A parallel iterative solver for large sparse linear systems enhanced with randomization and GPU accelerator, and its resilience to soft errors

Aygul Jamal

NNT : 2017SACLS269

Thèse de doctorat de l'Université Paris-Saclay préparée à l'Université Paris-Sud

Ecole doctorale n°580 :
Sciences et technologies de l'information et de la communication
Spécialité de doctorat : Informatique

par

Aygul JAMAL

A parallel iterative solver for large sparse linear systems enhanced with randomization and GPU accelerator, and its resilience to soft errors

Thèse présentée et soutenue au LRI, le 28 Septembre 2017.

Composition du Jury :

M.	Florent Hivert	Professeur Université Paris-Sud	(Président du jury)
Mme.	Fabienne Jezequel	Maître de Conférences, HDR Université Panthéon-Assas	(Rapportrice)
M.	Jose E. Roman	Professeur Universitat Politècnica de València	(Rapporteur)
Mme	Camille Coti	Maître de Conférences Université Paris 13	(Examinatrice)
M.	Marc Baboulin	Professeur Université Paris-Sud	(Directeur de thèse)
Mme.	Amal Khabou	Maître de Conférences Université Paris-Sud	(Co-Encadrante de thèse)

Résumé

Dans cette thèse de doctorat, nous abordons trois défis auxquels sont confrontés les solveurs d'algèbre linéaire dans la perspective des futurs systèmes exascale: accélérer la convergence en utilisant des techniques innovantes au niveau algorithmique, tirer profit des accélérateurs GPU (Graphics Processing Units) pour améliorer la performance des calculs sur les systèmes hybrides CPU/GPU, évaluer l'impact des erreurs plus fréquentes du fait de l'augmentation du parallélisme dans les supercalculateurs. Nous nous intéressons à l'étude des méthodes permettant d'accélérer la convergence et le temps d'exécution des solveurs itératifs pour les grands systèmes linéaires creux. Le solveur plus spécifiquement considéré dans ce travail est le "parallel Algebraic Recursive Multilevel Solver" (pARMS) qui est un solveur parallèle à mémoire distribuée basé sur les méthodes de Krylov.

Tout d'abord, nous proposons d'intégrer une technique de randomisation appelée "Random Butterfly Transformations (RBT)" qui a été appliquée avec succès pour éliminer le coût du pivotage dans la résolution des systèmes linéaires denses. Notre objectif est d'appliquer cette méthode dans le préconditionneur ARMS de pARMS pour résoudre plus efficacement le dernier système de complément de Schur dans l'application du processus récursif multi-niveaux. Les résultats expérimentaux montrent une amélioration de la convergence et de la précision par rapport aux implémentations existantes. En raison de possibles problèmes d'occupation mémoire pour certains problèmes tests, nous proposons également d'utiliser une variante "creuse" du RBT suivie d'un solveur direct creux (SuperLU), ce qui a pour effet d'améliorer le temps d'exécution.

Ensuite, nous expliquons comment une approche non intrusive peut être appliquée pour implémenter des calculs GPU dans le solveur pARMS, plus particulièrement dans la phase de préconditionnement locale qui représente une partie importante du temps de résolution. Puis nous comparons les versions purement CPU et hybrides CPU/GPU du solveur sur plusieurs problèmes tests issus d'applications physiques. Les expériences portant sur notre solveur hybride CPU/GPU utilisant le préconditionnement ARMS combiné avec le RBT, ou le préconditionnement ILU(0), montrent un gain de performance allant jusqu'à 30% sur les problèmes considérés dans nos expériences.

Enfin, nous étudions l'effet des fautes logicielles sur la convergence de la méthode itérative `GMRES flexible` (FGMRES) qui est utilisée pour résoudre le système préconditionné dans pARMS. Le problème ciblé dans nos expériences est un problème elliptique d'EDP sur une grille régulière. Nous considérons deux types de préconditionneurs: une factorisation LU incomplète à double seuil (ILUT) et le préconditionneur ARMS combiné avec la randomisation RBT. Nous considérons deux modèles de fautes logicielles différentes où nous perturbons la multiplication matrice-vecteur et l'application du préconditionneur, et nous comparons leur impact sur la convergence du solveur.

Mots-clés : Calcul haute performance, solveurs linéaires itératifs parallèles, solveur pARMS, preconditionnement, algorithmes randomisés, calculs sur GPU, GMRES Flexible, tolérance aux fautes, modèles de fautes logicielles.

Abstract

In this PhD thesis, we address three challenges faced by linear algebra solvers in the perspective of future exascale systems: accelerating convergence using innovative techniques at the algorithm level, taking advantage of GPU (Graphics Processing Units) accelerators to enhance the performance of computations on hybrid CPU/GPU systems, evaluating the impact of errors in the context of an increasing level of parallelism in supercomputers. We are interested in studying methods that enable us to accelerate convergence and execution time of iterative solvers for large sparse linear systems. The solver specifically considered in this work is the parallel Algebraic Recursive Multilevel Solver (pARMS), which is a distributed-memory parallel solver based on Krylov subspace methods.

First we integrate a randomization technique referred to as Random Butterfly Transformations (RBT) that has been successfully applied to remove the cost of pivoting in the solution of dense linear systems. Our objective is to apply this method in the ARMS preconditioner to solve more efficiently the last Schur complement system in the application of the recursive multilevel process in pARMS. The experimental results show an improvement of the convergence and the accuracy. Due to memory concerns for some test problems, we also propose to use a sparse variant of RBT followed by a sparse direct solver (SuperLU), resulting in an improvement of the execution time.

Then we explain how a non intrusive approach can be applied to implement GPU computing into the pARMS solver, more especially for the local preconditioning phase that represents a significant part of the time to compute the solution. We compare the CPU-only and hybrid CPU/GPU variant of the solver on several test problems coming from physical applications. The performance results of the hybrid CPU/GPU solver using the ARMS preconditioning combined with RBT, or the ILU(0) preconditioning, show a performance gain of up to 30% on the test problems considered in our experiments.

Finally we study the effect of soft fault errors on the convergence of the commonly used `flexible GMRES` (FGMRES) algorithm which is also used to solve the preconditioned system in pARMS. The test problem in our experiments is an elliptical PDE problem on a regular grid. We consider two types of preconditioners: an incomplete LU factorization with dual threshold (ILUT), and the ARMS preconditioner combined with RBT randomization. We consider two soft fault error modeling approaches where we perturb the matrix-vector multiplication and the application of the preconditioner, and we compare their potential impact on the convergence of the solver.

Keywords: High performance computing, parallel iterative linear solvers, pARMS solver, preconditioning, randomized algorithms, GPU computing, Flexible GMRES, fault tolerance, soft fault models.

Contents

List of Figures

List of Tables

Introduction

The recent trends in High-Performance Computing (HPC), as shown for instance in the Top 500 list[1], confirm the ever increasing development of systems combining multicore processors with accelerators (like GPUs or Intel Xeon Phi). This requires the adaptation or the redesign of many of the numerical linear algebra algorithms which are at the heart of scientific applications. In the perspective of the coming exascale systems in the years 2020s [1], we are interested in investigating innovative methods that take advantage of current highly parallel architectures and apply them to solvers used in major scientific applications.

In this PhD thesis, we focus on preconditioned iterative solvers for large sparse linear systems, and more specifically on solvers based on Krylov subspace methods. We are interested in preconditioning techniques, which improve the spectral properties of the coefficient matrix in order to solve the linear system more easily. The targeted solver for our work is the parallel Algebraic Recursive Multilevel Solver (pARMS), which is a distributed-memory parallel iterative solver and uses the Flexible Generalized Minimal Residual method (FGMRES) to solve the preconditioned system.

The randomized algorithms are now gaining ground in HPC applications and are more and more used to enhance linear algebra solvers since they can outperform deterministic methods while providing accurate results. We will use a randomized algorithm called Random Butterfly Transformations (RBT), to improve the convergence and the accuracy of pARMS.

We also propose a version of pARMS that can exploit the high level of parallelism provided by Graphics Processing Units (GPU). In this work, we integrate GPU functions into pARMS on top of the widespread MAGMA library[2] that has been recently extended to sparse matrix computations. The GPU implementation of pARMS will mainly concern the preconditioning phase that is dominant in the global computational time (preconditioners based on RBT or Incomplete LU factorization).

Another major trend in HPC is the increase of parallelism since a supercomputer may consist of several millions of computing units, with the drawback of increasing also the probability of occurence of hardware or software faults during the computation. It is then necessary to study the impact of these errors on a parallel solver such as pARMS.

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

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

This manuscript is organized as follows:

In Chapter 1, we introduce the main methods (direct and iterative) for solving linear systems with a specific focus on the Krylov subspace methods. We introduce the solver pARMS that will be the solver of choice for this PhD thesis. We also present the main existing parallel software libraries, the parallel architectures and programming paradigms that we will use in our work and experiments.

In Chapter 2, we use a randomized algorithm based on RBT randomization technique to enhance the preconditioning phase in the pARMS solver. Our experiments on test matrices from the Davis' collection showed an improvement in the number of iterations and accuracy. We also developed a sparse variant of RBT that enables us to improve the execution time by taking into account the potential sparsity of the last Schur complement system.

In Chapter 3, we present a hybrid CPU/GPU version of the pARMS solver where the preconditioning phase is performed on the GPU. This hybrid version of pARMS is implemented on top of the MAGMA library and concerns the ARMS preconditioner using Random Butterfly Transformations (RBT), and the Incomplete LU factorization preconditioning. Each preconditioning method provides a good performance for a given set of matrices.

In Chapter 4, we focus on the soft fault errors, and we present two types of fault models. We study the impact of these two soft fault models in the framework of the pARMS solver, where we evaluate the resilience of two preconditioners in the pARMS solver. Additionally, we present the experimental work and give the corresponding explanations.

The last chapter is a conclusion of this PhD thesis, which also proposes possible directions of research.

Chapter 1

Linear systems, software libraries, parallel architectures

1.1 Introduction

Applications with intensive computations, intensive memory usage and intensive data movement require the use of High Performance Computing (HPC). "HPC generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business" [2]. The traditional mono CPU computers can not deal with such a high computing demand. However, HPC allocates a high intensive work for a single problem by using multiple CPUs to decrease the execution time. Parallelism in todays computer architectures is ubiquitous whether in supercomputers or small portable devices such as smartphones or watches. The increasing complexity in CPU design has lead manufacturers to develop more sophisticated architectures. These new architectures that allow to increase the computational power vary from CPU multicores to accelerators such as graphics processing units (GPUs). Exploiting the full performance of such architectures for numerical problems has become very challenging.

In order to take fully advantage of these new architectures, software libraries are developed to give users access to efficient linear algebra computations. The first major numerical libraries to appear include the Basic Linear Algebra Subprograms (BLAS [3]) and the Linear Algebra Package (LINPACK [4]) in 1979. Since then, architecture specific libraries for dense or sparse linear algebra have been developed such as LAPACK [5] for cache-based CPUs, ScaLAPACK [6] for distributed architectures, PLASMA [7] for multicore architectures, and more recently MAGMA [8] for hybrid architectures using GPUs or XeonPhi.

In this thesis, we are interested in solving systems of linear equations, $Ax = b$, where the matrix A is sparse. Such systems may arise from the discretization of partial differential equations. There are several libraries that implement direct methods for solving sparse systems such as MUMPS [9], PARADISO [10], and SuperLU [11]. However, when the matrix A is sparse, the factors obtained from the matrix decomposition when using direct methods are denser than the input matrix. Moreover, direct methods are prohibitive in terms of memory and floating point operations when solving very large systems. These direct methods are also not easily parallelized on modern architectures. Thus, iterative methods, computing a sequence of approximate solutions for the system $Ax = b$ starting from an initial guess, are a good alternative. The Krylov subspace methods are among the most practical and popular iterative methods. The solvers we are considering in this thesis, ARMS [12] and pARMS [13] are based on some Krylov subspace methods. Our work aims at improving the performance of these two solvers on modern architectures.

This chapter presents an overview of the different linear algebra methods, the software libraries and the parallel architectures that we used in this thesis. The chapter is organized as follows. Section 1.2 introduces the main methods used to solve a given linear system $Ax = b$ including direct and iterative methods. Section 1.3 gives an overview on the Krylov subspace methods in particular the generalized minimal residual (GMRES) method. Section 1.4 presents some preconditioning techniques used in this thesis including preconditioners based on classical iterative methods as well as preconditioners based on matrix factorization such as incomplete LU factor-

ization. Section 1.5 describes the algebraic recursive multilevel solver (ARMS) [12] and its parallel version pARMS [13]. Section 1.6 describes the main linear algebra libraries that we have been using in this thesis. Section 1.7 recalls the main parallel architectures as well as the programming models of interest for this work. Section 1.8 briefly concludes this chapter.

1.2 Methods to solve general linear systems

In this thesis, we are interested in the solution of linear systems. A linear system, or a system of linear equations, is a set of m equations and n variables. Linear solvers are various: depending on the structure of the coefficient matrix, we use symmetric or non-symmetric solvers, depending on the number of non-zero entries, we use sparse or dense solvers, and depending on the properties of the solver, we can have direct or iterative solvers. According to our own needs, we choose the most suitable solver to solve the specific linear system, $Ax = b$, that we are considering, where A is $m \times n$ matrix, x is a column vector with n unknowns, b is a column vector with m entries.

Direct methods generally involve the decomposition of matrices followed by successive resolutions of triangular systems. Different methods of decomposition exist such as QR factorization, Cholesky, LDL$^{\mathrm{T}}$ or LU [14].

LU: is used to solve general systems and decomposes a matrix A into the product of a unit lower triangular matrix L and an upper triangular matrix U. It requires about $2n^3/3$ floating point operations (flops) for a square matrix of size $n \times n$.

LDL$^{\mathbf{T}}$: is used for symmetric matrices. A symmetric matrix A is decomposed as follows, $A = LDL^T$ where L is a unit lower triangular matrix and D is a diagonal (or block-diagonal) matrix. It requires about $n^3/3$ flops for a square matrix of size $n \times n$.

Cholesky: for symmetric positive definite matrices. The coefficient matrix A is factored as $A = LL^T$, where L is a lower triangular matrix with positive diagonal entries. It requires about $n^3/3$ flops for a square matrix of size $n \times n$.

QR: to solve full rank least squares problems in the case of overdetermined systems (the system has more equations than unknowns). A matrix A of size $m \times n$ is factored as $A = QR$, where Q is an m by m orthogonal matrix and R is an m by n upper triangular matrix. It requires about $2n^2(m - n/3)$ flops.

Direct methods can become impractical if the matrix A is large and sparse, since the factorization step can lead to dense factors. Thus, there are two approaches to solve a sparse problem, $Ax = b$. The first one is to use a direct method and adapt it to exploit the matrix sparsity pattern in order to minimize the fill-in. The second approach is the use of iterative methods. These methods generate a

sequence of approximate solutions x_k and essentially involve the matrix A through matrix-vector multiplication. Classical examples of iterative methods are the Jacobi method [15, Chapter 4], the Gauss-Seidel method [15, Chapter 4], the successive over-relaxation (SOR) method [15, Chapter 4], the generalized minimal residual method [15, Chapter 6], the conjugate gradient method [15, Chapter 6], etc. Among the reasons for the great interest in sparse solvers is that they allow to obtain numerical solutions to partial differential equations (PDE).

In the remainder, we give an overview of the methods that we use throughout this thesis, that is the LU factorization and some iterative methods including mainly Krylov subspace methods.

1.2.1 LU factorization

The LU factorization [14, Chapter 3]is very common in dense linear algebra. It decomposes the input matrix A into the product $L \times U$, where L is a lower triangular matrix with diagonal elements equal to one and U is an upper triangular matrix. As stated above, computing the LU decomposition of a square matrix of size $n \times n$ requires $\frac{2n^3}{3}$ floating point operations.

Algorithm 1 shows how the LU factorization can be performed in place which means that the input matrix A is overwritten by the output factors L and U.

Algorithm 1 In place LU factorization without pivoting

```
1: for k ← 1 to n − 1 do
2:     for i ← k + 1 to n do
3:         A(i, k) ← A(i, k)/A(k, k)
4:     end for
5:     for i ← k + 1 to n do
6:         for j ← k + 1 to n do
7:             A(i, j) ← A(i, j) − A(i, k) ∗ A(k, j)
8:         end for
9:     end for
10: end for
```

With the method described previously, if a 0 is found on the diagonal of the matrix, a division by zero will occur and the factorization will fail. Also if elements of small magnitude are on the diagonal, entries on the triangular factors will grow significantly. Moreover, rounding errors are unavoidable since the numerical precision is limited on a computer when finite precision arithmetic is used. These errors due to limited precision will be propagated and amplified by the division by very small values. Thus, the larger systems are more prone to rounding errors.

The stability of the LU factorization can be measured by the growth factor which indicates how large the entries of the matrix become during the elimination steps comparing to the largest entries of the initial input matrix. The growth factor of a matrix A of size $n \times n$ is defined as,

$$g_n(A) = \frac{\max_{i,j,k} |a_{ij}^{(k)}|}{\max_{i,j} |a_{i,j}|},$$

where $a_{ij}^{(k)}$ is the element of index (i, j) after the step number k of the elimination [16]. For this reason we need to move the largest element of the column on the diagonal position by swapping rows. This pivoting method is called partial pivoting. It is also possible to swap rows and columns, using the largest element of the matrix (complete pivoting).

Even though pivoting improves the numerical stability and requires no additional floating point operations, it involves irregular data movements due to the comparisons performed in the process of finding the pivot. If n is the size of the matrix, complete pivoting requires $\mathcal{O}(n^3)$ comparisons and partial pivoting $\mathcal{O}(n^2)$ comparisons. Therefore, even if complete pivoting has the best stability with a growth factor bound of $cn^{1/2}n^{1/4 \log n}$ [17] comparing to partial pivoting (2^{n-1}) [17], it is time-consuming, due to the comparisons and data movements. The choice of the pivoting strategy is then the result of a compromise between the numerical stability and the performance. One pivoting strategy that aims at meeting this compromise for large problems is the tournament pivoting that was first used in the communication avoiding LU factorization [18, 19]. Some other techniques aim at avoiding pivoting by performing a preprocessing step on the input matrix A such as the randomization of the matrix A [20]. More details about this technique will be given in Chapter 2.

Case of sparse LU factorization

As described in the previous paragraph, the numerical stability is a primary issue in the context of the LU factorization. Moreover, when the matrix A is sparse then the fill-in in the L and U factors becomes a concern as well. The challenge is to find out a trade-off such that the obtained LU decomposition is reasonably stable and the triangular factors L and U are close to being optimally sparse [21, Chapter 6]. Similar to other sparse direct methods, the sparse LU usually performs three steps in order to find the solution of a sparse linear system $Ax = b$. The first step consists in analyzing the sparse matrix A and defining a graph that represents the dependencies between computations on smaller dense blocks. This step aims at estimating and limiting the amount of computations and resources required for the factorization. In general, a preprocessing (reordering [21, Chapter 7] the rows and columns) is applied to the matrix in order to improve its numerical characteristics and reduce the fill-in. The second step is the factorization of the matrix, which consists in applying the corresponding computations to the graph that was produced in the first step. The third step is applying forward elimination then backward substitution to obtain the vector solution.

1.2.2 Iterative methods

The main principle of an iterative method is to generate a sequence of iterates x_k, which after a certain number of iterations converge to the solution of the linear

system, $Ax = b$. These methods essentially involve matrix-vector products, but are often combined with preconditioning techniques, where we consider for example, the linear system $M^{-1}Ax = M^{-1}b$, where $M^{-1}A$ has better convergence properties than A and where the system $M^{-1}y = z$, with $y = Ax$ and $z = M^{-1}b$ is easier to solve than the original system $Ax = b$. More details about various preconditioners and preconditioning techniques will be presented in Section 1.4. Stationary iterative methods define x_{k+1} as a linear function of x_k, that is, they can be written as $x_{k+1} = Gx_k + f$ for some iteration matrix G such as the Jacobi method [15, Chapter 4] and the Gauss-Seidel method [15, Chapter 4]. More advanced methods like Krylov subspace methods (e.g., conjugate gradient [15, Chapter 6], generalized minimal residual (GMRES) [15, Chapter 6]) build a subspace of increasing size at each iteration, in which the next approximate vector is found. We note that iterative methods can be combined with direct methods, either to build a preconditioner (e.g. incomplete LU factorizations), or to design hybrid methods such as the domain decomposition methods where a direct solver is used within each domain [22].

An iterative procedure should keep going until the convergence. This is impractical and usually unnecessary. Therefore, in general a stopping criteria is used to stop the iterative procedure when a pre-specified condition is met. One of the most used stopping criteria is based on the evolution of the residual vector $r_k = Ax_k - b$ along the iterations. Specifically, given a small $\varepsilon > 0$, the iterative procedure is stopped after the kth iteration when $||x_{k+1} - x_k|| \leq \varepsilon$, $||r_{k+1} - r_k|| \leq \varepsilon$, or $||r_k|| \leq \varepsilon$. A maximum number of iterations is also usually specified. If an iterative method does not meet the stopping criteria before reaching the maximum iteration number, it is considered to be inefficient. In this case, alternative methods should be considered or a preconditioner should be applied.

In the next section, we focus on the Krylov subspace methods.

1.3 Krylov subspace methods

Krylov subspace methods are general methods to solve sparse linear systems. They are named after the Russian mathematician and engineer Alexei Krylov who first introduced these methods in 1931. In this work, we focus on one variant of the Krylov subspace methods which is the generalized minimal residual (GMRES) method.

1.3.1 The Krylov subspace methods

A Krylov subspace of order-j $\mathcal{K}_j$ generated by a matrix A of size $n \times n$ and a vector y of size n is the linear subspace spanned by the images of the vector y under the first j powers of the matrix A. Thus, $\mathcal{K}_j(A, y) = span\{y, Ay, A^2y, \ldots, A^{j-1}y\}$. The Krylov subspace verifies the following properties:

$$\mathcal{K}_1(A, y) \subseteq \mathcal{K}_2(A, y) \subseteq \mathcal{K}_3(A, y) \subseteq \cdots \subseteq \mathcal{K}_n(A, y) \subseteq \ldots,$$
$$A\mathcal{K}_j(A, y) \subseteq \mathcal{K}_{j+1}(A, y).$$

The Krylov subspace methods are polynomial iterative methods that aim at

solving a given linear system $Ax = b$ by finding a sequence of iterates x_j for $j = 1, \dots k$ minimizing some error measure over the corresponding spaces,

$$x_0 + \mathcal{K}_j(A, r_0), \quad for \quad j = 1, \dots k,$$

where x_0 is the initial iterate or guess, $r_0 = b - Ax_0$ is the initial residual and $\mathcal{K}_j(A, r_0)$ is the Krylov subspace of order j generated by the matrix A and the residual vector r_0. Among the Krylov Subspace methods, we can cite the conjugate gradient (CG) [23], the generalized minimum residual (GMRES) [24], the bi-conjugate gradient (Bi-CG) [25].

The Krylov projection methods compute a sequence of approximate solutions $x_k \in x_0 + \mathcal{K}_k, \quad k = 1, 2, \dots$ to the linear system $Ax = b$, such that the k^{th} residual, $r_k = b - Ax_k$ verifies the Petrov-Galerkin constraint,

$$r_k \perp \mathcal{L}_k,$$

where $\mathcal{L}_k \subseteq \mathbb{R}^n$ (or $\mathcal{L}_k \subseteq \mathbb{C}^n$) is a well-defined subspace of dimension k. We note that $\mathcal{L}_k$ can be the same as the Krylov subspace $\mathcal{K}_k$ or different. The different choices for the subspace $\mathcal{L}_k$ results in different Krylov projection methods [15, Chapter 6]. For instance, GMRES is a Krylov projection method where $\mathcal{L}_k = A\mathcal{K}_k$. Thus, the different Krylov projection methods are defined by the subspace $\mathcal{L}_k$ and the two following conditions:

- the subspace condition: $x_k \in x_0 + \mathcal{K}_k$
- the Petrov-Galerkin condition: $r_k \perp \mathcal{L}_k$.

1.3.2 Generalized minimal residual (GMRES) method

The generalized minimal residual method (GMRES), introduced by Saad and Schultz [24], is a Krylov projection method for solving general linear systems, $Ax = b$. It computes approximate solutions $x_k \in x_0 + \mathcal{K}_k$, where x_0 is an initial guess such that,

$$\|r_k\|_2 = \|b - Ax_k\|_2 = min\{\|b - Ax\|_2, \quad \forall x \in x_0 + \mathcal{K}_k\}$$

We note that the residual vectors in GMRES do not form an orthonormal basis for the subspace $\mathcal{K}_k$. To build an orthonormal basis for $\mathcal{K}_k$, the Arnoldi process, as described in Algorithm 2, is used. At each Arnoldi iteration, a new basis vector is computed and orthonormalized against the previous vectors. Starting from a general, nonsymmetric matrix A of size $n \times n$, the Arnoldi process reduces A into an upper Hessenberg form by the similarity transformation

$$A = V_m H_m V_m^T \quad or \quad AV_m = V_m H_m,$$

where H_m is an upper Hessenberg matrix of size $m \times m$ obtained by deleting the last row of $\bar{H}_m$, the Hessenberg matrix of size $(m+1) \times m$ produced by the Arnoldi process and V_m is a matrix of size $n \times m$ with $V_m^T V_m = I$. We note that $m \leq n$ is the largest index such that $v_m \neq 0$. Since V_m satisfies $V_m^T V_m = I$, then $\{v_1, v_2, \dots, v_k\}$ form an orthonormal basis for $\mathcal{K}_k$. For $1 \leq k \leq m$, we have:

$$AV_k = V_{k+1} \bar{H}_k,$$

where V_k is a matrix with orthonormal columns of size $n \times k$ and $\bar{H}_{k-1}$ is an upper Hessenberg matrix of size $(k+1) \times k$. At the k^{th} iteration of the Arnoldi process, the vector $v_k = Av_{k-1}$ is computed and orthonormalized against $\{v_1, v_2, \ldots, v_{k-1}\}$. Thus, the last column of the matrix $\bar{H}_{k-1}$ is produced.

Algorithm 2 Arnoldi [15, Section 6.4]

1: Choose a vector v_1 such that $||v_1||_2 = 1$
2: **for** k = 1, 2, ..., m **do**,
3: **for** $i = 1, 2, ..., k$ **do**,
4: Compute $h_{ik} := (Av_k, v_i)$
5: **end for**
6: Compute $\omega_k := Av_k - \sum_{i=1}^{k} h_{ik} v_i$
7: $h_{k+1,k} = \|\omega_k\|_2$
8: **if** $h_{k+1,k} = 0$ **then** Stop
9: **end if**
10: $v_{k+1} = \frac{\omega_k}{h_{k+1,k}}$
11: **end for**

After building the orthonormal basis $\{v_1, v_2, \ldots, v_k\}$ for the subspace $\mathcal{K}_k$ using the Arnoldi process, the least squares problem $\min_{x \in x_0 + \mathcal{K}_k} \|b - Ax\|_2$ becomes $\min_{y \in \mathbb{R}^k} \|\beta_0 e1 - \bar{H}_k y\|_2$, where $x_k = x_0 + V_k y$, V_k is a matrix with orthonormal columns of size $n \times k$, $\bar{H}_k$ is an upper Hessenberg matrix of size $(k+1) \times k$, $\beta_0 = \|r_0\|_2$ and $e1 = [1 \quad 0 \quad 0 \ldots 0]^t$ is a vector of size $k+1$. Finding the optimal $y \in \mathbb{R}^k$ that minimizes $\|\beta_0 e1 - \bar{H}_k y\|_2$ is equivalent to solving the linear system $\bar{H}_k y = \beta_0 e1$.

Algorithm 3 presents the main process for the GMRES algorithm.

Algorithm 3 GMRES Algorithm [15]

1: Compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$, and $v_1 := \frac{r_0}{\beta}$
2: **for** k = 1, 2, ..., m **do**,
3: Compute $\omega_k := Av_k$
4: **for** $i = 1, 2, ..., k$ **do**,
5: $h_{ik} := (\omega_k, v_i)$
6: $\omega_k := \omega_k - h_{ik} v_i$
7: **end for**
8: $h_{k+1,k} = \|\omega_k\|_2$.
9: **if** $h_{k+1,k} = 0$ **then** set $m := k$ and goto line 13,
10: **end if**
11: $v_{k+1} = \frac{\omega_k}{h_{k+1,k}}$
12: **end for**
13: Define the $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ik}\}_{1 \le i \le m+1, 1 \le k \le m}$
14: Compute y_m, the minimizer of $\|\beta e_1 - \bar{H}_m y\|_2$, and $x_m = x_0 + V_m y_m$

At each iteration a new basis vector is computed and orthonomalized against all the previous vectors. Thus, the memory requirements and the number of floating point operations increase at each iteration. The GMRES method is known for its

superlinear convergence behavior, where the rate of convergence improves as the iterations proceed [26].

1.4 Preconditioners

Given a linear system $Ax = b$, where A is a matrix of size $n \times n$, preconditioning transforms the original linear system into a new system with the same solution by applying a preconditioner M. There are three types of preconditioning:

- Right preconditioner : $AM^{-1}y = b$, where $y = Mx$
- Left preconditioner : $M^{-1}Ax = M^{-1}b$
- Split preconditioning with $M = M_1M_2$: $M_1^{-1}AM_2^{-1}y = M_1^{-1}b$, where $y = M_2x$

A given preconditioner M is efficient if the preconditioned system has a faster rate of convergence than the original system, when solved using iterative methods. For that, M should be chosen such that the condition number of the preconditioned system is almost equal to 1. Moreover, the construction of the preconditioner M should not be expensive neither in terms of floating point operations nor in terms of communication. We note that the preconditioners could be operators on vectors so it is not necessary to compute the full matrices M and M^{-1} but their action on a vector. Thus, the preconditioner should be chosen such that its application to a vector is not expensive.

Finding a suitable preconditioner for a given sparse linear system, that satisfies the above conditions is not always an easy task. For systems obtained from the discretization of PDE's, it is possible to build preconditioners based on the geometry of the original problem. However, here, we will only discuss algebraic preconditioners that are defined based on the matrix A only.

The simplest algebraic preconditioners in terms of both construction and application are those based on the classical iterative methods like Jacobi, Gauss-Seidel and successive over relaxation (SOR). These preconditioners are based on splitting the initial matrix A as follows,

$$\begin{pmatrix} \ddots & & \\ & \ddots & -F \\ & D & \\ -E & & \ddots \\ & & & \ddots \end{pmatrix}$$

.

Here are some of those preconditioners:

- Jacobi preconditioner: $M = D$
- Forward Gauss Seidel preconditioner: $M = D - E$
- Backward Gauss Seidel preconditioner: $M = D - F$
- Successive over relaxation (SOR) preconditioner: $M = \frac{1}{\omega}D - E$.

A second type of preconditioners is based on an approximate factorization of the matrix A such as incomplete LU preconditioner and incomplete Cholesky precon-

ditioner. Incomplete LU preconditioners are based on incomplete LU factorizations where $A = LU + R$ and $M = LU$. In general, the complete LU factorization of a given sparse matrix A is a Gaussian elimination that results in a lower triangular factor L and an upper traingular factor U having more nonzero entries than A. In order to obtain sparse factors L and U and drop some entries, there are several incomplete LU factorizations. Based on different dropping strategies that satisfy some conditions such as the sparsity pattern or some drop tolerance, we have different ILU factorizations: zero fill-in referred to as $ILU(0)$, level of fill-in referred to as $ILU(k)$, threshold referred to as $ILUT$ and other variants.

$ILU(0)$ consists in taking the zero pattern of the preconditioner M to be precisely the zero pattern of the input matrix A. It means that $ILU(0)$ produces L and U factors that have the same sparsity pattern as the lower and upper triangular part of A. The $ILU(0)$ factorization of A is obtained by performing an LU factorization, where only the nonzero entries of A are modified. Details about $ILU(k)$ and $ILUT$ will be presented in Section 2.2.2. For a full description of the different ILU factorizations refer to [15, Sections 10.3 and 10.4].

A third type of preconditioners is based on domain decomposition of the unknowns. The subdomains can be overlapping, which is the case for restricted additive Schwarz preconditioner (RAS) or non-overlapping, which is the case for block Jacobi preconditioner (BJ). Then, the preconditioner is equal to the blocks of A restricted to the subdomains. Again, more details about these two preconditioners are presented in Section 2.2.2.

There are different other types of preconditioners that we do not address in this thesis such as algebraic multigrid preconditioners [27]. A detailed survey on preconditioning techniques is presented in [28].

For all the iterative methods presented in the previous sections, in particular the GMRES method, preconditioned versions are slightly different from the original methods. The matrix A is replaced by $M^{-1}A$, AM^{-1} or $M_1^{-1}AM_2^{-1}$ and b by $M^{-1}b$ or bM^{-1} depending on the type of the preconditioner used. For the Krylov subspace methods discussed in the previous section, the matrix A is multiplied by a vector. Thus, there is no need to explicitly multiply the matrices A and M^{-1}, we should only apply the preconditioner to a vector. We note that some variants of iterative methods allow the preconditioner to vary from one step to another. Such methods are called flexible. Among these methods, we cite the flexible GMRES (FGMRES) method that we will be using throughout this thesis.

1.5 Parallel algebraic recursive multilevel solver

In this section, we first present the algebraic recursive multilevel solver (ARMS) [12], then its parallel version (pARMS) [13], the two main solvers that we consider throughout this PhD work.

1.5.1 Algebraic recursive multilevel solver (ARMS)

ARMS is an algebraic recursive multilevel solver for general sparse linear systems where preconditioning is based on a multilevel partial elimination approach. Gen-

erally, to precondition a linear system in ARMS, we use a method named block incomplete LU factorization, which consists of an approximate Gaussian elimination process based on separating the original unknowns into a "coarse" and a "fine" set. The idea of independent or "group independent" sets is exploited to define this separation. The preconditioner ARMS is based on a block incomplete LU factorization with different dropping strategies. Block independent set orderings permute the original linear system $Ax = b$ into a 2×2 block structure as follows,

$$\begin{pmatrix} B_l & F_l \\ E_l & C_l \end{pmatrix} \begin{pmatrix} u_l \\ y_l \end{pmatrix} = \begin{pmatrix} f_l \\ g_l \end{pmatrix}, \tag{1.1}$$

where the submatrix B comes from a group-independent set reordering (see, e.g., [15]), thereby generating a block-diagonal matrix B_l.

At the first level, the reordered coefficient matrix is factorized approximately as follows,

$$\begin{pmatrix} B_l & F_l \\ E_l & C_l \end{pmatrix} \approx \begin{pmatrix} L_l & 0 \\ E_l U_l^{-1} & I \end{pmatrix} \times \begin{pmatrix} U_l & L_l^{-1} F_l \\ 0 & A_{l+1} \end{pmatrix} \tag{1.2}$$

where l is the level ranking, L_l and U_l forms the ILU factors of B_l, and A_{l+1} is an approximation to the Schur complement with respect to C_l,

$$A_{l+1} \approx C_l - (E_l U_l^{-1})(L_l^{-1} F_l), \tag{1.3}$$

where the system A_{l+1} is computed by eliminating the unknowns u_l associated with the block B_l. Then we obtain the following system,

$$A_{l+1} \times y_l = b_{l+1}, \tag{1.4}$$

where $b_{l+1} = g_l - E_l U_l^{-1} L_l^{-1} F_l$ is computed first, then the system is solved by a forward solver and a backward solver. We note that the unknown vector is $y_l = \begin{pmatrix} u_{l+1} \\ y_{l+1} \end{pmatrix}$.

The coefficient matrix for the resulting "reduced system" is the approximate Schur complement A_{l+1} [29]. A recursion can now be exploited, such that dropping is applied to A_{l+1} to limit the fill-in followed by the reordering of the resulting reduced system into the form (1.1) using the group-independent set ordering again. This process is repeated for several recursion levels until the approximate Schur complement system is small enough or until a maximum number of recursion levels is reached. Then, the last Schur complement may be solved by a direct or an iterative solver. Note that the sparsification of the Schur complement may be undertaken at each recursion level to keep down the preconditioning costs. Algorithm 4 describes the ARMS solver as detailed in [12].

Algorithm 4 ARMS-solve(A_l, b_l) – Recursive Multi-Level Solution

1: Solve $L_l \; f'_l = f_l$
2: Descend, i.e., compute $b_{l+1} := g_l - E_l U_l^{-1} \; f'_l$
3: **if** $l = last\_lev$ **then**
4: Solve $A_{l+1} y_l = b_{l+1}$ using GMRES + $ILUT$ factors
5: **else**
6: y_l = ARMS-solve(A_{l+1}, b_{l+1})
7: **end if**
8: Ascend, i.e., compute $f''_l = f'_l - L_l^{-1} F_l \; y_l$
9: Back-Substitute $u_l = U_l^{-1} f''_l$

1.5.2 Parallel implementation of ARMS (pARMS)

The parallel Algebraic Recursive Multilevel Solver (pARMS) is a memory distributed parallel solver. The parallel implementation of ARMS is based on the domain decomposition method [30]. In pARMS, each process reads the whole matrix. The matrix graph is then partitioned using distributed site expansion (DSE) [31]. A simple partitioning routine splits the vertices into two part, then splits the one with more vertices into another two part and so on. After applying the partitioning method, the master process assigns a local submatrix to each of the slave processes. Once these submatrices are received, the global system is preconditioned using a global preconditioner such as the restrictive additive Schwarz preconditioner (RAS), the Schur complement based preconditioner (SCHUR) and the block-Jacobi preconditioner (BJ). Then the global system is solved using the flexible generalized minimal residual (FGMRES) method [15].

Note that, at each FGMRES iteration, each process exchanges interface variables with neighboring processes. Furthermore, in FGMRES solver, each process has the possibility to choose a local preconditioner such as $ILU(0)$, $ILU(K)$, $ILUT$ and ARMS. More details about the pARMS solver as well as the local preconditioners will be presented in Chapter 2.

1.6 Linear algebra libraries and sparse matrix storage

Since it is important to "get it right" and "get it right fast", it is always suggested to use numerical libraries of proven code to handle classical linear algebra operations. Here we present the libraries that we used in this work: LAPACK [5], MAGMA [32] and SuperLU [11].

1.6.1 LAPACK

The linear algebra package (LAPACK) [5] is a successor to both LINPACK [4] and EISPACK [33], and achieves better performance. LAPACK provides solvers for systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. To implement these solvers, it also provides the basic associated computation such as matrix factorizations (LU, QR, Cholesky etc)

or the estimation of condition numbers. To perform all the matrix, vector and scalar computations, LAPACK uses the Basic Linear Algebra Subprograms (BLAS) routines [3]. Therefore the performance of the LAPACK library depends on the implementation of the BLAS used [34]. Most of the numerical linear algebra libraries developed afterwards are based on BLAS and LAPACK, offering different implementations of the same routines and operations.

1.6.2 MAGMA

Matrix algebra on GPU and multicore architectures (MAGMA) [32] is an accelerator-focused linear algebra library developed at the University of Tennessee. It provides back-ends for NVIDIA GPUs, Intel's Xeon Phi manycore accelerators (MIC) and any OpenCL-compatible system such as AMD GPUs. Similarly to LAPACK, MAGMA [35,36] is being build as a community effort, incorporating the newest developments in hybrid algorithms and scheduling techniques. It aims at redesigning the dense linear algebra algorithms in LAPACK to fully exploit the power of current heterogeneous systems of multi/manycore CPUs and accelerators. MAGMA provides also a large variety of solvers, preconditioners, and eigensolvers for sparse linear systems. More details about the MAGMA sparse library will be given in Chapter 3, where we used some MAGMA routines to improve the performance of the pARMS solver.

1.6.3 SuperLU

SuperLU [11,37] is a general purpose library for the direct solution of large, sparse and nonsymmetric systems of linear equations. The library uses MPI, OpenMP and CUDA to support various forms of parallelism. SuperLU routines perform an LU decomposition and triangular system solves through forward and backward substitutions. The matrix columns may be preordered (before factorization) either through library or user supplied routines. This preordering for sparsity is completely separate from the factorization. Working precision iterative refinement subroutines are provided for improved backward stability. Routines are also provided to equilibrate the system, estimate the condition number, calculate the relative backward error and estimate error bounds for the refined solutions. We mainly used routines from SuperLU in Chapter 2.

1.6.4 Sparse matrix storage formats

According to existing research works [38,39], the sparse matrix storage formats can significantly affect the performance of iterative methods for solving sparse linear systems. All formats aim at decreasing the memory footprint of the matrices by reducing the number of explicitly stored zeros. This strategy is taken to extremes in the compressed sparse row storage (CSR) [40], where only nonzeros are stored. However, the CSR storage format does not necessarily provide a good performance on streaming architectures like GPUs. This has motivated a number of new formats, and in particular, the one standing out is ELLPACK [41], where padding of the different rows with zeros is applied to get a uniform row-length suitable for

coalesced memory accesses of the matrix and instruction parallelism. However, the ELLPACK (or ELL) format incurs a storage overhead, which is determined by the maximum number of nonzeros aggregated in one row. Depending on the particular problem, the overheads in using ELLPACK may result in poor performance, despite the coalesced memory access that is highly favorable for streaming processors. One workaround is given by the sliced ELLPACK format [42]. It splits the original matrix into row blocks, each stored using the ELL format. This format is usually referred to as SELL, or SELLP in case of the matrix with additional zeros to match the GPU thread block dimensions [43]. We note that the MAGMA library supports all the three formats: CSR, ELL, and SELL/SELLP. However, in this thesis we only used the CSR format since it is the format supported by the ARMS and pARMS solvers. Figure 1.1 illustrates the CSR format for a sparse matrix A with 9 rows, 9 columns, and 20 non-zero entries.

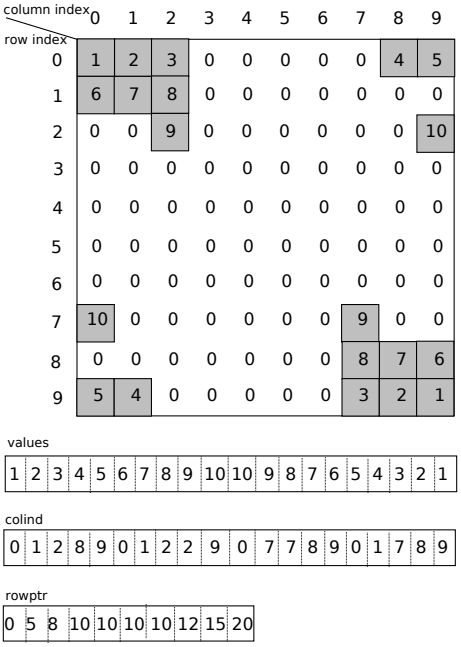


Figure 1.1: CSR Format

1.7 Parallel architectures and programming models in HPC

In the domain of modern high performance computing (HPC), parallelism is a critical issue. Different types of parallel architectures exist. In this Section, we present the architectural components and programming paradigms that we used during this PhD work.

1.7.1 Distributed memory systems

Many of the current supercomputers are based on distributed memory systems. A distributed memory system consists of multiple independent nodes connected by a given network. Each node has its own private memory and autonomous computational capabilities. The nodes connected together form a cluster. The nodes exchange data by passing messages between processors using the network. Each node can be composed of multiple CPUs and may contain accelerators. Figure 1.2 [1] illustrates an example of a distributed memory cluster.

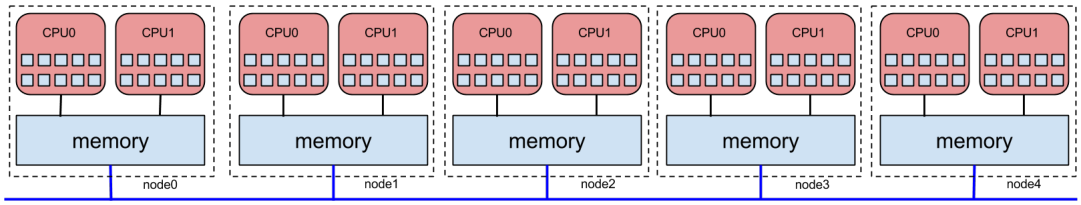


Figure 1.2: Distributed Memory Cluster

1.7.2 Multi-core processors

A multi-core processor [44] is a computing component with two or more independent processing units. Multi-cores compute instructions simultaneously, so they can accelerate the whole computation time. The processor manufacturers faced a technology limitation to their traditional approach of boosting clock speeds and increasing throughput due to power dissipation problems. This limitation leads to the development of multi-core architectures and hyper-threading as a solution which is a turning point in terms of software development. While constructors might increase the frequency in processors freely, the performance of a program would naturally increase at each new generation of processor. This went on until the end of the Intel Pentium 4 processors in 2006 with frequencies around 3.6 GHz.

The common concept behind a multi-core processor is to let each core have an L1 independent cache memory and put a shared L2 cache on the die which is the interface to the main memory. Figure 1.3 illustrates a multi-core system with two levels of cache. Modern CPUs generally have two levels of cache in each core and a shared L3 cache. Having multiple CPU cores on the same die allows for a highly efficient cache coherent system at a much higher clock rate. Parallel programming models take advantage of multi-cores. For instance, the application programming interface Open multi-processing (OpenMP) [45] can be used on multi-core platforms.

1.7.3 Graphics Processing Units (GPUs)

Applications are becoming more demanding in workload and complexity as architectures become more powerful. CPUs have shown their limits in terms of computational power as they tend to be more general. This technological gap has led to

[1] https://wiki.hpc.tulane.edu/trac/wiki/cypress /Programming/Mpi

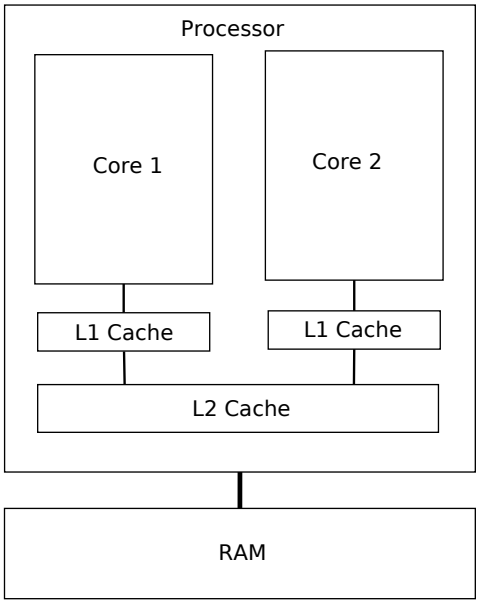


Figure 1.3: Different levels of cache in multi-core processors

the development of new architectures that are more specialized and can therefore provide more computational power for specific applications. The graphics processing units (GPUs) are an example of these new type of architectures that has been emerging during the last decade.

The GPUs are specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 1.4.

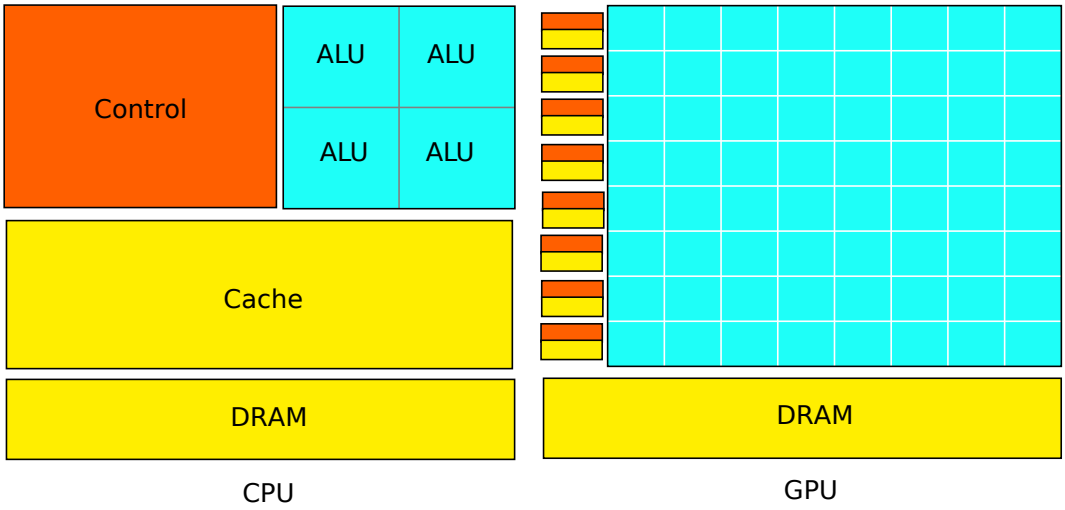


Figure 1.4: The GPU devotes more transistors to data processing

Before the 2000's, GPUs were only used to perform or accelerate graphical computation for 2D and 3D pictures. In 2001, Larsen developed one of the first example of non-graphical computation on a GPU with a matrix-matrix product [46] using the 8-bits integer texture maps. In 2003, the introduction of 32-bits floating-point values allowed a real progress in matrix computations on GPUs [47]. In 2007, NVIDIA released the Compute Unified Device Architecture (CUDA) programming platform [48] providing a virtual instruction set, allowing the development of general purpose applications.

GPU has become a common occurrence in HPC and is often used in supercomputer architectures. Thus, out of the ten most powerful supercomputers in the latest TOP500 [49] ranking (June 2017), two use GPU accelerators (the second and the third machine).

A graphic card has a certain number of Streaming Multiprocessors (SMX). For instance, K20Xm has 14, K40Xm has 15 and P100 has 56.

Figure 1.5 shows a block diagram of NVIDIA's Pascal GPU and Figure 1.6 shows a block diagram of the Pascal GPU's streaming multiprocessor [2].

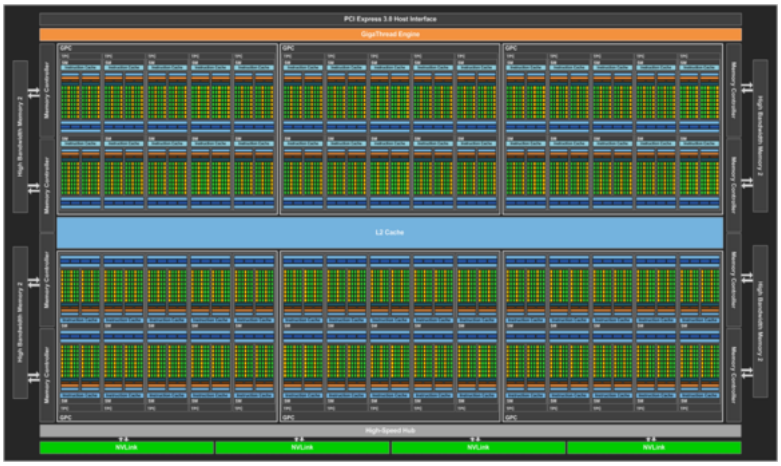


Figure 1.5: A block diagram of Nvidia's Pascal GPU

We note that GPUs do not operate on the computer main memory. Often a GPU is connected to its own off-chip memory (the GPU global memory) which is used to store data. The size of this graphics memory varies but it is currently about 3 to 12 Gigabytes for high-end GPUs. Before the GPU starts to work on a given data, that data needs first to be moved to the GPU main memory. The speed of that operation depends on the connection between the main memory and the graphics board via the PCI Express bus.

When used in HPC, GPUs can have over an order of magnitude higher memory bandwidth and higher computation power (in terms of GFflop/s) than CPUs. However, programming efficiently on GPU-based architectures remains a critical challenge. In this thesis, we showed some solutions to efficiently improve the performance of the iterative pARMS solver using GPUs as accelerators.

[2] Figures 1.5, 1.6 are from: https://devblogs.nvidia.com/parallelforall/inside-pascal/

Figure 1.6: A block diagram of the Pascal GPU's streaming multiprocessors

1.7.4 Programming models for parallel architectures

In this thesis, we are mainly interested in the pARMS solver which is a distributed memory sparse iterative solver based on the message passing interface (MPI) library. We improve the performance of pARMS using MAGMA routines written with CUDA, an application programming interface model created by NVIDIA. In the following subsections, we give some details about these two programming models, MPI and CUDA.

Message Passing Interface (MPI)

Message passing interface (MPI) [50] is a standardized and portable communication protocol based on the agreements of the MPI Forum. MPI is a library for distributed memory environments. The objective of the MPI implementation is to establish a portable, efficient and flexible standard for distributed programs. The standard includes point-to-point message-passing, collective communications, group communicator concepts and process topologies etc.

Even if the MPI programming interface has been standardized, practical library implementations will differ in which version and characteristics of the standard they support. The way of MPI programs are compiled and executed on different platforms may be different. There are two main MPI implementations: MPICH and Open MPI [51].

Compute Unified Device Architecture (CUDA)

When it comes to hardware accelerators, programming a parallel chip that is not a CPU can be difficult and challenging. This applies to any kind of accelerator, such as GPUs or FPGAs. Compute Unified Device Architecture (CUDA) [52] is an

application programming interface for NVIDIA graphics processing units. It can be described as an extension to high level languages (C, C++). Moreover, CUDA allows users to adapt complex scientific applications to GPU architectures by rewriting codes with respect to the single program multiple data (SPMD) programming model. One major limitation of CUDA is that it only targets NVIDIA GPUs while OpenCL is heterogeneous and can be ran on various GPUs, CPUs and other processors.

Figure 1.7[3] presents an example of CUDA processing flow. This process consists of four steps. First, data is copied from the CPU main memory to the GPU memory. Then the CPU instructs the process to the GPU by calling the GPU kernel. At the third step, each thread executes the program in parallel. Finally, the result is copied from the GPU memory to the main memory.

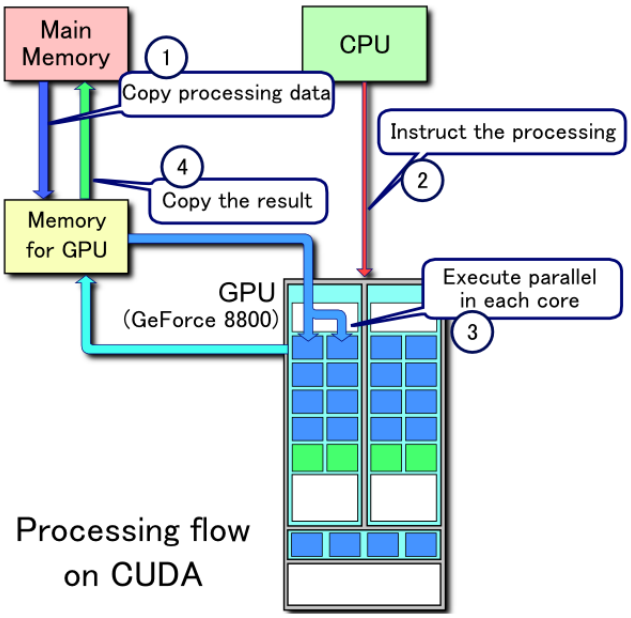


Figure 1.7: CUDA processing flow

1.8 Conclusion

In this chapter, we discussed the different methods for solving general linear systems of equations including both direct and iterative methods. For the direct method, we mainly detailed the LU factorization. For the iterative methods, we focused on the Krylov subspace methods, in particular the generalized minimal residual (GMRES) method. We also discussed the main preconditioning techniques used in iterative solvers.

Then, we briefly introduced the algebraic recursive multilevel solver (ARMS), which is a sparse iterative solver and its parallel version pARMS. Our primary objective throughout this thesis is to improve the performance of these two solvers.

[3] https://en.wikipedia.org/wiki/CUDA

We also described the numerical libraries used in the framework of this thesis depending on the targeted architectures. This included LAPACK, MAGMA and SuperLU.

Finally, we presented the parallel architectures relative to the machines used to run the experiments performed in this PhD work, as well as the main programming models. We mainly gave an overview of the challenges faced by the programmer because of the different types of parallelism and programming paradigms.

Chapter 2

Using randomization in the pARMS solver

2.1 Introduction to randomized algorithms

A randomized algorithm is an algorithm that makes a random decision rather than a deterministic decision, and its behavior can vary even with a fixed input. The advantage of a random algorithm is that no input can reliably produce worst-case results since it produces a different solution for each execution of the algorithm. This type of algorithm is often used in situations where there is no available deterministic or fast enough algorithm. Randomized algorithms are widely used in numerical linear algebra since they can outperform deterministic methods while providing accurate results [53]. Below are some examples of randomized algorithms that have been implemented in recent years.

The first example is related to the random sampling. Computationally expensive or NP hard problems can sometimes be solved by using randomization and sampling techniques which help us to obtain provably accurate algorithms. For instance, Drineas et al. [54] focus on approximating a matrix multiplication $A \times B$ based on an appropriate random sampling of the columns of A and of the rows of B, each of them scaled properly. For this randomized matrix multiplication, the authors provide an estimate of the error (in Frobenius norm) and the resulting algorithm can be implemented without storing the matrices in RAM.

Another example concerns the approximation of the Gram product AA^T by a small number of outer products of columns of A (see [55]). The authors also present probabilistic bounds for the two-norm relative error due to randomization for the Monte Carlo matrix multiplication algorithm [54] that samples outer products. The bounds depend on the rank of the matrix but not on the matrix dimensions.

We can also find examples of randomized algorithms in [53], with applications to low-rank matrix approximation, or the solution of linear least-squares problems, and can possibly work on Terabyte (TB) data. Regarding the solution of linear least-squares, we mention the randomized approach given in [56] which provides an implementation that is faster than LAPACK. Finally, other examples of randomized algorithms can be found in [57] which describes fast sparse matrix algorithms for over-constrained least-squares regression, low-rank approximation, approximation of leverage scores, and l_p-regression. In general these algorithms run in $\mathcal{O}(nnz(A))$ operations.

In this chapter, we propose to use a randomized algorithm based on Random Butterfly Transformations (RBT) in the Algebraic Recursive Multilevel Solver (ARMS) to improve the preconditioning phase in the iterative solution of sparse linear systems. We propose more specifically to integrate the RBT technique in the parallel distributed version of ARMS (pARMS). The RBT approach was initially described by Parker [58] and then revisited in [20] for general dense systems and [59] for symmetric indefinite systems. It has also been applied recently to a sparse direct solver in a preliminary paper [60]. The main interest of this technique is to avoid the data communication cost due to partial pivoting (swap of rows) in the Gaussian elimination. Our objective is here to apply this technique to solve more efficiently the last Schur complement system in the recursive multilevel process of pARMS. We will present experimental results on some matrices from the Davis' collection that show an improvement in the convergence and accuracy of the results when compared

with existing implementations of the pARMS preconditioner.

This chapter is organized as follows. In Section 2.2, we present the parallel distributed implementation of ARMS. In Section 2.3, we describe the Random Butterfly Transformations as used for the solution of dense linear systems. In Section 2.4, we explain how RBT can be used for solving the last Schur complement system in the recursive process of pARMS. In Section 2.5, we present some experiments where we use "sparse" RBT to solve the last Schur complement system considered as sparse. Section 2.6 is a brief conclusion of this chapter.

2.2 Parallel implementation of ARMS

2.2.1 Data distribution

The parallel Algebraic Recursive Multilevel Solver (pARMS) is a memory-distributed parallel solver. Each MPI process reads the whole matrix at the same time, then uses the Distributed Site Expansion (DSE) method [31] to distribute the initial matrix. Each processor uses a **local preconditioner** to factorize the local matrix, or a local solver to solve the local system. To solve the interface variables, adjacent processes need to exchange some data. Then, the preconditioned system is solved using the Flexible Generalized Minimal Residual method (FGMRES) [15].

Figure 2.1 outlines distributed linear system solution using pARMS [13]. First, the initial matrix A is distributed among the processors, using a graph partitioning method. In Figure 2.1, each column of blocks depicts one processor, hence there are five processors shown. Second, each processor solves its part of the system in parallel to construct its portion of the global preconditioner. Then FGMRES solves the preconditioned system with a given accuracy.

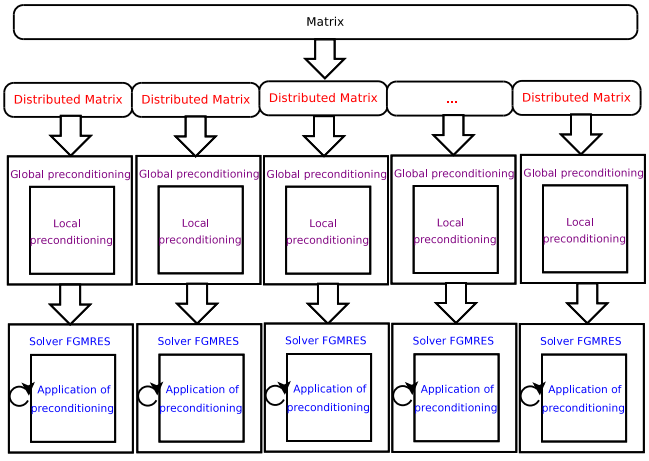


Figure 2.1: Sketch of the distributed linear system solution using pARMS on five processors.

When considering the parallel implementation, it is important to specify how the matrix is distributed and handled in parallel. In particular, our pARMS implemen-

tation partitions the whole matrix on a single processor using the DSE technique, which is rather simple yet effective in constructing well-balanced subdomains with small interfaces [31]. Although partitioning the entire matrix by a single processor lacks scalability, we note here that this is done by the driver routine, which may be adapted to an application matrix size and format at hand. Given a distributed matrix, Figure 2.2 shows the per-subdomain division of variables into internal, interdomain interface, and external (residing on the neighboring processors) sets.

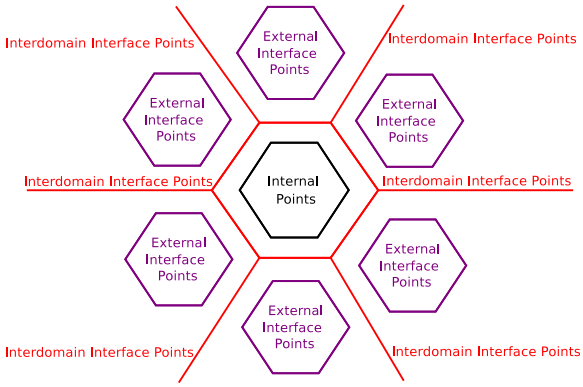


Figure 2.2: Per-subdomain view of equation variables-points.

2.2.2 Preconditioners in the pARMS environment

In pARMS, the "local" preconditioners mentioned in Section 2.2 can be: three local preconditioners based on Incomplete LU (ILU) factorization, and one multilevel preconditioner called Algebraic Recursive Multilevel Solver (ARMS).

An ILU preconditioner is constructed by performing an approximate Gaussian Elimination (GE) [61] on a sparse matrix A and dropping certain nonzero entries of the factorization according to different dropping strategies. A dropping strategy that relies on levels of the matrix fill-in results in a factorization called $ILU(K)$.

The preconditioner $ILU(0)$ is obtained by performing the LU factorization of A and dropping all fill-in elements generated during the process. Conversely, if the nonzeros are dropped according to their numerical value magnitudes, then the resulting factorization is called ILU with the threshold or—if combined with the dropping strategy based on the number of remaining nonzero—with dual threshold ($ILUT$) and is performed as follows. In the algorithm $ILUT(k, \tau)$, there are two important rules. (1) If an element is less than relative tolerance τ_i ($\tau \times$ the norm of the ith row), it is dropped. (2) Keep only the k largest elements in the L and U parts of the row along with the diagonal element.

The other preconditioner available is the ARMS preconditioner [12], which is based on a block incomplete LU factorization with different dropping strategies. This block factorization consists of an approximate GE process separating the unknowns into two sets; and an idea of independent or "group independent" set is exploited to define the separation. Hence, the original linear system $Ax = b$ is

permuted into the form:

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \times \begin{pmatrix} u \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}, \tag{2.1}$$

where the submatrix B corresponds to group-independent set reorderings, thereby generating a block-diagonal matrix B. Thus, it is convenient to eliminate the u variable to obtain a system with only y variable. The coefficient matrix for the resulting "reduced system" is the Schur complement $S = C - EB^{-1}F$ [29]. A recursion can now be exploited, such that dropping is applied to S to limit the fill-ins followed by the reordering of the resulting reduced system into the form (2.1) by using the group-independent set reordering again. This process is repeated for several levels of recursion until the Schur complement system is small enough or until a maximum number of recursion levels is reached. Then, the last Schur complement may be solved by a direct or an iterative solver. Note that the sparsification of the Schur complement may be undertaken at each level of recursion, to keep down the preconditioning costs.

Figure 2.3 depicts a representation using Matlab of the recursive process in ARMS where an initial matrix A (top left) is first reordered to give the matrix on the top right. At the first level of decomposition, we obtain the matrices B, F, E and C, as denoted in Equation (2.1). Note that the number of nonzero entries of each matrix of this figure is indicated is the x-legend. Then the matrix on the bottom left of Figure 2.3 corresponds to the level-1 Schur complement of C (after reordering) and the matrix on bottom right corresponds to the last Schur complement (if we have 2 levels) which is possibly dense enough and can be factored using a dense LU factorization. Our objective in this chapter is to apply ARMS, enhanced with Random Butterfly Transformations (RBT) to alleviate the extra work associated with pivoting that may be required in the Schur complement matrix S due to its poor conditioning.

We outline now three global preconditioners types available in pARMS: Block-Jacobi preconditioner (BJ), Schur complement preconditioner (SCHUR), and Schur complement based Restrictive Additive Schwartz preconditioner (SchurRAS). BJ is the simplest global preconditioner because it does not take into account the interface information between neighboring subdomains [30]. SCHUR relates equations associated with the local and interdomain interface points [62]. SchurRAS is constructed from the local ARMS preconditioners in each subdomain using an overlap similar to a standard RAS preconditioner [63] and acting on the Schur complement system as shown in [64]. Specifically, for each of the three preconditioner types, the following algorithms may be implemented in each subdomain.

Block-Jacobi preconditioner:
1. Update local residual: $r_i = (b - Ax)_i$,
2. Solve: $A_i\delta_i = r_i$,
3. Update local solution: $x_i = x_i + \delta_i$.

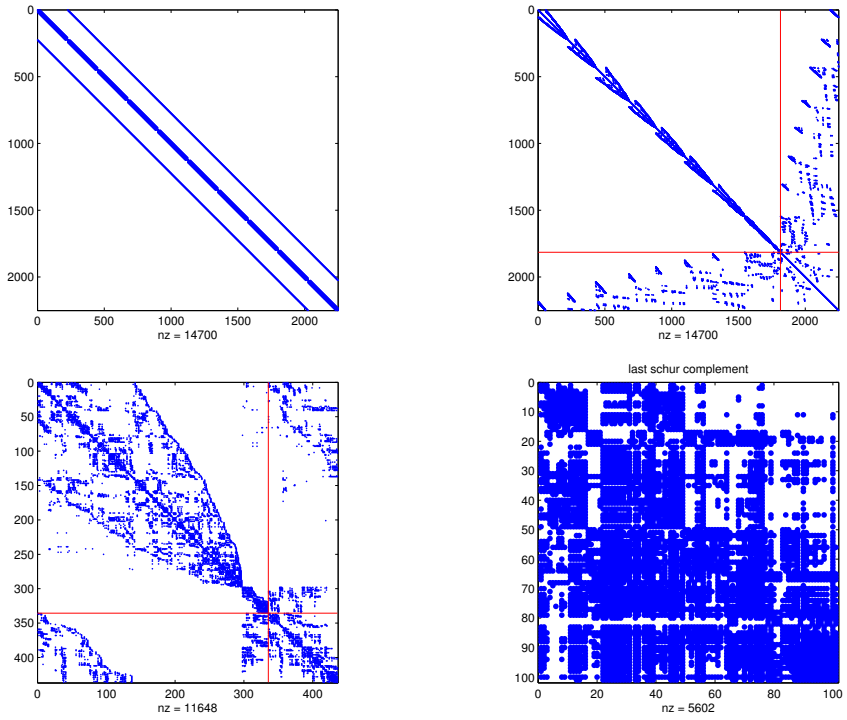


Figure 2.3: Application of the ARMS preconditioner : Initial matrix A (top left), re-ordered matrix (top right), level-1 Schur complement (bottom left), level-2 Schur complement (bottom right)

SCHUR preconditioner:
1. From Equation (2.1) compute: $g_i' = g_i - E_i B_i^{-1} f_i$,
2. Solve: $S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i'$, where $S_i = C_i - E_i B_i^{-1} F_i$ and N_i is a set of neighboring subdomains,
3. Back substitute: u_i with $B_i u_i = f_i - E_i y_i$.

SchurRAS preconditioner:
1. Compute local right-hand side g_i'.
2. Solve local Schur complement system extended with rows for all external variables $y_{i,ext}$.
3. Back substitute: u_i with $B_i u_i = f_i - E_i y_i$.

Note that the local solves in step 2 of BJ, SCHUR, and SchurRAS may be accomplished using the incomplete LU or ARMS procedures, described above.

2.3 Random Butterfly Transformations (RBT)

2.3.1 Context

With the evolution of recent computer architectures, the growing gap between communication and computation efficiency makes communication very expensive (at a cost of one communication, we can generally perform thousands of arithmetical operations). This has required the rethinking of most of numerical libraries in order to take advantage of current parallel architectures which are commonly based on multicore processors [65], possibly with accelerators [36], such as Graphics Processing Units (GPU) or Intel Xeon Phi.

When solving square linear systems $Ax = b$ using Gaussian elimination (e.g., via LU factorization), we commonly use partial pivoting to avoid having zero or too-small numbers on the diagonal. This technique is implemented in current linear algebra libraries and ensures stability [66]. However, partial pivoting requires communication (search for pivots, swapping of rows). For example, on a hybrid CPU/GPU system, the LU algorithm in the MAGMA library [36] spends more than 20% of the factorization time in pivoting for a random dense matrix of size $10,000 \times 10,000$ [20].

As an alternative to pivoting, an approach based on randomization called Random Butterfly Transformation (RBT) [58] was recently revisited. Following the RBT method, A is transformed into a matrix that would be sufficiently random to avoid pivoting (with a probability close to 1). Then the transformed matrix is factorized with Gaussian elimination with no pivoting, which can significantly reduce the amount of communication [67]. We can obtain satisfying accuracy with an additional computational cost of $\mathcal{O}(n^2)$ operations, which is negligible compared to the cost of the factorization ($\mathcal{O}(n^3)$ operations). This method has been successfully applied to dense linear systems for either general [20] or symmetric indefinite [59,68] systems, in the context of direct methods based on matrix factorizations.

2.3.2 Preliminary definitions

In this section, we present the main notions used in the definition of a Random Butterfly Transformation.

A **butterfly matrix** is an $n \times n$ matrix of the form:

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix} \tag{2.2}$$

where $n \geq 2$ and R_0 and R_1 are random diagonal and nonsingular $n/2 \times n/2$ matrices.

A **recursive butterfly** $n \times n$ matrix of depth d is a product of the form [20]:

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & B_{2^{d-1}}^{n/2^{d-1}} \end{pmatrix} \times \cdots \times \begin{pmatrix} B_1^{<n/4>} & 0 & 0 & 0 \\ 0 & B_2^{<n/4>} & 0 & 0 \\ 0 & 0 & B_3^{<n/4>} & \\ 0 & 0 & 0 & B_4^{<n/4>} \end{pmatrix} \times \begin{pmatrix} B_1^{<n/2>} & 0 \\ 0 & B_2^{<n/2>} \end{pmatrix} \times B_1^n, \quad (2.3)$$

where $B_i^{<n/2^{k-1}>}$ are butterflies of size $n/2^{k-1} \times n/2^{k-1}$, $k = 2, ..., d$ and $B^{<n>}$ is butterfly of size $n \times n$.

Note that this definition requires that n is a multiple of 2^d which can be always obtained by "augmenting" the matrix A with additional 1's on the diagonal. Note also that it differs from the definition of a recursive butterfly given in [58] where $d = log_2 n$ and the first term of $W^{<n,d>}$ is a diagonal matrix of size n (and thus we have $log_2 n + 1$ terms). Here, we give an example of a recursive butterfly matrix $W^{<4,2>}$ of size $n \times n$ and depth $d = 2$. It is obtained by the product of two butterfly matrices $\binom{B_1^{<2>}\ 0}{0\ B_2^{<2>}}$ of size $n \times n$ and depth $d = 1$ and $B^{<4>}$ of size $n \times n$ and depth $d = 2$.

$$W^{<4,2>} = \begin{pmatrix} B_1^{<2>} & 0 \\ 0 & B_2^{<2>} \end{pmatrix} \times B^{<4>} =$$

$$\frac{1}{2} \begin{pmatrix} r_1^{<2>} & r_2^{<2>} & 0 & 0 \\ r_1^{<2>} & -r_2^{<2>} & 0 & 0 \\ 0 & 0 & r_3^{<2>} & r_4^{<2>} \\ 0 & 0 & r_3^{<2>} & -r_4^{<2>} \end{pmatrix} \times \begin{pmatrix} r_1^{<4>} & 0 & r_3^{<4>} & 0 \\ 0 & r_2^{<4>} & 0 & r_4^{<4>} \\ r_1^{<4>} & 0 & -r_3^{<4>} & 0 \\ 0 & r_2^{<4>} & 0 & -r_4^{<4>} \end{pmatrix}$$

$$= \frac{1}{2} \begin{pmatrix} r_1^{<2>} r_1^{<4>} & r_2^{<2>} r_2^{<4>} & r_1^{<2>} r_3^{<4>} & r_2^{<2>} r_4^{<4>} \\ r_1^{<2>} r_1^{<4>} & -r_2^{<2>} r_2^{<4>} & r_1^{<2>} r_3^{<4>} & -r_2^{<2>} r_4^{<4>} \\ r_3^{<2>} r_1^{<4>} & r_4^{<2>} r_2^{<4>} & -r_3^{<2>} r_3^{<4>} & -r_4^{<2>} r_4^{<4>} \\ r_3^{<2>} r_1^{<4>} & -r_4^{<2>} r_2^{<4>} & -r_3^{<2>} r_3^{<4>} & r_4^{<2>} r_4^{<4>} \end{pmatrix} \quad (2.4)$$

A **Random Butterfly Transformation** (RBT) of depth d of an $n \times n$ matrix A corresponds to the calculation of the product $A_r = U^T A V$. It consists of a "multiplicative preconditioning" $U^T A V$ where the matrices U and V are recursive butterfly matrices.

The random values in U and V are $e^{r/10}$ where r is randomly chosen in $[-\frac{1}{2}, \frac{1}{2}]$ from a uniform distribution. This guarantees a small condition number for U and V [20].

2.3.3 Solving linear systems using RBT

We solve the general linear system $Ax = b$ as follows:

1. Compute the randomized matrix $A_r = U^T AV$, with U and V recursive butterfly matrices
2. Factorize A_r with Gaussian Elimination with No Pivoting (GENP)
3. Solve $A_r y = U^T b$
4. Compute the solution $x = Vy$

We summarize in Algorithm 5 the procedure to solve $Ax = b$, where A is a general dense matrix, using Random Butterfly Transformation with d recursions and the LU factorization with no pivoting.

Algorithm 5 Dense Random Butterfly Transformation Algorithm (d recursions)

Transform the original matrix to a dense matrix whose size is multiple of 2^d (with possibly additional 1's on the diagonal)
Generate recursive butterfly matrices U and V
Perform randomization to update the matrix A and obtain the matrix $A_r = U^T AV$
Factorize the randomized matrix with GENP
Compute $U^T b$ to solve $A_r y = U^T b$, then $x = Vy$
Restore original size of vector

2.3.4 Main results

In the original work by Parker [58], $d = \log_2 n$ and it is proved that, given two recursive butterfly matrices U and V, the matrix $U^T AV$ can be factored into LU without pivoting with probability 1 in exact arithmetic, or with probability $1 - \mathcal{O}(2^{-t})$ using t-bit floating point numbers.

RBT was extensively studied for dense matrices. In particular it has been shown in [20] that:

- In practice, $d = 1$ or $d = 2$ is enough to obtain a satisfactory accuracy (in most cases a few steps of iterative refinement [66, p. 232] can recover the digits that have been lost).
- Each recursive butterfly matrix can be stored compactly into an $n \times d$ matrix.
- The RBT transformation $U^T AV$ is cheap to compute ($\mathcal{O}(dn^2)$ operations) and can be implemented very efficiently on current parallel architectures (multicore processors, GPUs).
- The condition number of the matrix A is almost unchanged by the application of RBT.

Implementations of RBT are proposed for multicore architectures in the dense linear algebra library PLASMA [7] and for hybrid CPU/GPU or CPU/Intel Xeon Phi architectures [69] in the MAGMA [36] library.

2.4 Integration of dense RBT in pARMS

2.4.1 Method

We describe in this section how Random Butterfly Transformations can be integrated into pARMS. Our goal is to find the last level of preconditioning and then replace the original LU factorization by the RBT pre-processing. Note that RBT usually concerns dense linear systems, while ARMS addresses sparse matrices. So we have to convert the last Schur complement which is a sparse matrix into a dense format, before using RBT. Then after randomizing the last Schur complement A with recursive butterfly matrices U and V, the dense matrix is factorized using a LAPACK-like [5] routine that performs Gaussian elimination without pivoting, followed by two triangular solves. The pARMS solver manages the parallel part by using global preconditioning with MPI instructions, while the local part of the code, more precisely the local preconditioning phase does not use MPI instructions. In particular, the application of RBT and the no-pivoting factorization that follows are sequential operations at the node/core level, that could be achieved by LAPACK-like routines. The parallelism is entirely managed by pARMS. The local preconditioning can be based on $ilu0$, $iluk$, $ilut$ or $arms$. The essential part resides in the last Schur complement, where we implemented RBT and the preconditioned matrix is then used in FGMRES in order to solve the linear system. In the remainder of this manuscript, the resulting local preconditioner will be called $arms\_rbt$. In our implementation, we will consider a value of $d = 1$ for the number of recursions in the implementation of RBT and we will not use iterative refinement.

2.4.2 Numerical experiments

This section presents the results obtained by integrating RBT into the pARMS solver. The experiments have been carried out using one node (2 twelve-core AMD MagnyCours Opteron 6172 processors running at 2.10GHz) of the Hopper machine located at NERSC[1].

In these experiments, we used matrices from the Davis' collection [70, 71] to test the performance of different preconditioners. The first matrix (`Sherman5`) is a real non-symmetric matrix of size $3,312$ (nnz = 20,793). `Sherman5` arises from a three dimensional simulation model on a $n_x \times n_y \times n_z$ grid using a seven-point finite-difference approximation with n_c equations and unknowns per grid block, where n_x is 16, n_y is 23, n_z is 3, n_c is 3. The second matrix (`Raefsky3`) is a real non-symmetric matrix of size $21,200$ (nnz =1,488,768), which arises from a fluid structure interaction turbulence problem. The third matrix (`Cant`) is a real symmetric matrix that comes from a 2D/3D FEM problem, of size 62,451 (nnz = 2,034,917). For these three matrices, we study the results obtained when using a global Schur complement-based preconditioner with the following local preconditioners: $ilu0$, $iluk$, $ilut$, $arms$, or $arms\_rbt$.

The pARMS parameters are chosen for these matrices following the guidance for the local ARMS preconditioner, as explained in [12], for example. Certain parame-

[1] http://www.nersc.gov (note that Hopper has retired since December 2016)

ters influence considerably the size and density of the last Schur Complement, which, in turn, affects greatly the performance of RBT. Since the RBT for dense matrices is used in this work, it is desirable that the last Schur complement remains dense while being relatively small. Hence, parameter values for the number of ARMS levels and the ARMS independent block size were chosen such that a small Schur complement is obtained. In particular, the former parameter was small (equals two) while the latter was large (allowing to form the blocks up to the entire local matrix size). At the same time, the drop tolerance for the last Schur complement was kept quite low (0.001) as well as all the other intermediate-level drop tolerances, so that there is close to none sparsification of the Schur complement.

The experiments are performed using 4 to 12 cores, and we use one MPI process per core and no multi-threading. In fig. 2.4, we compare the number of iterations required for convergence. We observe that, for matrices `Sherman5` (fig. 2.4a), `Raefsky3` (fig. 2.4b), *arms_rbt* performs better than the other local preconditioners. For `Cant`, *arms_rbt* converges in fewer iterations than the other preconditioners do so when using up to eight cores. This observation suggests that *arms_rbt* may be a more versatile preconditioner to use for obtaining superior convergence. Note that, for different numbers of subdomains (one per MPI process) in a given matrix, the obtained parallel preconditioning varies leading to the differences in the number of iterations to converge.

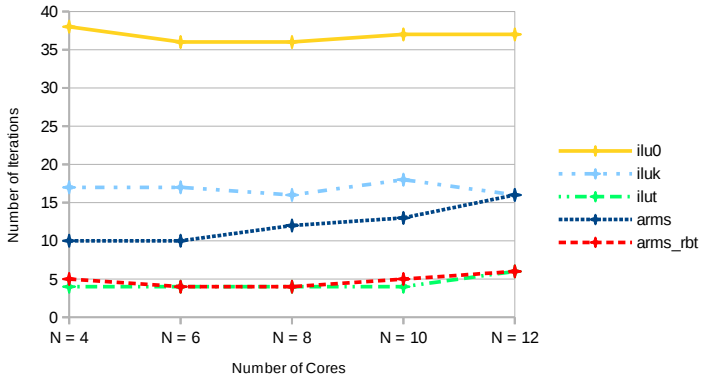
Note that for these tests, *arms_rbt* requires more time to solve the system since a dense matrix solver was used to solve the last Schur complement system in pARMS. However, as it will be illustrated in the next section, this solution remains faster than using a "sparse" variant of RBT based on SuperLU [37]. Figure 2.5 represents the residual obtained with the five local preconditioners. We observe that these preconditioners provide us with a similar accuracy, *arms_rbt* being more accurate for the matrix `Raefsky3`.

Figure 2.6 shows the execution time in solving the three test problems. In every case, *arms_rbt* requires more time to solve the system since, in our preliminary implementation, a dense-matrix solver was used to solve the last Schur complement system in pARMS. In the next section we study the use of a "sparse" variant of RBT based on the sparse direct solver SuperLU [37] in order to decrease the execution time. Note that the execution time does not decrease significantly with the number of cores due to a larger amount of communication when increasing the number of subdomains and MPI processes.

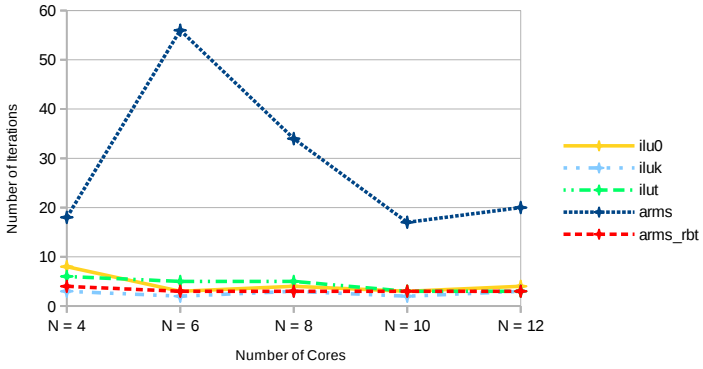
2.5 Some experiments using sparse RBT in pARMS

2.5.1 Motivation and issues in using sparse RBT

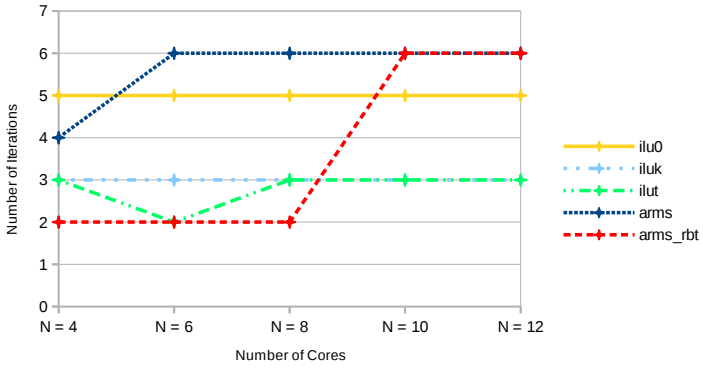
Depending on the test problems and parameters, the last Schur complement system in ARMS can be considered as sparse and it is natural to investigate to use a sparse direct solver to solve the corresponding linear system. However, for sparse direct solvers, the issue of pivoting in sparse direct solvers using LU factorization is often a serious bottleneck to achieve performance and scalability. A number of relaxed pivoting algorithms have been proposed in the past but none of them have shown



(a) `Sherman5`

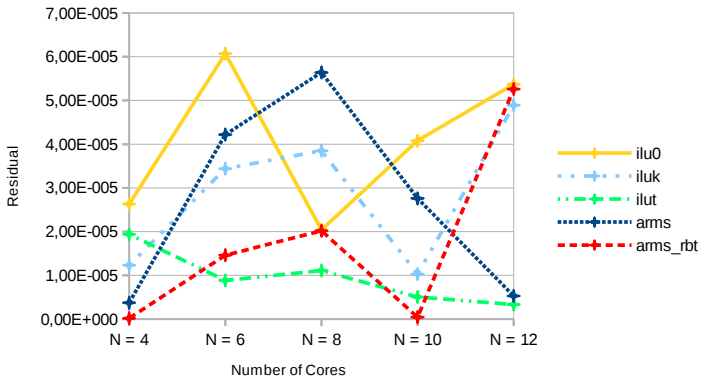


(b) `Raefsky3`

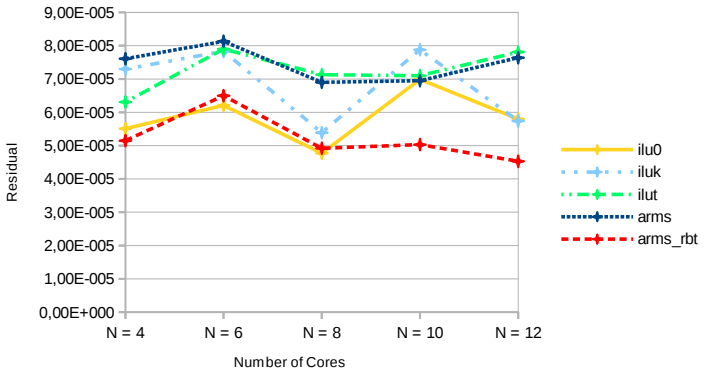


(c) `Cant`

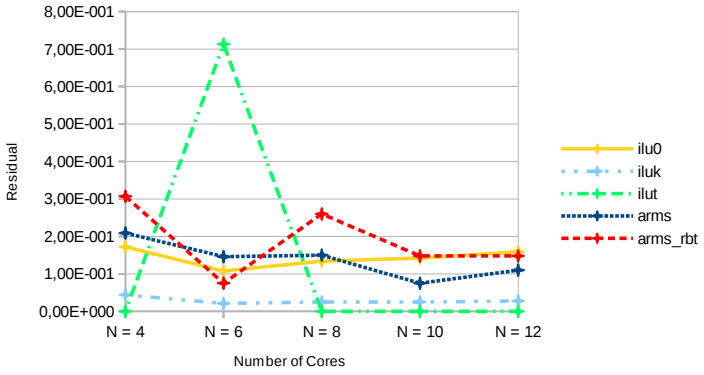
Figure 2.4: Iterations required for convergence with five choices of local preconditioner.



(a) `Sherman5`

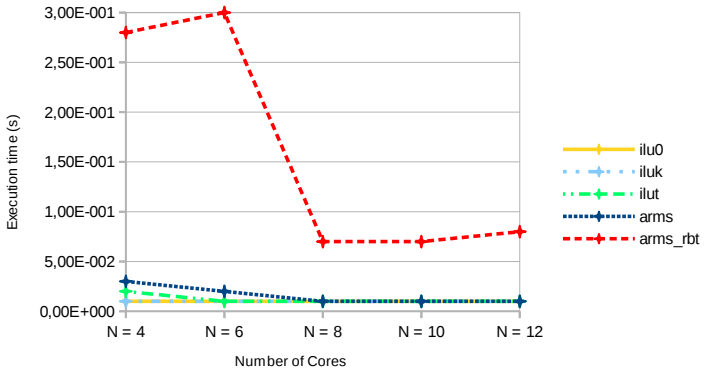


(b) `Raefsky3`

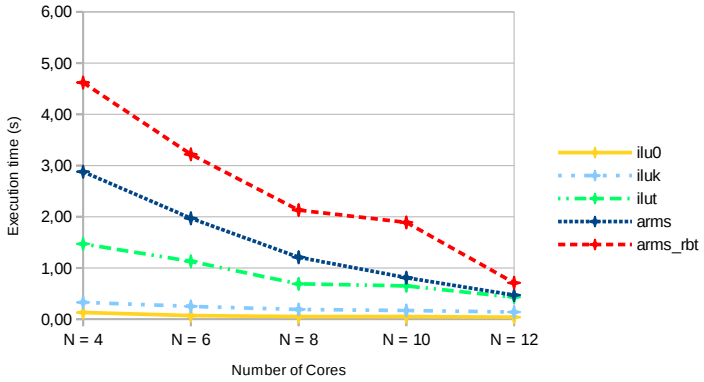


(c) `Cant`

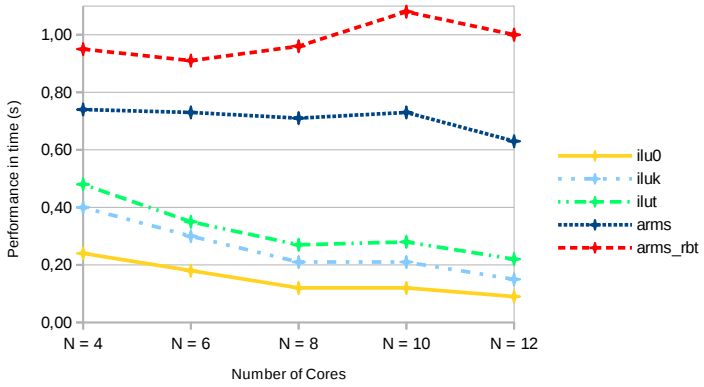
Figure 2.5: Residual for test problems with five choices of local preconditioner.



(a) `Sherman5`



(b) `Raefsky3`



(c) `Cant`

Figure 2.6: Execution time for test problems with five choices of local preconditioner.

promise of scalable implementation. In general dynamic pivoting causes dynamic change of L & U structures and might have dramatic consequence on performance. For example, if we consider the MUMPS factorization with 128 processes for the nonlinear optimization problem NLPKKT80, it requires 639 seconds using the partial threshold pivoting whereas it would require 87 seconds if we do not pivot (when the matrix is modified to be diagonally dominant).

A possibility of using RBT as an alternative to pivoting in sparse solvers has been explored in [60]. As mentioned in Section 2.3, for dense computations, the additional cost of applying RBT is limited to storing and applying RBT prior to the factorization but in the sparse case it modifies the nonzero structure of the randomized matrix. Indeed, the number of nonzeros in the transformed matrix U^TAV can be up to 4^d times the number of nonzeros in A in the worst case. This increase in nonzeros may lead to an even larger increase in the size of the LU factors and thus to prohibitive costs. To minimize this fill-in, we therefore limit our study to $d = 1$ which corresponds to a practical setting commonly used in the dense case.

Regarding the sparsity ordering, two different strategies when applying RBT have been studied in [60] :

- Strategy 1: First apply sparsity ordering, then apply RBT (i.e., $A = U^T(QAQ^T)V$, where Q is the fill-reducing permutation).
- Strategy 2: First apply RBT, then apply sparsity ordering (i.e., $A = Q(U^TAV)Q^T$).

Strategy 2 reduces the fill-in but then the matrix $A = Q(U^TAV)Q^T$ is not anymore guaranteed to be factorizable without pivoting (see counter-example in [60]). It has been observed in practice that

- Strategy 2 is as stable as Strategy 1.
- Strategy 2 reduces fill-in substantially.

Algorithm 6 describes how a sparse variant of RBT that combines RBT with the sparse direct solver SuperLU is applied to the last Schur complement system in the ARMS preconditioner. We point out that, similarly to Section 2.4, since RBT is applied locally, we use the sequential version of SuperLU (and not the MPI version SuperLU_dist [72]).

The re-ordering option that we chose in SuperLU is *Multiple Minimum Degree* (MMD) [73] applied to the structure of $A^T + A$ which gave the best results in terms of runtime for our test matrices.

Algorithm 6 Sparse Random Butterfly Transformation Algorithm ($d = 1$)

Modify the sparse matrix (CSR format, see Section 1.6.4) to have a size that is a multiple of 2
Generate recursive butterfly matrices U and V
Randomize A and obtain the matrix $A_r = U^TAV$
Factorize the randomized matrix with SuperLU with no pivoting (note that SuperLU also performs the ordering of A_r)
Compute U^Tb to solve $A_ry = U^Tb$, the solve $x = Vy$ by using SuperLU
Restore original size of vector

In the next section, we evaluate on several cores the performance of *arms_rbt*

combined with SuperLU.

2.5.2 Numerical experiments

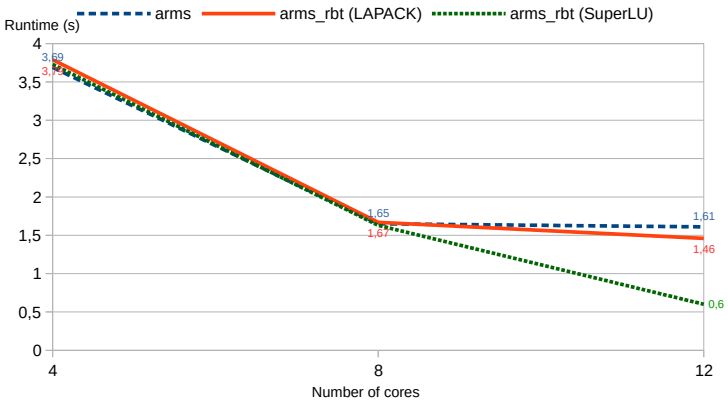
These experiments were carried on two Intel(R) Xeon(R) CPU E5645 processors at 2.40GHz, each CPU has 6 cores, and each CPU has 2M cache. We use one MPI process per core, and there is no multi-threading. We consider the following matrices from the Davis' collection [70]:

- `Raefsky3`, real non-symmetric matrix of size $21,200$ ($nnz = 1,488,768$) described in Section 2.4.2,
- `Cop20k_A`, real symmetric indefinite matrix of size $121,192$ ($nnz = 2,624,331$) which comes from an FEM accelerator cavity problem.

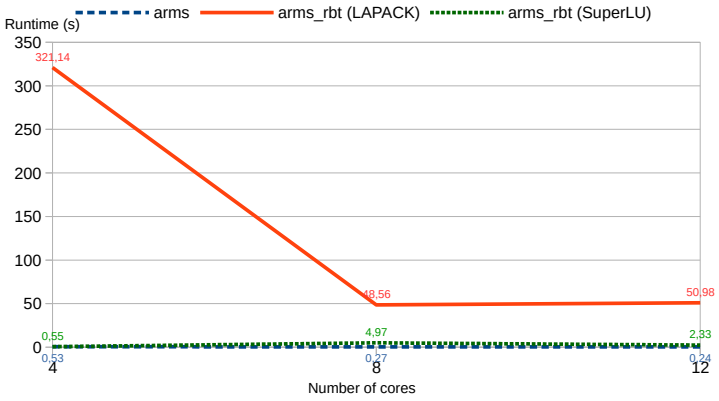
In our experiments we compare the performance of the preconditioners `arms_rbt` studied in Section 2.4.2 (dense RBT + LAPACK LU with no pivoting) with the sparse variant (sparse RBT + SuperLU with no pivoting). We point out that we obtained the same number of iterations for the two variants. Therefore we focus here on the comparison of the execution time. In Figure 2.7, we compare the performance of dense `arms_rbt` and sparse `arms_rbt` in terms of runtime for the matrices `Raefsky3` and `Cop20k_A`, respectively. We use 4-12 cores with one MPI process per core. We observe that the sparse variant of `arms_rbt` is always faster than the dense one. The difference is even more significant for `Cop20k_A` which is a bigger matrix. This can be explained by the fact that for these test matrices, the sparse variant takes into account the sparsity of the last Schur complement in the ARMS preconditioning technique, resulting in less arithmetical operations.

2.6 Conclusion

In this chapter, we have illustrated how the parallel distributed solver pARMS can benefit from the integration of the randomization technique based on the Random Butterfly Transformation (RBT) in solving the last Schur complement system in the recursive multilevel process of ARMS. The RBT preprocessing combined with a LAPACK-style routine that performs Gaussian elimination with no pivoting enabled us to enhance the convergence. Our experiments on test matrices from the Davis' collection showed an improvement in the number of iterations and accuracy. However, our integration of RBT in pARMS requires an implementation that may adjust the sparsity of the last Schur complement matrix based on the available memory and on the performance characteristics of its (direct) solver at hand. This constraint motivated additional experiments using a sparse variant of RBT followed by a factorization with no pivoting from a direct sparse solver (SuperLU). In our experiments, this sparse variant of RBT provided similar performance in terms of iteration count and accuracy while improving significantly the execution time. In the next chapter, we study how runtime performance can be improved by using GPU computing for the preconditioning phase of pARMS.



(a) `Raefsky3`



(b) `Cop20k_A`

Figure 2.7: Execution time for 2 test problems.

Chapter 3

Accelerating pARMS using Graphics Processing Units

3.1 Introduction

Graphic Processing Units (GPUs) are special-purpose architectures. Their highly parallel structure makes them more efficient than general-purpose CPUs. GPUs architectures are very different from multicore CPUs and provide more computational power by consuming less energy, which makes them suitable for improving the performance of large problems [1].

For example, a K40 NVIDIA GPU has a double precision peak performance of 1,689 Gflop/s for a thermal design power (TDP) of 235 W. According to benchmarks in the MAGMA library [36], optimized large dense matrix computations, e.g., matrix-matrix multiplications, reach 1,200 Gflop/s for a power draw of about 200 W, i.e., ≈ 6 Gflop/W. In contrast, two Sandy Bridge E5-2670 CPUs have about the same TDP ($2 \times 115 = 230$ W) as the K40 but for a peak of 333 Gflop/s, which translates to only 1.4 Gflop/W for the Sandy Bridge CPU. However, to achieve high performance, the algorithms must be designed for high parallelism and high "flops to data" ratio, while maintaining a low number of flops and exploiting the hardware features of the hybrid CPU/GPU architecture.

In recent years, several packages which include GPU implementations have been developed for iterative methods to solve sparse linear systems. As examples, we can mention the **cuSPARSE** library [74], which is a collection of routines for sparse linear algebra computations on NVIDIA GPUs. It provides subroutines, such as the sparse matrix-vector product kernel, matrix conversion routines, preconditioning techniques and some basic iterative solvers. We can also mention **ViennaCL** [75], which is a free open-source linear algebra library written in C++. **ViennaCL** is built on the three programming models CUDA, OpenCL and OpenMP, and it allows to obtain high performance for all three hardware architectures. It provides subroutines for dense or sparse linear systems, such as sparse matrix-vector and sparse matrix-matrix products, pipelined iterative solvers and various preconditioners.

For the GPU kernels integrated into our implementation, we will use the **MAGMA** library [8, 36], which is a public domain linear algebra library for heterogeneous architectures. In addition to being well-known for the dense linear algebra (e.g., factorizations and solvers for linear systems, least squares and eigen problems), MAGMA also provides a large variety of solvers, preconditioners, and eigen solvers for sparse linear systems. Comprehensive support for NVIDIA GPUs is provided, as well as some basic routines and functionalities in OpenCL and for Intel's Xeon Phi manycore accelerators (MIC).

In this chapter, we illustrate how the pARMS solver can be adapted for heterogeneous CPU/GPU architectures. We aim at improving the performance of pARMS by using GPU computing. The tasks performed on the GPU are related to the preconditioning of each part of the distributed matrix ("local preconditioning") which is handled in the distributed version by each MPI process. This preconditioning phase represents the main computational part and we show in this chapter how it can take advantage of GPU capabilities by using efficient kernels based on randomization or incomplete LU factorization. Note that the solving phase remains on the CPU.

We will use MAGMA routines for integrating preconditioning based either on ARMS and Random Butterfly Transformations (RBT) or Incomplete LU factoriza-

tion into pARMS. We analyze the performance improvement obtained using each method on some test matrices. Each preconditioning method ensures a good performance for a given set of matrices.

This chapter is organized as follows. In Section 3.2, we present the parallel implementation of ARMS. In Section 3.3, we evaluate the preconditioners available in the MAGMA library in order to choose the most efficient one. In Section 3.4, we detail the integration of GPU kernels into pARMS. Section 3.5 presents the numerical experiments in order to show the advantage of using GPU computing in the framework of the pARMS solver. Section 3.6 concludes the chapter with a reminder of the main contributions as well as some future work.

3.2 Parallel implementation of ARMS

Algorithm 7 outlines the steps to go through to solve a given linear system $Ax = b$, using the Schur complement (SC) preconditioning in the pARMS package. First, each processor loads the input matrix and performs Distributed Site Expansion (DSE) partitioning [31], which attributes to each process a set of equations corresponding to the rows of the global linear system, as well as the associated unknown variables. We note that when the number of processors is not a power of 2, the load balance between the different MPI processes is not ensured. As depicted in Figure 3.1, in the rows assigned to each processor, two parts may be distinguished: a local submatrix A_i that acts only on the local variables (u_i) and an external interface matrix X_i that acts only on the external interface variables, which are communicated from neighboring processors at each matrix-vector multiplication.

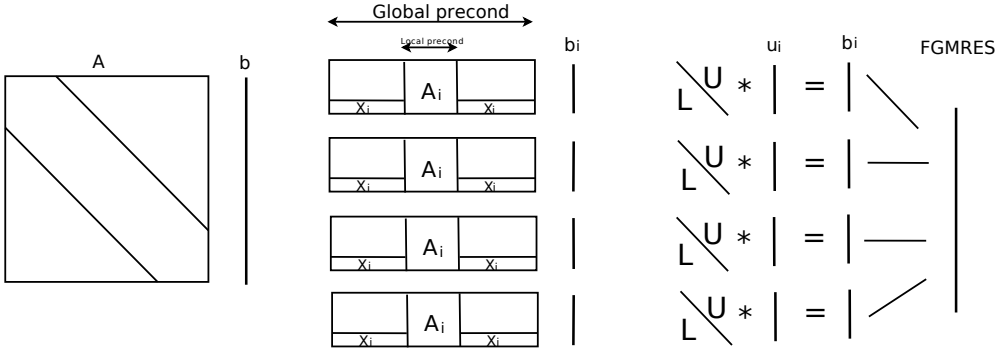


Figure 3.1: Sketch of the distributed linear system solution using pARMS (example using four processors).

To construct the global SC preconditioning, first the local LU factorization is performed on the local matrix held by each processor (Step 4 in Algorithm 7) in order to precondition the internal part of the local system and to obtain a factorization of the local Schur complement matrices $S_i = C_i - E_i B_i^{-1} F_i$, where $A_i = \begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix}$. More details about this step can be found in [62]. Then the different parts of the

distributed SC are constructed and *left to reside* in each processor (Step 5). In essence, the global SC system is never assembled or gathered in one processor and is being solved using a preconditioned GMRES method. Its distributed implementation, employed to solve the global SC, may be viewed as an "inner" accelerator with respect to the "outer" accelerator FGMRES used to solve the original linear system with the input matrix A. The solution of the global SC system yields a different preconditioning at each iteration, and hence the need for the outer flexible GMRES.

Algorithm 7 The linear system solution using a global Schur Complement preconditioning in pARMS

1. Each processor loads the sparse matrix A.

2. Partition the input matrix A using DSE partitioner.

3. Each processor exchanges boundary variables with neighboring processors.

4. Each processor performs local LU factorization on its local submatrix $A_i = L_i \times U_i$ (see Fig. 3.1).

5. Each processor constructs its portion S_i of the global Schur complement system from the result of the local LU factorization and the external interface submatrix X_i.

6. Solve the global Schur complement system iteratively by distributed GMRES as "inner" solver.

7. Each processor back-substitutes its interface variables to recover the internal variables.

8. Each processor calculates its local residual.

9. If the global residual norm is not small enough, repeat from Step 6.

3.3 Evaluation of MAGMA preconditioners for GPUs

MAGMA provides various preconditioners like the ILU [76, 77] factorization which uses "exact" triangular solves based on level scheduling techniques [15], or the approximate ILU (AILU) [78] which uses different numbers of Jacobi sweeps in the approximate triangular solves. MAGMA also provides the Jacobi, Incomplete Cholesky or approximate incomplete Cholesky preconditioners.

In this section, we compare the performance of different preconditioners available in the MAGMA library in order to identify which preconditioner gives the best performance and then will be a good candidate to be integrated into pARMS, in addition to our new `arms_rbt` preconditioner defined in Chapter 2 and which we also developed for GPU. We evaluate and compare their performance on a matrix

coming from a EDF (Electricité de France) application and using the GMRES solver or the preconditioned GMRES (PGMRES) solver from the MAGMA library. We consider the two following metrics of performance:

- the number of iterations necessary to the convergence,
- the execution time to reach the convergence.

3.3.1 Experimental framework

In our experiments, we analyze convergence and performance of the GMRES using different preconditioners when solving the real-world EDF problems. The corresponding test matrix `edf` comes from a computational fluid dynamics (CFD) application [79] and is generated using a regular hexahedral mesh. This matrix is symmetric and of size 16,384. In particular, to preserve the properties of the original matrices, we increase the size by simply duplicating the original matrix 128 times on the diagonal then we obtain our test matrix `edfx128`.The following experimental results pave the way for the later experiments in this chapter, in other words, they help us decide which kind of preconditioners to use to accelerate pARMS solver. Furthermore, the comparison results also contribute to the validation of some routines in the MAGMA library.

Table 3.1: Solvers and parameters used in the experiments

Solver	Preconditioner	Matrix Format	levels	sweeps	Max iterations	Relative tolerance
GMRES	No	CSR	–	–	500	1.0e-10
PGMRES	ILU	CSR	(0-2)	–	500	1.0e-10
PGMRES	AILU	CSR	(0-2)	(1-10)	500	1.0e-10

These experiments were carried out in double precision arithmetic on a system composed of one NVIDIA Kepler K40m GPUs and a dual Intel Xeon E5-2620 system. The GPU implementations are based on CUDA version 7.5 [80] and MAGMA version 2.0.0 [32]. We use one MPI process per core and no multi-threading. All the MPI processes are sharing the same GPU. The parameters applied for the execution of the evaluated solvers are summarized in Table 3.1.

Table 3.2: Test matrix

Matrix	Dimension	# non-zeros	Structure
`edfx128`	2,097,152	14,155,776	symmetric

3.3.2 Results

Figures 3.2 and 3.3 represent the performance of the different solvers given in Table 3.1 on the matrix `edfx128` in terms of number of iterations for convergence and execution time, respectively. The execution time is computed as the sum of the transfer time between CPU and GPU, the preconditioning time and the solving time.

These solvers include preconditioners such as the ILU with 0-2 levels and the AILU with 0-2 levels & 1-10 sweeps. For the ILU preconditioner, the solid lines

correspond to ILU(0), the dashed lines to ILU(1), and the dotted lines to ILU(2). Exact ILU uses exact triangular solves based on level scheduling and approximate ILU uses different numbers of Jacobi sweeps. Note that we obtained similar results on other (smaller) non-symmetric matrices coming from EDF.

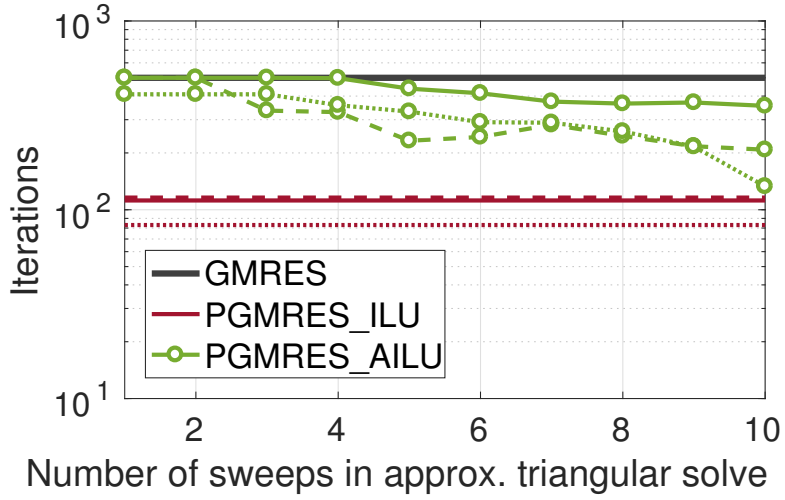


Figure 3.2: Number of iterations for various preconditioners on the `edfx128` matrix.

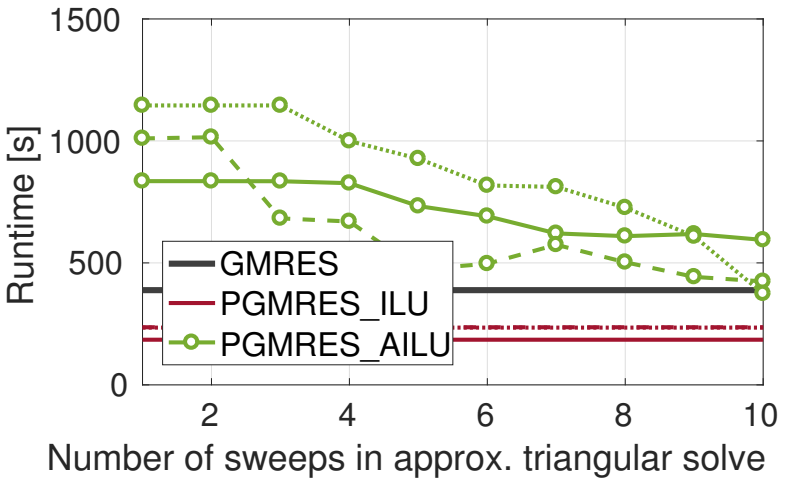


Figure 3.3: Execution time for various preconditioners on the `edfx128` matrix.

We observe in Figure 3.2 that using the ILU or AILU preconditioner decreases the number of iterations of GMRES significantly. The exact ILU preconditioner requires less iterations than the Approximate ILU preconditioner, with best performance for ILU(2). Using AILU requires some additional iterations of the outer GMRES solver, but depending on the fill-in level, 10 sweeps in the approximate

triangular solves enables us to get closer to the exact ILU iteration count.

Regarding the execution time, we observe in Figure 3.3 that the approximate triangular solves do not accelerate the ILU-preconditioned GMRES. Also, the lower iteration count for higher fill-in levels (1-2) is not reflected in the execution time, and despite a lower iteration count, the ILU(2) using exact triangular solves needs more time than the ILU(0). This comes from the higher cost of the preconditioner setup and data transfers and also by a higher fill-in that makes the sparse triangular solves more expensive. Then, for using an incomplete LU factorization preconditioner, the runtime winner for this test problem is the ILU(0) preconditioned solver. The metric of execution time is more relevant in industrial applications. Therefore, despite a slightly larger number of iterations (compared to ILU(2)), ILU(0) will be chosen in the next section, together with our new preconditioner `arms_rbt`, to be integrated in the hybrid CPU/GPU version of pARMS .

3.4 Integration of GPU kernels into pARMS

In our hybrid CPU/GPU approach, we use GPU computing in the preconditioning step of the pARMS solver. We mainly focus on the two following functions. The first function is related to the local preconditioner ARMS where we notice that the last Schur complement system becomes denser compared to the previous levels, and thus needs more time to be solved. Similarly to [81], we propose the use of Random Butterfly Transformation to avoid pivoting when solving the last Schur complement system.

The second function is related to the local preconditioner $ILU(0)$ which represents, by profiling the pARMS solver execution time, a significant part of the global time for a large set of test matrices. For instance, we analyze the time breakdown for solving the test problem `flame2p3d80x4` (described in Section 3.5.1) using one MPI process, where we use block Jacobi as a global preconditioner, ILU0 as a local preconditioner, and FGMRES to solve the global preconditioned system. We observe in Figure 3.4 that, for this test matrix, the preconditioning application (see Steps 6 and 7 of Algorithm 7) represents about 27% of the total time needed to solve the problem. We note that, for other test matrices, the preconditioning represents between 20% and 50% of the time for solution. This observation motivated our interest in accelerating the preconditioning phase using GPU computing.

Figure 3.5 presents the tree of functions calls, and analyzes the performance of the pARMS solver. It is obtained using the Valgrind [82] tool. Such a figure helps us to integrate various GPU kernels in different parts of the pARMS code. Note that Valgrind was also used to obtain in Figure 3.4 the time spent in each function.

3.4.1 GPU implementation of Random Butterfly Transformations in ARMS

In Chapter 2, we explained how the RBT technique can be applied to the ARMS preconditioning where the LU factorization is replaced by the RBT preprocessing in the solution of the last Schur complement. First, the last Schur complement which is a sparse matrix is converted into a dense matrix. Then, RBT is applied

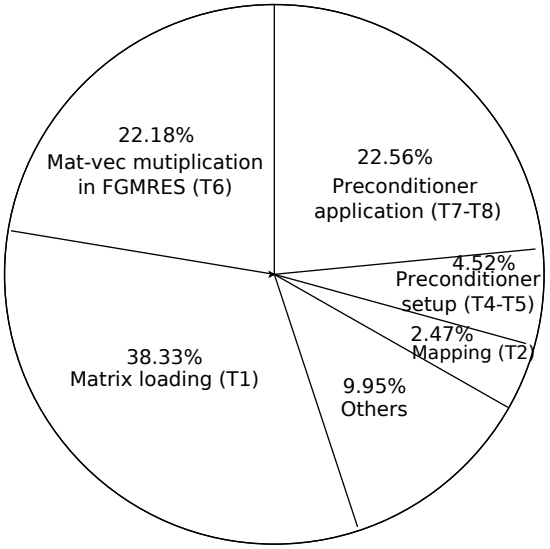


Figure 3.4: Time breakdown for the pARMS solver (matrix `flame2p3d80x4`) and corresponding task numbers in Algorithm 7.

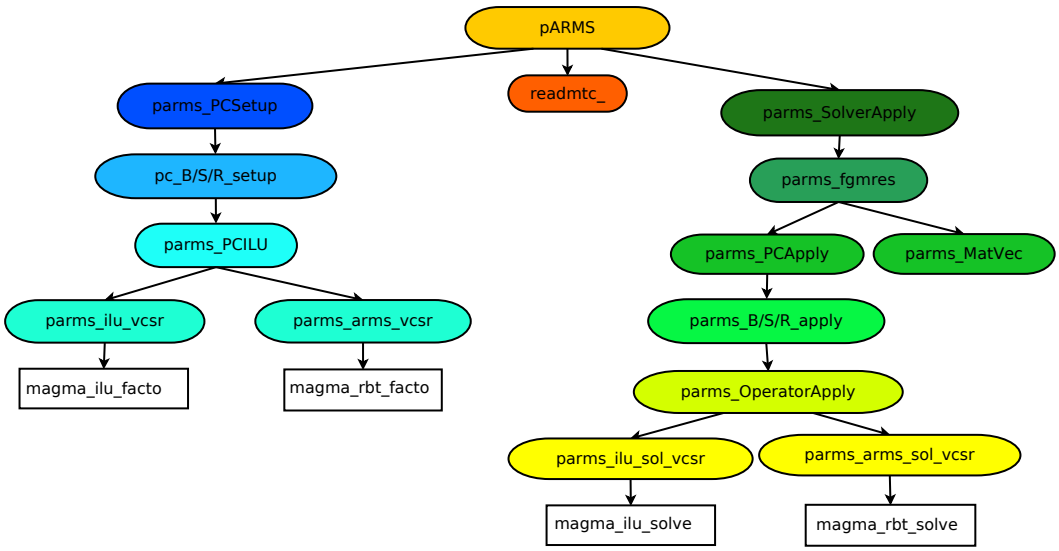


Figure 3.5: Tree of function calls in pARMS obtained with Valgrind profiling.

and the dense matrix is randomized using two recursive butterfly matrices. Finally, the randomized dense matrix is factorized using a LAPACK-like [5] routine that performs Gaussian elimination with no pivoting, followed by two triangular solves.

To apply RBT to the last Schur complement S, we generate two recursive butterfly matrices W_1 and W_2 of size $n \times n$. We set d to 2 since, as explained in [20], two recursions are in general sufficient to obtain satisfactory results. Instead of solving the initial system, $S\ x = b$, using Gaussian elimination with partial pivoting, we first solve the randomized system, ${W_1}^T\ S\ W_2\ z = {W_1}^T\ b$, using Gaussian elimination with no pivoting, then the system $W_2\ z = x$.

In our GPU implementation, the RBT preprocessing is performed as follows: using the MAGMA routines: `dgerbt`, `dgetrf_nopiv_gpu` and `dgetrs_nopiv_gpu`, following the three steps below:

1. Randomize the last Schur complement in ARMS (converted to dense format) using `dgerbt`.
2. Factorize the last Schur complement system with `dgetrf_nopiv_gpu` to obtain the upper triangular matrix U and the lower triangular matrix L.
3. With L and U triangular matrices obtained in the second step, we solve the triangular systems by using `dgetrs_nopiv_gpu`, $Ly = b$, $Ux = y$.

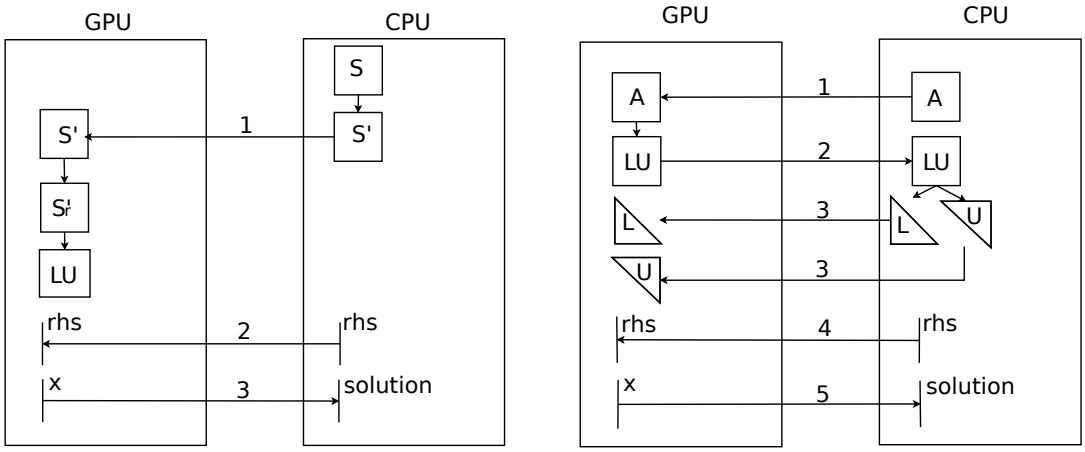
In Figure 3.6a, we describe the data movements (numbered chronologically) between the CPU and the GPU in our implementation of the solution of the last Schur complement using RBT (S denotes the last Schur complement, S' is a dense format of S, and S'_r is the randomized matrix, rhs is the right-hand side of the system).

We note that to get performance improvement using the GPU implementation, the last Schur complement should not be too small, as it will be illustrated in the numerical experiments section.

3.4.2 GPU implementation for $ILU(0)$ in pARMS

In this implementation, we use MAGMA routines to perform the $ILU(0)$ factorization and the triangular solves. The routine `magma_dcumilusetup` prepares the ILU preconditioner via the cuSPARSE library to factorize the local system of each processor. Then the routines `magma_dapplycumilu_l` and `magma_dapplycumilu_r` perform the left and right triangular solves, respectively, by using the $ILU(0)$ preconditioner.

In Figure 3.6b, we describe the data movements between the CPU and the GPU for the $ILU(0)$ preconditioning in pARMS, where A denotes the local matrix held by a processor. In this figure, the data movements are numbered chronologically. Note that the factorized LU system is sent back to the CPU host to perform the separation between the L and U factors (steps 2 and 3).



(a) *arms_rbt* (b) *magma_ilu0*

Figure 3.6: CPU-GPU communication for *arms_rbt* and *magma_ilu0* in pARMS.

3.5 Numerical experiments

3.5.1 Experimental framework

The experiments were carried out in double precision arithmetic on the system described in Section 3.3.1. We use one MPI process per core, no multi-threading and all the MPI processes share the same GPU.

The first test matrix `edf` was already described in Section 3.3.1. The second test matrix `flame2p3d80` arizes from using a finite-difference scheme with local approximation (called FLAME) [83] to screened electrostatic interactions of spherical colloidal particles governed by the Poisson-Boltzmann equation (PBE). Specifically, this matrix corresponds to a two-particle simulation on an 80^3 grid and has a regular sparsity structure but it is not symmetric and not diagonally-dominant due to the characteristics of FLAME. This simulation has been studied in [84], where it has been shown that parallel preconditioning is imperative for its solution and that even modest levels of ILU fill-in already yield a good convergence. The third test matrix `epb3` is a matrix from the University of Florida Sparse Matrix Collection [71]. It represents a large case of a plate-fin heat exchanger.

Here, we test these matrices on hybrid CPU/GPU architectures. As detailed in the previous section, we use GPU computing to perform the local preconditioning. We point out that for our experiments, we increase the size of the original matrices `edf`, `flame2p3d80`, and `epb3` in order to study the scalability. In particular, to preserve the properties of the original matrices, we increase the size by simply duplicating the original matrix several times on the diagonal. For example, to obtain the matrix `flame2p3d80x4`, which is four times larger, we duplicate the original `flame2p3d80` three times along the diagonal. Table 3.3 contains the size and number of nonzeros of the "scaled" test matrices.

Table 3.3: The set of test matrices.

Matrix \ Characteristics	Dimension	# non-zeros	Structure
`edfx128`	2,097,152	14,155,776	symmetric
`flame2p3d80x4`	2,125,764	13,808,916	unsymmetric
`epb3x64`	5,415,488	29,672,000	unsymmetric

3.5.2 RBT combined with ARMS preconditioning

In this section, we compare the execution time of the original pARMS solver (CPU) to that of the hybrid CPU/GPU version which uses RBT to solve the last Schur complement system in the recursive process, as described in the previous section.

In Figure 3.7, we evaluate the weak scaling of the CPU and CPU/GPU solvers. The tests are performed on the following matrices: `edfx16`, `edfx32`, `edfx64` and `edfx128`. Here the number of non-zeros of the sparse matrix increases with the number of cores, that is with the number of MPI processes used. The performance of `arms_rbt` is better than `arms`, except for 2 cores due to the small size of the problem that does not enable us to take advantage of the GPU. Figure 3.7 shows that using 12 MPI processes and one GPU, the execution time decreases by 30%. We add that for this specific test, the size of the last Schur complement held by each MPI process ranges from 2828 to 3724.

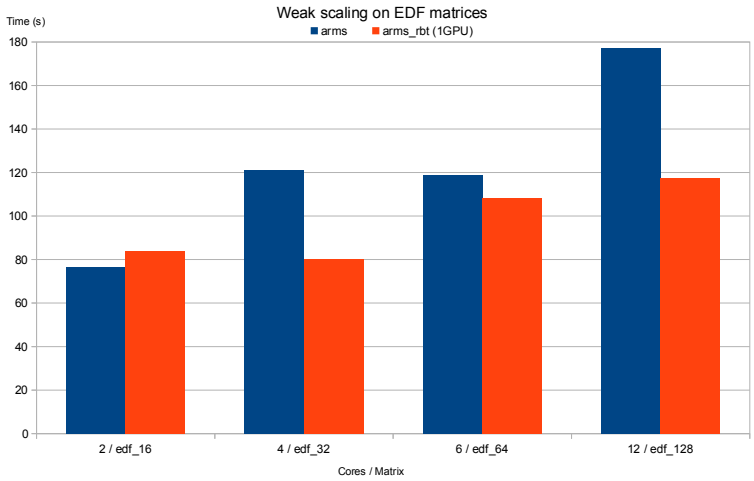


Figure 3.7: Execution time for `arms` and `arms_rbt` on different `edf` matrices.

Figure 3.8 shows that `arms_rbt` performs better than `arms` on the `epb3x64` matrix using different number of MPI processes varying from 6 to 12. In the best case, using `arms_rbt` decreases the execution time by 25%. We note that we can not use less than 6 MPI processes for this problem since the last Schur complement is too large to fit into the GPU memory.

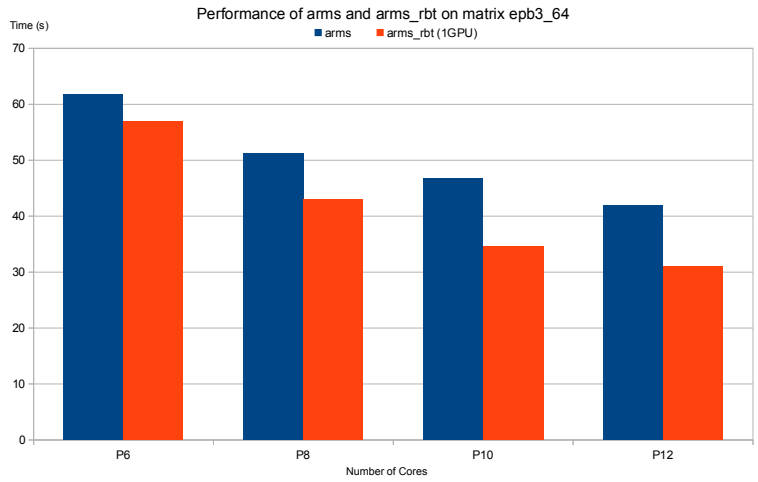


Figure 3.8: Execution time for `arms` and `arms_rbt` on the `epb3x64` matrix.

3.5.3 $ILU(0)$ preconditioning

We compare the execution time for the two following solvers:

- pARMS with $ILU(0)$ local preconditioning on the CPU, referred to as `pARMS_ilu0`.
- pARMS with $ILU(0)$ local preconditioning on the GPU, referred to as `magma_ilu0`.

Figure 3.9 displays the execution time of the `pARMS_ilu0` and the `magma_ilu0` solvers for the `edfx128` matrix. We note that for this matrix, using more than 6 MPI processes does not bring any advantage. Indeed the processes are sharing the same GPU, which increases the communication amount to perform between the CPU and the GPU with respect to the problem size.

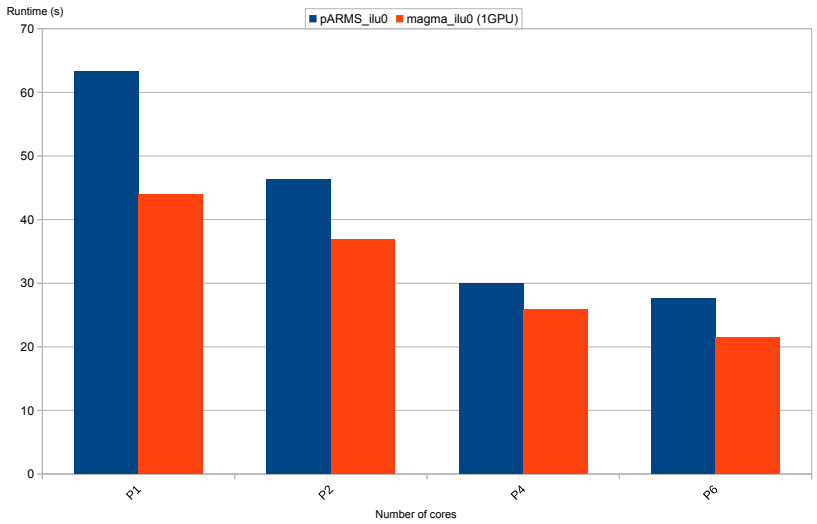


Figure 3.9: The execution time for pARMS with $ILU(0)$ on the `edfx128` matrix.

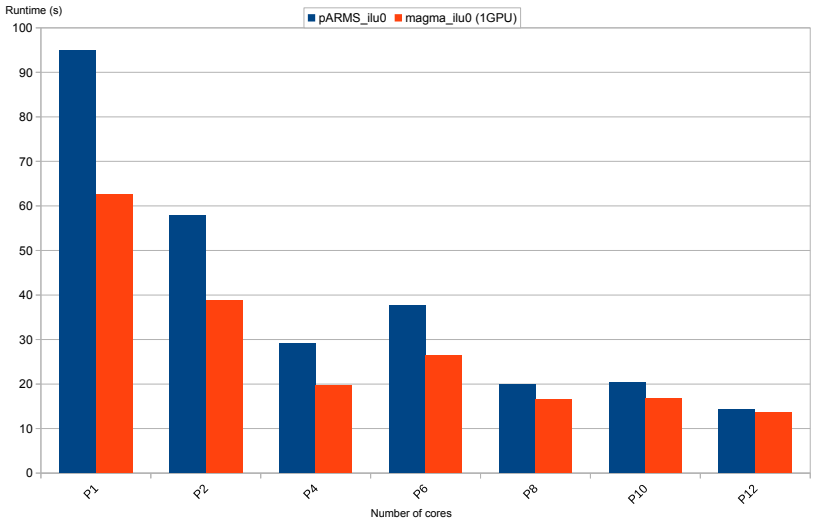


Figure 3.10: The execution time for pARMS with $ILU(0)$ on the `flame2p3d80x4` matrix.

Figure 3.10 shows the execution time of the `pARMS_ilu0` and the `magma_ilu0` solvers for the `flame2p3d80x4` matrix. The best improvement is obtained using one MPI process and it is about 30%. In this figure, we observe that when we use 6 MPI processes, the execution time of the `pARMS_ilu0` solver increases with respect to the use of 4 MPI processes. This is because of the DSE partitioning that does not ensure a good load balance between the processes when using a number of processes which is not a power of 2.

3.6 Conclusion

In this chapter, we have illustrated how a non intrusive approach can be applied to integrate GPU computing into the pARMS solver, more specifically for the local preconditioning phase that represents a significant part of the time to solve a given sparse linear system. The CPU-only and the hybrid CPU/GPU solvers have been compared on several test problems from physical applications. The performance results of the hybrid CPU/GPU solver using the ARMS preconditioning combined with RBT, or the $ILU(0)$ preconditioning, show a performance gain up to 25% and 30%, respectively, on the test problems considered in this paper. In a future work, extensive testing will be performed on other matrices and other local preconditioners (e.g., $ILUT$), also by using several nodes of a cluster of GPUs. Moreover, in the case where the last Schur complement is too large and sparse, we will investigate the use of RBT applied to the sparse matrix (without conversion to dense format) and the factorization using a sparse direct solver on GPU.

Chapter 4

Resilience of pARMS to soft errors

4.1 Introduction

Fault tolerance is a major concern in high-performance computing. Indeed, the prevalence of faults is expected to increase as the level of concurrency in HPC platforms continue to become larger and larger [85–87]. As the typical HPC environment is moving towards "Exascale", the mean time between failures (MTBF) will continue to decrease dramatically [87]. Faults are typically divided into two distinct categories: hard faults and soft faults [88, 89].

Hard faults are usually due to negative effects on the physical hardware components of the system which cause the loss of one or several processors. The key characteristic of all hard faults is that they cause program interruption. Thus they are difficult to deal with from an algorithmic standpoint. However, as hardware components continue to grow both smaller and faster, they (generally) become more prone to error, and the algorithms and software packages used in HPC environments need to be able to respond to sudden and unexpected changes in both the quantity and quality of the physical resources that may be available for use.

The other category of faults, soft faults, can cause wrong arithmetic or abnormal storage. In general, soft faults do not immediately cause program interruption, although program interruption may occur as a result of the damage caused by a soft fault. Another key feature of soft faults, is that they can be detected (though the cost of detection may be prohibitively high) during program execution. Most of the time, soft faults refer to some data corruption.

While hard faults, resulting often in processor breakdowns, are in general easy to detect, this is not the case for soft faults usually referred to as *silent faults*. Common strategies to deal with soft fault are based on replication strategies of computing resources or redundancy of computations, resulting in significant overheads. A classical technique for fault corrections is *checkpoint/restart* (see, e.g., [90]) which saves data periodically on a storage device and when a fault occurs, restart from the most recent checkpoint. Another classical approach called *Algorithm-Based Fault Tolerance* (ABFT) (see, e.g., [91] for iterative methods) focuses on introducing checksums or other encoding schemes into the algorithm itself.

In this chapter, we consider only soft faults. They are typically divided into various categories based on how long their effect is felt by the resident program. For a long time, the impact of soft faults was measured by the injection of bit flips into the data structures used by the algorithm in question. However, recent research efforts (e.g., [87, 88, 92, 93]) have focused on modeling the impact of soft faults with a slightly more generalized numerical approach that quantifies the potential impact of a bit flip – which is dependent on where the bit flip occurs – and generating an appropriately sized fault using a more numerically-based scheme. The experiments conducted in this chapter aim at adapting two existing, generalized, numerical soft fault models to study a particular class of soft faults ("sticky" faults) that has not been examined extensively in the past.

The remaining sections of this chapter are organized as follows: In Section 4.2, we present the general motivations for our study of soft faults and a brief overview of related studies. In Section 4.3, we recall the FGMRES algorithm and indicate at which step of the algorithms are injected the perturbations. In Section 4.4, we

detail the fault models used throughout this chapter. In Section 4.5, we evaluate the PBSFM model on the `arms_rbt` preconditioner. In Section 4.6, we present the experimental results for the two fault models. Finally, Section 4.7 concludes this chapter and gives possible directions for future work.

4.2 Motivations and related work

The easiest instance of a soft fault to understand occurs in the form of a bit flip. Then researchers have typically relied upon the direct injection of bit flips into their routines in order to simulate the occurrence of a soft fault [94, 95]. On the other hand, in the work by Elliot, Hoemmen, and Mueller [87, 88, 92], faults are modeled in a more general sense. In this approach, during the injection of a fault into the result of a specific important operation inside of the algorithm – in the case of an iterative solver (i.e. FGMRES) this could be a sparse matrix-vector multiply or in the application of the preconditioner – instead of flipping a bit inside of the resultant data structure, a study ([95]) was conducted to quantify the impact of a single bit flip and this analysis was used in [87] in order to create a numerical soft fault model that injects a fault in a more general sense. This fault injection methodology is described in section 4.4.1.

In the classification of soft faults that is presented in [88, 89], soft faults are divided into the following three categories based upon how they affect program execution: transient, sticky, and persistent. Transient faults are defined as faults that occur only once, sticky faults indicate a fault that recurs for some period of time but where computation eventually returns to a fault-free state, and persistent faults arise when the fault is permanent. Note that a fault in any of these three classifications should be detectable during program execution, and as such, it should be possible to ameliorate the negative impact caused by the fault.

Scenarios that could cause a persistent fault include a stuck bit in memory, or the Intel Pentium FDIV bug [88, 96]. Traditional analysis of potential persistent type errors has rested more in the hardware domain than in the algorithmic domain, with analysis of both processor based faults [97, 98] and memory based faults [99]. An example of a situation that can create a sticky fault, that is provided in [89], is the incorrect copy of data from one location to another. The incorrect bit pattern present in the faulty copy of the data will remain incorrect for an indefinite amount of time, but will be corrected if and when the data is copied over again – assuming that the later copy routine executes correctly. It is also important to note that in the case of a sticky fault, the fault can be corrected by means of a direct action. Transient errors are typically caused by solitary bit flips, which may be caused by different issues (e.g. radiation, hardware malfunction, data cache set incorrectly, etc).

Whether research chooses to model faults using bit flips or adopt a more numerical approach, much of the previous work on the impact of silent data corruption (SDC) has to do with the modeling of transient errors. The goal of the study that is detailed here is to adapt both a numerical soft fault model for transient soft faults and a perturbation based soft fault model for persistent soft faults so that each of

the two models is capable of modeling the potential impact of a sticky fault.

4.3 Perturbation of the FGMRES algorithm

In this section, a brief background of both the GMRES algorithm with flexible preconditioning (FGMRES) is given, and then an overview of the two preconditioners that were focused on in this study is provided.

The FGMRES algorithm, as described in [15], is provided in Algorithm 8. FGMRES is similar in its nature to the standard GMRES with the notable exception of allowing the preconditioner to change in each iteration by storing the result of each preconditioning operation (cf. matrix Z_m in line 13). FGMRES was selected in this study because it is used in pARMS to solve the global system (see Sections 1.5 and 3.2). It is a robust, popular iterative solver which is proven to converge under variable preconditioning - including converging in situations where the variable preconditioning comes as a result of some sort of perturbation or anomaly in the preconditioning operation. In this study specifically, such a perturbation is due to injected faults via one of the two fault models that is used in the experiments.

Algorithm 8 FGMRES as given in [15]

1: ▷ In: A Linear system $Ax = b$ and an initial guess at the solution, x_0
2: ▷ Out: An approximate solution x_n for some $n \geq 0$
3: $r_0 := b - Ax_0$;
4: $\beta := ||r_0||_2, v_1 := r_0/\beta$;
5: **for** $j = 1, 2, \ldots, m$ **do**
6: $z_j = M_j^{-1} v_j$;
7: $w = Az_j$;
8: **for** i = 1, 2, . . . , j **do**
9: $h_{i,j} := (w, v_i)$;
10: $w := w - h_{i,j} v_i$;
11: **end for**
12: $h_{j+1,j} := ||w||_2, v_{j+1} := w/h_{j+1,j}$;
13: $Z_m := [z_1, \ldots, z_m], \bar{H}_m := h_{i,j}{}_{1 \leq i \leq j+1; 1 \leq j \leq m}$; ;
14: **end for**
15: $y_m := argmin_y ||\bar{H}_m y - \beta e_1||_2, x_m := x_0 + Z_m y_m$;
16: **if** Convergence was reached **then**
17: x_m
18: **else**
19: set $x_0 = x_m$, go to Line 3
20: **end if**

In particular, faults were injected at two distinct points inside of the FGMRES algorithm; line 3, termed here as the *outer matrix-vector* operation, and line 6, which is the application of the preconditioner.

In our study, we will consider the three following preconditioners:

- ILUT, see Section 1.4,

- `arms`, see Section 1.5.1,
- `arms_rbt`, see Section 2.4.

4.4 Fault models

We assume a fault model for Silent data corruption. Here we note that in practice, the SDCs are rare, even at extreme scales of parallelism. However, it is worth to study how a single SDC impacts an algorithm or a computing result. The ancient SDC model considers a silent bit flip. In this work, our objective is to analyze the effects of SDC in specific algorithms, then compare the resilience of different types of preconditioners.

Injecting bit flips is not interesting, because it only needs setting the memory location equal to any value. Besides, the bit flips errors produce numeric values, non-numeric infinity and not a number values. In our work, we apply a different approach, SDC impacts only the critical computations of an algorithm. We are not interested in detecting binary errors, however we consider bit flips as numerical errors and evaluate how these errors relate to the fundamentals.

In this section, a description of each of the fault models that are included in the comparison is provided. Particular note is made to distinguish each of the models from one another. As noted earlier, the two main sticky fault models that were utilized in this study were an adapted version of the numerical soft model presented in [87, 88, 92] - termed "Numerical Soft Fault Model" (NSFM) due to the origins of this model in seeking a numerical estimation of a fault (rather than modeling faulty behavior directly), and an adapted version of the model given in [93] - which will be referred to as the "Perturbation Based Soft Fault Model" (PBSFM) due to its modeling of faults as small random perturbations [88, 89, 96]. A fuller description of each of the two soft fault models follows in the next two subsections. Note that in the remainder of the chapter, the Euclidean norm is used.

4.4.1 Numerical Soft Fault Model (NSFM)

The approach given by detailed in [87, 88, 92] generalizes the simulation of soft faults by disregarding the actual source of the fault and allowing the fault injector to create as large or as small a fault as necessary for the experiment. In the experiments conducted in [87, 88, 92] faults are typically defined as either:

- a scaling of the contribution of the result of the preconditioner application for the Message Passing Interface (MPI) process in which a fault was injected,
- a permutation of the components of the vector result of the preconditioner application for the MPI process in which a fault was injected,

or a combination of either of these two effects. Note that if α is the scaling factor used, and if x is the original vector and if $\hat{x}$ is the vector with a fault injected into it than we have three scenarios:

1. $\alpha = 1$: $||x||_2 = ||\hat{x}||_2$
2. $0 \leq \alpha < 1$: $||x||_2 > ||\hat{x}||_2$
3. $\alpha > 1$: $||x||_2 < ||\hat{x}||_2$

The adaptation that was made to extend this model to be applicable in a "sticky" sense was to inject a fault into a single MPI process in the exact same manner at every iteration in which a fault is simulated. The analysis that was performed in [87, 88, 92] details the impact of the NSFM in the case where it is modeling transient soft faults with various scaling values. The impact of this fault model relative to the impact of a single bit flip is given in [87] and shows that regardless of where the bit flip occurs, the NSFM will perform comparably to the worst case scenario induced by a traditional bit flip. Analysis to show the impact of a bit flip based on where in the storage of a floating point number it occurs is given by [95].

4.4.2 Perturbation Based Soft Fault Model (PBSFM)

The approach proposed in [93] is similar in spirit to the NSFM proposed above. It selects a single MPI process and injects a small random perturbation into each element of the vector. Thus, if the vector to be perturbed is x and the size of the perturbation factor is ϵ, then we first generate a random number $r_\epsilon \in (-\epsilon, \epsilon)$ then, we set $x_i = x_i + r_\epsilon$ for all i. The resultant vector, call it $\hat{x}$, is thus perturbed away from the original vector x.

Since the FGMRES algorithm works at minimizing the norm of the residual, and this can be directly affected by the norm of certain steps inside of the FGMRES algorithm, there are three variants to the PBSFM:

1. The sign of x_i is not taken into account. In this variant, $||x||_2 \approx ||\hat{x}||_2$.
2. If $x_i \geq 0$ then $r_\epsilon \in (-\epsilon, 0)$ and if $x_i < 0$ then $r_\epsilon \in (0, \epsilon)$. Here, $||x||_2 \geq ||\hat{x}||_2$.
3. If $x_i \leq 0$ then $r_\epsilon \in (-\epsilon, 0)$ and if $x_i > 0$ then $r_\epsilon \in (0, \epsilon)$. Here, $||x||_2 \leq ||\hat{x}||_2$.

Using these three variants allows the PBSFM to possess some level over the norm of the vector that it is injecting a fault into, and therefore an added level of control on how a fault may affect the convergence of the FGMRES algorithm.

4.4.3 Comparison of soft fault models

In looking to see which of the two fault models induces a "larger" fault, then in general it will be the case that the NSFM will create a larger difference between a given data structure with a fault injected and the same data structure in a fault free environment.

Examining this for the test problem (section 4.4.4) used in these experiments, the result of the outer matrix-vector operation is a zero vector initially and as FGMRES progresses closer to the solution, this vector will begin to approach the original right hand side of the equation, b. In this problem, the entries in the final iterates of Ax_i before convergence will have entries, b_i, where $-0.01 \leq b_i \leq 0.01$ forms a loose bound on all entries of $b = Ax_i$. To show the potential difference in magnitude between a given vector b and a vector $\hat{b}$ representing the vector b with a fault injected, 10000 random vectors were generated in Matlab for vectors b of varying sizes (to represent varying problem sizes) and the norm of $|b - \hat{b}|$ was calculated for each of the two fault models. These results are shown in table 4.1.

Additionally, the NSFM allows slightly more exact statements to be made concerning the effect of the injected fault on the norm, as the norm will be the exact

Vector Size	$\|\|b - \hat{b}\|\|^2$ - NSFM	$\|\|b - \hat{b}\|\|^2$ - PBSFM
10	2.2223	9.0351e-04
100	5.1826	0.0029
1,000	17.1997	0.0091
10,000	53.8458	0.0289
100,000	172.3676	0.0913
1,000,000	543.9308	0.2887

Table 4.1: Difference in the effect of each of the fault models on random vectors with values similar to those found in the result of the outer matrix-vector operation. Note: The scaling factor in the NSFM was set to 1.0 and the fault size in the PBSFM was set to 5×10^{-4}. Columns 2 and 3 represent average differences over 10,000 runs.

same for all but the affected subdomains, where the norm of that section is controlled explicitly. However, the size of the fault – measured as a difference from a fault free run – is in general dependent only on the problem size in the case of the NSFM. On the other hand, statements concerning the norm are inherently not exact when the PBSFM is used, as the norm of the faulty subdomain is not precisely controlled, but the difference from a fault free run - i.e. the "size" of the fault - is easier to control by way of simply adjusting the bounds on the perturbation that is used.

In general, one of a limited number of outcomes is most likely to occur when a fault occurs during the execution of an iterative solver ([87, 94]).

- The solver will converge in approximately the same number of iterations, with an error in the final solution.
- The solver will converge in approximately the same number of iterations, with no error in the final solution.
- The solver will converge in more iterations than in a fault free run; with or without an error in the final solution.
- The progress of the solver towards the solution will stagnate, and it will fail to converge.

4.4.4 Test problems

In this section, we give a brief summary of the test problem that was used in the experiments that follow in this chapter. The test problem comes directly from the pARMS library [13], and represents the discretization of the following elliptic 2D partial differential equation,

$$-\Delta u + 100\frac{\partial}{\partial x}(e^{xy}u) + 100\frac{\partial}{\partial y}(e^{-xy}u) - 10u = f$$

on a square region with Dirichlet boundary conditions, using a five-point centered finite-difference scheme on an $n_x \times n_y$ grid, excluding boundary points. The mesh is mapped to a virtual $p_x \times p_y$ grid of processors, such that a subrectangle of $xnmesh = n_x/p_x$ points in the x direction and $ynmesh = n_y/p_y$ points in the y

direction is mapped to a processor. The size of the problem can be monitored by changing the size of the mesh that was used in the creation of the domain.

The mesh sizes that were considered corresponded to a "small" problem of $xnmesh = ynmesh = 200$ and a "large" problem variant with $xnmesh = ynmesh = 400$. Both of these two problem sizes were run on a $p_x = p_y = 20$ grid of 400 total processors. This leads to problem sizes of,

- Small: $n_x \times n_y = xnmesh \times p_x \times ynmesh \times p_y = 16,000,000$
- Large: $n_x \times n_y = xnmesh \times p_x \times ynmesh \times p_y = 64,000,000$

As pointed out in section 4.4.3 the NSFM creates larger faults in some sense for larger problem sizes, whereas the size of the fault injected by the PBSFM scales much more evenly with problem size.

4.5 Preliminary evaluation of the PBSFM fault model.

The resilience of `arms` along with ILUT has been already evaluated with PBSFM in [100] but it is natural to study how our new `arms_rbt` preconditioner behaves with the same error model. In this section, we present the experiments that were conducted to evaluate the PBSFM model when using the `arms_rbt` preconditioner, and compare it with `arms`.

4.5.1 Experiment description

We inject soft faults using the PBSFM model in the main FGMRES algorithm for *matrix-vector* operation (residual vector r_0 in Algorithm 8), or in the preconditioner application (vector z_j in Algorithm 8) or both of them. Then we evaluate the effect of these errors in terms of number of iterations, when using the `arms` and `arms_rbt` preconditioners, respectively. Similarly to [100], we inject errors for several iterations, starting from the fifth iteration, then we compute the number of additional iterations required to reach convergence. The test problem has been described in Section 4.4.4, with the parameters given in Table 4.2. The value of the perturbations are $\epsilon = 1e-6, 5e-6, 1e-5, 5e-5$.

These experiments were carried out in double precision arithmetic on a system composed of two dual Intel Xeon E5645 processors. In total, we have 12 physical cores, and the hyper-threading is disabled. We use one single thread MPI process per core.

4.5.2 Results

For each figure, the x-axis represents the percentage of iterations of FGMRES that have been perturbed by errors, compared to the iterations required for a fault-free execution. Then the y-axis corresponds to the number of additional iterations required to obtain convergence.

In Figure 4.1, we have injected errors in the *outer matrix-vector* operation in the FGMRES algorithm and we compare the effect of these errors when using `arms` and `arms_rbt`.

Table 4.2: The set of test parameters.

Parameters ╲ Matrix	`2D elliptic`
Tolerance for inner iteration	0.01
Tolerance for outer iteration	1.0e-6
Number of levels for ARMS, ARMSRBT	2
Block size for block independent sets	1000
Tolerance used in independent set	0.1
Krylov subspace size for outer iteration	20
Outer fgmres iteration	500
Number of processors in the x direction for PDE problem	2
Number of processors in the y direction for PDE problem	2
$xnmesh$, $n_x = xnmesh \times p_x$ for PDE problem	400
$ynmesh$, $n_y = ynmesh \times p_y$ for PDE problem	400

In Figure 4.2, we have injected errors in the application of the preconditioner in the FGMRES algorithm and we compare the effect of these errors when using `arms` and `arms_rbt`.

In Figure 4.3, we have injected errors both in the *outer matrix-vector* operation and the preconditioner application and we compare the effect of these errors when using `arms` and `arms_rbt`.

We observe that, for each type of perturbation, the extra number of iterations due to errors is more significant for `arms` than for `arms_rbt`. This illustrates that, for the 2D elliptic test problem, `arms_rbt` is more resilient than `arms` to errors when using the PBSFM. Thus in the next section, we will use the `arms_rbt` preconditioner to compare the two models NSFM and PBSFM.

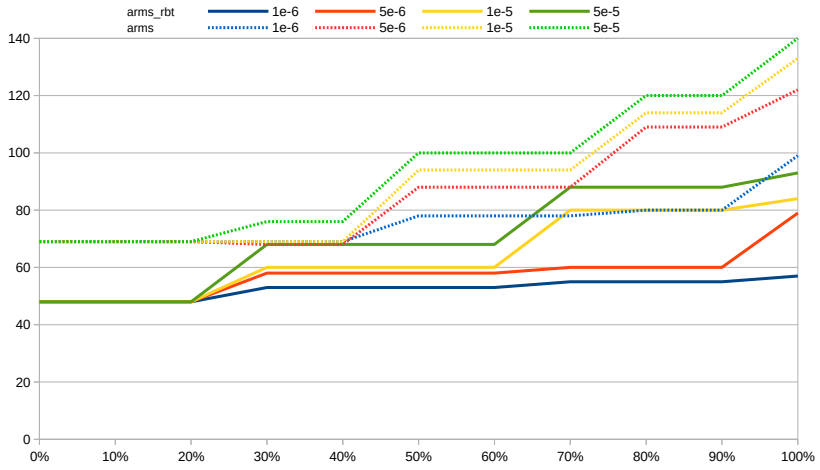


Figure 4.1: *Outer matrix-vector* perturbations in FGMRES, using `arms` and `arms_rbt`, on matrix 2D elliptic

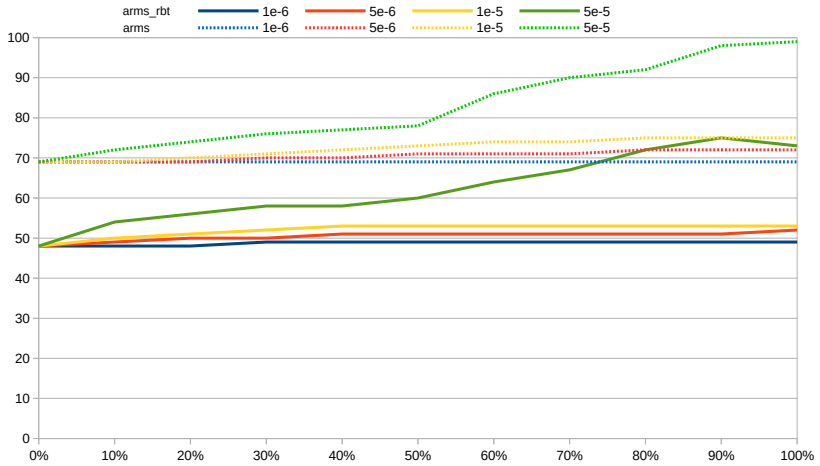


Figure 4.2: Preconditioner perturbations in FGMRES, using `arms` and `arms_rbt`, on matrix 2D elliptic

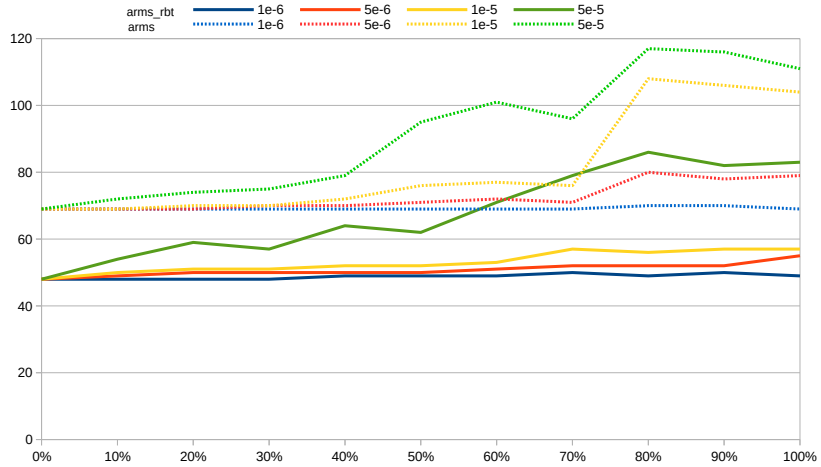


Figure 4.3: Combination of *outer matrix-vector* and preconditioner in FGMRES, using `arms` and `arms_rbt`, on matrix 2D elliptic

4.6 Comparison of NSFM and PBSFM fault models

In this section, we present the experiments that were conducted to compare the two fault models considered in our study. The results are presented as a comparison of the effects of a sticky soft fault as modeled by both the PBSFM and the NSFM fault models. Experiments are run on 400 cores of the computing platform Edison located at NERSC, which is a Cray XC30. It has a peak performance of 2.57 Petaflop/sec, with 134,064 cores and 5586 nodes. Each node has two sockets, with a 12-core Intel "Ivy Bridge" processor at 2.4 GHz. It has 357 Terabytes of memory and 7.56 Petabytes of disk storage.

4.6.1 Experiment description

For both the small and large problem, the performed tests included a fault-free run, a series of runs using the NSFM model and a series of runs using the PBSFM model. For the NSFM, the variable that will have the largest impact upon the fault injected is the scaling factor α while for the PBSFM the largest contributor to the impact of the fault is the size of the perturbation ϵ. For these experiments, three values of both α and ϵ were used: $\alpha = 1/2, 1, 2$, and $\epsilon = 1e^{-3}, 5e^{-4}, 1e^{-4}$. The NSFM runs using $\alpha = 1/2$, $\alpha = 1$, and $\alpha = 2$, were compared to the three variants of PBSFM that decreases the norm, that leaves the norm approximately the same (referred to as "neutral" in the remainder), and that increases the norm, respectively (see Section 4.4). Sticky faults were conservatively defined to be present during the first 1000 iterations of the iterative solver execution. For the fault-free test, the small problem converged in roughly 1500 iterations, and the large problem in approximately 3500 iterations. Note that all the runs of FGMRES were performed multiple times and the average was taken.

4.6.2 Results

The plots are only presented for the neutral norm variants of the fault models in Figures 4.4, 4.5, 4.6 and 4.7. To be specific, this involves the variants of the PBSFM where the norm remains approximately the same, and the version of the NSFM where the scaling factor α is set to 1 (but complete results for variants that decrease or increase the norm will be given in Tables 4.3 and 4.4). Each figure shows the number of iterations for five different fault methods: a nominal (fault-free) run, a PBSFM run with a "small" fault ($1e^{-4}$), a PBSFM run with a "medium" fault ($5e^{-4}$), a PBSFM with a "large" fault ($1e^{-3}$), and a NSFM run with $\alpha = 1$.

Figure 4.4 depicts the effects in terms of iteration count for the various soft faults injected into the outer matrix-vector operation of the FGMRES algorithm, when solving the small problem. In this figure, we observe that for the neutral variants, for both the `arms_rbt` and ILUT preconditioners, the NSFM has a more negative effect on the convergence of the FGMRES algorithm than the PBSFM. For instance, compared to the fault-free runs, the NSFM runs needed more than 1000 additional iterations to converge for both preconditioners while the additional number of iterations is at most around 150 for the different PBSFM variants. Figure 4.5 shows, for the small problem, the results when the faults are injected into the vector resulting from the preconditioner application.

Figure 4.6 displays the number of iterations to convergence when injecting faults into the outer matrix-vector operation for the large problem. As in Figure 4.4, the results in Figure 4.6 show a steady increase in the delay in the convergence of FGMRES from the nominal case to the PBSFM cases (ordered by the increasingly sized faults), then to the faults simulated by the NSFM case. The plots in Figure 4.7 depict the injection of faults into the result of the preconditioning operation for the large problem.

For a fault-free case, the FGMRES algorithm converged in fewer iterations when using the `arms_rbt` preconditioner compared to the ILUT preconditioner. This remained true when faults were injected into the application of the preconditioner,

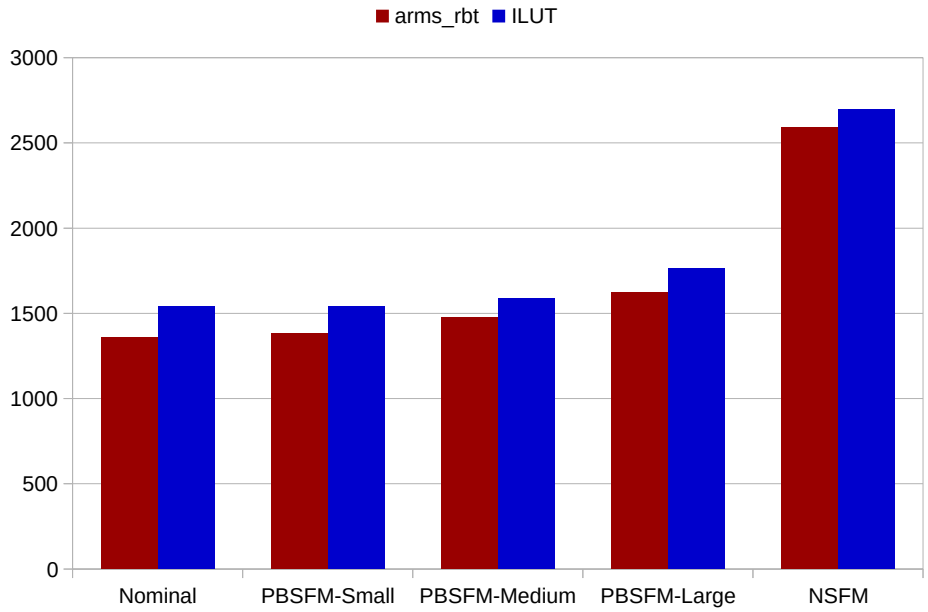


Figure 4.4: Number of iterations for the small problem for soft faults injected at the outer matrix-vector operation using `arms_rbt` and ILUT preconditioners.

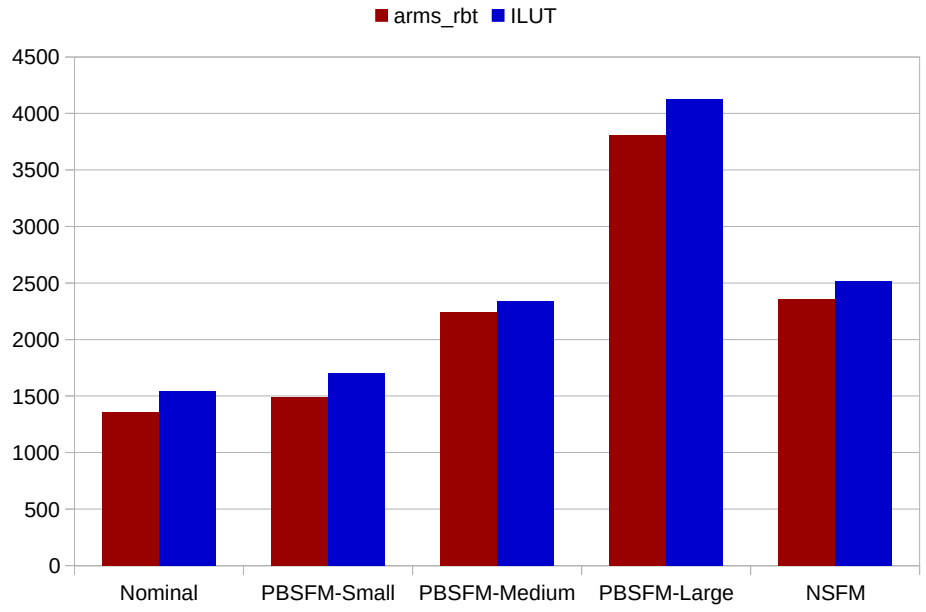


Figure 4.5: Number of iterations for the small problem for soft faults injected, at the application of the preconditioner using `arms_rbt` and ILUT preconditioners.

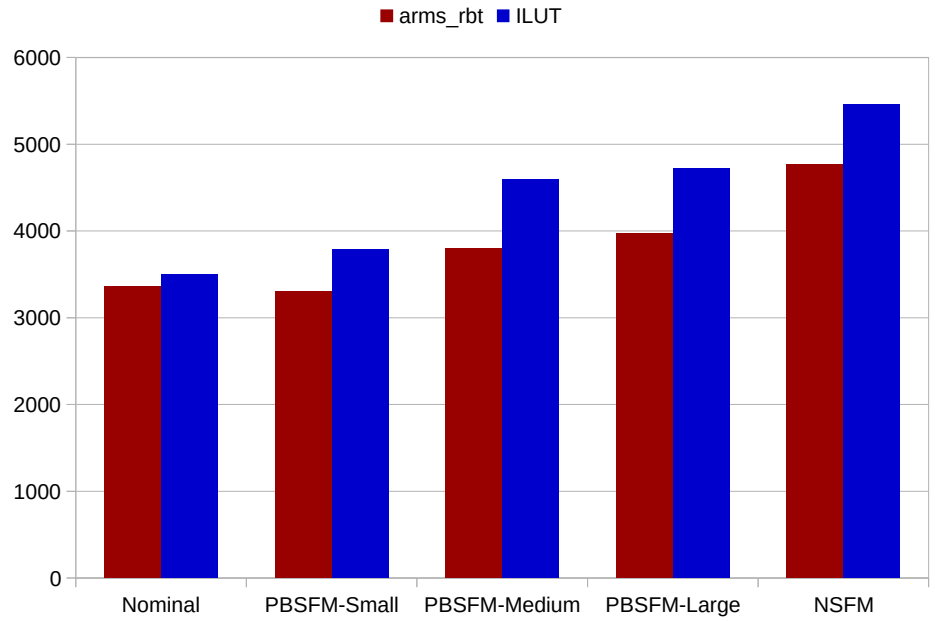


Figure 4.6: Number of iterations for the large problem for soft faults injected at the outer matrix-vector operation using `arms_rbt` and ILUT preconditioners.

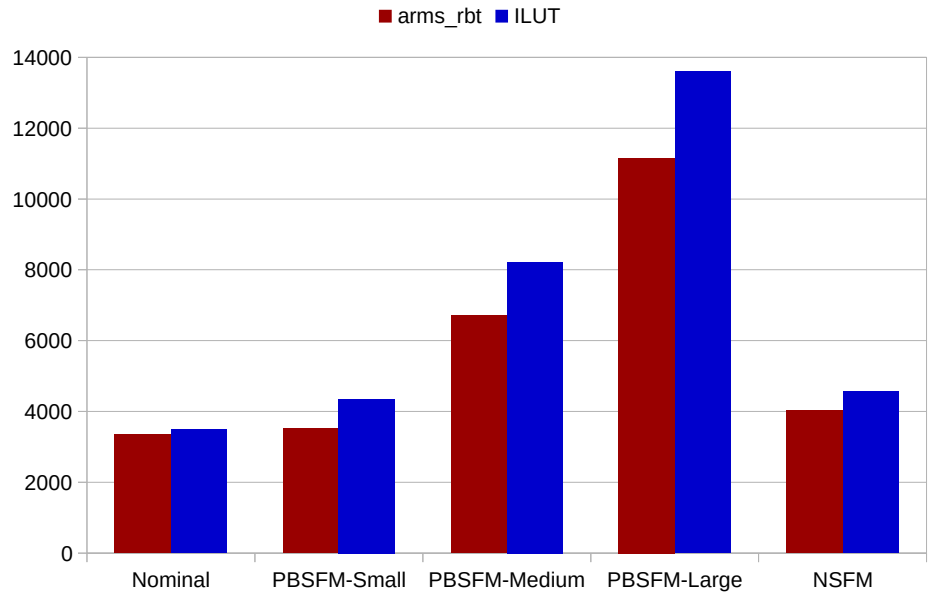


Figure 4.7: Number of iterations for the large problem for soft faults injected at the application of the preconditioner using `arms_rbt` and ILUT preconditioners.

		$\|\|\ \|\|_2$	Nominal		PBSFM-Small		PBSFM-Medium		PBSFM-Large		NSFM	
			SP	LP	SP	LP	SP	LP	SP	LP	SP	LP
ILUT	matvec	=	1542	3496	1380	3300	1477	3797	1624	3969	2590	4768
		−	1542	3496	2236	3807	2318	4170	2352	4380	2565	4660
		+	1542	3496	2241	3603	2326	4140	2358	4386	2637	4788
	precond	=	1542	3496	1487	3523	2243	6703	3811	11156	2355	4022
		−	1542	3496	1499	3280	2155	5163	2782	7639	2324	4093
		+	1542	3496	1499	3518	2168	5162	2780	7735	†	†

Table 4.3: Full results with `ILUT` preconditioner for the small (`SP`) and large (`LP`) problems with the neutral, decrease, and increase norm variants.

		$\|\|\ \|\|_2$	Nominal		PBSFM-Small		PBSFM-Medium		PBSFM-Large		NSFM	
			SP	LP	SP	LP	SP	LP	SP	LP	SP	LP
arms_rbt	matvec	=	1359	3357	1538	3790	1585	4594	1764	4727	2698	5456
		−	1359	3357	2323	4199	2426	4810	2459	7639	2697	5375
		+	1359	3357	2339	3825	2423	4655	2459	5059	2646	5426
	precond	=	1359	3357	1700	4349	2336	8221	4125	13607	2518	4550
		−	1359	3357	1706	4010	2201	6063	2925	9492	2570	4493
		+	1359	3357	1657	3989	2205	6061	2927	9005	†	†

Table 4.4: Full results with `arms_rbt` preconditioner for the small (`SP`) and large (`LP`) problems with the neutral, decrease, and increase norm variants.

however the injection of faults into the outer matrix-vector operation caused FGMRES to converge in roughly the same number of iterations whether it was preconditioned with ILUT or with `arms_rbt`. This suggests that for faults occuring at the outer matrix-vector operation, the advantage of the `arms_rbt` preconditioner is not as present as it is elsewhere. Note that, for both ILUT and `arms_rbt` preconditioners, faults injected into the outer matrix-vector operation had a larger impact than did identical faults injected into the resulting vector from the preconditioner application. Similar results were obtained in [93]. In addition, the impact of the faults injected using each of the two soft fault models with effects on the norm seems to be more pronounced in the PBSFM case; although, this is clearly adjustable through the use of the parameters available to both soft fault models. For instance, using larger values for α in the NSFM may provide a better comparison.

Complete results, including PBSFM variants that decrease or increase the norm, are provided in Tables 4.3 and 4.4 for all the experiments. Indeed, these tables give the full results for the `ILUT` (Table 4.3) and `arms_rbt` (Table 4.4) preconditioners for the small (`SP`) and large (`LP`) problems with the neutral, decrease, and increase norm variants in rows represented by signs =, −, and +, respectively. The † symbol indicates that the corresponding solver does not converge. We recall that in NSFM, the cases =, −, and + correspond to $\alpha = 1$, $1/2$, and 2, respectively.

When comparing the two fault models presented here directly, it is evident that

the NSFM has a larger negative impact on the convergence of the iterative FGMRES than the PBSFM in most scenarios. In every instance tested except for preconditioner faults on the larger problem size, the comparable version of the NSFM delayed convergence longer than the PBSFM did. This is in part due to the fact that the NSFM moves the vector where a fault is injected much further from its original location than the PBSFM does (see, e.g.,Table 4.1). In summary, for recurring faults specifically, the PBSFM offers a greater level of fine-tuned control over the fault impacts. However, the size of the fault in the PBSFM does not seem to have large impact on the convergence of FGMRES in the runs that attempted to manipulate the norm.

4.7 Conclusion

In this chapter, we have illustrated how soft errors can affect the performance of the FGMRES algorithm, which is a key kernel of the pARMS solver. We used two types of error models for sticky soft faults, and we applied them to the FGMRES algorithm. First, we have evaluated the soft fault based called PBSFM on a 2D elliptic problems where errors are injected at some steps of the FGMRES algorithm, namely the *outer matrix-vector* operation and/or the preconditioner application using `arms` and `arms_rbt`. For this problem, we have shown that the preconditioner `arms_rbt` is more resilient than $ILUT$ to such errors. In the future, it would be beneficial to better quantify the potential impact of both sticky and persistent faults that originate in a real-world environment. It may also be helpful to consider a wider range of scaling factors for the NSFM, as well as a wider range of fault sizes for the PBSFM in order to cover a larger spectrum of potential impacts due to the presence of faults.

Conclusion and future work

Conclusion

The first part of this thesis was devoted to a general presentation of concepts, software and tools that have been used for this PhD work. More specifically, the first chapter recalled the main principles in solving dense or sparse linear systems and focused on (preconditioned) Krylov iterative methods, known for their robustness, including the FGMRES and pARMS solvers which are considered in our study. We also presented some existing software libraries, and the various storage formats for sparse matrices. Moreover, we detailed the targeted parallel architectures and the main paradigms to program these architectures.

Our main contributions are about the enhancement of the distributed memory sparse iterative solver pARMS:

The first contribution is related to the use of Random Butterfly Transformations (RBT) to improve the performance of the pARMS solver. We used the RBT method to solve the last Schur complement system in the application of the ARMS preconditioner, resulting in a new `arms_rbt` preconditioner. The experimental results showed an improvement of the convergence and accuracy on test matrices from the Davis collection. We also implemented a sparse variant of the RBT to address the case where the last Schur complement is too large to be randomized as a dense matrix, which resulted in faster execution time.

The second contribution concerns the use of GPU accelerators to improve the performance of the pARMS solver on current hybrid CPU/GPU architectures. We identified the functions of pARMS that can benefit from GPU computing on CPU/GPU hybrid architectures. We implemented a GPU version of our new `arms_rbt` preconditioner, and we integrated a MAGMA version of the ILUT preconditioner, which provided an acceleration up to 30% on pARMS simulations on test problems coming from real-world applications (EDF).

The third contribution is a study of the effects of soft errors in the FGMRES algorithm which is used in pARMS to solve the preconditioned system. We consider two types of soft models: the Perturbation Based Soft Fault Model (PBSFM) and the Numerical Based Soft Fault Model (NBSFM). We first evaluated the resilience of pARMS preconditioners using the PBSFM model on a test problem by quantifying the increase of the number of iterations with the increase of the error magnitude and we showed that `arms_rbt` is more resilient than `arms`. Then we applied the PBSFM and NSFM models on `arms_rbt` and ILUT on bigger test matrices and showed that FGMRES is more affected by NSFM than PBSFM model, and that `arms_rbt` is more resilient than ILUT.

Future work and perspectives

The sparse variant of RBT enabled us to enhance the `arms_rbt` preconditioning for some test matrices. The fact that we use only one recursion in the randomization process limits the increase of fill-in in the randomized matrix. However it would be useful to combine RBT with fill-reducing techniques (e.g., reverse Cuthill-McKee methods) in order to control the fill-in of the last Schur complement, and consequently reduce the factorization time of this matrix.

The hybrid CPU-GPU version of pARMS is limited to the two preconditioners `arms_rbt` and $ILU(0)$ and it is necessary in a future version to integrate other preconditioners in order to address a wider range of test matrices. Additional optimization can be obtained by using CUBLAS routines for some specific kernels since we can handle the memory copies explicitly and then more efficiently than MAGMA libraries. Also, a track to improve performance is the extended use of kernel fusion techniques [101] that merge into a single kernel consecutive vector operations sharing some of the input or output data, such that data, once loaded into the fast multiprocessor memory, is reused. This would reduce memory traffic and then improve the performance of our hybrid solver.

Tolerance to hard faults and recovery techniques were not considered in this PhD work. Future work would for instance concern the use of Interpolation Restart (IR) strategies, which recover from a node crash in parallel distributed systems. In the pARMS solver, hard fault failures could also be managed by recent techniques specific to MPI (e.g., User Level Failure Mitigation (ULFM) [102]). To address the case of damaged data, recovery techniques investigated in [103] could be implemented in pARMS.

Appendix

Appendix A

Dense and sparse routines from the MAGMA library

In this appendix, we describe the main routines of MAGMA, which are used in our experiments. They concern the factorization and the triangular solves for dense and sparse matrices. We also mention the different kernels involved in the Random Butterfly Transformation (RBT) approach.

Randomization, factorization and solve routines for dense linear systems

The routine `magmablas_dprbt_q` randomizes a square general matrix using RBT.
magmablas_dprbt_q (magma_int_t n, double $*dA$, magma_int_t $ldda$, double $*du$, double $*dv$, magma_queue_t $queue$)

n	[in]	The number of columns and rows of the matrix dA
dA	[in, out]	Double precision array, dimension (n, $ldda$), on entry, the n-by-n matrix dA, on exit $dA = duT * dA * d\_V$
$ldda$	[in]	The leading dimension of the array dA
du	[in]	Double precision array, dimension (n, 2), the $2 * n$ vector representing the random butterfly matrix U
dv	[in]	Double precision array, dimension (n, 2), the $2 * n$ vector representing the random butterfly matrix V
$queue$	[in]	Execution queue

The routine `magma_dgetrf_nopiv_gpu` computes an LU factorization of a general m-by-n matrix A without any pivoting.
magma_dgetrf_nopiv_gpu (magma_int_t m, magma_int_t n, magmaDouble_ptr dA, magma_int_t $ldda$, magma_int_t $*info$)

m	[in]	The number of rows of the matrix A
n	[in]	The number of columns of the matrix A
dA	[in, out]	Double precision array on the GPU, dimension ($ldda$, n), on entry, the m-by-n matrix to be factored, on exit, the factors L and U from the factorization $A = L * U$
$ldda$	[in]	The leading dimension of the array A
$info$	[out]	Return code

The routine `magma_dgetrs_nopiv_gpu` solves a system of linear equations $A * X = B$, $A^T * X = B$, or $A^H * X = B$ with a general n-by-n matrix A using the LU factorization computed by dgetrf_nopiv_gpu.
magma_dgetrs_nopiv_gpu (magma_trans_t *trans*, magma_int_t n, magma_int_t *nrhs*, magmaDouble_ptr dA, magma_int_t *ldda*, magmaDouble_ptr dB, magma_int_t *lddb*, magma_int_t $*info$)

trans	[in]	MagmaNoTrans: $A * X = B$ (No transpose) MagmaTrans: $A^T * X = B$ (Transpose) MagmaConjTrans: $A^H * X = B$ (Conjugate transpose)
n	[in]	The order of the matrix A
nrhs	[in]	The number of right hand sides
dA	[in]	Double precision array on the GPU, dimension (*ldda*, n). The factors L and U from the factorization $A = L * U$ as computed by dgetrf_nopiv_gpu
ldda	[in]	The leading dimension of the array A
dB	[in, out]	Double precision array on the GPU, dimension (*lddb*, *nrhs*), on entry, the right hand side matrix B, on exit, the solution matrix X
lddb	[in]	The leading dimension of the array B
$info$	[out]	Return code

Factorization and solve routines for sparse linear systems

Below are the MAGMA sparse routines that we used in our experiments.

The `magma_dcumilusetup` is a factorization routine that prepares the ILU preconditioner using the cuSPARSE library.
magma_dcumilusetup (magma_d_matrix A, magma_ d_preconditioner $*precond$, magma_queue_t *queue*)

A	[in]	Input matrix
precond	[in, out]	Preconditioner parameters
queue	[in]	Execution queue

The routine `dapplycumilu_l` performs the left triangular solves using the ILU preconditioner.
magma_dapplycumilu_l (magma_d_matrix b, magma_d_matrix $*x$, magma_d_preconditioner $*precond$, magma_queue_t *queue*)

b	[in]	RHS
x	[in,out]	Vector to precondition
precond	[in,out]	Preconditioner parameters
queue	[in]	Execution queue

The routine `dapplycumilu_r` performs the right triangular solves using the ILU preconditioner.

magma_dapplycumilu_r (magma_d_matrix b, magma_d_matrix $*x$, magma_d_preconditioner $*precond$, magma_queue_t $queue$)

b	[in]	RHS
x	[in, out]	Vector to precondition
$precond$	[in, out]	Preconditioner parameters
$queque$	[in]	Execution queue

Appendix B

Synthèse en français

Dans cette thèse de doctorat, nous abordons trois défis auxquels sont confrontés les solveurs itératifs d'algèbre linéaire dans la perspective des futurs systèmes exascale, visant à effectuer 10^{18} opérations arithmétiques par seconde (flop/s) vers 2020 - 2022. Le premier défi est d'accélérer la convergence vers la solution en utilisant des techniques innovantes au niveau algorithmique. Le deuxième défi est de tirer profit des accélérateurs (par exemple les GPU - Graphics Processing Units) pour améliorer la performance des calculs sur les architectures parallèles hétérogènes. Le troisième défi est d'étudier l'impact des erreurs plus fréquentes du fait de l'augmentation du parallélisme dans les super-calculateurs. Nous nous intéressons à l'étude des méthodes permettant d'accélérer la convergence et le temps d'exécution des solveurs itératifs pour les grands systèmes linéaires creux. Le solveur plus spécifiquement considéré dans ce travail est le "parallel Algebraic Recursive Multilevel Solver" (pARMS) qui est un solveur parallèle à mémoire distribuée basé sur les méthodes de Krylov et propose plusieurs types de préconditionneurs standards parmi lesquels Schwarz additif restreint (RAS), Jacobi parallèle par blocs (BJ), et un préconditionnement basé sur le complément de Schur (SCHUR).

Tout d'abord, nous proposons d'intégrer une technique de randomisation appelée "Random Butterfly Transformations (RBT)" qui a été appliquée avec succès pour éliminer le coût du pivotage dans la résolution des systèmes linéaires denses (généraux et symétriques indéfinis). Notre objectif est d'appliquer cette méthode dans le préconditionneur ARMS de pARMS pour résoudre plus efficacement le dernier système de complément de Schur dans l'application du processus récursif multi-niveaux. Les résultats expérimentaux montrent une amélioration de la convergence et de la précision par rapport aux implémentations existantes. En raison de possibles problèmes d'occupation mémoire pour certains problèmes tests, nous proposons également d'utiliser une variante "creuse" du RBT à laquelle est associée un solveur direct creux (SuperLU) pour la résolution du dernier système de complément de Schur, ce qui a pour effet d'améliorer le temps de calcul de la solution.

Ensuite, nous expliquons comment une approche non intrusive peut être appliquée pour implémenter des calculs GPU dans le solveur pARMS, plus particulièrement dans la phase de préconditionnement locale (ARMS ou factorisation LU incomplète) qui représente une partie importante du temps de résolution. Pour intégrer des noyaux GPU dans pARMS, nous utilisons la bibliothèque d'algèbre linéaire MAGMA développée pour les architectures hybrides comprenant des processeurs classiques et des accélérateurs de type GPU ou Xeon Phi. Puis nous comparons les versions purement CPU et hybrides CPU/GPU du solveur sur plusieurs problèmes tests issus d'applications physiques. Les expériences portant sur notre solveur hybride CPU/GPU qui utilise le préconditionnement ARMS combiné avec le RBT, ou

le préconditionnement ILU(0), montrent un gain de performance allant jusqu'à 30% sur les problèmes considérés dans nos expériences.

Enfin, nous étudions l'effet des fautes logicielles sur la convergence de la méthode itérative `GMRES flexible` (FGMRES) qui est utilisée pour résoudre le système préconditionné dans pARMS. Le problème ciblé dans nos expériences est un problème elliptique d'équation aux dérivées partielles sur une grille régulière à deux dimensions. Nous considérons deux types de préconditionneurs: une factorisation LU incomplète à double seuil (ILUT) et le préconditionneur ARMS combiné avec la randomisation basée sur le RBT. Nous considérons deux modèles de fautes logicielles différentes (modèle de faute logicielle numérique - NFSM - et modèle de faute logicielle basée sur les perturbations - PBSFM), pour lesquels nous perturbons la multiplication matrice-vecteur et l'application du préconditionneur dans l'algorithme FGMRES. Pour chacun de ces modèles, nous comparons l'impact des fautes logicielles sur la convergence du solveur. L'effet négatif des modèles d'erreurs NSFM et PBSFM dépend de la position d'injection des erreurs dans le solveur FGMRES. L'injection d'erreurs répétitives sur un nombre donné d'itérations pendant l'application du préconditionneur a un effet plus négatif sur la convergence que la multiplication matrice-vecteur.

List of publications

[A] M. Baboulin, A. Jamal, and M. Sosonkina, "Using Random Butterfly Transformations in Parallel Schur Complement-Based Preconditioning," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, Lodz, Poland, Sept 2015, pp. 661–666.

[B] A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina, "A hybrid CPU/GPU approach for the parallel algebraic recursive multilevel solver pARMS," in *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Timisoara, Romania, Sept 2016, pp. 411–416.

[C] E. Coleman, A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina, "A Comparison of Soft-Fault Error Models in a Parallel Iterative Solver based on FGMRES," accepted at the 12th International Conference on Parallel Processing and Applied Mathematics (PPAM 2017), Sept. 10-13, 2017, Lublin, Poland.

Bibliography

[1] J. Dongarra, J. Kurzak, P. Luszczek, T. Moore, and S. Tomov, "Numerical algorithms and libraries at exascale," *HPCwire*, October 2015. [Online]. Available: http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/

[2] insideHPC, "What is high performance computing?" [Online]. Available: http://insidehpc.com/hpc-basic-training/what-is-hpc/

[3] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[4] J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*. SIAM, 1979, vol. 8.

[5] E. Andersen, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchev, and D. Sorensen, "LAPACK user's guide," 3rd. Ed. 1999 SIAM, Philadelphia, 1999.

[6] 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*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[7] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, "PLASMA users' guide," Technical report, ICL, UTK, Tech. Rep., 2009.

[8] 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," Paris-Sud University (France) and University of Tenessee (USA), Poster at Supercomputing (SC'08), Austin USA, November 15, 2008, http://www.lri.fr/ baboulin/SC08.pdf.

[9] P. R. Amestoy, J.-Y. L'Excellent, F.-H. Rouet, and W. M. Sid-Lakhdar, "Modeling 1d distributed-memory dense kernels for an asynchronous multifrontal sparse solver," in *High Performance Computing for Computational Science – 11th International Conference VECPAR, USA*. Springer, 2014, pp. 156–169.

[10] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with pardiso," *Computational Science—ICCS 2002*, pp. 355–363, 2002.

[11] X. Li, J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki, "SuperLU Users' Guide," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-44289, September 1999.

[12] Y. Saad and B. Suchomel, "ARMS: an algebraic recursive multilevel solver for general sparse linear systems," *Numerical Linear Algebra with Applications*, vol. 9, no. 5, pp. 359–378, 2002.

[13] Z. Li, Y. Saad, and M. Sosonkina, "pARMS: a parallel version of the algebraic recursive multilevel solver," *Numerical linear algebra with applications*, vol. 10, no. 5-6, pp. 485–509, 2003.

[14] G. H. Golub and C. F. V. Loan, *Matrix Computations*. Baltimore: The Johns Hopkins University Press, 1996, third edition.

[15] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[16] D. R. Kincaid and E. W. Cheney, *Numerical analysis: mathematics of scientific computing*. American Mathematical Soc., 2002, vol. 2.

[17] J. H. Wilkinson, "Error Analysis of Direct Methods of Matrix Inversion," *J. ACM*, vol. 8, no. 3, pp. 281–330, 1961.

[18] L. Grigori, J. Demmel, and H. Xiang, "Communication avoiding Gaussian elimination," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 29.

[19] ——, "CALU: a communication optimal LU factorization algorithm," *SIAM J. Matrix Anal. and Appl.*, vol. 32, pp. 1317–1350, 2011.

[20] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating Linear System Solutions Using Randomization Techniques," *ACM Trans. Math. Softw.*, vol. 39, no. 2, pp. 8:1–8:13, Feb 2013.

[21] T. A. Davis, *Direct methods for sparse linear systems*. SIAM, 2006.

[22] L. Giraud, A. Haidar, and S. Pralet, "Using multiple levels of parallelism to enhance the performance of domain decomposition solvers," *Parallel Computing*, vol. 36, no. 5, pp. 285–296, 2010.

[23] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*. NBS, 1952, vol. 49, no. 1.

[24] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[25] C. Lanczos, "Solution of systems of linear equations by minimized iterations," *J. Res. Natl. Bur. Stand*, vol. 49, pp. 33–53, 1952.

[26] H. A. Van der Vorst and C. Vuik, "The superlinear convergence behaviour of gmres," *Journal of computational and applied mathematics*, vol. 48, no. 3, pp. 327–341, 1993.

[27] U. M. Yang, "Parallel algebraic multigrid methods-high performance preconditioners," *Lecture Notes in Computational Science and Engineering*, vol. 51, p. 209, 2005.

[28] M. Benzi, "Preconditioning techniques for large linear systems: a survey," *Journal of computational Physics*, vol. 182, no. 2, pp. 418–477, 2002.

[29] Z.-H. Cao, "Constraint Schur complement preconditioners for nonsymmetric saddle point problems," *Appl. Numer. Math.*, vol. 59, no. 1, pp. 151–169, 2009.

[30] B. F. Smith, P. E. Bjørstad, and W. D. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. New York, NY, USA: Cambridge University Press, 1996.

[31] Y. Saad and M. Sosonkina, "Non-standard parallel solution strategies for distributed sparse linear systems," in *Parallel Computation: 4th International ACPC Conference*, ser. Lecture Notes in Computer Science, P. Z. *et al.*, Ed., vol. 1557. Springer-Verlag, 1999, pp. 13–27.

[32] "Matrix Algebra on GPU and Multicore Architectures MAGMA Web:." [Online]. Available: http://icl.cs.utk.edu/magma/

[33] D. Cline and Meyering, "Eispack is a collection of fortran subroutines that compute the eigenvalues and eigenvectors of nine classes of matrices," 1970s. [Online]. Available: http://www.netlib.org/eispack/

[34] J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley, "A Proposal for a Set of Parallel Basic Linear Algebra Subprograms," in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*. Springer, 1996, pp. 107–114.

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

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

[37] X. S. Li, "An overview of SuperLU: Algorithms, implementation and user interface," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 302–325, 2005.

[38] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner, "Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications," *Computer Graphics Forum*, vol. 32, no. 1, pp. 16–26, 2013.

[39] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013, pp. 273–282.

[40] Y. M. H. A. Brameller, Ronald Norman Allan, *Sparsity: its practical application to systems analysis*. Pitman, 1976.

[41] "ITPACKV 2D User Guide," https://www.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.

[42] A. Monakov, A. Lokhmotov, A. Avetisyan, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *HiPEAC*, 2010, pp. 111–125.

[43] H. Anzt, S. Tomov, and J. Dongarra, "Energy efficiency and performance frontiers for sparse computations on GPU supercomputers," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2015, pp. 1–10.

[44] P. Gepner and M. F. Kowalik, "Multi-Core Processors: New Way to Achieve High System Performance," in *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, Sept 2006, pp. 9–13.

[45] "The OpenMP API specification for parallel programming version 4.0," 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[46] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM, 2001, pp. 55–55.

[47] A. Moravánszky, "Dense matrix algebra on the GPU," *ShaderX2: Shader Programming Tips and Tricks with DirectX*, vol. 9, pp. 352–380, 2003.

[48] C. Nvidia, "Compute Unified Device Architecture programming guide," 2007.

[49] TOP500, "Top500 list in november 2016." [Online]. Available: https://www.top500.org/lists/

[50] M. Forum, "Mpi documents." [Online]. Available: http://mpi-forum.org/docs

[51] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, Sep 2004, pp. 97–104.

[52] C. P. C. Platform, "CUDA presentation." [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[53] M. W. Mahoney *et al.*, "Randomized algorithms for matrices and data," *Foundations and Trends in Machine Learning*, vol. 3, no. 2, pp. 123–224, 2011.

[54] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast Monte Carlo algorithms for matrices I: Approximating matrix multiplication," *SIAM Journal on Computing*, vol. 36, no. 1, pp. 132–157, 2006.

[55] J. T. Holodnak and I. C. Ipsen, "Randomized approximation of the Gram matrix: Exact computation and probabilistic bounds," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 1, pp. 110–137, 2015.

[56] A. Avron, P. Maymounkov, and S. Toledo, "Blendenpick: Supercharging LAPACK's least-squares solvers," *SIAM J. Sci. Comput.*, vol. 32, pp. 1217–1236, 2010.

[57] K. L. Clarkson and D. P. Woodruff, "Low rank approximation and regression in input sparsity time," in *STOC '13 Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 81–90.

[58] D. S. Parker, "Random butterfly transformations with applications in computational linear algebra," University of California Los Angeles, CA USA, Tech. Rep. CSD-950023, 1995.

[59] M. Baboulin, D. Becker, G. Bosilca, A. Danalis, and J. Dongarra, "An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems," *Parallel Computing*, vol. 40, no. 7, pp. 213–223, 2014.

[60] M. Baboulin, X. S. Li, and F.-H. Rouet, "Using random butterfly transformations to avoid pivoting in sparse direct methods," in *High Performance Computing for Computational Science - VECPAR 2014 - 11th International Conference, Eugene, OR, USA, June 30 - July 3, 2014, Revised Selected Papers*, 2014, pp. 135–144.

[61] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "A survey of recent developments in parallel implementations of gaussian elimination," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1292–1309, May 2014.

[62] Y. Saad and M. Sosonkina, "Distributed Schur Complement Techniques for General Sparse Linear Systems," *SIAM J. Scientific Computing*, vol. 21, pp. 1337–1356, 1999.

[63] X.-C. Cai and M. Sarkis, "A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems," *SIAM J. Sci. Comput.*, vol. 21, no. 2, pp. 792–797, Sep. 1999.

[64] Z. Li and Y. Saad, "SchurRAS: A Restricted Version of the Overlapping Schur Complement Preconditioner," *SIAM J. Sci. Comput.*, vol. 27, no. 5, pp. 1787–1801, Nov. 2005.

[65] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, Jan 2009.

[66] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[67] 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," in *International Conference on Computational Science (ICCS 2012)*, ser. Procedia Computer Science, vol. 9. Elsevier, 2012, pp. 17–26.

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

[69] M. Baboulin, A. Khabou, and A. Rémy, "A Randomized LU-based Solver Using GPU and Intel Xeon Phi Accelerators," in *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, 2015, pp. 175–184.

[70] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[71] "The University of Florida sparse matrix collection." [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices/

[72] X. S. Li and J. W. Demmel, "Superlu_dist: a scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Transactions on Mathematical Software*, vol. 29, no. 9, pp. 110–140, 2003.

[73] J. W. H. Liu, "Modification of the Minimum-degree Algorithm by Multiple Elimination," *ACM Trans. Math. Softw.*, vol. 11, no. 2, pp. 141–153, 1985.

[74] *CUsparse Toolkit Documentation v7.5*, NVIDIA Corporation, Sep 2015. [Online]. Available: http://docs.nvidia.com/cuda/cusparse/

[75] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.

[76] E. Chow, H. Anzt, and J. Dongarra, "Asynchronous iterative algorithm for computing incomplete factorizations on GPUs," in *High Performance Computing - 30th International Conference, ISC 2015, Frankfurt, Germany*, 2015, pp. 1–16.

[77] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[78] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *European Conference on Parallel Processing*. Springer, 2015, pp. 650–661.

[79] H. Anzt, M. Baboulin, J. Dongarra, F. Fournier, Y. Hulsemann, A. Khabou, and Y. Wang, "Accelerating the Conjugate Gradient Algorithm with GPU in CFD Simulations," *VECPAR*, 2016.

[80] *CUDA Toolkit Documentation v7.5*, NVIDIA, Mar 2016. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[81] M. Baboulin, A. Jamal, and M. Sosonkina, "Using random butterfly transformations in parallel Schur complement-based preconditioning," in *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Lódz, Poland*, Sep 2015, pp. 649–654.

[82] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[83] I. Tsukerman, "A class of difference schemes with flexible local approximation," *J. Comput. Phys.*, vol. 211, no. 2, pp. 659–699, 2006.

[84] M. Sosonkina and I. Tsukerman, "Parallel solvers for flexible approximation schemes in multiparticle simulation," in *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, Proceedings, Part I*, ser. Lecture Notes in Computer Science, V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, Eds., vol. 3991. Springer, 2006, pp. 54–62.

[85] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, 2014.

[86] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams *et al.*, "The landscape of parallel computing research: A view from Berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[87] J. Elliott, M. Hoemmen, and F. Mueller, "A Numerical Soft Fault Model for Iterative Linear Solvers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, 2015, pp. 271–274.

[88] ——, "Evaluating the impact of SDC on the GMRES iterative solver," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1193–1202.

[89] P. Bridges, K. Ferreira, M. Heroux, and M. Hoemmen, "Fault-tolerant linear solvers via selective reliability," *arXiv preprint arXiv:1206.1390*, 2012.

[90] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI," *Future Generation Computer Systems*, vol. 24:1, pp. 73–84, 2008.

[91] P. W. L. Chen, D. Tao and Z. Chen, "Extending checksum-based ABFT to tolerate soft errors online in iterative methods," in *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 344–351.

[92] J. Elliott, M. Hoemmen, and F. Mueller, "Tolerating silent data corruption in opaque preconditioners," *CoRR*, vol. abs/1404.5552, 2014.

[93] E. Coleman and M. Sosonkina, "Evaluating a perturbation-based soft fault model on preconditioned iterative methods," *Independent Study Report - Old Dominion University*, 2016.

[94] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 155–164.

[95] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," *preprint*, 2013.

[96] A. Edelman, "The mathematics of the Pentium division bug," *SIAM review*, vol. 39, no. 1, pp. 54–67, 1997.

[97] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Trace-based microarchitecture-level diagnosis of permanent hardware faults," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 22–31.

[98] F. Bower, D. Sorin, and S. Ozev, "Online diagnosis of hard faults in microprocessors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 2, p. 8, 2007.

[99] B. Schroeder, E. Pinheiro, and W. Weber, "DRAM errors in the wild: a large-scale field study," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1. ACM, 2009, pp. 193–204.

[100] E. Coleman and M. Sosonkina, "Evaluating a Persistent Soft Fault Model on Preconditioned Iterative Methods," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016, p. 98.

[101] J. Aliaga, J. Perez, and E. Quintana-Orti, "Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers," in *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, 2015, pp. 675–686.

[102] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[103] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon, "Numerical recovery strategies for parallel resilient Krylov linear solvers," *Numerical Linear Algebra with Applications*, vol. 23, pp. 888–905, 2016.