPU and the result is sent back to the GPU.
3: The GPU updates (2) (next panel).
4: The updated panel (2) is sent back to the CPU to be factored while the GPU updates the rest of the matrix (3).

The technique consisting of factoring (2) while still updating (3) is often referred to as *look-ahead* [69]. In the current implementation of MAGMA, the panel factorization is performed using Gaussian Elimination with Partial Pivoting (GEPP) but this algorithm is general enough to be applicable to many forms of LU factorizations, where the distinction can be made based on the form of pivoting that they employ. Depending on the problem size n and on the hardware used, MAGMA proposes a default value for the parameter b (width of the panel). Note that the design of this hybrid LU factorization avoids communicating by having only panels transferred between CPU and GPU ($\mathcal{O}(n * b)$ data vs $\mathcal{O}(n * n * b)$ computation in the updates), enabling also the total overlap of the panel computation by the updates for n large enough.

However communication can still be reduced by considering another pivoting technique to factor the panel on the CPU. This technique called *tournament pivoting* was introduced in the last years in the context of CALU, a communication-avoiding LU factorization algorithm [51]. With this strategy, the panel factorization, referred to as TSLU (Tall Skinny LU), can be efficiently parallelized as follows. The panel is partitioned into P_r blocks. From each block, a set of local pivots is selected in parallel using GEPP. A tournament is used on the P_r local sets to select a set of global pivots. These global pivots are moved to the diagonal positions, and then the LU factorization with no pivoting of the entire panel is performed. The tournament is implemented as a reduction operation, with GEPP being the operator used at each step of the reduction. This pivoting strategy turns out to be very efficient for factoring the panel due to its particular "tall and skinny" structure.

Whatever the pivoting strategy chosen for factoring the panel, the hybrid LU factorization can also be represented as a sequence of DAGs (Directed Acyclic Graphs) as depicted

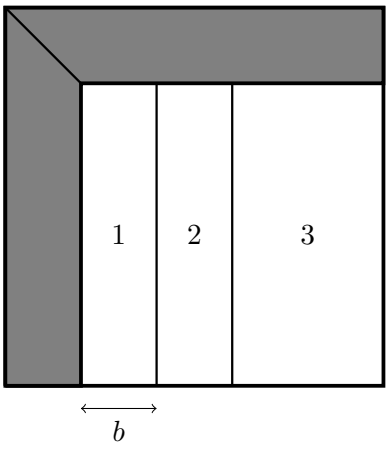


Figure 1.3: Block splitting in hybrid LU factorization

in Figure 1.4. We consider that the matrix is initially stored on the GPU. Black tasks represent the factorization of the panel using multithreaded CALU and the gray tasks represent the update of the trailing submatrix in the GPU. At each step of the factorization, the block corresponding to the panel is transfered to the CPU and factored using GEPP or CALU. Once the panel is factored, it is sent back to the GPU in order to update the trailing submatrix. The GPU updates in priority the column block corresponding to the next panel. Note that, similarly to [94], the data transfer between CPU and GPU is overlapped by computation.

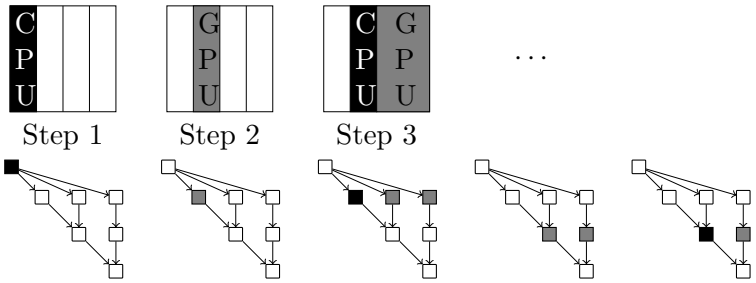


Figure 1.4: Task splitting in hybrid LU factorization.

1.2.3 Numerical experiments on hybrid LU solvers

1.2.3.1 Performance results on single and multi GPUs

In this section we present performance results for the hybrid LU factorization algorithm described in Section 1.2.2. The GPU device is an NVIDIA Fermi Tesla C2050 with 448 CUDA cores running at 1.15 GHz and 2687 MB memory. The multicore host is a 48 cores system (4 sockets $\times$ 12 cores) Magny-Cours AMD Opteron 6172 (2.1 GHz). The Fermi GPU used in these experiments achieves single and double precision peak performance of

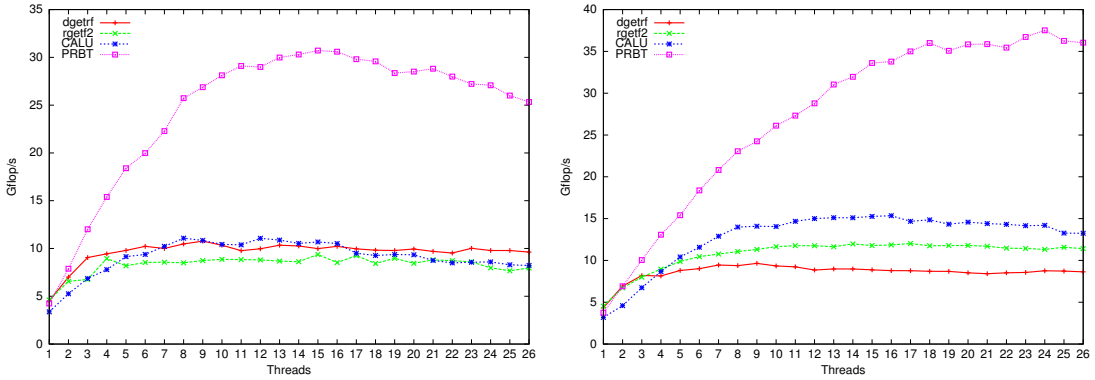


Figure 1.5: Comparison of CPU multi-threaded panel factorizations.
Matrix size = 5120, panel size = 256 (left) - Matrix size = 10240, panel size = 320 (right)

respectively 1030 Gflop/s and 515 Gflop/s. All computations are performed on random matrices and in double precision arithmetic.

Let us first compare the kernels used on the multicore for factoring the panel. In Figure 1.5, we compare the performance of the panel factorization for the following routines:

- CALU factorization routine that we modified to develop the hybrid solver called H-CALU (see [10]) and linked with the sequential version of MKL for the required BLAS and LAPACK routines.
- MKL implementation of the LAPACK routine `dgetrf`, used in the MAGMA implementation of LU for factoring the panel.
- A recursive routine for GEPP `rgetf2` (linked with MKL multithreaded BLAS) described in [54] and known to give good performance on "tall and skinny" matrices.
- PRBT (solver based on randomization [13]) where the panel is factored using the routine `dgetrf_nopiv` that performs Gaussian Elimination with No Pivoting (GENP).

This performance is measured by summing the total number of flops executed in factoring successively each panel throughout the factorization and dividing it by the time spent during these steps. This performance (expressed in Gflop/s) is plotted in Figure 1.5 for the factorization of two square matrices, each associated with a given panel size (parameter b defined in Section 1.2.2, corresponding to the number of columns for the panel).

For factoring the panel, we consider different numbers of threads (one CPU core being used for each thread) varying from 1 to 26. Note that using more than 26 threads does not provide us with better performance, due to the too-large amount of communication involved in the panel factorization. The panel size b considered in Figure 1.5 for each matrix size corresponds to a value empirically tuned in order to provide the best global factorization time for each matrix when using a hybrid implementation.

The performance of PRBT, based on a GENP routine can be considered here as a "peak" performance for the panel factorization. We also observe in Figure 1.5(b) that CALU is faster for a larger ratio rows/columns. Moreover, CALU and PRBT have better

scalability properties. This can be explain by the fact that CALU minimizes communication thanks to its pivoting strategy and PRBT does not pivot at all. The plateau observed for each curve after a certain number of threads corresponds to cases where the amount of communication becomes too large and cannot be overlapped by computation. For $n = 5120$, CALU, `dgetrf` and `rgetf2` give similar performance. However, when the matrix size increases and then the panel becomes more "tall and skinny", CALU outperforms the two other solvers and achieves a reasonable fraction of the PRBT rate. This good behavior of CALU for factoring the panel was already mentioned in [37].

We compare in Figure 1.6 three hybrid LU factorization routines implemented as described in Algorithm 1.2.1. The first one (`magma_dgetrf`) is implemented in MAGMA 1.1, where the panel is factored using the MKL routine `dgetrf` that performs GEPP. For the second routine (H-CALU), the panel is factored using the CALU routine mentioned in Section 1.2.2. These two routines are compared with the PRBT solver that prior to factorization performs randomization referred to as Partial Random Butterfly Transformation (PRBT) [13]. Following randomization, Gaussian elimination with no pivoting is applied which is implemented as a very efficient fully BLAS 3 algorithm. Then we expect that the Gflop/s performance of the PRBT solver will provide us with an upper bound for other LU solvers on hybrid CPU/GPU architectures. More details on this randomization technique will be given in Chapter 2.

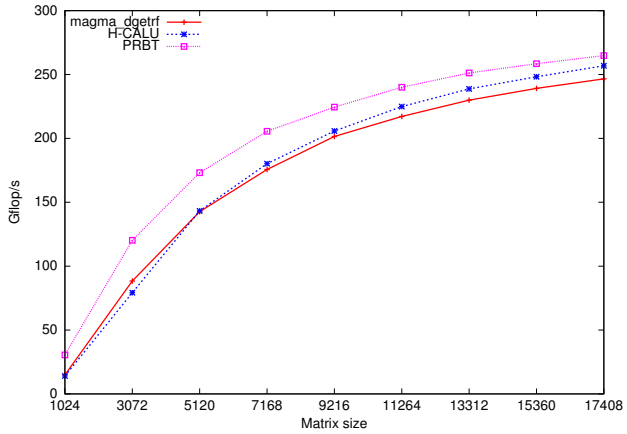


Figure 1.6: Performance for hybrid LU factorization AMD + Tesla C2050, 16 threads.

As expected, PRBT outperforms the other routines because it does not pivot and the randomization time is negligible. We can also observe that in the range 1024-5120, H-CALU gives similar performance as MAGMA but it is slightly faster for matrix sizes larger than 5120. This trend can be explained by the fact that, for matrix sizes smaller than 5120, the panels are not enough "tall and skinny" to take advantage of the CALU algorithm. We notice that the difference of performance observed for the panel in Figure 1.5 has a moderate impact on the whole factorization since the update phase performed on the GPU represents the bulk of the computation. We observe that asymptotically, the performance of the three routines should be close because communication become negligible compared to the $O(n^3)$ computations for large dimensions. Finally we point out that, if we use only multicore machines without GPU, then other solvers can be considered (see *e.g.* recursive

tile version in [39]).

Following recent work on one-sided factorization algorithms [102], we developed recently a version of PRBT that uses multiple GPUs. The matrix to factorize is distributed on the GPUs using a 1-D block-cyclic column layout [21, p. 58]). At each step the current panel is downloaded from the GPU that owns it to the CPU to be factored. When the CPU finishes the panel factorization, it sends it to all GPUs. This panel is stored in a temporary space allocated on each GPU (except for the GPU that owns this panel from the data distribution) and the GPUs update their trailing submatrix. The GPU that owns the next panel, updates in priority the part of the trailing submatrix that corresponds to the next panel and sends it to the CPU. Using this algorithm, we can compare in Figure 1.7 the performance of the LU with partial pivoting and no pivoting. It shows that using multiple GPUs is interesting only when we consider large systems since for smaller sizes, the communication cost between CPU and GPUs is significant. Note also that the no-pivoting factorization is much more scalable than the partial pivoting factorization. Indeed, the latter does not take full advantage of the multiple GPUs since the pivoting is performed on the CPU. This justifies again the interest of using techniques to avoid pivoting on these architectures. This aspect will be developed in Chapter 2.

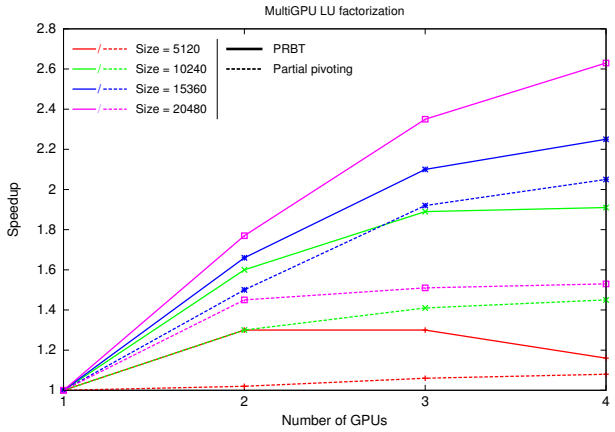


Figure 1.7: Performance for multiGPU LU factorization.

1.2.3.2 Accuracy of hybrid LU implementations

The following experiments on accuracy were achieved on the machine described at the beginning of Section 1.2.3.1. First we study the backward error obtained for the linear system solution computed with the solvers LU MAGMA (based on the factorization routine `magma_dgetrf`), H-CALU and PRBT, using random matrices. The quantity plotted in Figure 1.8 corresponds to the componentwise backward error given in [2, p. 78] and expressed by

$$\omega = \max_i \frac{|Ax - b|_i}{(|A| \cdot |x| + |b|)_i},$$

where x is the computed solution. We observe that the backward errors are very similar for the three hybrid solvers.

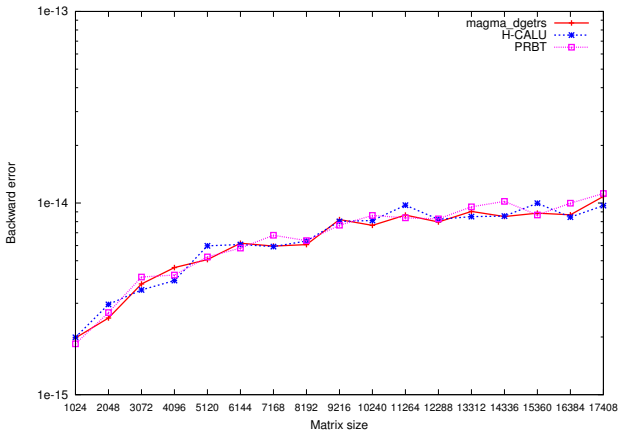


Figure 1.8: Comparison of componentwise backward error

Now we present additional experiments using LAPACK test cases given in [22]. Table 1.1 lists the 11 matrices used in our experiments (size 512, all in double precision). In this table, ε denotes the machine precision and κ is the infinity-norm condition number of the matrix. We report in Table 1.2 the componentwise backward error obtained for the three hybrid solvers (LU MAGMA, H-CALU and PRBT) and the no pivoting case. Iterative refinement (in the working precision) is added if necessary using the LAPACK routine `dgerfs` on the multicore host. The iterative refinement is based on the stopping criterion given in [89] ($\omega \leq (n+1)\varepsilon$), with a maximum of 5 iterations. Matrices 5 to 7 are singular and have at least one row and column equal to zero. These are used in [22] to test the error return codes. For the ill-conditioned matrix 9, the backward error for PRBT is slightly less accurate than the other solvers. Matrix 10 is scaled to near underflow and the three solvers give similar results but less acurate than for other matrices. For all the other matrices, the three solvers give the same accuracy. Tests on accuracy for specific matrix collections can be found in [13] and [51] respectively for PRBT and CALU.

Table 1.1: Test matrices

1	Diagonal	7	Last $n/2$ columns zero
2	Upper triangular	8	Random, $\kappa = \sqrt{0.1/\varepsilon}$
3	Lower triangular	9	Random, $\kappa = 0.1/\varepsilon$
4	Random, $\kappa = 2$	10	Scaled near underflow
5	First column zero	11	Scaled near overflow
6	Last column zero		

1.3 Mixed precision algorithms

The interest of mixed precision algorithms comes from the observation that, in many cases, a single precision solution of a problem can be refined to the point where double precision

Table 1.2: Componentwise Backward Error

Matrix Type	MAGMA LU (`magma_dgetrf`)	H-CALU	PRBT	No pivoting
1	0.0	0.0	1.42e-16	0.0
2	1.32e-16	1.32e-16	4.02e-16	6.19e-16
3	1.85e-16	1.85e-16	2.46e-16	2.14e-16
4	2.16e-16	2.76e-16	2.93e-16	1.13e-11
5	-	-	-	-
6	-	-	-	-
7	-	-	-	-
8	2.10e-16	3.76e-16	2.64e-16	2.94e-12
9	2.70e-16	6.37e-16	1.16e-13	1.41e-13
10	7.60e-14	7.40e-14	4.01e-14	2.42e-11
11	2.27e-16	2.11e-16	2.41e-16	2.90e-11

accuracy is achieved. The refinement can be accomplished, for instance, by means of the Newton's algorithm [104] which computes the zero of a function $f(x)$ according to the iterative formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1.1}$$

In general, we would compute a starting point and $f'(x)$ in single precision arithmetic and the refinement process will be computed in double precision arithmetic.

If the refinement process is cheap enough compared to the initial computation of the solution then double precision accuracy can be achieved nearly at the same speed as the single precision accuracy. In the resulting method, the bulk of the operations is performed in 32-bit arithmetic, then we postprocess the 32-bit solution by refining it into a solution that is 64-bit accurate. This approach can be applied to linear solvers, either direct or iterative and for dense or sparse matrices. If we consider the example of the solution of linear systems, then due to round-off errors, the computed solution x carries a numerical error amplified by the condition number of the coefficient matrix A. In order to improve the computed solution, we can apply an iterative process which produces a correction to the computed solution at each iteration, which then yields the method that is commonly known as the *iterative refinement* algorithm [5, 89]. As mentioned in [34], the non-linearity of the round-off errors makes the iterative refinement process equivalent to the Newton's method applied to the function $f(x) = b - Ax$. Provided that the system is not too ill-conditioned, the algorithm produces a solution correct to the working precision. Iterative refinement in single/double precision is a fairly well understood concept and was analyzed by [74, 91, 101]. There are also techniques called extra-precise iterative refinements that can be used to compute error bounds for linear systems [32] or overdetermined least squares problems [33].

The method of iterative refinement can be modified to use a mixed precision approach for solving linear systems. If we consider for instance the solution of general square systems, then the factorization $PA = LU$ and the solution of the triangular systems $Ly =$

Pb and $Ux = y$ are computed using single precision arithmetic. The residual calculation and the update of the solution are computed using double precision arithmetic applied to the original double precision coefficients (see Algorithm 1.3.1). The most computationally expensive operation, the factorization of the coefficient matrix A, is performed using single precision arithmetic and takes advantage of its higher speed. The only operations that must be executed in double precision are the residual calculation and the update of the solution (they are denoted with an ε_d in Algorithm 1.3.1). We observe that the only operation with computational complexity of $\mathcal{O}(n^3)$ is handled in single precision, while all operations performed in double precision are of at most $\mathcal{O}(n^2)$ complexity. The coefficient matrix A is converted to single precision for the LU factorization and the resulting factors are stored in single precision while the initial coefficient matrix A needs to be kept in memory. Therefore, one drawback of the following approach is that the it uses 50% more memory than the standard double precision algorithm.

Then the method given in Algorithm 1.3.1 accelerates the solution of linear systems (either sparse or dense) if:

1. single precision computation is significantly faster than double precision computation (which is the case for current multicore and GPU architectures).

2. the iterative refinement procedure converges in a small number of steps.

3. the cost of each iteration is small compared to the cost of the system factorization. If the cost of each iteration is too high, then a low number of iterations will result in a performance loss with respect to the full double precision solver. In the sparse case, for a fixed matrix size, both the cost of the system factorization and the cost of the iterative refinement step may substantially vary depending on the number of non-zeroes and on the matrix sparsity structure. In the dense case, results are more predictable.

Note that the choice of the stopping criterion in the iterative refinement process is critical. Formulas for the error computed at each step of Algorithm 1.3.1 can be obtained for instance in [32, 77]. The componentwise backward error used for instance [2, p. 78] is expressed by

$$\omega = \max_i \frac{|A\hat{x} - b|_i}{(|A| \cdot |\hat{x}| + |b|)_i},$$

where $\hat{x}$ is the computed solution. Then the stopping criterion can be, as suggested in [89], ($\omega \leq (n+1)\varepsilon_d$).

Let us apply this method to the solution of general dense linear systems, using the hybrid LU factorization. The LU factorization in single precision is performed using the routine `magma_dgetrf` detailed in Section 1.2.2. The iterations are performed on the GPU and the CPU is used only in the factorization step for the panels. We present in Figure 1.9 performance results for a hybrid mixed precision iterative refinement solver for dense matrices and we compare them with the performance of single and double precision solvers. These experiments were carried out on the multicore+GPU system described in Section 1.2.3.1 (Magny-Cours+Fermi). We use random matrices and the iterative refinement converged in 3 iterations

These results show that the mixed precision iterative refinement method can run very close to the speed of the full single precision solver while delivering the same accuracy

Algorithm 1.3.1 Mixed precision iterative refinement using LU factorization

1: $LU \leftarrow PA$ (ε_s)
2: solve $Ly = Pb$ (ε_s)
3: solve $Ux_0 = y$ (ε_s)
4: **do** $k = 1, 2, ...$
5: $r_k \leftarrow b - Ax_{k-1}$ (ε_d)
6: solve $Uz_k = y$ (ε_s)
7: $x_k \leftarrow x_{k-1} + z_k$ (ε_d)
8: **check convergence**
9: **done**

as the full double precision one. For small problem sizes the cost of even a few iterative refinement iterations is high compared to the cost of the factorization and thus the mixed precision iterative solver is less efficient than the double precision one.

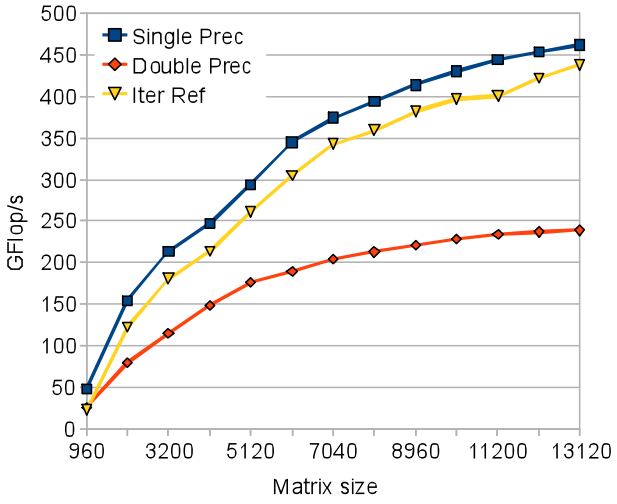


Figure 1.9: Performance for mixed precision LU-based solver on Fermi (C2050).

Chapter 2

Accelerating linear system solutions with randomized algorithms

2.1 Introduction to randomization for linear systems

The last several years saw the development of randomized algorithms in high-performance computing applications. This increased interest is motivated by the fact that the resulting algorithms are able to outperform deterministic methods while still providing very accurate results (see e.g. random sampling algorithms that can be applied to least squares solutions or low-rank matrix approximation [73]). In addition to being easier to analyze, the main advantage of such algorithms is that they can lead to much faster solution by performing a smaller number of floating-point operations (e.g. [7]), or by involving less communication (e.g. [13]). As a result, they potentially allow domain scientists to address larger simulations (which also contributes to get more accurate results). Other applications of statistical techniques to accelerate linear algebra calculations can be found for instance for solving linear systems using Monte Carlo methods [36], or for computing condition estimates [4, 66] (this application will be presented in Section 3.2.2 in the next chapter). Recently, randomization has also been applied to more general matrix decompositions [56].

However, to be of full interest for real-world applications, randomized algorithms must be able to exploit the computing capabilities of current parallel machines, which can commonly achieve performance of more than one Tflop/s per node. Since randomized algorithms are supposed to be useful for very large problems, the main challenge for them is to exploit efficiently computing units like multicore systems or GPUs and their associated memories. Another important requirement is to be able to schedule efficiently such algorithms on these architectures.

Let us develop in this chapter how randomized algorithms can be useful to enhance linear system solutions. In such solvers, a classical way to ensure stability is to use pivoting. This technique aims at preventing divisions by zero or by too-small quantities in the process of Gaussian Elimination (GE). In the case of general linear systems, we solve a linear system $Ax = b$ using a factorization

$$PA = LU \tag{2.1}$$

where P is a permutation matrix, L is unit lower triangular and U is upper triangular. The solution x can be computed by successively solving $Ly = Pb$ and $Ux = y$. The Gaussian Elimination with Partial Pivoting (GEPP) procedure permutes rows of the input matrix so that large nonzero matrix elements are moved to the diagonal to be used as "pivot". There is no floating-point operation in pivoting but it involves communication due to irregular data movements ($\mathcal{O}(n^2)$ comparisons for the partial pivoting, where n is the matrix size). GEPP turns out to be very stable in practice and has been implemented in standard linear algebra libraries (e.g. LAPACK and ScaLAPACK).

With the advent of architectures such as multicore processors or Graphics Processing Units (GPU), the growing gap between communication and computation efficiency made the communication overhead due to pivoting more critical. Moreover, in the LAPACK implementation of GEPP, rows are swapped at once during pivoting, which inhibits the exploitation of more asynchronicity between block operations. Several pivoting techniques, potentially less stable than partial or complete pivoting, have been used to minimize the communication like pairwise pivoting [90] or threshold pivoting [40] (see [96] for a stability analysis of these pivoting techniques). In particular pairwise pivoting has been implemented in algorithms for multicore machines [27] but this generates a significant overhead since the rows are swapped in pairs of blocks. We also mention, for multithreaded architectures, a pivoting technique called incremental pivoting in [84] based on principles used for out-of-core solvers. Another pivoting technique has been proposed in [51] that minimizes the number of messages exchanged during the factorization, leading to a new class of algorithms often referred to as "communication-avoiding" algorithms. More specifically for GPUs, the pivoting overhead was reduced by using an innovative data structure [99].

To illustrate the cost of pivoting, we plot in Figure 2.1 the percentage of time due to pivoting in LU factorization (MAGMA implementation) for several sizes of random matrices on a current hybrid CPU/GPU machine (in double precision arithmetic). We observe that pivoting can represent more than 40% of the global factorization time for small matrices and although the overhead decreases with the size of the matrix, it still represents 17% for a matrix of size $10,000$.

For symmetric indefinite systems, we use in general diagonal pivoting methods [25] where a block-LDL$^{\mathrm{T}}$ factorization is obtained such that

$$PAP^T = LDL^T \tag{2.2}$$

where P is a permutation matrix, L is unit lower triangular and D is block-diagonal, with blocks of size 1×1 or 2×2; all matrices are of size $n \times n$. If no pivoting is applied, *i.e.* $P = I$, D becomes diagonal. The solution x can be computed by successively solving the triangular or block-diagonal systems $Lz = Pb$, $Dw = z$, $L^T y = w$, and ultimately we have $x = P^T y$. This pivoting method turns out to be very stable in practice and is implemented in current serial dense linear algebra libraries (*e.g.* LAPACK). It requires between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons.

One of the differences between symmetric and nonsymmetric pivoting is that, independently from the pivoting technique used, columns and rows must be interchanged in the symmetric case while only rows must be swapped in the nonsymmetric case, as illustrated in Figure 2.2. This in itself makes pivoting more expensive in terms of data movement for symmetric matrices. Interchanging rows and columns also compromises data locality since noncontiguous data blocks must be moved however data are stored. There is

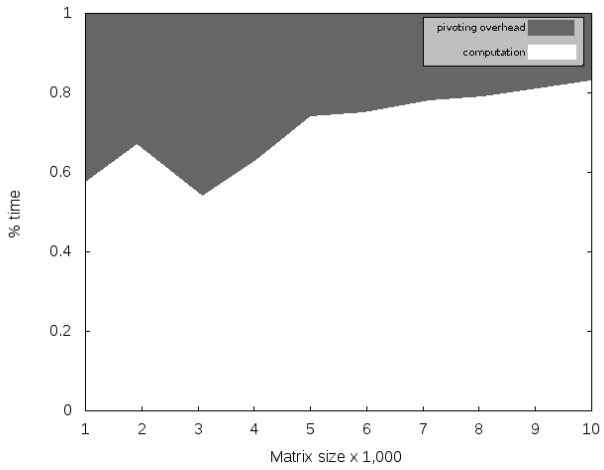


Figure 2.1: Cost of pivoting in LU factorization (CPU 1 × Quad-Core Intel Core2 Processor Q9300 @ 2.50 GHz GPU C2050 — 14 Multiprocessors (× 32 CUDA cores) @ 1.15 GHz).

also an increase of data dependencies, which inhibits parallelism, both in interchanging columns/rows and in searching for pivots. For nonsymmetric matrices, pivots are most commonly searched in a single column (partial pivoting) while for symmetric matrices the search may be extended to the diagonal and further. The fact that pivoting remains a bottleneck for linear system solutions is a motivation to study an alternative to pivoting thanks to randomization.

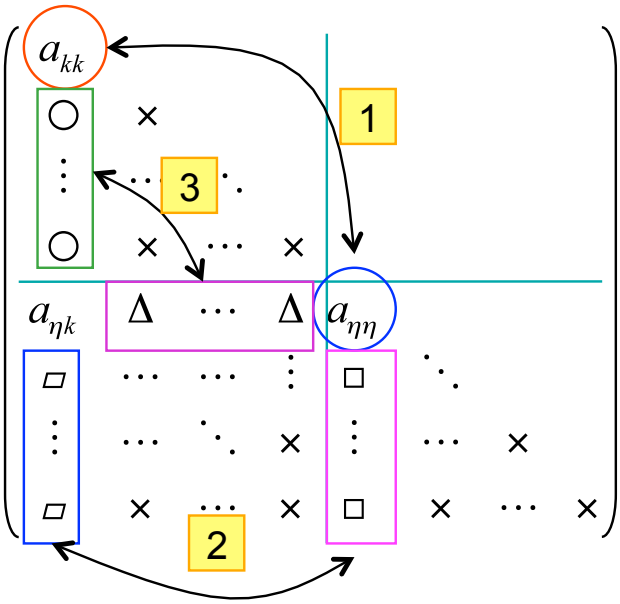


Figure 2.2: Symmetric pivoting in LDL^T factorization.

While many implementations of the LDL $^{\mathrm{T}}$ factorization have been proposed for sparse solvers on distributed and shared memory architectures [35, 47, 57, 88], there is no par-

allel implementation in the current dense linear algebra libraries ScaLAPACK [21], PLASMA [78], MAGMA [94], and FLAME [53]. These libraries have implemented solutions for the common Cholesky, LU and QR factorizations but none of them introduced a solution for indefinite symmetric matrices in spite of the gain of flops it could provide for these cases. The main reason for this comes from the algorithms used for pivoting in LDL$^{\mathrm{T}}$, which are difficult to parallelize efficiently. To our knowledge, the only research in the subject has been done by Strazdins [92] and the procedure is available in the OpenMP version of MKL [62]. At the same time, these types of linear systems are commonly encountered in optimization problems coming from physics of structures, acoustics, and electromagnetism, among others. Symmetric indefinite systems also result from linear least squares problems when they are solved via the augmented system method [20, p. 77]. This has motivated our research [8, 19] for proposing a parallel solver based on randomization for dense symmetric indefinite systems, after having applied randomization techniques for general dense systems [13].

We describe in this chapter an approach based on randomization where the original matrix A is transformed into a matrix that would be sufficiently "random" so that, with a probability close to 1, pivoting is not needed. We illustrate it by considering the case of symmetric indefinite systems.

2.2 How to avoid pivoting in linear systems using randomization

2.2.1 Symmetric Random Butterfly Transformation (SRBT)

Let us recall here the main definitions and results related to the randomization approach that is used for dense symmetric indefinite systems. The randomization of the matrix is based on a technique described in [82], revisited in [13] for general systems and applied in [19] for symmetric indefinite systems. We consider a symmetric system $Ax = b$ that can be transformed as

$$Ax = b \equiv \underbrace{U^T A U}_{A_r} \; \underbrace{U^{-1} x}_{y} = \underbrace{U^T b}_{c}. \tag{2.3}$$

Then the procedure to solve $Ax = b$ via randomization is the following:

1. Compute $A_r = U^T A U$, with U a random matrix,
2. Factorize $A_r = LDL^T$ (without pivoting),
3. Solve $A_r y = U^T b$ and compute $x = Uy$.

The random matrix U is chosen among a particular class of matrices called *recursive butterfly matrices* and the resulting transformation is referred to as **Symmetric Random Butterfly Transformation** (SRBT). To be of interest, this randomization should be cheap ($\mathcal{O}(n^2)$ operations and efficiently implemented) and the LDL$^{\mathrm{T}}$ with no pivoting should be fast (close to a "Cholesky speed"). Another requirement for this randomized solver is to provide us with an accuracy similar to the LDL$^{\mathrm{T}}$ Bunch-Kaufman algorithm [24] (implemented for instance in LAPACK).

A butterfly matrix is defined as any n-by-n matrix of the form:

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix} \tag{2.4}$$

where $n \geq 2$ and R and S are random diagonal and nonsingular $n/2$-by-$n/2$ matrices. A recursive butterfly matrix U of size n and depth d is a product of the form

$$U = U_d \times \cdots \times U_1, \tag{2.5}$$

where U_k $(1 \leq k \leq d)$ is a block diagonal matrix expressed as

$$U_k = \begin{pmatrix} B_1 & & \\ & \ddots & \\ & & B_{2^{k-1}} \end{pmatrix} \tag{2.6}$$

each B_i being a butterfly matrix of size $n/2^{k-1}$. In particular U_1 is a butterfly as defined in Formula (2.4). Note that this definition requires that n is a multiple of 2^d which can always be obtained by "augmenting" the matrix A with additional 1's on the diagonal.

We generate the random diagonal values used in the butterflies as $e^{\rho/10}$, where ρ is randomly chosen in $[-\frac{1}{2}, \frac{1}{2}]$. This choice is suggested and justified in [82] by the fact that the determinant of a butterfly has an expected value 1. Then the random values r_i used in generating butterflies are such that

$$e^{-1/20} \leq r_i \leq e^{1/20}.$$

Using these random values, it is shown in [8] that the 2-norm condition number of the randomized matrix A_r verifies

$$\mathrm{cond}_2(A_r) \leq 1.2214^d \mathrm{cond}_2(A), \tag{2.7}$$

and thus, for small values of d, the 2-norm condition number of the initial matrix A will be kept almost unchanged by the randomization.

We recall also that the LDL$^{\mathrm{T}}$ algorithm without pivoting is potentially unstable [60, p. 214], due to a possibly large growth factor. We can find in [82] explanations about how recursive butterfly transformations modify the growth factor of the original matrix A. To ameliorate this potential instability, we systematically add in our method a few steps of iterative refinement in the working precision as indicated in [60, p. 232].

We also point out that SRBT requires a limited extra storage since a butterfly matrix and a recursive butterfly matrix can be stored in a packed storage using a vector and a matrix, respectively.

2.2.2 Efficient SRBT algorithm for symmetric matrices

For simplicity, n (order of matrices U and A) is supposed to be a multiple of 2^d hereafter (if not the system is augmented with additional 1's on the diagonal). Following Equation (2.3) two kernels are required in order to transform the system $Ax = b$:

$$A_r = U^T A U \tag{2.8}$$

$$c = U^T b \tag{2.9}$$

After solving $A_r y = c$, x is obtained by

$$x = Uy \tag{2.10}$$

Equations (2.9) and (2.10) can be taken as particular cases of Equation (2.8), where the U matrix on the right side of A is an identity matrix. This means that the data dependencies described in what follows are similar but simpler for Equations (2.9) and (2.10). Since the implementation uses the same principle for all three operations, only $U^T AU$ is detailed.

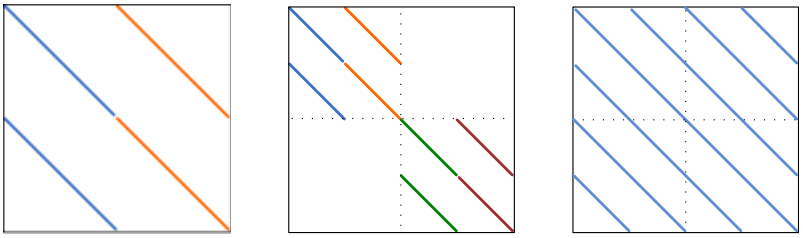
Using the definition of the recursive matrix U given in Equation (2.5), the randomized matrix A_r can be expanded as

$$A_r = U_1^T \times U_2^T \times \cdots \times U_d^T \times A \times U_d \times \cdots \times U_2 \times U_1.$$

Note that U_i is a sparse matrix, with sparsity pattern as shown in Figure 2.3, and that the matrix product of U_i results in a different sparsity pattern, which depends on the number of levels of recursion. To avoid storing the product of U_i and to maintain the symmetry, the computation can be performed by recursively computing

$$A_r^{(i-1)} = U_i^T A^{(i)} U_i, \tag{2.11}$$

where $A^{(d)} = A$ and $A_r = A_r^{(0)}$.



(a) U_1 (b) U_2 (c) $U_2 \times U_1$

Figure 2.3: Sparsity pattern of matrix U.

It can be observed that, for each level, $U_i^T A^{(i)} U_i$ can be written as blocks given by $B_i^T A_{ij} B_j$. For instance, for the second level

$$\begin{aligned} U_2^T A^{(2)} U_2 &= \begin{bmatrix} B_1^T & \\ & B_2^T \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_1 & \\ & B_2 \end{bmatrix} \\ &= \begin{bmatrix} B_1^T A_{11} B_1 & B_1^T A_{12} B_2 \\ B_2^T A_{21} B_1 & B_2^T A_{22} B_2 \end{bmatrix} \end{aligned}$$

Hence, the so-called *core kernel* of a random butterfly transformation is given by

$$B_i^T A_{ij} B_j \tag{2.12}$$

where A_{ij} is a block of A and B_* is a random butterfly matrix, both of size $m \times m$. The block A_{ij} can either be symmetric (diagonal block, *i.e.* $i = j$) or non-symmetric (off-diagonal block, *i.e.* $i \neq j$).

Recalling that B has a well defined structure

$$B = \begin{bmatrix} R & S \\ R & -S \end{bmatrix} \text{ and } B^T = \begin{bmatrix} R & R \\ S & -S \end{bmatrix}$$

where R and S are diagonal matrices, and given that A_{ij} is divided into four submatrices of same size, such as

$$A_{ij} = \begin{bmatrix} TL & TR \\ BL & BR \end{bmatrix}$$

and that

$$\begin{aligned} W_{TL} &= (TL + BL) + (TR + BR), \\ W_{BL} &= (TL - BL) + (TR - BR), \\ W_{TR} &= (TL + BL) - (TR + BR), \\ W_{BR} &= (TL - BL) - (TR - BR). \end{aligned}$$

Equation (2.12) can be written as

$$B_i^T A_{ij} B_j = \begin{bmatrix} R \cdot W_{TL} \cdot R & R \cdot W_{TR} \cdot S \\ S \cdot W_{BL} \cdot R & S \cdot W_{BR} \cdot S \end{bmatrix}.$$

Note that only the signs differ in calculating each W_*. Hence, all four cases can be generalized as

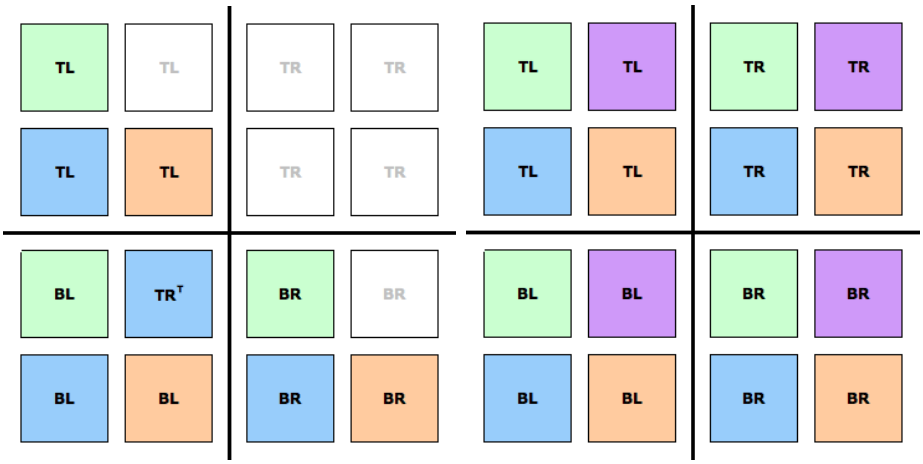
$$W = (TL \circ BL) \circ (TR \circ BR), \tag{2.13}$$

where the operator $\circ$ denotes an addition or a substraction. Equation (2.13) shows that each matrix W_* depends on all four submatrices of A_{ij}. More specifically, any given element of W depends on four elements of A. Therefore, the data dependencies among elements could be depicted as:

$$\underbrace{\left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 2 & 3 \\ 2 & 4 & 5 & 2 & 4 & 5 \\ 3 & 5 & 6 & 3 & 5 & 6 \\ \hline 1 & 2 & 3 & 1 & 2 & 3 \\ 2 & 4 & 5 & 2 & 4 & 5 \\ 3 & 5 & 6 & 3 & 5 & 6 \end{array}\right]}_{Symmetric} \quad \underbrace{\left[\begin{array}{ccc|ccc} 1 & 4 & 7 & 1 & 4 & 7 \\ 2 & 5 & 8 & 2 & 5 & 8 \\ 3 & 6 & 9 & 3 & 6 & 9 \\ \hline 1 & 4 & 7 & 1 & 4 & 7 \\ 2 & 5 & 8 & 2 & 5 & 8 \\ 3 & 6 & 9 & 3 & 6 & 9 \end{array}\right]}_{General}$$

where same numbers means these elements depend on each other. For the symmetric case, the strictly upper triangular part is not calculated, and for this work, not stored either. Hence, elements above the diagonal ($i < j$) must be computed as their transpose, *i.e.* A_{ij} is read and written as A_{ji}.

This method can be extended to blocks, or tiles to comply with the tiled LDL$^{\mathrm{T}}$ factorization that will be presented in Section 2.2.3. If each number above is taken as a tile (or block), the data dependencies can be sketched as in Figure 2.4.



(a) Symmetric kernel (b) General kernel

Figure 2.4: SRBT core kernel ($B_i^T A_{ij} B_j$), data dependencies among tiles of A_{ij}, given by matching colors.

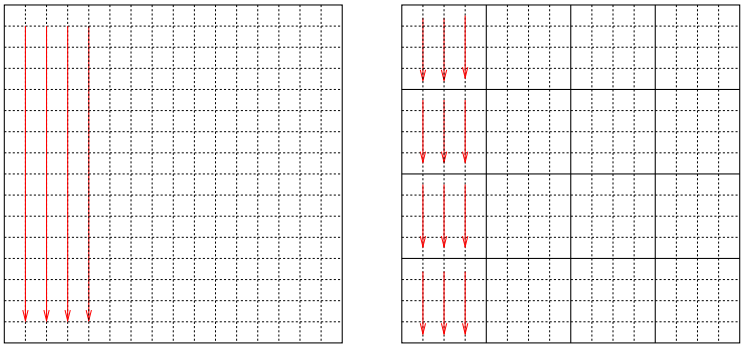
Regarding the computational cost of randomization, it depends on the order of the matrix to be transformed, n, and on the number of recursion levels, d. Using Equation (2.11), for each recursion level i we compute a matrix of the form

$$\begin{pmatrix} B_1^T A_{11}^{(i)} B_1 & \cdots & B_1^T A_{p1}^{(i)T} B_p \\ \vdots & \ddots & \vdots \\ B_p^T A_{p1}^{(i)} B_1 & \cdots & B_p^T A_{pp}^{(i)} B_p \end{pmatrix}, \tag{2.14}$$

where $p = 2^{i-1}$. Each block-matrix expressed in (2.14) requires p symmetric kernels and $p(p-1)/2$ general (nonsymmetric) kernels operating on matrices of size n/p. Therefore, the number of operations involved in randomizing A by an SRBT of depth d is

$$\begin{aligned} C(n,d) &\simeq \sum_{i=1}^{d} \left(p \cdot 2(n/p)^2 + p(p-1)/2 \cdot 4(n/p)^2 \right) \\ &= 2dn^2 \end{aligned}$$

We will consider a number of recursions d such that $d < \log_2 n \ll n$. Numerical tests, performed on a collection of matrices from the Higham's Matrix Computation Toolbox [60], will described in Section 2.3.1. They show that, in practice, $d = 2$ enables us to achieve satisfying accuracy. Similarly to the product of a recursive butterfly by a matrix, the product of a recursive butterfly by a vector does not require the explicit formation of the recursive butterfly since the computational kernel will be a product of a butterfly by a vector, which involves $\mathcal{O}(n)$ operations. Then the computation of $U^T b$ and Uy can be performed in $\mathcal{O}(dn)$ flops and, for small values of d, can be neglected compared to the $\mathcal{O}(n^3)$ cost of the factorization.



(a) Column-major (LAPACK) (b) Tile layout (PLASMA)

Figure 2.5: Column-major and tile layout sketch.

2.2.3 Tiled LDL$^{\mathrm{T}}$ factorization

In order to increase parallelism on multicore machines, we use a tiled algorithm that starts by decomposing A in $NT \times NT$ tiles (blocks), such as

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1,NT} \\ A_{21} & A_{22} & \dots & A_{2,NT} \\ \vdots & \vdots & \ddots & \vdots \\ A_{NT,1} & A_{NT,2} & \dots & A_{NT,NT} \end{bmatrix}_{N \times N}, \tag{2.15}$$

where each A_{ij} is a tile of size $NB \times NB$. The same decomposition can be applied to L and D. With this decomposition and using the principle of the Schur complement, a series of tasks can be generated to calculate each L_{ij} and D_{ii}.

The decomposition into tiles allows the computation to be performed on small blocks of data that fit into cache. This leads to the need of a reorganization of data formerly given in a column major layout, as depicted in Figure 2.5. The tile layout reorders data in such a way that all data of a single block is contiguous in memory. Thus the decomposition of the computation can either be statically scheduled to take advantage of cache locality and reuse or be dynamically scheduled based on dependencies among data and computational resources available.

The tiled algorithm for the LDL$^{\mathrm{T}}$ factorization is based on the following operations:

`xSYTRF`: This LAPACKbased subroutine is used to perform the LDL$^{\mathrm{T}}$ factorization of a symmetric tile A_{kk} of size $NB \times NB$ producing a unit triangular tile L_{kk} and a diagonal tile D_{kk}.
Using the notation *input* $\longrightarrow$ *output*, the call `xSYTRF`(A_{kk} , L_{kk} , D_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk}, D_{kk} = \mathrm{LDL}^{\mathrm{T}} \ (A_{kk})$$

`xSYTRF2`: This subroutine first calls `xSYTRF` to perform the factorization of A_{kk} and then multiplies L_{kk} by D_{kk}. The call `xSYTRF2`(A_{kk} , L_{kk} , D_{kk} , W_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk}, D_{kk} = \text{LDL}^{\text{T}}\ (A_{kk}),$$
$$W_{kk} = L_{kk} D_{kk}$$

`xTRSM`: This BLAS subroutine is used to apply the transformation computed by xSYTRF2 to an A_{ik} tile by means of a triangular system solve. The call `xTRSM`$(W_{kk},\ A_{ik})$ performs

$$W_{kk}, A_{ik} \longrightarrow L_{ik} = A_{ik} W_{kk}^{-T}$$

`xSYDRK`: This subroutine is used to update the tiles A_{kk} in the trailing submatrix by means of a matrix-matrix multiply. It differs from `xGEMDM` by taking advantage of the symmetry of A_{kk} and by using only the lower triangular part of A and L. The call `xSYDRK`$(A_{kk},\ L_{ki},\ D_{ii})$ performs

$$A_{kk}, L_{ki}, D_{ii} \longrightarrow A_{kk} = A_{ik} - L_{ki} D_{ii} L_{ki}^{T}$$

`xGEMDM`: This subroutine is used to update the tiles A_{ij} for $i \neq j$ in the trailing submatrix by means of a matrix-matrix multiply. The call `xGEMDM`$(A_{ij},\ L_{ik},\ L_{jk},\ D_{kk})$ performs

$$A_{ij}, L_{ik}, L_{jk}, D_{kk} \longrightarrow A_{ij} = A_{ij} - L_{ik} D_{kk} L_{jk}^{T}$$

Given a symmetric matrix A of size $N \times N$, NT as the number of tiles, such as in Equation (2.15), and making the assumption that $N = NT \times NB$ (for simplicity), where $NB \times NB$ is the size of each tile A_{ij}, then the tiled LDL^{T} algorithm can be described as in Algorithm 2.2.1. A graphical representation of Algorithm 2.2.1 is depicted in Figure 2.6.

Algorithm 2.2.1 Tile LDL^{T} Factorization

```
1: for k = 1 to NT do
2:     xSYTRF2( A_kk , L_kk , D_kk , W_kk )
3:     for i = k + 1 to NT do
4:         xTRSM( W_kk , A_ik )
5:     end for
6:     for i = k + 1 to NT do
7:         xSYDRK( A_kk , L_ki , D_ii )
8:
9:         for j = k + 1 to i − 1 do
10:            xGEMDM( A_ij , L_ik , L_jk , D_kk )
11:        end for
12:    end for
13: end for
```

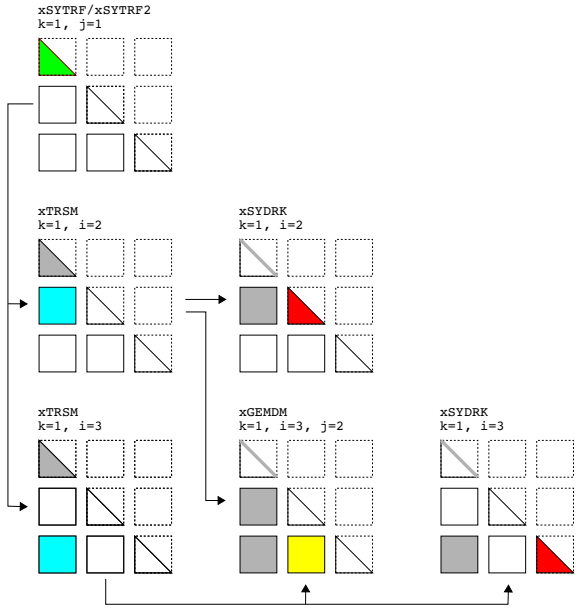


Figure 2.6: Graphical representation with dependencies of one repetition of the outer loop in Algorithm 2.2.1 with $NT = 3$.

2.2.4 Scheduling issues

Following the approach presented in [26], Algorithm 2.2.1 can be represented as a Directed Acyclic Graph (DAG) where nodes are elementary tasks that operate on one or several $NB \times NB$ blocks and where edges represent the dependencies among them. A dependency occurs when a task must access data outputted by another task either to update or to read them. Figure 2.7 shows a DAG for the tiled $\mathrm{LDL}^{\mathrm{T}}$ factorization when Algorithm 2.2.1 is executed with $NT = 4$. Once the DAG is known, the tasks can be scheduled asynchronously and independently as long as the dependencies are not violated.

This dynamic scheduling results in an out-of-order execution where idle time is almost completely eliminated since only very loose synchronization is required between the threads. Figure 2.8 (a) shows the execution trace of Algorithm 2.2.1 where tasks are dynamically scheduled, based on dependencies in the DAG, and run on 8 cores of the *MagnyCours-48*machine (described in Section 2.3.2). The tasks were scheduled using QUARK [103], which is the scheduler available in the PLASMA library. Each row in the execution flow shows which tasks are performed and each task is executed by one of the threads involved in the factorization. The trace follows the same color code as Figure 2.6.

Figure 2.8 (b) shows the trace of Algorithm 2.2.1 using static scheduling, which means that each core's workload is predetermined. The synchronization of the computation for correctness is enforced by a global progress table. The static scheduling technique has two important shortcomings. First is the difficulty of development. It requires full understanding of the data dependencies in the algorithm, which is hard to acquire even by an experienced developer. Second is the inability to schedule dynamic algorithms, where the complete task graph is not known beforehand. This is the common situation for eigenvalue algorithms, which are iterative by nature. Finally, it is almost impossible

with the static scheduling to overlap simply and efficiently several functionalities like the factorization and the solve that are often called simultaneously. However for a single step, as can be seen in Figure 2.8, the static scheduling on a small number of cores may outrun the dynamic scheduling due to better data locality and cache reuse. It is important to highlight that developing an efficient static scheduling can be very difficult and that the dynamic scheduler notably reduces the complexity of programing tiled algorithms.

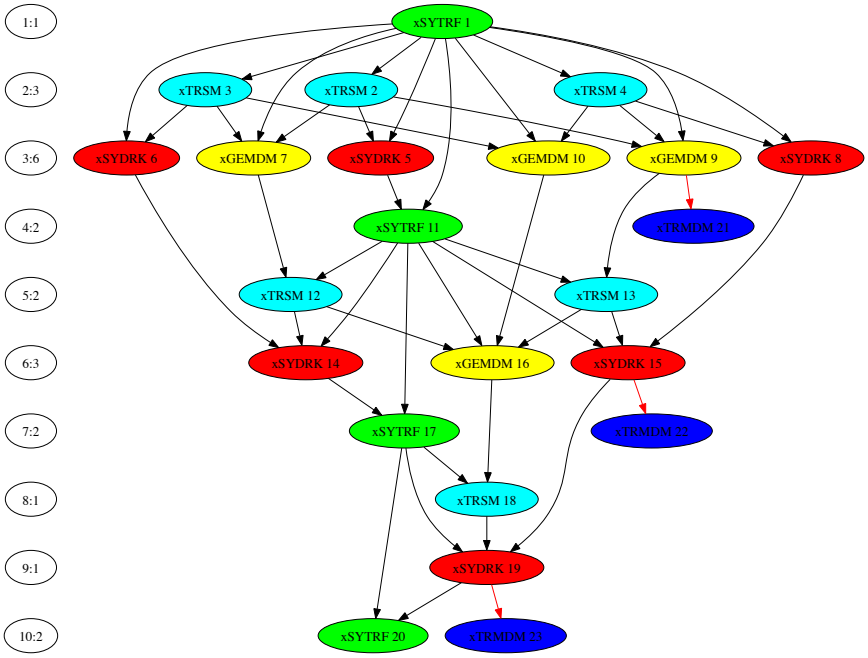
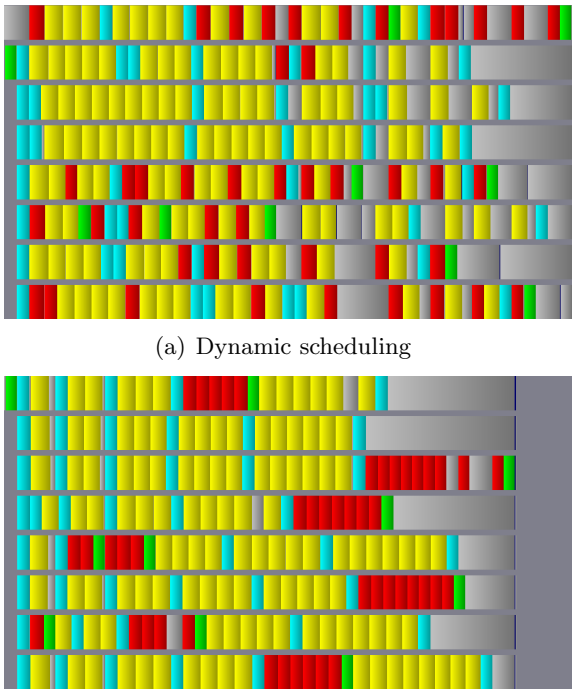


Figure 2.7: DSYTRF DAG; $NT = 4$.

2.3 Numerical experiments

2.3.1 Accuracy results

Experiments to measure the accuracy of each procedure described in the previous sections were carried out using Matlab version 7.12 (R2011a) on a machine with a precision of $2.22 \cdot 10^{-16}$. Table 2.1 presents accuracy comparisons of linear systems solved using the factors of A calculated by $\mathrm{LDL}^{\mathrm{T}}$ with: no pivoting (NP), partial pivoting (PP), tile-wise pivoting (TP), and the Symmetric Random Butterfly Transformation followed by no pivoting (SRBT NP) and tile-wise pivoting (SRBT TP). For tile-wise pivoting (TP), the matrices have 64 tiles (8×8). The partial pivoting corresponds to the Bunch-Kaufman algorithm as it is implemented in LAPACK. The tile-wise pivoting corrsponds to a tiled algorithm where pivoting is performed within a tile (the LAPACK-like factorization is ap-



(a) Dynamic scheduling

(b) Static scheduling

Figure 2.8: Traces of tiled LDL$^{\mathrm{T}}$ (*MagnyCours-48*with 8 threads).

plied to the tile A_{kk}). Tile-wise pivoting does not guarantee the accuracy of the solution; it strongly depends on the matrix to be factorized and how the pivots are distributed. However, it guarantees numerical stability of the factorization of each tile A_{kk}, as long as an appropriate pivoting technique is applied. Note that for all experiments the rook pivoting [6] achieves the same accuracy as the partial pivoting and therefore is not listed.

All matrices are of size 1024×1024, either belonging to the Matlab gallery or the Higham's Matrix Computation Toolbox [60] or generated using Matlab function `rand`. Matrices $|i-j|$, $max(i,j)$ and `Hadamard` are defined in the experiments performed in [82]. Matrices `rand1` and `rand2` correspond to random matrices with entries uniformly distributed in $[0,1]$ with all and $1/4$ of the diagonal elements set to 0, respectively. Matrices `rand0` and `rand3` are also random matrices, where the latter has its diagonal elements scaled by $1/1000$.

For all test matrices, we suppose that the exact solution is $x = [1\ 1 \ldots 1]^T$ and we set the right-hand side $b = Ax$. In Table 2.1, the 2-norm condition number of each matrix is listed. Note that we also computed the condition number of the randomized matrix which, similarly to [13], is of same order of magnitude as cond A and therefore is not listed. For each LDL$^{\mathrm{T}}$ solver, the component-wise backward error is reported. The latter is defined in [77] and expressed as

$$\omega = \max_i \frac{|A\hat{x} - b|_i}{(|A| \cdot |\hat{x}| + |b|)_i},$$

where $\hat{x}$ is the computed solution.

As explained in Section 2.2.1, the random entries used to generate the butterflies are

Table 2.1: Component-wise backward error for LDL$^{\mathrm{T}}$ solvers on a set of test matrices of size 1024×1024 and 64 tiles (8×8) when applicable.

Matrix	Cond A	NP	PP	TP	SRBT	
					NP (IR)	TP (IR)
`condex`	$1 \cdot 10^{2}$	$5 \cdot 10^{-15}$	$6 \cdot 10^{-15}$	$7 \cdot 10^{-15}$	$6 \cdot 10^{-15}$ (0)	$4 \cdot 10^{-15}$ (0)
`fiedler`	$7 \cdot 10^{5}$	Fail	$2 \cdot 10^{-15}$	$7 \cdot 10^{-15}$	$9 \cdot 10^{-15}$ (0)	$1 \cdot 10^{-15}$ (0)
`orthog`	$1 \cdot 10^{0}$	$8 \cdot 10^{-1}$	$1 \cdot 10^{-14}$	$5 \cdot 10^{-1}$	$3 \cdot 10^{-16}$ (1)	$4 \cdot 10^{-16}$ (1)
`randcorr`	$3 \cdot 10^{3}$	$4 \cdot 10^{-16}$	$3 \cdot 10^{-16}$	$4 \cdot 10^{-16}$	$5 \cdot 10^{-16}$ (0)	$3 \cdot 10^{-16}$ (0)
`augment`	$5 \cdot 10^{4}$	$7 \cdot 10^{-15}$	$4 \cdot 10^{-15}$	$8 \cdot 10^{-15}$	$2 \cdot 10^{-16}$ (1)	$5 \cdot 10^{-15}$ (0)
`prolate`	$6 \cdot 10^{18}$	$8 \cdot 10^{-15}$	$8 \cdot 10^{-16}$	$2 \cdot 10^{-15}$	$2 \cdot 10^{-15}$ (0)	$1 \cdot 10^{-15}$ (0)
`toeppd`	$1 \cdot 10^{7}$	$5 \cdot 10^{-16}$	$7 \cdot 10^{-16}$	$6 \cdot 10^{-16}$	$2 \cdot 10^{-16}$ (0)	$1 \cdot 10^{-16}$ (0)
`ris`	$4 \cdot 10^{0}$	Fail	$3 \cdot 10^{-15}$	$8 \cdot 10^{-1}$	$6 \cdot 10^{-1}$ (10)	$6 \cdot 10^{-1}$ (10)
$\lvert i-j \rvert$	$7 \cdot 10^{5}$	$2 \cdot 10^{-15}$	$2 \cdot 10^{-15}$	$7 \cdot 10^{-15}$	$1 \cdot 10^{-14}$ (0)	$1 \cdot 10^{-15}$ (0)
`max(i,j)`	$3 \cdot 10^{6}$	$2 \cdot 10^{-14}$	$2 \cdot 10^{-15}$	$5 \cdot 10^{-15}$	$1 \cdot 10^{-14}$ (0)	$1 \cdot 10^{-15}$ (0)
`Hadamard`	$1 \cdot 10^{0}$	0	0	0	$7 \cdot 10^{-15}$ (0)	$4 \cdot 10^{-15}$ (0)
`rand0`	$2 \cdot 10^{5}$	$1 \cdot 10^{-12}$	$7 \cdot 10^{-14}$	$1 \cdot 10^{-13}$	$1 \cdot 10^{-15}$ (1)	$1 \cdot 10^{-15}$ (0)
`rand1`	$2 \cdot 10^{5}$	Fail	$1 \cdot 10^{-13}$	$2 \cdot 10^{-11}$	$1 \cdot 10^{-15}$ (1)	$1 \cdot 10^{-15}$ (0)
`rand2`	$1 \cdot 10^{5}$	Fail	$5 \cdot 10^{-14}$	$6 \cdot 10^{-13}$	$1 \cdot 10^{-15}$ (1)	$2 \cdot 10^{-15}$ (0)
`rand3`	$8 \cdot 10^{4}$	$4 \cdot 10^{-13}$	$7 \cdot 10^{-14}$	$4 \cdot 10^{-13}$	$1 \cdot 10^{-15}$ (1)	$1 \cdot 10^{-15}$ (0)

NP: LDL$^{\mathrm{T}}$ with No Pivoting
PP: LDL$^{\mathrm{T}}$ with Partial Pivoting
TP: LDL$^{\mathrm{T}}$ with Tile-wise Pivoting
SRBT: Symmetric Random Butterfly Transformation followed by LDL$^{\mathrm{T}}$ without pivoting
IR: Iterative refinement number of iterations

chosen as $exp(\frac{r}{10})$ where r is randomly chosen in $[-\frac{1}{2}, \frac{1}{2}]$ (matlab instruction `rand`). The number of recursions d used in the SRBT algorithm has been set to 2. Hence, the resulting cost of SRBT is $\sim 4n^2$ operations (see end of Section 2.2.2). To improve the stability, iterative refinement (in the working precision) is added when SRBT is used. Similarly to [5, 89], the iterative refinement algorithm is called while $\omega > (n+1)u$, where u is the machine precision. The number of iterations (IR) in the iterative refinement process is also reported in Table 2.1.

For all matrices, except `orthog` and `ris` with TP and `ris` with SRBT, the factorization with both tile-wise pivoting and randomization achieves satisfactory results. Iterative refinement turns out to be necessary in a few cases when using SRBT but with never more than one iteration (except for `ris` for which neither TP nor SRBT have achieved accurate results). SBRT TP shows slightly better results than SRBT NP. The former only requires iterative refinement for one of the test cases while the latter for a few. For matrix `prolate`, all methods result in a small backward error. However, the solution cannot be accurate at all due to the large condition number. Note that when matrices are orthogonal (`orthog`) or proportional to an orthogonal matrix (`Hadamard`), LDL$^{\mathrm{T}}$ must not be used. Also, `toeppd` is positive definite and would normally be solved by Cholesky and not LDL$^{\mathrm{T}}$. These three test cases have been used only for testing purposes. In the case of the integer-valued matrix `Hadamard` , SRBT destroys the integer structure and transforms the initial matrix into a real-valued one. For the four random matrices, TP achieves results slightly less accurate than SRBT. However, in these cases iterative

refinement added to TP would enable us to achieve an accuracy similar to SRBT.

TP and SRBT are always more accurate than NP but they both failed to produce results as accurate as PP for at least one of the test matrices. Nevertheless, despite the reduced number of test cases, they cover a reasonable range of matrices, including those with zeros on the diagonal. Test case `rand1` has only zeros on the diagonal and was accurately solved by both techniques. This case fails at the very first step of the LDL$^{\mathrm{T}}$ method without pivoting. Test case `orthog` has been solved accurately with SRBT but not with TP. For this particular case, when the pivot search is applied on the full matrix, rows/columns 1 and n are permuted, then rows/columns 2 and $n-1$ are permuted, and so forth. In others, the pivots are spread far apart and the tile-wise pivoting cannot reach them, *i.e.* there are not *good enough* pivots within each tile.

2.3.2 Performance results

We present numerical experiments for our parallel LDL$^{\mathrm{T}}$ solver where the randomization by SRBT is computed as described in Section 2.2.2 with a maximum of 2 recursions. The LDL$^{\mathrm{T}}$ algorithm presented in Section 2.2.3 has been implemented by following the software development guidelines of PLASMA, the Parallel Linear Algebra Software for Multicore Architectures library [78]. In the remainder of this section, our solver for symmetric indefinite systems will be designated as SRBT-LDL$^{\mathrm{T}}$.

The numerical results that follow are presented for both a static and a dynamic scheduler (see Section 2.2.4) and have been carried out using the *MagnyCours-48*system. This machine has a NUMA architecture and is composed of four AMD Opteron 6172 Magny-Cours CPUs running at 2.1GHz with twelve cores each (48 cores total) and 128GB of memory. The theoretical peak of this machine is 403.2 Gflop/s(8.4 Gflop/sper core) in double precision. Comparisons are made against version 10.3.2 of the Intel MKL [62] library for multicore, and against the reference LAPACK 3.2 from Netlib, linked with the same MKL BLAS multi-threaded library. SRBT-LDL$^{\mathrm{T}}$ is linked with the sequential version of MKL for the required BLAS and LAPACK routines. This version of MKL achieves 7.5 Gflop/son a `DGEMM` (matrix-matrix multiplication) on one core. Unless otherwise stated, the measurements were carried out using all the 48 cores of the *MagnyCours-48*system and run with `numactl -interleaved=0-#`. Also, a tile size of $NB = 250$ and an inner-blocking size of $IB = 125$.

Figure 2.9 shows the performance in Gflop/s of our SRBT solver against both MKL and LAPACK LDL$^{\mathrm{T}}$ routines xSYTRS (for real and double real) and xHETRS (for complex and double complex). Note that we compare solvers that do not perform the same operations because our solver does randomization (SRBT) and LDL$^{\mathrm{T}}$ with no pivoting while the other two solvers include pivoting. However, a definite matrix has been chosen for performance comparison so that no permutations are actually made by MKL and LAPACK(only search for pivot is performed). In our randomized solver, we use a tile layout where data is stored in block $NB \times NB$ (tiles) and performance is reported with dynamic and static scheduling for four arithmetic precisions (real, double real, complex, double complex). The static scheduling usually outruns the dynamic one, mostly due to the overhead of the dynamic scheduler. As mentioned before, the progress table for SRBT-LDL$^{\mathrm{T}}$ is quite efficient, exposing the overhead caused by the dynamic scheduler. We observe that SRBT-LDL$^{\mathrm{T}}$ is about twice faster than MKL and four times faster than

LAPACKfor all the four arithmetic precisions presented in Figure 2.9.

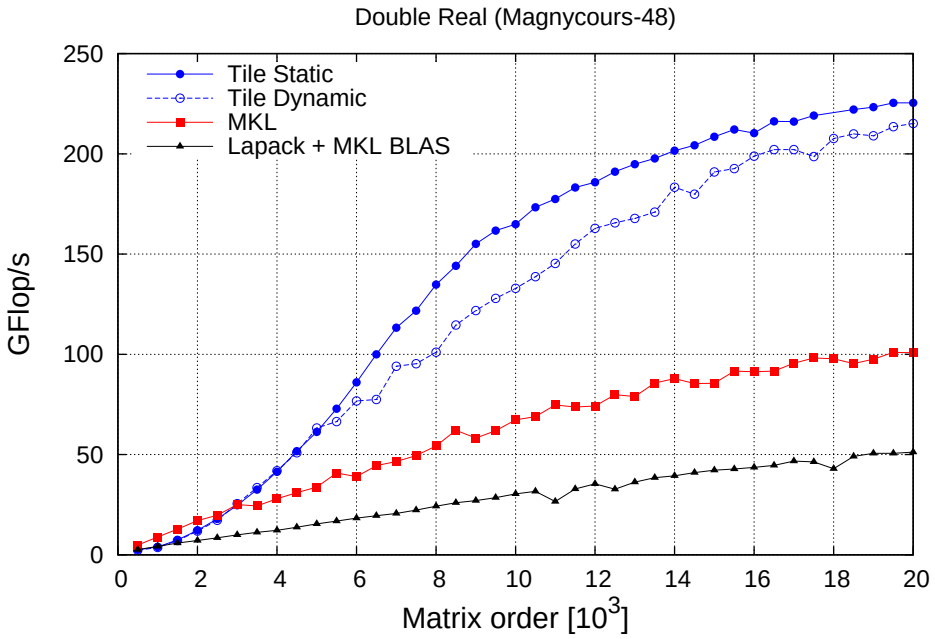


Figure 2.9: Performance of SRBT solver against MKL and LAPACK (double precision)

Let us now study specifically the performance of the tiled LDL^{T} factorization routine described in Section 2.2.3 that represents the bulk of the computation our solver. This performance is compared with that of the LU (`DGETRF`) and Cholesky (`DPOTRF`) factorization routines from the version 2.4.1 of the PLASMA library. Since there is no LDL^{T} factorization in PLASMA and by analogy to LAPACK, the tiled LDL^{T} factorization routine is named here `DSYTRF`double precision real arithmetic. Figure 2.10 reports the execution time of `DSYTRF`, `DPOTRF`and `DGETRF`with dynamic and static scheduling. The static scheduling scheme usually delivers the highest performance. This happens mostly because there is no overhead on scheduling the tasks and, as mentioned before, the LDL^{T} algorithm lends itself a quite efficient progress table. As expected, LDL^{T} is noticeably faster than LU and only moderately slower than Cholesky. This clearly states that it is advantageous, in terms of time, to factorize a symmetric matrix using `DSYTRF`(instead of `DGETRF`) and also that `DSYTRF`(instead of `DPOTRF`) can be used on decomposing SPD matrices (or diagonally dominant matrices, because they do not require pivoting) with no substantial time overhead.

The parallel speedup or scalability of `DSYTRF`is shown in Figure 2.11 for matrices of order 5000, 10000 and 20000 [N], both for dynamic and static scheduling. As anticipated, the parallel speedup increases as the matrix order increases. This happens because the bigger the matrix, the more tasks are available to be executed concurrently, resulting in higher scalability.

The parallel performance actually depends on several factors, one of them being the tile size [NB]. The performance reported previously has been obtained with $NB = 250$ and $IB = 125$, where IB stands for the internal blocking size that is used by the

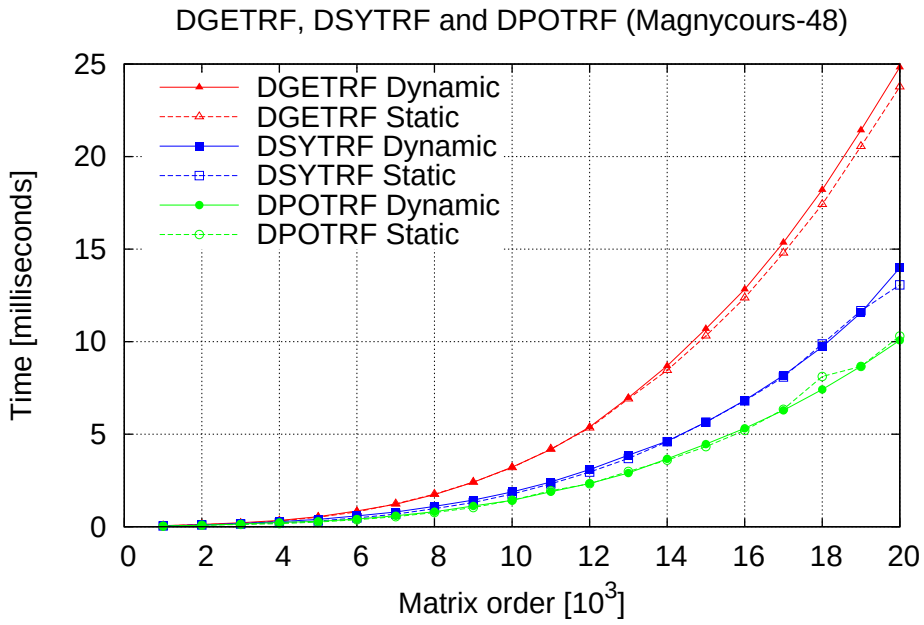


Figure 2.10: Execution time of Cholesky (PLASMA), LU (PLASMA) and tiled LDL ^{T} , dynamic (solid line) and static (dashed line) scheduling - double precision.

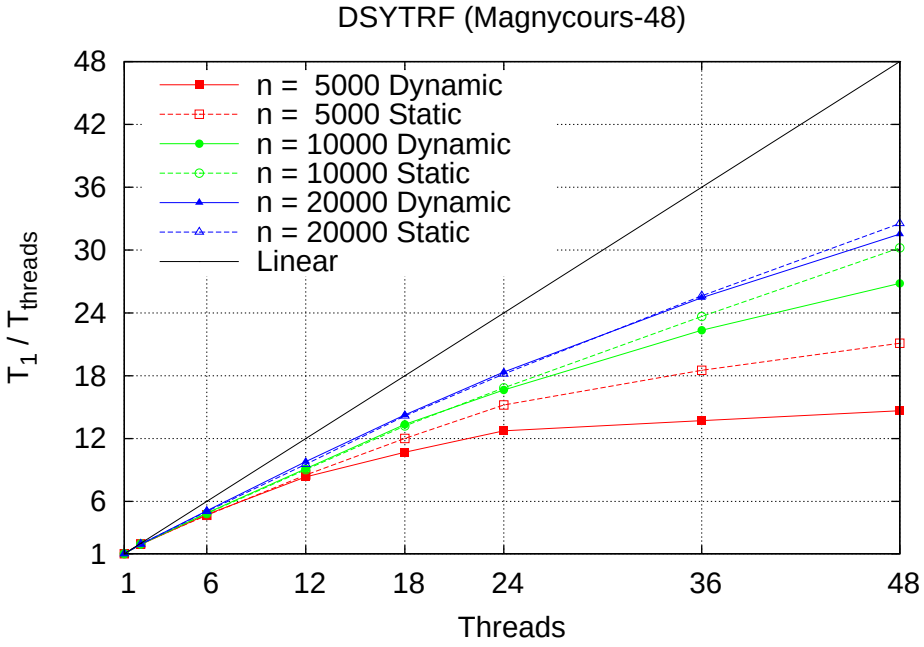


Figure 2.11: Parallel speed-up; dynamic (solid line) and static (dashed line) scheduling

blocked BLAS routines. As depicted in Figure 2.12, 250 is not necessarily the optimal tile size. In order to achieve optimal performance, NB and other parameters must be tuned accordingly to the size of the matrix to be decomposed and the number of threads.

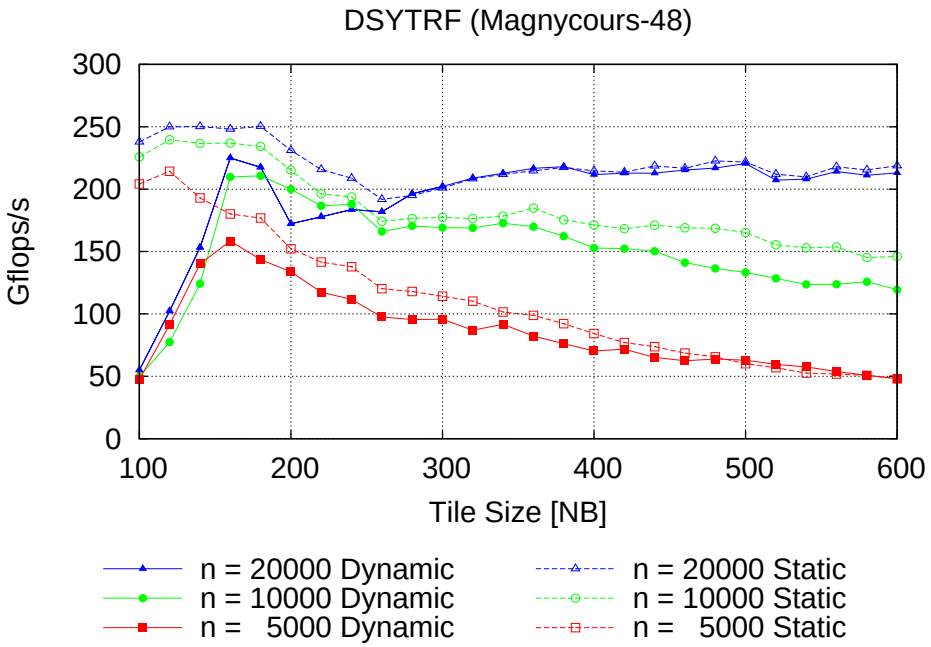


Figure 2.12: Tile-size performance of tiled LDL ^{T}

Chapter 3

Using condition numbers to assess numerical quality in high-performance computing applications

3.1 Introduction to condition numbers

In addition to performing fast computations on today's parallel systems, one major challenge for the high-performance computing community is to provide physicists with a reliable indicator for the difficulty to solve a numerical problem with a satisfying accuracy. Indeed most numerical algorithms are implemented in finite-precision floating-point arithmetic generating rounding and truncation errors. Other types of errors might also affect the numerical quality of solutions: errors due to instrumental measurements, model errors (*e.g.* errors coming from linearization when the initial problem is nonlinear, or simplification in the physics)... More generally, controlling rounding errors in numerical algorithms has always been a major concern in scientific computing and numerical validation has been the subject of extended research in recent years [18, 68, 70, 71, 87]. For instance some specific numerical methods have been developed to study rounding-error propagation using interval arithmetic [86] or stochastic arithmetic [70].

The method of error analysis that we consider in this chapter is based on an approach referred to as *backward error analysis* in [101] in which various types of errors can be considered. The notion of "sensitivity" of a solution to change in data was initially introduced by Alan Turing [97] as a way to measure the mathematical difficulty to solve a problem. This sensitivity is commonly measured using a quantity named *condition number* which is defined by Rice [85] as the maximum amplification factor between a small perturbation in the data and the resulting change in the problem solution. Evaluating the condition number of a problem is crucial for assessing the accuracy of a computed solution (assuming the underlying algorithm is stable).

Errors on solutions are classically measured using *forward errors* while errors on data are measured using *backward errors*. Let us recall briefly how these quantities are defined by considering the following mathematical problem where the solution x can be expressed

as a function g of a data y:

$$(\mathcal{P}) \quad x = g(y),$$

the data and solution spaces E and G being equipped respectively with norms $\|\cdot\|_E$ and $\|\cdot\|_G$. Then the computed solution $\tilde{x}$ of $(\mathcal{P})$ satisfies a "perturbed" problem

$$(\mathcal{P}') \quad \tilde{x} = g(y + \Delta y),$$

where $(\mathcal{P}')$ is a nearby problem for which $\tilde{x}$ is an exact solution. Then the *absolute forward error* is expressed as $\|x - \tilde{x}\|$. We often consider the *relative forward error* $\|x - \tilde{x}\|/\|x\|$ which has the advantage of being independent to scaling. The *backward error* measures the change in data and corresponds to the distance between $(\mathcal{P})$ and $(\mathcal{P}')$. If we have an approximate solution $\tilde{x}$, it can be expressed as

$$\eta(\tilde{x}) = \inf\{\|\Delta y\| : \tilde{x} = g(y + \Delta y)\}.$$

If g is differentiable, which is the case for most linear algebra problems, then the *absolute condition number* of g at $y \in E$ is defined in [43] by

$$\kappa(y) = \left|\left|\left|g'(y)\right|\right|\right| = \max_{z \neq 0} \frac{\|g'(y).z\|_G}{\|z\|_E}, \tag{3.1}$$

where $|||\cdot|||$ denotes the operator norm subordinated to the norms $\|\cdot\|_E$ and $\|\cdot\|_G$. Note that $\kappa(y)$ can be normalized using the expression $\kappa(y)\|y\|_E/\|g(y)\|_G$ resulting in a so-called *relative condition number*.

A common rule of thumb [60] gives the relationship between these quantities as

$$\|x - \tilde{x}\| \sim \eta(\tilde{x}) \times \kappa(y). \tag{3.2}$$

A a result the condition number can help predicting the error on the solution. It can also indicate in a linear solver if regularization (or iterative refinement) could be necessary to improve the quality of the solution. Of course, since the condition number is a measure of sensitivity at first order, Equation (3.2) will be accurate only if the first order asumption is realistic.

The main difficulty of this approach comes from the possibility or not to compute or estimate the condition number of a given problem. In general it requires that the solution can be expressed as an explicit function of the data (or expressed implicitely as $F(x, y) = 0$) and that this function is differentiable [85]. So in general condition numbers are known for only a limited class of mostly linear problems and not for all parameter sensitivity problems. The targetted applications in our research for condition number calculations are linear algebra solvers which are at the heart of many HPC applications. For most of these solvers, the function g defining the problem $(\mathcal{P})$ is differentiable. The condition number can then be expressed like in Equation (3.1) *i.e.* as an operator norm of a linear function. This measure is an attainable bound in the limit as $\Delta y \to 0$, and may therefore be approximate depending on the size of the perturbations. In general, the larger the ill-condition of the problem, the smaller the perturbations should be for this measure to provide a good bound, or guide to the possible solution change. Note that the forward error predicted by Equation (3.2) by the condition number corresponds to the "worst case" and if it is not too large, it enables us to assess the numerical quality

of a solution or, on the contrary, to warn the user that some perturbations can generate a significant error. The choice of metrics used to measured errors on data and solution space and their differentiability is also important.

In this chapter we describe how condition numbers can be computed or estimated for some linear algebra problems, namely linear least squares and linear systems. In Section 3.2, we study the conditioning of linear least squares (LLS) problems. We provide computable expressions and statistical estimates for the conditioning of LLS problems and we show that the resulting computational cost is affordable and much cheaper than the cost for the solution itself. We finally describe how these condition numbers can be computed in practice using current parallel libraries respectively for distributed memory and heterogeneous multicore+GPU computers. In Section 3.3, we study the conditioning of the total least squares (TLS) problem. We provide computable estimates for the conditioning of the TLS problem and we show on some experiments the limitation of the first order approach when using Equation (3.2) to predict the error on the solution.

3.2 Computing least squares condition numbers

3.2.1 Least squares conditioning

We study in this section the conditioning of the overdetermined LLS problem *i.e.* the problem $(\mathcal{P})$ mentioned in Section 3.1 can be expressed here as

$$\min_{x \in \mathbb{R}^n} \|Ax - b\|_2, \tag{3.3}$$

with $A \in \mathbb{R}^{m \times n}, m \geq n$ and $b \in \mathbb{R}^m$. Supposing that A has full column rank, we have a unique solution $x = (A^T A)^{-1} A^T b = A^\dagger b$ ($A^\dagger$ denotes the pseudo-inverse of A). Note that consistent linear systems can be considered as special cases of LLS where we have $m = n$ and a residual $r = b - Ax$ equal to zero.

The data and solution spaces are then respectively $E = \mathbb{R}^{m \times n} \times \mathbb{R}^m$ and $G = \mathbb{R}^n$ and the function g for which we want to compute the condition number is

$$\begin{array}{rccc} g \ : & \mathbb{R}^{m \times n} \times \mathbb{R}^m & \longrightarrow & \mathbb{R}^n \\ & A, b & \longmapsto & g(A,b) = x(A,b) = (A^T A)^{-1} A^T b. \end{array} \tag{3.4}$$

Since A has full rank n, g is continuously differentiable in a neighbourhood of (A, b).

LLS are in general more sensitive to change (or errors) in data than linear systems, in particular when the right-hand side is too far from the column space (see *e.g.* [63, p. 98]). It is then crucial to be able to assess the quality of the solutions obtained after solving these problems in practical applications. It was shown in [44] that the 2-norm condition number of the matrix A, defined as $\kappa_2(A) = \|A\|_2 \|A^+\|_2$ plays a significant role in the sensitivity analysis of least squares problems. It was later proved in [100] that the sensitivity of LLS problems is proportional to $\kappa_2(A)$ when the residual vector is small and to $\kappa_2(A)^2$ otherwise. Then [43] provided a closed formula for the condition number of linear least squares problems, using the Frobenius norm to measure the perturbations of A. Since then many results on normwise LLS condition numbers have been published (see *e.g.* [4, 20, 41, 49, 50]).

It was observed in [59] that the normwise condition number approach can lead to a loss of information. Indeed it consolidates all sensitivity information into a single number while in some cases this sensitivity can vary significantly among the different solution components (some examples for LLS are presented in [12, 66]). To overcome this issue, it was proposed the notion of "componentwise" condition numbers or condition numbers for the solution components [29]. Note that this approach must be distinguished from the componentwise metric also applied to LLSP for instance in [15, 31]. This approach was generalized by the notion of *partial* or *subspace* condition numbers, *i.e.* the conditioning of $L^T x$ with $L \in \mathbb{R}^{n\times k}, k \leq n$, proposed for instance in [4] for least squares or [28] for linear systems. When L is a canonical vector, it is equivalent to the condition number of a specific component, while when L is the identity matrix, it is the same as the classical condition number mentioned above. The motivation for computing the conditioning of $L^T x$ can be found for instance in [4, 12] for normwise LLS condition numbers.

Even though the condition numbers provide interesting information about the quality of the computed solution, they are expected to be calculated in an acceptable time compared to the cost for the solution itself. Computing the exact (subspace or not) condition number requires $\mathcal{O}(n^3)$ flops when a QR factorization (or normal equations) has been used to solve the LLS and can be reused to compute the conditioning [4, 12]. Although this cost is affordable compared to the cost for solving the problem ($\mathcal{O}(mn^2)$ flops), statistical estimates can reduce this cost to $\mathcal{O}(n^2)$ [4, 52, 65, 66]. The theoretical quality of the statistical estimates can be formally measured by the probability to give an estimate in a certain range around the exact value.

In general we are interested in computing the LLS condition numbers for two special cases. The first case is when L is the identity matrix (conditioning of the solution) and the second case is when L is a canonical vector e_i (conditioning of a solution component).

Regarding the choice of norm to measure perturbations, we proposed in several papers [4, 12] to use the Euclidean norm for the solution space and, for the data space, a product norm defined by

$$\|(\Delta A, \Delta b)\|_F = \sqrt{\alpha^2 \|\Delta A\|_F^2 + \beta^2 \|\Delta b\|_2^2},\ \alpha, \beta > 0,$$

where $\|\cdot\|_F$ denotes the Frobenius matrix norm, and α, β are two positive real numbers. This product norm has the advantage of being very flexible because the coefficients α and β allow us to monitor the perturbations on A and b. For instance, large values of α (resp. β) enable us to obtain condition number problems where mainly b (resp. A) are perturbed. For instance we can address the case where only b (resp. A) is perturbed by choosing $\alpha = +\infty$ and $\beta = 1$ (resp. $\alpha = 1$ and $\beta = +\infty$) Another classical choice corresponds to the case where perturbations on the data ΔA and Δb are measured relatively to the original data A and b, *i.e.* $\alpha = \frac{1}{\|A\|_F}$ and $\beta = \frac{1}{\|b\|_2}$.

In the remainder of this chapter we consider the case where $\alpha = \beta = 1$, which has the advantage of providing us with simplified formulas and, as suggested in [49], to be appropriate to estimate the forward error obtained when the LLS problem is solved via with normal equations. We also make the assumption that the LLS has already been solved with either the normal equations method or a QR factorization. Then the solution x and the residual r have been already computed and the R-factor of the QR factorization of A is available (we recall that the Cholesky factor of the normal equations is, in exact arithmetic, the R-factor of the QR factorization up to some signs).

We can obtain from [12] the following closed formula for the condition number of the LLS solution:

$$\kappa_{LS} = \|R^{-1}\|_2 \left(\|R^{-1}\|_2^2 \|r\|_2^2 + \|x\|_2^2 + 1\right)^{\frac{1}{2}}. \tag{3.5}$$

This equation requires the computation of the smallest singular value of the matrix A (or R). This involves $\mathcal{O}(n^3)$ flops if we compute the full SVD of A (or R). But $\|R^{-T}\|_2$ can also be approximated by other matrix norms $\|R^{-1}\|_1$ or $\|R^{-1}\|_\infty$ (using *e.g.* [60, p. 293]).

Using again [12], we can express in Equation (3.6) the condition number of the component $x_i = e_i^T x$ and then calculate a vector $\kappa_{CW} \in \mathbb{R}^n$ with components κ_i being the exact condition number for the i th component expressed by

$$\kappa_i = \left(\|R^{-1}R^{-T}e_i\|_2^2 \|r\|_2^2 + \|R^{-T}e_i\|_2^2 (\|x\|_2^2 + 1)\right)^{\frac{1}{2}} \tag{3.6}$$

The computation of each κ_i requires two triangular solves ($R^T y = e_i$ and $Rz = y$) corresponding to $2n^2$ flops. If we want to compute all κ_i, it is more efficient to solve $RY = I$ and compute YY^T, which requires about $2n^3/3$ flops.

3.2.2 Statistical condition estimation

It is possible to reduce the computational cost for estimating the LLS conditioning by using statistical techniques. This approach was initially proposed in [52, 65, 66] and reduces the cost to $\mathcal{O}(n^2)$ flops. The theoretical quality of such statistical estimates can be formally measured by the probability to give an estimate in a certain range around the exact value. After the randomization method described in Chapter 2, these statistical techniques give another illustration on how statistics can be applied to enhance numerical linear algebra calculations. These techniques are very promising and are currently under development for integration into public domain libraries LAPACK and MAGMA.

We propose an algorithm that, similarly to what was proposed in [28] for linear systems, enables us to estimate the condition number of the LLS solution using the method called *small-sample theory* [65] that provides statistical condition estimates for matrix functions. Using this method, we can derive Algorithm 3.2.1 that computes a statistical estimate $\bar{\kappa}_{LS}$ for κ_{LS} given in Equation (3.5). The accuracy of this estimate can be tweaked by modifying the number q of considered random samples. The computation of $\bar{\kappa}_{LS}$ involves the computation of the QR factorization of an $n \times q$ matrix for $\mathcal{O}(nq^2)$ flops. It also involves q times two $n \times n$ triangular solves, each triangular system being solved in $\mathcal{O}(n^2)$ flops. Then the total computational cost is $\mathcal{O}(qn^2)$ flops (if $n \gg q$).

It is also possible to estimate the conditioning of the components of x by using an approach based on [66]. Algorithm 3.2.2 corresponds to the algorithm that uses unstructured perturbations and it can be compared with the exact value given in Equation (3.6). Algorithm 3.2.2 computes a vector $\bar{\kappa}_{CW} = (\bar{\kappa}_1, \cdots, \bar{\kappa}_n)$ containing the statistical estimate for the κ_i's. Depending on the needed accuracy for the statistical estimation, the number of random perturbations $q \geq 1$ applied to the input data in Algorithm 3.2.2 can be adjusted. This algorithm involves two $n \times n$ triangular solves with q right-hand sides, which requires about qn^2 flops.

We can compare the statistical estimates with their corresponding exact values on random LLS problem with known solution using of variant of the method given in [81].

Algorithm 3.2.1 Statistical condition estimation for linear least squares solution (SCE_LLS)

Require: $q \geq 1$, the number of samples
  Generate q vectors $z_1, z_2, ..., z_q \in \mathbb{R}^n$ with entries in $\mathcal{U}(0,1)$
  Orthonormalize the vectors z_i using a QR factorization
  for $j = 1$ to q **do**
    Compute $\kappa_j = \left(\|R^{-1}R^{-T}z_j\|_2^2\|r\|_2^2 + \|R^{-T}z_i\|_2^2(\|x\|_2^2 + 1)\right)^{\frac{1}{2}}$
  end for
  Compute $\bar{\kappa}_{LS} = \frac{\omega_q}{\omega_n}\sqrt{\sum_{j=1}^{q}\kappa_j^2}$ with $\omega_q = \sqrt{\frac{2}{\pi(q-\frac{1}{2})}}$

Algorithm 3.2.2 Componentwise statistical condition estimate for linear least squares (SCE_LLS_CW)

Require: $q \geq 1$, the number of perturbations of input data
  for $j = 1$ to q **do**
    Generate $S_j \in \mathbb{R}^{n\times n}$, $h_j \in \mathbb{R}^n$ and $g_j \in \mathbb{R}^n$ with entries in $\mathcal{N}(0,1)$
    Compute $u_j = R^{-1}(g_j - S_j x + \|Ax - b\|_2 R^{-T}h_j)$
  end for
  Let $p = m(n+1)$, compute vector $\bar{\kappa}_{CW} = \frac{\sum_{i=1}^{q}|u_j|}{q\omega_p\sqrt{p}}$ with $\omega_q = \sqrt{\frac{2}{\pi(q-\frac{1}{2})}}$

The matrix A is generated using

$$A = Y\begin{pmatrix} D \\ 0 \end{pmatrix}Z^T,\ Y = I - 2yy^T,\ Z = I - 2zz^T$$

where $y \in \mathbb{R}^m$ and $z \in \mathbb{R}^n$ are random unit vectors and $D = n^{-l}diag(n^l, (n-1)^l, (n-2)^l, \cdots, 1)$. We have $x = (1, 2^2, ..., n^2)^T$ and $\text{cond}_2(A) = n^l$. The residual vector is given by $r = Y\begin{pmatrix} 0 \\ v \end{pmatrix}$ where $v \in \mathbb{R}^{m-n}$ is a random vector of norm n_r and the right-hand side is given by $b = Y\begin{pmatrix} DZx \\ v \end{pmatrix}$. We generate such random LLS problems of size $m \times n$ with $m = 9984$ and $n = 2496$.

In Figure 3.1 we compare the estimate $\bar{\kappa}_{LS}$ obtained via Algorithm 3.2.1 with the exact condition number κ_{LS} computed using Equation (3.5) and another estimate $\hat{\kappa}_{LS}$ given in [4] using one sample ($q = 1$) and for several values of $\text{cond}_2(A)$. We observe that the two statistical estimates lie within factor 1.7 of the exact value.

The accuracy of componentwise statistical estimates can be shown in Figure 3.2. We set $\text{cond}_2(A) = 2.5 \cdot 10^3$ and $n_r = 1$. Figure 3.2 (a) depicts the conditioning for all LLS solution components, computed as $\kappa_i/|x_i|$ where each κ_i is computed using Equation (3.6). The interest of the componentwise approach is visible here since the sensitivity to perturbations of each solution component varies significantly (from 10^2 to 10^8) while the normalized condition number of the solution computed using Equation 3.5 is $\kappa_{LS}/\|x\|_2 = 2.5 \cdot 10^3$. Then we represent in Figure 3.2 (b) the ratio between the statistical condition estimate computed via Algorithm 3.2.2, considering two samples ($q = 2$), and the exact value computed using Equation (3.6). The ratio is computed as an average on

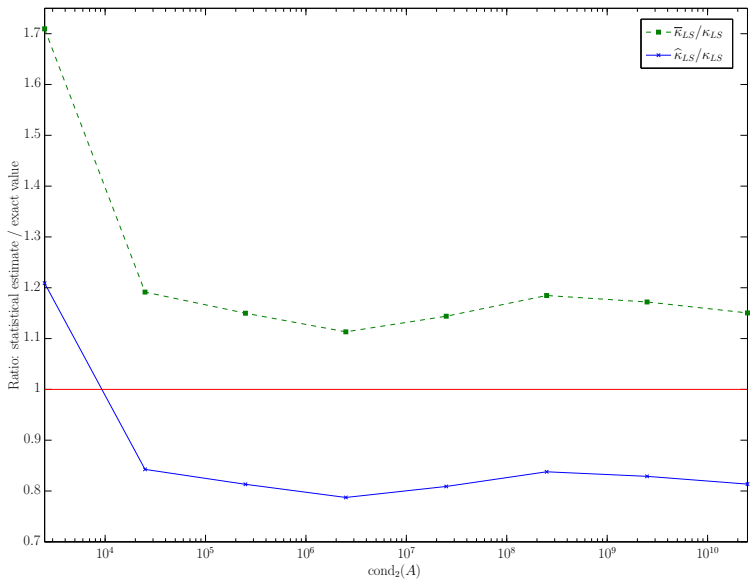


Figure 3.1: Ratio between statistical and exact condition numbers

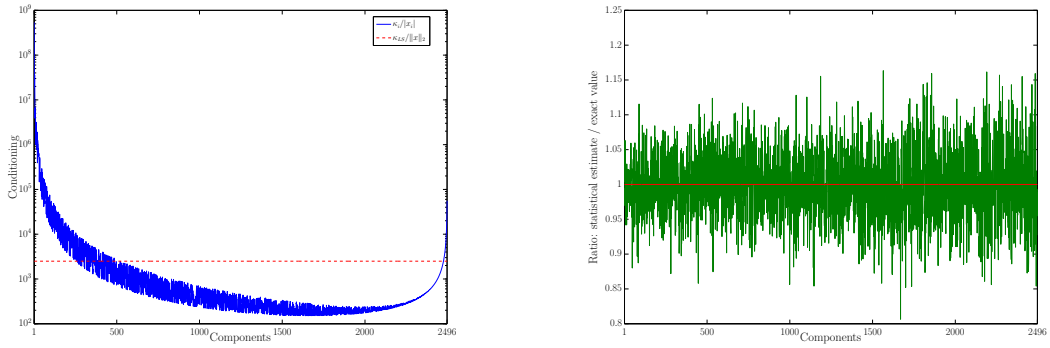
100 random problems. We observe that this ratio is lower than 1.2 and that the quality of the estimate is similar for all solution components. Other tests on accuracy of statistical condition estimates can be found in [17].

3.2.3 Using HPC libraries to evaluate least squares conditioning

In this section we describe how to compute the quantities defined in Sections 3.2.1 and 3.2.2 using the high-performance linear algebra libraries (Sca)LAPACK and MAGMA.

3.2.3.1 Computation with (Sca)LAPACK

ScaLAPACK being designed for parallel distributed memory machines, it enables us to address large size LLS problems. Note that the routines that we use to compute condition numbers have similar nomenclature in both libraries LAPACK and ScaLAPACK. We suppose that we have used the (Sca)LAPACK routine (P)DGELS that solves the LLSP using a QR factorization of A. Note that, as already mentionned in Section 1.3, it is possible to have a more accurate solution using extra-precise iterative refinement [33]. We give in Table 3.1 the (Sca)LAPACK routines used for computing the condition numbers of an LLS solution or its components and the corresponding number of flops. We observe that the cost for computing all the κ_i's or estimating κ_{LS} is always $\mathcal{O}(n^3)$. When $m \gg n$, this cost is affordable if we compare it to the cost of the least squares solution using Householder QR factorization ($2mn^2 - 2n^3/3$) or the normal equations ($mn^2 + n^3/3$). However, on some new architectures (*e.g.* GPU), the implementation of SVD or eigenvalue algorithms might not be efficient compared to a QR factorization and the cost for computing κ_{LS} can be expensive in practice.



(a) $\kappa_i/|x_i|$ (b) Ratio $\bar{\kappa}_i/\kappa_i$

Figure 3.2: Componentwise conditioning (left) and comparison with statistical estimate (right).

Table 3.1: Computation of least squares conditioning with (Sca)LAPACK

condition number	linear algebra operation	(Sca)LAPACK routines	flops count
κ_{LS}	estimate $\|R^{-1}\|_{1 \text{ or } \infty}$ compute $\left\|R^{-1}\right\|_F$	(P)DTRCON (P)DTRTRI	n^2 $n^3/3$
$\bar{\kappa}_{LS}$	generate random orthogonal vectors 2 triangular solves	(P)DTRSV	n^2
κ_i	$R^T y = e_i$ and $Rz = y$	(P)DTRSV	$2n^2$
all $\kappa_i,\ i = 1, n$	$RY = I$ and compute YY^T	(P)DPOTRI	$2n^3/3$
$\bar{\kappa}_{CW}$	generate random vectors 2 triangular solves	(P)DTRSV	n^2

For estimating κ_{LS}, we need to have an estimate of $\left\|A^{\dagger}\right\|_2$ *i.e.* $\left\|R^{-1}\right\|_2$. To avoid the computation of an SVD of R^{-1}, a possibility is to approximate $\left\|R^{-1}\right\|_2$ using other matrix norms. For instance, $\|R^{-1}\|_1$ or $\|R^{-1}\|_\infty$ can be estimated using Higham modification [60, p. 293] of Hager's [55] method as it is implemented in the (Sca)LAPACK routine (P)DTRCON. The cost is $\mathcal{O}(n^2)$.

3.2.3.2 Computation with MAGMA

In many physical applications, LLS problems are expressed using a statistical model often referred to as *linear statistical model* where we have to solve

$$b = Ax + \epsilon,\ A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m, \text{rank}(A) = n,$$

where ϵ is a vector of random errors having expected value $E(\epsilon) = 0$ and variance-covariance $V(\epsilon) = \sigma_b^2 I$. In statistical language, the matrix A is called the regression matrix and the unknown vector x is called the vector of regression coefficients.
Following the Gauss-Markov theorem [105], the least squares estimate $\hat{x}$ is the linear unbiased estimator of x satisfying

$$\|A\hat{x} - b\|_2 = \min_{x\in\mathbb{R}^n} \|Ax - b\|_2,$$

with minimum variance-covariance equal to

$$C = \sigma_b^2 (A^T A)^{-1}. \tag{3.7}$$

The diagonal elements c_{ii} of C give the variance of each component $\hat{x}_i$ of the solution. The off-diagonal elements c_{ij}, $i \neq j$ give the covariance between $\hat{x}_i$ and $\hat{x}_j$. Then instead of computing condition numbers (which are notions more commonly handled by numerical linera algebra practitioners) physicists often compute the variance-covariance matrix whose entries are intimately correlated with condition numbers κ_i and κ_{LS} mentioned previously. In the following we describe how we can compute totally or partially the variance-covariance matrix. To our knowledge, there is no existing routine in public domain libraries LAPACK or ScaLAPACK to compute the variance-covariance contrary to the NAG library [93]. This is why we also propose an implementation based on the MAGMA library in order to take advantage of GPU computing. More details about how to compute the variance-covariance matrix using (Sca)LAPACK can be found in [12].

When the variance-covariance matrix has been computed, the condition numbers described in Section 3.2 can be easily obtained. Indeed, we can use the fact that $\left\|R^{-1}\right\|_2^2 = \frac{\|C\|_2}{\sigma_b^2}$, $\|R^{-T}e_i\|_2^2 = \frac{c_{ii}}{\sigma_b^2}$, and $\|R^{-1}R^{-T}e_i\|_2 = \frac{\|C_i\|_2}{\sigma_b^2}$ where C_i and c_{ii} are respectively the i th column and the i th diagonal element of the matrix C. Then by replacing respectively in Equations (3.5) and (3.6) we get the formulas

$$\kappa_{LS} = \frac{\|C\|_2^{1/2}}{\sigma_b}((m-n)\|C\|_2 + \|x\|_2^2 + 1)^{1/2}, \tag{3.8}$$

and

$$\kappa_i = \frac{1}{\sigma_b}((m-n)\|C_i\|_2^2 + c_{ii}(\|x\|_2^2 + 1))^{1/2}. \tag{3.9}$$

Note that, when $m > n$, $\frac{1}{m-n}\|r\|_2^2$ is an unbiased estimate of σ_b^2 [20, p. 4].

Let us now evaluate the performance for computing LLS condition numbers from the variance-covariance matrix using the MAGMA library for heterogeneous and hybrid architectures (release 1.2.1). In our implementation, MAGMA is linked with the libraries MKL 10.3.8 and CUDA 4.1 respectively for multicore and GPU. The tests have been achieved on a multicore processor Intel Xeon E5645 (2 sockets $\times$ 6 cores) running at 2.4 GHz (the cache size per core is 12 MB and the size of the main memory is 48 GB). This system hosts two GPU NVIDIA Tesla C2075 running at 1.15 GHz with 6 GB memory each.

We observe in Figure 3.3 that the computation of the variance-covariance matrix and of the conditioning of the components are significantly faster than the problem solution with respectively a time factor of 3 and 2, this factor increasing with the problem size. The κ_i's are computed with the variance-covariance matrix (using Equation (3.9)). The time overhead between the computation of the κ_i's and the variance-covariance computation comes from the computation of the norms of the columns (routine cublasDnrm2) which is non optimal due to communication cost. As expected, the routines that compute the statistical condition estimates for the solution and for all solution components outperform the other routines. Note that we did not mention on this graph the performance for computing κ_{LS} (using Equation (3.8)). Indeed this involves an eigenvalue decomposition of the variance-covariance matrix (MAGMA routine magma_dsyevd_gpu), which turns out to be much slower than the LLS solution (MAGMA routine magma_dgels3_gpu) in spite of a smaller number of arithmetic operations. This illustrates the fact, even though the theoretical number of flops for computing κ_{LS} is much smaller that for computing x ($\mathcal{O}(n^3)$ vs $\mathcal{O}(mn^2)$), having an efficient implementation on the targetted architecture is essential to take advantage of the gain in flops. In the particular case of condition estimation, this confirms the interest of considering statistical estimates on these architectures. More generally, this application gives another illustration of how statistical techniques accelerate linear algebra calculations on current parallel architectures, similarly to the randomization approach described in Chapter 2.

3.3 The total least squares approach

3.3.1 Conditioning of the total least squares

We study in this section the conditioning of the total least squares (TLS) problem $Ax \simeq b$, where a certain type of measurement errors affecting A is considered. This case is treated by the statistical model referred to as Errors-In-Variables model (see *e.g.* [61, p. 230] and [20, p. 176]), where we have the relation

$$(A + E)x = b + \epsilon.$$

In general it is assumed in this model that the rows of $[E, \epsilon]$ are independently and identically distributed with common zero mean vector and common covariance matrix. The corresponding linear algebra problem, discussed originally in [45], can be expressed as:

$$\min_{E,\epsilon} \|(E, \epsilon)\|_F, \quad (A + E)x = b + \epsilon. \tag{3.10}$$

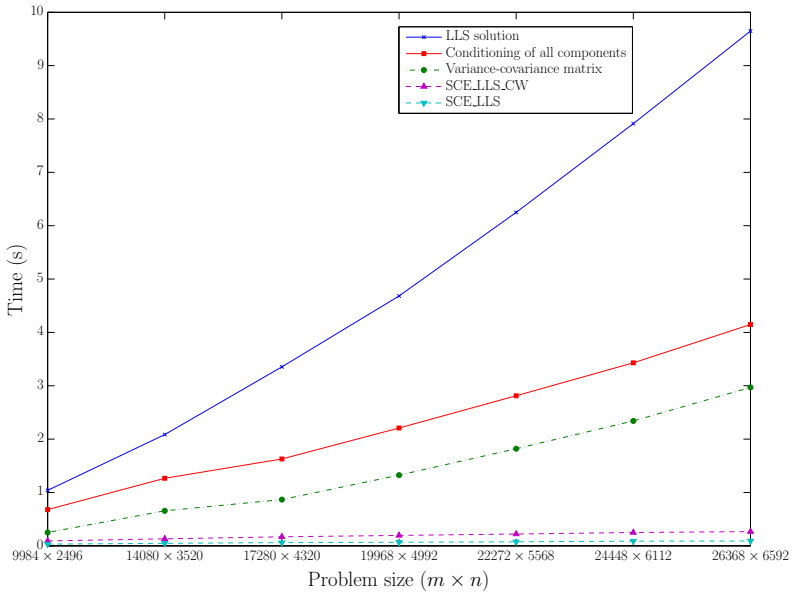


Figure 3.3: Performance for computing LLS condition numbers with MAGMA

As mentioned in [61, p. 238], the TLS method enables us to obtain a more accurate solution when entries of A are perturbed under certain conditions.

Let $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$, with $m > n$. Following [61], we consider the two singular value decompositions of A, and $[A,b]$: $A = U'\Sigma' V'^T$ and $[A,b] = U\Sigma V^T$. We also set $\Sigma = \mathrm{diag}(\sigma_1,\ldots,\sigma_{n+1})$, $\Sigma' = \mathrm{diag}(\sigma'_1,\ldots,\sigma'_n)$, where the singular values are in nonincreasing order, and define $\lambda_i = \sigma_i^2$, and $\lambda'_i = \sigma_i'^2$. From [20, p. 178], we have the interlacing property

$$\sigma_1 \geq \sigma'_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq \sigma'_n \geq \sigma_{n+1}. \tag{3.11}$$

We consider the total least squares problem expressed in Equation (3.10) and we assume in this text that the *genericity* condition $\sigma'_n > \sigma_{n+1}$ holds (for more information about the "nongeneric" problem see *e.g.* [61, 80]). From [61, Theorems 2.6 and 2.7], it follows that the TLS solution x exists, is unique, and satisfies

$$x = \left(A^T A - \lambda_{n+1} I_n\right)^{-1} A^T b, \tag{3.12}$$

where I_n denotes here the $n \times n$ identity matrix.

In addition, $\begin{bmatrix} x \\ -1 \end{bmatrix}$ is an eigenvector of $[A,b]^T[A,b]$ associated with the simple eigenvalue λ_{n+1}, *i.e.* $\sigma'_n > \sigma_{n+1}$ guarantees that λ_{n+1} is not a semi-simple eigenvalue of $[A,b]^T[A,b]$. As for linear least squares problems, we define the total least squares residual $r = b - Ax$, which enables us to write

$$\lambda_{n+1} = \frac{1}{1+x^T x}\begin{bmatrix} x^T, & -1 \end{bmatrix}\begin{bmatrix} A^T A & A^T b \\ b^T A & b^T b \end{bmatrix}\begin{bmatrix} x \\ -1 \end{bmatrix} = \frac{r^T r}{1+x^T x}. \tag{3.13}$$

As mentioned in [61, p. 35], the TLS solution is obtained by scaling the last right singular vector v_{n+1} of $[A, b]$ until its last component is -1 and, if $v_{i,n+1}$ denotes the ith component of v_{n+1}, we have

$$x = -\frac{1}{v_{n+1,n+1}}[v_{1,n+1}, \ldots, v_{n,n+1}]^T. \tag{3.14}$$

The TLS method involves an SVD computation and the computational cost is higher than that of a classical LLS problem (about $2mn^2 + 12n^3$ as mentioned in [46, p. 598], to be compared with the approximately $2mn^2$ flops required for LLS solved via Householder QR factorization). However, there exist faster methods referred to as "partial SVD" (PSVD) that calculate only the last right singular vector or a basis of the right singular subspace associated with the smallest singular values of $[A, b]$ (see [61, p. 97]).

Similarly to Section 3.2.1, we measure the perturbations of the data A and b using the product norm defined on $\mathbb{R}^{m\times n} \times \mathbb{R}^m$ by $\|(A, b)\|_F = \sqrt{\|A\|_{\mathrm{F}}^2 + \|b\|_2^2}$ and we take the Euclidean norm $\|x\|_2$ for the solution space $\mathbb{R}^n$. The TLS solution x can be expressed as a function of the data A and b using the mapping g defined by

$$\begin{array}{rccc} g & : & \mathbb{R}^{m\times n} \times \mathbb{R}^m & \longrightarrow & \mathbb{R}^n \\ & & (A, b) & \longmapsto & g(A, b) = x = (A^T A - \lambda_{n+1} I_n)^{-1} A^T b. \end{array}$$

Since λ_{n+1} is simple, g is a Fréchet-differentiable function of A and b, and the genericity assumption ensures that the matrix $(A^T A - \lambda_{n+1} I_n)^{-1}$ is also Fréchet-differentiable in a neighborhood of (A, b). As a result, g is Fréchet-differentiable in a neighborhood of (A, b). Note that we studied a more general case in [16] by considering the conditioning of $L^T x$, where L is an $n \times k$ matrix, with $k \leq n$. Using an approach already applied for LLS problem in Section 3.2.1, the computation of the condition number of the TLS problem is based on the evaluation of the Fréchet dérivative of g. Under the genericity assumption, we showed in [16] that g is Fréchet differentiable in a neighborhood of (A, b) and that its derivative g' is expressed by

$$g'(A, b).(\Delta A, \Delta b) = B_\lambda^{-1} \left(A^T + \frac{2xr^T}{1 + x^T x} \right) (\Delta b - \Delta A x) + B_\lambda^{-1} \Delta A^T r, \tag{3.15}$$

with $B_\lambda = A^T A - \lambda_{n+1} I_n$. Then, setting $D_\lambda = B_\lambda^{-1} \left(A^T + \frac{2xr^T}{1+x^T x} \right) \in \mathbb{R}^{n\times m}$, the TLS condition number κ_{TLS} can be expressed as

$$\kappa_{TLS} = \left\| \mathcal{M}_{g'} \right\|_2, \tag{3.16}$$

where

$$\mathcal{M}_{g'} \;=\; \left[-x^T \otimes D_\lambda + \left(r^T \otimes (B_\lambda^{-1}) \right) P, \quad D_\lambda \right] \in \mathbb{R}^{n\times(nm+m)}$$

and $\otimes$ denotes the Kronecker product of two matrices [48, p. 21].

Then computing κ_{TLS} reduces to computing the 2-norm of the $n \times (nm + m)$ matrix $\mathcal{M}_{g'}$. For large values of n or m, it is not possible to build explicitly the generally dense matrix $\mathcal{M}_{g'}$. Iterative techniques based on the power method [60, p. 289] or on the Lanczos method [46] are better suited. These algorithms involve however the computation

of the product of $\mathcal{M}_{g'}^T$ by a vector $y \in \mathbb{R}^n$. This operation can be performed using the adjoint operator of g'. We also used in [15] the technique consisting of computing condition numbers by working on the dual spaces.

Using the scalar products $\text{trace}(A_1^T A_2) + b_1^T b_2$ and $y_1^T y_2$ respectively on $\mathbb{R}^{m\times n} \times \mathbb{R}^m$ and $\mathbb{R}^n$, the adjoint of g' can be expressed as

$$\begin{array}{rcl} g'^*(A,b) & : \mathbb{R}^n \longrightarrow & \mathbb{R}^{m\times n} \times \mathbb{R}^m \\ & y \longmapsto & \left(-D_\lambda^T y x^T + r y^T B_\lambda^{-1}, D_\lambda^T y\right). \end{array} \tag{3.17}$$

Using (3.15) and (3.17), we can now write in Algorithm 3.3.1 the iteration of the power method ([60, p. 289]) to compute the TLS condition number κ_{TLS}. In this algorithm we assume x and λ_{n+1} are available, and we iterate (A_p, b_p) to approach the optimal $(\Delta A, \Delta b)$ that realizes (3.1).

Algorithm 3.3.1 Condition number of TLS problem

Require: Select initial vector $y \in \mathbb{R}^n$
  for $p = 1, 2, \ldots$ **do**
    $(A_p, b_p) = \left(-D_\lambda^T y x^T + r y^T B_\lambda^{-1}, D_\lambda^T y\right)$
    $\nu = \|(A_p, b_p)\|_F$
    $(A_p, b_p) \leftarrow (\frac{1}{\nu} \cdot A_p, \frac{1}{\nu} \cdot b_p)$
    $y = B_\lambda^{-1}\left(A^T + \frac{2xr^T}{1+x^T x}\right)(b_p - A_p x) + B_\lambda^{-1} A_p^T r$
  end for
  $\kappa_{TLS} = \sqrt{\nu}$

The quantity ν computed by Algorithm 1 is the largest eigenvalue of $\mathcal{M}_{g'}\mathcal{M}_{g'}^T$. Since $\kappa_{TLS} = \left\|\mathcal{M}_{g'}\right\|_2$ then the condition number κ_{TLS} is also the largest singular value of $\mathcal{M}_{g'}$ *i.e.* $\sqrt{\nu}$. As mentioned in [46, p. 331], the algorithm will converge if the initial y has a component in the direction of the corresponding dominant eigenvector of $\mathcal{M}_{g'}\mathcal{M}_{g'}^T$. When there is an estimate of this dominant eigenvector, the initial y can be set to this estimate but in many implementations, y is initialized as a random vector. The algorithm is terminated by a "sufficiently" large number of iterations or by evaluating the difference between two successive values of ν and comparing it to a tolerance given by the user.

In practice the TLS solution is obtained by Equation (3.14) and involves an SVD computation. This is why we proposed in [16] a formula for κ_{TLS} that can be computed with quantities that may be already available from the solution process. In particular it is shown that

$$\kappa_{TLS} = (1 + \|x\|_2^2)^{\frac{1}{2}} \left\| D' \left[V'^T, \quad 0_{n,1}\right] V \left[D, \quad 0_{n,1}\right]^T \right\|_2, \tag{3.18}$$

where

$$D' = \text{diag}\left((\sigma_1'^2 - \sigma_{n+1}^2)^{-1}, \ldots, (\sigma_n'^2 - \sigma_{n+1}^2)^{-1}\right) \text{ and}$$

$$D = \text{diag}\left((\sigma_1^2 + \sigma_{n+1}^2)^{\frac{1}{2}}, \ldots, (\sigma_n^2 + \sigma_{n+1}^2)^{\frac{1}{2}}\right),$$

($0_{n,1}$ denotes the zero column vector of length n).

In many applications, an upper bound would be sufficient to give an estimate of the conditioning of the TLS solution. This upper bound, also given in [16], is

$$\bar{\kappa}_{TLS} = (1 + \|x\|_2^2)^{\frac{1}{2}} \frac{(\sigma_1^2 + \sigma_{n+1}^2)^{\frac{1}{2}}}{(\sigma_n'^2 - \sigma_{n+1}^2)}.$$

3.3.2 Limitation of the first-order approach

As explained in Section 3.1, the condition number, defined as the norm of the Fréchet derivative of the solution, is a first order term. In the following example, we show the limitation of this approach in providing good error bounds, depending on the conditioning of the problem and on the size of the perturbations.

We consider the TLS problem $Ax \approx b$ where $[A, b]$ is defined by

$$[A, b] = Y \begin{pmatrix} D \\ 0 \end{pmatrix} Z^T \in \mathbb{R}^{m \times (n+1)}, Y = I_m - 2yy^T, Z = I_{n+1} - 2zz^T,$$

where $y \in \mathbb{R}^m$ and $z \in \mathbb{R}^{n+1}$ are random unit vectors, $D = diag(n, n-1, \ldots, 1, 1 - e_p)$ for a given parameter e_p. The quantity $\sigma_n' - \sigma_{n+1}$ measures the distance of our problem to nongenericity and, due to Equation (3.11), we have in exact arithmetic

$$\sigma_n' - \sigma_{n+1} \leq \sigma_n - \sigma_{n+1} = e_p.$$

Then by varying e_p, we can generate different TLS problems and by considering small values of e_p, it is possible to study the behavior of the TLS condition number in the context of close-to-nongeneric problems. The parameter e_p that enables us to vary the conditioning of the problem. The TLS solution x is computed using an SVD of $[A, b]$ and Equation (3.14).

For each value of e_p, we consider random relative perturbations $(\Delta A, \Delta b)$ such that $\frac{\|(\Delta A, \Delta b)\|_F}{\|(A,b)\|_F} = 10^{-q}$. Note that, for this problem, the exact solution $x = g(A, b)$ is known by construction and, with the notation of Example 1, is equal to $Z(1 : n, n + 1)/Z(n + 1, n + 1)$. Let $\tilde{x}$ be the computed solution. For several values of $(e_p, 10^{-q})$, we report in Table 3.2 the condition number κ_{TLS}, the relative forward error, the relative error at first order, and the "worst case" relative error estimate (that corresponds to the product of the relative condition number by the relative perturbation of data).

The rows of Table 3.2 are sorted by increasing condition numbers and increasing relative perturbations. We observe that, when the problem is well-conditioned, the first order enables us to predict the forward error for all sizes of perturbation considered here. When the condition number increases, only small perturbations provide consistency between the forward error and the first order error. This indicates that, the larger the ill-condition of the problem, the less reliable the first order approach is when large perturbations are considered. We also notice that, for all experiments, the error estimate based on the condition number overestimates the first order error with an order of magnitude $\mathcal{O}(10)$, which corresponds to the ratio between $\|g'(A, b)\| \times \|(\Delta A, \Delta b)\|_2$ and $\|g'(A, b).(\Delta A, \Delta b)\|_2$ where $\|g'(A, b)\|$ denotes the operator norm of the linear function $g'(A, b)$.

Table 3.2: First order estimation for TLS forward error for various conditioning and perturbation size.

e_p	q	κ_{TLS}	$\frac{\|\tilde{x}-x\|_2}{\|x\|_2}$	$\frac{\|g'(A,b).(\Delta A,\Delta b)\|_2}{\|x\|_2}$	$\frac{\kappa_{TLS}\times 10^{-q}}{\|x\|_2}$
$1 \cdot 10^{0}$	15	$1.18 \cdot 10^{0}$	$5.61 \cdot 10^{-15}$	$5.61 \cdot 10^{-15}$	$1.00 \cdot 10^{-13}$
	12	”	$4.82 \cdot 10^{-12}$	$4.82 \cdot 10^{-12}$	$1.00 \cdot 10^{-10}$
	9	”	$5.67 \cdot 10^{-9}$	$5.67 \cdot 10^{-9}$	$1.00 \cdot 10^{-7}$
	6	”	$4.50 \cdot 10^{-6}$	$4.50 \cdot 10^{-6}$	$1.00 \cdot 10^{-4}$
	3	”	$3.54 \cdot 10^{-3}$	$3.71 \cdot 10^{-3}$	$1.00 \cdot 10^{-1}$
$1 \cdot 10^{-4}$	15	$8.36 \cdot 10^{3}$	$2.29 \cdot 10^{-11}$	$2.22 \cdot 10^{-11}$	$7.10 \cdot 10^{-10}$
	12	”	$8.46 \cdot 10^{-9}$	$8.46 \cdot 10^{-9}$	$7.10 \cdot 10^{-7}$
	9	”	$1.99 \cdot 10^{-5}$	$1.99 \cdot 10^{-5}$	$7.10 \cdot 10^{-4}$
	6	”	$2.10 \cdot 10^{-2}$	$2.06 \cdot 10^{-2}$	$7.10 \cdot 10^{-1}$
	3	”	$8.95 \cdot 10^{0}$	$1.09 \cdot 10^{1}$	$7.10 \cdot 10^{2}$
$1 \cdot 10^{-8}$	15	$8.36 \cdot 10^{7}$	$2.11 \cdot 10^{-7}$	$2.22 \cdot 10^{-7}$	$7.10 \cdot 10^{-6}$
	12	”	$8.47 \cdot 10^{-5}$	$8.46 \cdot 10^{-5}$	$7.10 \cdot 10^{-3}$
	9	”	$2.31 \cdot 10^{-1}$	$1.99 \cdot 10^{-1}$	$7.10 \cdot 10^{0}$
	6	”	$4.39 \cdot 10^{0}$	$2.06 \cdot 10^{2}$	$7.10 \cdot 10^{3}$
	3	”	$9.27 \cdot 10^{0}$	$1.09 \cdot 10^{5}$	$7.10 \cdot 10^{6}$
$1 \cdot 10^{-12}$	15	$8.36 \cdot 10^{11}$	$3.61 \cdot 10^{-3}$	$2.21 \cdot 10^{-3}$	$7.11 \cdot 10^{-2}$
	12	”	$7.39 \cdot 10^{0}$	$8.42 \cdot 10^{-1}$	$7.11 \cdot 10^{1}$
	9	”	$3.62 \cdot 10^{0}$	$1.98 \cdot 10^{3}$	$7.11 \cdot 10^{4}$
	6	”	$4.40 \cdot 10^{0}$	$2.05 \cdot 10^{6}$	$7.11 \cdot 10^{7}$
	3	”	$9.27 \cdot 10^{0}$	$1.08 \cdot 10^{9}$	$7.11 \cdot 10^{10}$

Conclusion and perspectives

In this HDR thesis we presented several research axes to improve speed and reliability of numerical linear algebra solvers on modern architectures.

We showed that taking advantage of current multicore+GPU systems by designing "heterogenous-aware" algorithms can be a very efficient way to enhance dense linear algebra computations. In particular, splitting properly the workload between the components and minimizing the cost of pivoting can lead to fast linear system solvers. Using mixed precision algorithms is also an appropriate method to exploit fast single precision arithmetic on current processing units. Working on hybrid solvers is a continuous effort in our research which is currently performed in the framework of an ongoing PhD thesis on multiGPU algorithms (Adrien Rémy, started end of 2011 and funded by the French Ministry of Research).

We also illustrated how randomization techniques can accelerate the solution of dense linear systems on multicore architectures possibly accelerated by GPUs. We considered the example of dense symmetric indefinite systems but we also applied this method to general dense systems in [13]. An extension to sparse systems is also under investigation (collaboration with Dr Xiaoye Sherry Li - Lawrence Berkeley National Laboratory - USA) but we must be careful about the possible fill-in introduced by butterfly transformations in sparse matrices. The resulting solvers are based on a computationally cheap randomization technique followed by an efficient factorization without pivoting. In addition to providing us with satisfying performance results, this method gives accurate results. Our current task regarding randomization kernels concerns the progressive integration into public domain libraries PLASMA and MAGMA. We point out that randomized algorithms represent a hot topic in scientific computing in general and in numerical linear algebra in particular. In addition to recent publications in ACM TOMS and IPDPS'12, we organized in June 2012 a minisymposium on this subject at the SIAM Conference on Applied Linear Algebra (Valencia) and also we gave two invited plenary talks in recent conferences (*ACM HPC'12 - March 2012* in Orlando and *Workshop on solutions of indefinite systems - April 2012* in Eindhoven).

Providing error analysis tools is crucial for HPC applications and this corresponds to a strong demand from physicists. With the rewritting of the old generation libraries, the routines based on backward error analysis that were developed in (Sca)LAPACK must be adapted. Moreover, some approaches as the computation of problem conditioning were not considered in these libraries. Also assessing numerical quality is essential when combined with new algorithms (*e.g.* randomized) for which there are few existing results on stability in the literature. The computable expressions and routines for the conditioning of least squares problems that we described in this manuscript fill this gap. These formulas can

also be applied to linear systems, which are special cases of overdetermined least squares.

Subsequent to the work described in this HDR thesis, we give below four examples of research directions.

Getting closer to physical applications

One of the main requirement for our research to be useful and visible is to be applied to physical applications. In particular we plan to apply some of the concepts detailed in this manuscript to computational fluid dynamics. Our motivation is to develop an efficient solver for the numerical simulations of incompressible fluid flows on heterogeneous parallel architectures. We focus here on the solution of large sparse linear systems coming from the discretization of Helmholtz and Poisson equations that represent the major part of the computational time for solving the Navier-Stokes equations. The discretization scheme that we use for solving these equations results in block-tridiagonal systems that are difficult to solve efficiently on heterogenous platforms. We are currently investigating techniques to accelerate variants of Thomas algorithms using vectorization techniques [42] or cyclic reduction [67]. First results using this approach are very promising (see Figure 3.4 that shows an acceleration of more than 50%).

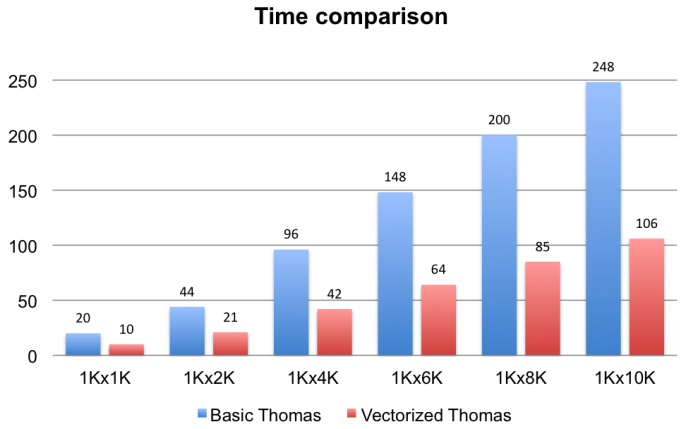


Figure 3.4: Performance of vectorized Thomas algorithm.

The algorithms developed for this research will be validated on physical applications studied at LIMSI[1] in the area of numerical simulations of incompressible flows [83], the criteria for this evaluation being based on accuracy and performance. This research is the purpose of a PhD thesis (Yushan Wang, started end of 2011, and funded by Digiteo[2]).

Still more effort on numerical libraries

The method of error analysis based on evaluating condition numbers that we presented in Section 3 enables us to evaluate the numerical quality of least-squares/linear system

[1] Laboratoire d'Informatique pour la Mécanique et les Sciences de l'Ingénieur, `http://www.limsi.fr/`

[2] `http://www.digiteo.fr`

solutions at a very affordable computational cost. However, as mentioned at the end of Section 3.2.3.2, there remains important issues for the performance of some calculations on current architectures like GPUs. In particular having efficient kernels for SVD or eigenvalue solvers on GPUs remains crucial for some computations (note that for distributed systems using MPI, efficient eigenvalue or SVD solvers can be found in [58]). We can illustrate this by comparing in Figure 3.5 the time for computing an LLS solution and its conditioning using LAPACK and MAGMA. We observe that MAGMA provides faster solution and condition number but, contrary to LAPACK, the computation of the condition number is slower than the time for the solution, in spite of a smaller flops count.

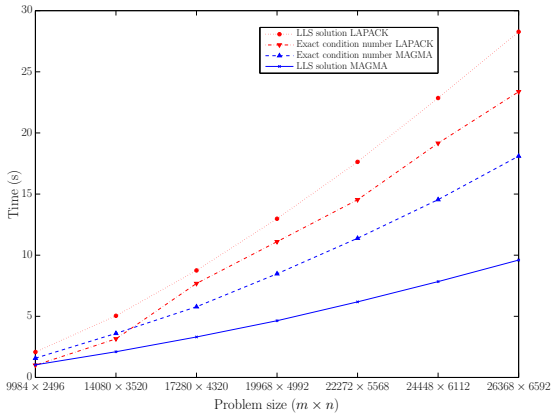


Figure 3.5: Time for LLS solution and condition number

This confirms the interest of randomized algorithms also when applied to condition estimation. For instance the statistical estimates described in Section 3.2.2 would be useful in numerical libraries especially in the changing HPC landscape where existing algorithms might rapidly become sub-optimal when implemented on new architectures. Developing statistical condition estimates and integrating them into libraries is the object of an ongoing collaboration with Pr Alan Laub - University of California Los Angeles (UCLA) - USA.

Addressing very large size simulations

A major challenge for the randomized algorithms presented in Chapter 2 is to be able to solve very large problems. As a matter of fact, large-scale linear algebra solvers from standard parallel distributed libraries like ScaLAPACK [21] often suffer from expensive inter-node communication costs. An important requirement is to be able to schedule these algorithms dynamically on highly distributed parallel systems. In particular we must point out that even though randomizing removes the communication due to pivoting, applying recursive butterflies also generates communication, especially if we use multiple nodes to perform the randomization. This communication must be of course minimized or at least overlapped by computation. Below are very promising preliminary performance results for SRBT obtained on a cluster of 16 nodes, each node consisting of 2 quadcore processors. These very recent results have been obtained using the DAGUE runtime

system [23]. We observe in Figure 3.6 that the performance for LDL^T is similar to that of the Cholesky algorithm, and close to the practical peak of the platform. This work on efficiently implementing randomized algorithms on clusters of multicore is an ongoing collaboration with University of Tennessee.

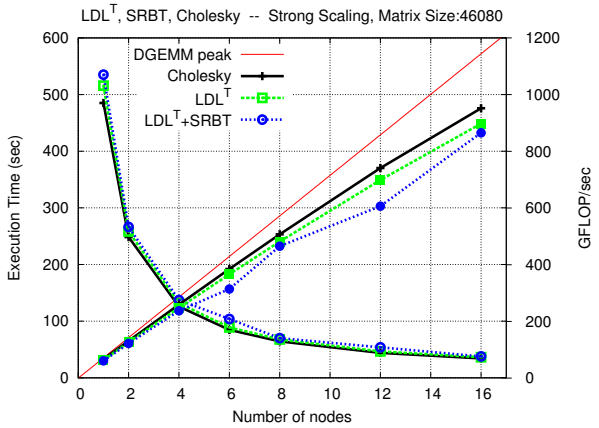


Figure 3.6: Performance of SRBT on clusters of multicore (46080x46080 matrix)

More applications for error analysis tools

We mentioned in Section 3.1 that the method based on condition numbers applies mainly to linear algebra problems. A track of research would be to combine our approach with other existing techniques, like for instance models of error propagation based on stochastic arithmetic [70]. It would then be possible, depending on the type of problem and of the existence of a condition estimate, to propose a specific algorithm. This would require an optimization of the routines for error propagation simulations that usually suffer from large overhead. A first step for this research would consist of comparing the accuracy and computational times between the different approaches. In a second phase, the study would be extended to nonlinear problems. Here again having industrial applications is essential and we plan to apply numerical quality tools to HPC applications developed at ONERA in aerodynamics and energetics and for which it is crucial to have an efficient numerical validation. The ultimate goal is to develop a specific library for error estimation so that physicits can control in a systematic way roundoff errors in linear algebra calculations but also can optimize the parameters related to the convergence of their iterative solvers.

Bibliography

[1] *Basic Linear Algebra Subprograms Technical Forum Standard*, Int. J. of High Performance Computing Applications **16** (2002), no. 1.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK users' guide*, 3 ed., SIAM, Philadelphia, 1999.

[3] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, *Communication-Avoiding QR decomposition for GPUs*, Tech. report, 2011, LAPACK Working Note 240, proceedings of IPDPS'11.

[4] M. Arioli, M. Baboulin, and S. Gratton, *A partial condition number for linear least-squares problems*, SIAM J. Matrix Anal. and Appl. **29** (2007), no. 2, 413–433.

[5] M. Arioli, J. W. Demmel, and I. S. Duff, *Solving sparse linear systems with sparse backward error*, SIAM J. Matrix Anal. and Appl. **10** (1989), no. 2, 165–190.

[6] C. Ashcraft, R. G. Grimes, and J. G. Lewis, *Accurate symmetric indefinite linear equation solvers*, SIAM J. Matrix Anal. and Appl. **20** (1998), no. 2, 513–561.

[7] A. Avron, P. Maymounkov, and S. Toledo, *Blendenpick: Supercharging LAPACK's least-squares solvers*, SIAM Journal on Scientific Computing **32** (2010), 1217–1236.

[8] M. Baboulin, D. Becker, and J. Dongarra, *A parallel tiled solver for dense symmetric indefinite systems on multicore architectures*, Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012), 2012, pp. 14–24.

[9] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, *Accelerating scientific computations with mixed precision algorithms*, Computer Physics Communications **180** (2009), no. 12, 2526–2533.

[10] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov, *A class of communication-avoiding algorithms for solving general dense linear systems on cpu/gpu parallel machines*, International Conference on Computational Science (ICCS 2012), Procedia Computer Science, vol. 9, Elsevier, 2012, pp. 17–26.

[11] M. Baboulin, J. Dongarra, J. Demmel, S. Tomov, and V. Volkov, *Enhancing the performance of dense linear algebra solvers on gpus in the magma project*, Poster at Supercomputing (SC'08), Austin USA, November 15, 2008, http://www.lri.fr/ baboulin/SC08.pdf.

[12] M. Baboulin, J. Dongarra, S. Gratton, and J. Langou, *Computing the conditioning of the components of a linear least-squares solution*, Numerical Linear Algebra with Applications **16** (2009), no. 7, 517–533.

[13] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, *Accelerating linear system solutions using randomization techniques*, ACM Trans. Math. Softw. **39** (2012), no. 2.

[14] M. Baboulin, J. Dongarra, and S. Tomov, *Some issues in dense linear algebra for multicore and special purpose architectures*, 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'08), Lecture Notes in Computer Science, vol. 6126-6127, Springer-Verlag, 2008.

[15] M. Baboulin and S. Gratton, *Using dual techniques to derive componentwise and mixed condition numbers for a linear function of a linear least squares solution*, BIT Numerical Mathematics **49** (2009), no. 1, 3–19.

[16] ———, *A contribution to the conditioning of the total least squares problem*, SIAM Journal on Matrix Analysis and Applications **32** (2011), no. 3, 685–699.

[17] M. Baboulin, S. Gratton, R. Lacroix, and A. J. Laub, *Efficient computation of condition estimates for linear least squares problems*, Research Report RR-8065, INRIA, 2012, Submitted to *Numerical Algorithms*.

[18] J.-C. Bajard, P. Langlois, D. Michelucci, G. Morin, and N. Revol, *Floating-point geometry: toward guaranteed geometric computations with approximate arithmetics*, SPIE Optics & Photonics 2008 Symposium (San Diego, CA, USA), vol. 7074, 2008.

[19] D. Becker, M. Baboulin, and J. Dongarra, *Reducing the amount of pivoting in symmetric indefinite systems*, 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011) (Heidelberg) (Roman Wyrzykowski et. al., ed.), Lecture Notes in Computer Science, vol. 7203, Springer-Verlag, 2012, pp. 133–142.

[20] Å. Björck, *Numerical methods for least squares problems*, SIAM, Philadelphia, 1996.

[21] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK users' guide*, SIAM, Philadelphia, 1997.

[22] S. Blackford and J. Dongarra, *Installation Guide for LAPACK*, Tech. report, 1999, LAPACK Working Note 41, revised version 3.0.

[23] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack J. Dongarra, *DAGuE: A generic distributed DAG engine for high performance computing*, Parallel Computing (2011), http://dx.doi.org/10.1016/j.parco.2011.10.003.

[24] J. R Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Math. Comput. **31** (1977), 163–179.

[25] J. R Bunch and B. N. Parlett, *Direct methods for solving symmetric indefinite systems of linear equations*, SIAM J. Numerical Analysis **8** (1971), 639–655.

[26] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, *The impact of multicore on math software*, (2006), In Proceedings of PARA 2006, Workshop on state-of-the art in scientific computing.

[27] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Computing **35** (2009), 38–53.

[28] Y. Cao and L. Petzold, *A subspace error estimate for linear systems*, SIAM J. Matrix Anal. and Appl. **24** (2003), 787–801.

[29] S. Chandrasekaran and I. C. F. Ipsen, *On the sensitivity of solution components in linear systems of equations*, SIAM J. Matrix Anal. and Appl. **16** (1995), no. 1, 93–112.

[30] B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters, and T. Priol, *Parallel Computing: From Multicores and GPUs to Petascale*, IOS Press, 2010.

[31] F. Cucker, H. Diao, and Y. Wei, *On mixed and componentwise condition numbers for Moore-Penrose inverse and linear least squares problems*, Mathematics of Computation **76** (2007), no. 258, 947–963.

[32] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, *Error bounds from extra-precise iterative refinement*, ACM Trans. Math. Softw. **32** (2006), no. 2, 325–351.

[33] J. Demmel, Y. Hida, X. S. Li, and E. J. Riedy, *Extra-precise iterative refinement for overdetermined linear least squares problems*, ACM Trans. Math. Softw. **35** (2009), no. 4, 1–32.

[34] J. W. Demmel, *Applied numerical linear algebra*, SIAM, 1997.

[35] J. W. Demmel, J. R. Gilbert, and X. S. Li, *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination*, SIAM J. Matrix Anal. and Appl. **20** (1999), no. 4, 915–952.

[36] I. Dimov, *Monte carlo methods for applied scientists*, Word Scientific, 2008.

[37] S. Donfack, L. Grigori, and A. K. Gupta, *Adapting communication-avoiding LU and QR factorizations to multicore architectures*, Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–10.

[38] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Numerical linear algebra for high-performance computers*, SIAM, 1998.

[39] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszcsek, *Achieving numerical accuracy and high performance using recursive tile LU factorization*, Tech. report, 2011, LAPACK Working Note 259.

[40] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Clarendon Press, Oxford, 1986.

[41] L. Eldén, *Perturbation Theory for the Least Squares Problem with Linear Equality Constraints*, SIAM J. Numerical Analysis **17** (1980), 338–350.

[42] J. Falcou and J. Sérot, *E.V.E.: An object oriented SIMD library*, Scalable Computing: Practice and Experience **6** (2005), no. 4, 31–41.

[43] A. J. Geurts, *A contribution to the theory of condition*, Numerische Mathematik **39** (1982), 85–96.

[44] G. Golub and J. Wilkinson, *Note on the iterative refinement of least squares solution*, Numerische Mathematik **9** (1966), no. 2, 139–148.

[45] G. H. Golub and C. F. Van Loan, *An analysis of the Total Least Squares problem*, SIAM J. Numerical Analysis **17** (1980), 883–893.

[46] ———, *Matrix computations*, The Johns Hopkins University Press, Baltimore, 1996, Third edition.

[47] N. I. M. Gould, J. A. Scott, and Y. Hu, *A numerical evaluation of sparse solvers for symmetric systems*, ACM Trans. Math. Softw. **33** (2007), no. 2, 10:1–10:32.

[48] A. Graham, *Kronecker products and matrix calculus with application*, Wiley, New York, 1981.

[49] S. Gratton, *On the condition number of linear least squares problems in a weighted Frobenius norm*, BIT Numerical Mathematics **36** (1996), no. 3, 523–530.

[50] J. F. Grcar, *Adjoint formulas for condition numbers applied to linear and indefinite least squares*, Technical Report LBNL-55221, Lawrence Berkeley National Laboratory, 2004.

[51] L. Grigori, J. Demmel, and H. Xiang, *CALU: a communication optimal LU factorization algorithm*, SIAM J. Matrix Anal. and Appl. **32** (2011), 1317–1350.

[52] T. Gudmundsson, C. S. Kenney, and A. J. Laub, *Small-sample statistical estimates for matrix norms*, SIAM J. Matrix Anal. and Appl. **16** (1995), 776–792.

[53] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, *Formal Linear Algebra Methods Environment*, ACM Trans. Math. Softw. **27** (2001), no. 4, 422–455.

[54] F. G. Gustavson, *Recursion leads to automatic variable blocking for dense linear-algebra algorithms*, IBM Journal of Research and Development **41** (1997), no. 6, 737–755.

[55] W. W. Hager, *Condition estimates*, SIAM J. Sci. Statist. Comput. **5** (1984), no. 2, 311–316.

[56] N. Halko, P. G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review **53** (2011), 217–288.

[57] P. Hénon, P. Ramet, and J. Roman, *On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes*, In PPMA'05, *LNCS* **3911** (2005), 1050–1057.

[58] V. Hernandez, J. E. Roman, and V. Vidal, *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems*, ACM Trans. Math. Softw. **31** (2005), no. 3, 351–362.

[59] N. J. Higham, *A survey of componentwise perturbation theory in numerical linear algebra*, In W. Gautschi editor, Mathematics of Computation 1943-1993: A Half Century of Computational Mathematics, volume 48 of Proceedings of Symposia in Applied Mathematics (1994), 49–77, American Mathematical Society, Providence, RI, USA.

[60] ———, *Accuracy and stability of numerical algorithms*, 2 ed., SIAM, Philadelphia, 2002.

[61] S. Van Huffel and J. Vandewalle, *The total least squares problem: computational aspects and analysis*, SIAM, Philadelphia, 1991.

[62] Intel, *Math Kernel Library (MKL)*, `http://www.intel.com/software/products/mkl/`.

[63] I. C. F. Ipsen, *Numerical matrix analysis: Linear systems and least squares*, SIAM, Philadelphia, 2009.

[64] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. Peterson, *Power aware computing on GPUs*, SAAHPC '12, Argonne, IL, 2012.

[65] C. S. Kenney and A. J. Laub, *Small-sample statistical condition estimates for general matrix functions*, SIAM J. Sci. Comput. **15** (1994), 36–61.

[66] C. S. Kenney, A. J. Laub, and M. S. Reese, *Statistical condition estimation for linear least squares*, SIAM J. Matrix Anal. and Appl. **19** (1998), no. 4, 906–923.

[67] H. Kim, S. Wu, L. Chang, and W. Hwu, *A scalable tridiagonal solver for GPUs*, ICPP '11 Proceedings of the 2011 International Conference on Parallel Processing, ACM, 2011, pp. 444–453.

[68] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller, *On the computation of correctly-rounded sums*, IEEE Transactions on Computers **61** (2012), no. 3, 289–298.

[69] J. Kurzak and J. Dongarra, *Implementing linear algebra routines on multi-core processors with pipelining and a look ahead*, Tech. report, 2006, LAPACK Working Note 178.

[70] J.-L. Lamotte, J.-M. Chesneaux, and F. Jézéquel, *Cadna_c: A version of cadna for use with c or c++ programs*, Computer Physics Communications **181** (2010), no. 11, 1925–1926.

[71] Philippe Langlois, Matthieu Martel, and Laurent Thévenoux, *Accuracy versus time: a case study with summation algorithms*, 4th International Workshop on Parallel and Symbolic Computation (PASCO'10) (New York, NY, USA), ACM, 2010, pp. 120–130.

[72] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. J. Dongarra, *Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy*, Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006.

[73] M. W. Mahoney, *Randomized algorithms for matrices and data*, Foundations and Trends in Machine Learning **3** (2011), no. 2, 123–224.

[74] C. B. Moler, *Iterative refinement in floating point*, J. ACM **14** (1967), no. 2, 316–321.

[75] R. Nath, S. Tomov, and J. Dongarra, *An improved MAGMA GEMM for Fermi GPUs*, International Journal of High Performance Computing Applications **24** (2010), no. 4, 511–515.

[76] NVIDIA, *NVIDIA CUDA C Programming Guide*, 04/16/2012, Version 4.2.

[77] W. Oettli and W. Prager, *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides*, Numerische Mathematik **6** (1964), 405–409.

[78] University of Tennessee, *PLASMA users' guide, parallel linear algebra software for multicore architectures, version 2.3*, 2010.

[79] J. M. Ortega and C. H. Romine, *The ijk forms of factorization methods II. Parallel systems*, Parallel Computing **7** (1988), no. 2, 149–162.

[80] C. Paige and Z. Strakoš, *Core problems in linear algebraic systems*, SIAM J. Matrix Anal. and Appl. **27** (2006), no. 3, 861–875.

[81] C. C. Paige and M. A. Saunders, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Softw. **8** (1982), 43–71.

[82] D. S. Parker, *Random butterfly transformations with applications in computational linear algebra*, Technical Report CSD-950023, Computer Science Department, UCLA, 1995.

[83] B. Podvin, Y. Fraigneau, F. Lusseyran, and P. Gougat, *A reconstruction method for the flow past an open cavity*, J. Fluids Engineerings **128** (2006), 531–540.

[84] G. Quintana-Orti, E. S. Quintana-Orti, R. A. van de Geijn, F. G. van Zee, and E. Chan, *Programming algorithms-by-blocks for matrix computations on multithreaded architectures*, ACM Trans. Math. Softw. **36** (2009), no. 3, 1–26.

[85] J. Rice, *A theory of condition*, SIAM J. Numerical Analysis **3** (1966), 287–310.

[86] S. Rump, *Fast and parallel interval arithmetic*, BIT Numerical Mathematics **39** (1999), no. 3, 534–554.

[87] ———, *Verification methods: Rigorous results using floating-point arithmetic*, Acta Numerica **19** (2010), 287–449.

[88] O. Schenk and K. Gärtner, *On fast factorization pivoting methods for symmetric indefinite systems*, Elec. Trans. Numer. Anal. **23** (2006), 158–179.

[89] R. D. Skeel, *Iterative refinement implies numerical stability for Gaussian elimination*, Math. Comput. **35** (1980), no. 151, 817–832.

[90] D. C. Sorensen, *Analysis of pairwise pivoting in Gaussian elimination*, IEEE Trans. Comput. **34** (1984), 274–278.

[91] G. W. Stewart, *Introduction to matrix computations*, Academic Press, 1973.

[92] P. E. Strazdins, *A dense complex symmetric indefinite solver for the Fujitsu AP3000*, Technical Report TR-CS-99-01, The Australian National University, 1999.

[93] The Numerical Algorithms Group, *NAG library manual, Mark 21*, NAG, 2006.

[94] S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems*, Parallel Computing **36** (2010), no. 5&6, 232–240.

[95] S. Tomov, R. Nath, and J. Dongarra, *Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing*, Parallel Computing **36** (2010), no. 12, 645–654.

[96] L. N. Trefethen and R. S. Schreiber, *Average-case stability of Gaussian elimination*, SIAM J. Matrix Anal. and Appl. **11** (1990), no. 3, 335–360.

[97] A. Turing, *Rounding-off errors in matrix processes*, Quart. J. Mech. and Applied Math. **1** (1948), 287–308.

[98] V. Volkov and J. Demmel, *Using GPUs to accelerate linear algebra routines*, Poster at PAR lab winter retreat, January 9, 2008, http://www.eecs.berkeley.edu/~volkov/volkov08-parlab.pdf.

[99] V. Volkov and J. W. Demmel, *LU, QR and Cholesky factorizations using vector capabilities of GPUs*, Technical Report UCB/EECS-2008-49, University of California, Berkeley, 2008, Also LAPACK Working Note 202.

[100] P.-Å. Wedin, *Perturbation theory for pseudo-inverses*, BIT **13** (1973), 217–232.

[101] J. H. Wilkinson, *Rounding errors in algebraic processes*, vol. 32, Her Majesty's Stationery Office, London, 1963.

[102] I. Yamazaki, S. Tomov, and J. Dongarra, *One-sided dense matrix factorizations on a multicore with multiple GPU accelerators*, International Conference on Computational Science (ICCS 2012), Procedia Computer Science, vol. 9, Elsevier, 2012, pp. 37–46.

[103] A. YarKhan, J. Kurzak, and J. Dongarra, *QUARK users guide: QUeueing And Runtime for Kernels*, Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory, 2011.

[104] T. J. Ypma, *Historical development of the Newton–Raphson method*, SIAM Review **37** (1995), no. 4, 531–551.

[105] M. Zelen, *Linear estimation and related topics*, Survey of numerical analysis (J.Todd, ed.), McGraw-Hill, New York, 1962, pp. 558–584.