Minimizing communication for incomplete factorizations and low-rank approximations on large scale computers

Sébastien Cayrols

Sorbonne Université
Inria

Doctoral School **Sciences Mathématiques de Paris Centre**

University Department **Laboratoire Jacques-Louis Lions**

Thesis defended by **Sébastien Cayrols**

Defended on **18th February, 2019**

In order to become Doctor from Sorbonne Université

Academic Field **Applied Mathematics**

Minimizing communication for incomplete factorizations and low-rank approximations on large scale computers

Thesis supervised by Laura Grigori

Committee members

Referees	Patrick Amestoy	Professor at ENSEEIHT-IRIT	
	Timothy A. Davis	Professor at Texas A&M University	
Examiners	Frederic Hecht	Professor at Sorbonne Université	Committee President
	Jennifer Scott	Professor at STFC-RAL	
	Marc Baboulin	Professor at LRI	
Supervisor	Laura Grigori	Senior Researcher at Inria	

Sorbonne Université
Inria

Doctoral School **Sciences Mathématiques de Paris Centre**

University Department **Laboratoire Jacques-Louis Lions**

Thesis defended by **Sébastien Cayrols**

Defended on **18th February, 2019**

In order to become Doctor from Sorbonne Université

Academic Field **Applied Mathematics**

Minimizing communication for incomplete factorizations and low-rank approximations on large scale computers

Thesis supervised by Laura Grigori

Committee members

Referees	Patrick Amestoy	Professor at ENSEEIHT-IRIT	
	Timothy A. Davis	Professor at Texas A&M University	
Examiners	Frederic Hecht	Professor at Sorbonne Université	Committee President
	Jennifer Scott	Professor at STFC-RAL	
	Marc Baboulin	Professor at LRI	
Supervisor	Laura Grigori	Senior Researcher at Inria	

Sorbonne Université
Inria

École doctorale **Sciences Mathématiques de Paris Centre**

Unité de recherche **Laboratoire Jacques-Louis Lions**

Thèse présentée par **Sébastien Cayrols**

Soutenue le **18 février 2019**

En vue de l'obtention du grade de docteur de Sorbonne Université

Discipline **Mathématiques appliquées**

Minimisation des commmunications lors de factorisations incomplètes et d'approximations de rang faible dans le contexte des grands supercalculateurs

Thèse dirigée par Laura Grigori

Composition du jury

Rapporteurs	Patrick Amestoy	professeur à l'ENSEEIHT-IRIT	
	Timothy A. Davis	professeur au Texas A&M University	
Examinateurs	Frederic Hecht	professeur à Sorbonne Université	président du jury
	Jennifer Scott	professeur au STFC-RAL	
	Marc Baboulin	professeur au LRI	
Directeur de thèse	Laura Grigori	directeur de recherche à l'Inria	

Keywords: ilu factorization, low-rank approximation, preconditioner, reducing communication

Mots clés : factorisation ilu, approximation de rang faible, préconditionneur, réduction des communications

This thesis has been prepared at the following research units.

Laboratoire Jacques-Louis Lions

4 place Jussieu
75005 Paris
France

☎ +33 1 44 27 42 98
Web Site `http://ljll.math.upmc.fr/`

INRIA

2 Rue Simone Iff
75012 Paris
France

☎ +33 1 39 63 55 11
Web Site `http://www.inria.fr/`

Maison de la Simulation

Batiment 565
91191 Gif-sur-Yvette
France

☎ +33 69 08 08 00
Web Site `http://http://www.maisondelasimulation.fr/index.php/`

The user is often another piece of software rather than human and we consider what influence this may have on the design of our software.

Iain S. Duff

Acknowledgements

A mon sens, les remerciements sont comme un miroir reflétant le passé. Ils constituent une collection des moments marquants de ma thèse et je me suis attelé à les rédiger avec soin.

Tout d'abord, Laura, rien de tout ceci n'aurait pu se réaliser sans ta volonté, ta patience et ta bienveillance. Tu m'as permis d'intégrer le monde des Mathématiques appliquées ainsi que celui du calcul haute performance. Je me rappelle ce cours de master 2 que tu avais donné à Orsay, puis cette offre de stage de master au sujet de l'implémentation parallèle d'un préconditionneur appelé CA-ILU(0). A ce moment-là, le jargon *préconditionneur*, *communication avoiding*, ou encore *factorisation incomplète* m'était totalement inconnu. Malgré cela, tu as su prendre le temps de me guider et me donner les outils pour comprendre et évoluer dans ce domaine si dynamique et enrichissant. Je t'en serai toujours reconnaissant. Je veux aussi te dire merci d'avoir été si compréhensive. Ma situation personnelle a parfois été compliquée et tu m'as aidé et accompagné à ta manière sans restriction, MERCI !

En 2013, ma thèse commença à la fois au sein de l'équipe ALPINES et de la Maison de la Simulation, un jeune laboratoire créé sur le plateau de Saclay. Durant mes deux années au laboratoire, j'y ai rencontré beaucoup de gens intéressants : Michel Kern, Julien Bigot, Julien Derouillat, Pierre Kestener, Martial Mancip, Fabien Rozar ou encore Pierre-Elliot Becue, par exemple. Par ailleurs, je souhaite remercier France Boillod-Cerneux, Serge Petition et Nahid Emad pour leur soutien et leurs conseils. Bien entendu, je n'aurais pas pu bien aborder ma thèse sans les membres de l'équipe ALPINES, et notamment Sophie Moufawad et Rémi Lacroix qui m'ont accompagné à mes débuts. Sophie a su m'apporter ses lumières sur mes questions mathématiques, alors que Rémi se concentrait surtout sur les aspects d'implémentations. J'ai une pensée toute particulière pour l'humour de Frédéric Hecht et la gentillesse de Frédéric Nataf, ainsi que pour leur impressionnant savoir qu'ils ont bien voulu partager avec moi. Au cours de ma première année, j'ai eu l'opportunité de présenter l'avancée de mon travail à la conférence PMAA 2014, à Lugano. J'ai eu la chance d'y aller en accompagnie de Pierre Jolivet. Pierre, tu es un travailleur acharné et ton travail est le juste reflet de ton talent, faisant de toi un exemple. Tu as toujours été présent, même après la fin de ta thèse, et pour ça, je voudrais encore te remercier. Je remercie enfin tous les membres de l'équipe ALPINES à commencer par Xavier Clayes, ainsi que tous les autres membres, anciens comme actuels. Il avait aussi été entendu dès le départ que je serai une journée par semaine au laboratoire Jacques-Louis Lions. J'ai eu la chance d'y rencontrer Khashayar Dadras, qui m'a non seulement assisté dans l'utilisation de HPC2, mais aussi apporté un soutien de tous les instants. Je garde en mémoire nos grandes discussions !

Ma deuxième année a été marquée par l'invitation de Jim Demmel à rejoindre l'Aspire Lab à Berkeley pendant 5 mois, ainsi que le groupe Bebop. I was so impressed by discovering the university and the American way of life. Jim, I am so thankful that you invited me to join your bebop group. I learnt a lot by being part of your bebop group meeting, every week, and by taking your CS267 course every Tuesday and Thursday. At the same time, I really started to learn English and I think it helped me to go through my fear of speaking. I would like to thank you so much for all of that. J'ai aussi eu l'opportunité de travailler avec Radek Stompor sur un code d'astrophysique, dans le contexte du mapmaking. Je le remercie pour cela et aussi pour sa gentillesse. A mon retour en France, l'INRIA se préparait à déménager à Paris et Laura préparait l'organisation de SIAM PP 16 à Paris.

L'année suivante fut un tournant important lorsqu'il fut décidé que je serai de manière permanente à l'INRIA à partir du mois de mai 2016. J'y ai surtout fait deux très belles rencontres, Olivier Tissot et Hussam Al Daas. Que dire à leur propos ? Je pourrais rédiger un chapitre entier tellement vous m'avez apporté tous les deux. Je ne vous remercierai jamais assez pour tout ce que vous avez fait pour moi. Hussam, nos cours de math autant que nos discussions sur

la langue française sont ancrés pour toujours dans ma mémoire. Ton aide sur la preuve du preconditionneur autant que sur l'implémentation de TSQR te voudrait la médaille du dévouement. Olivier, trouver le nom de notre *library* te permettra surement un jour de passer à la postérité pour un nom si farfelus mais tellement génial. Tu as surtout été d'un soutien sans faille, une branche solide sur laquelle on peut s'appuyer sans crainte. Vous serez toujours bien plus que deux co-bureau pour moi et je vous en remercie encore !

La dernière année a vu arriver Jan Papez dans l'équipe. Jan, you became a real plus for the team and much more for our small co-worker group. Your kindness, your smile, and your empathy make you so special, and I will always consider you. Par ailleurs, j'ai une pensée toute particulière pour Zakaria et nos discussions de thèse, ainsi que Hao pour son style incomparable. A l'approche de la fin de ma thèse, Michel Kern s'est révélé être d'un grand soutien, me prodiguant de précieux conseils. Et j'ose croire que sans toi, les choses auraient été bien differentes et je t'en remercie pour cela.

To the members of my jury, I would like to thank you for taking of your time to be part of this adventure. Especially, I thank Patrick Amestoy and Tim Davis for reviewing my manuscript. You did an amazing job, and your feedback helped me to enhance the document. Jennifer Scott, I am so glad that you decided to join my jury. Marc, il était important pour moi que tu fasses partie de mon jury. Mon aventure a aussi commencé avec tes cours de master 2. J'ai encore en mémoire un de tes exercice de TD où nous devions essayer de paralléliser le produit matrice vecteur, sur le papier. Ce fût un réel déclencheur pour moi.

Enfin, je me dois de remercier Laurence Bourcier pour tout ce que tu as fait tout au long de ces quatre années. Ta gentillesse, ta bonne humeur, mais aussi ton aide en toutes circonstances font de toi une collaboratrice hors pair.

I must have a few words to Iain Duff, who offered me the possibility to finish writing and submitting my thesis in the best conditions ever. Iain, I owe you so much and I will thank you forever for that.

Tout au long de ces années, j'ai eu la grande chance d'être soutenu et encouragé par mes amis et ma famille. Mes pensées vont à Mathieu et Maud, Jeremie et Elodie, Sonia ainsi que Sébastien. Merci d'avoir été présents et vous-même pendant tout ce temps. Je pense aussi à mes beaux parents. Ils ont toujours été là pour ma famille et moi, et ce, dans tous les moments importants, un immense merci !

Je n'aurais pu réussir sans le soutien inconditionel de ma (futur) femme, Magali. Tu as su être ma motivation, mon aide de camp, ma première fan. Tu m'as permis de me dépasser, de ne rien lâcher et de finir dans les meilleurs conditions possibles, et pour cela, je t'en serai eternellement reconnaissant ! Je te remercie chaque jour de m'avoir donné le plus beau des fils, Louis. Sans vos sourires quotidiens, sans nos moments d'insouciance et de légèreté, les dernières années n'auraient pas été aussi merveilleuses.

Enfin, il me reste une personne à remercier et pas uniquement pour son soutien durant ces années de thèse mais aussi durant toute ma scolarité et bien plus encore. Merci maman pour avoir été toujours là pour moi. Tu seras toujours tellement importante dans ma vie et le plus grand des exemples à suivre. Je me souviens que durant mes longues études tu m'avais dit à plusieurs reprises : *il va bien falloir qu'un jour tu te mettes au travail.* Et bien, je crois qu'avec ce manuscrit je peux enfin dire que je m'y suis mis (un peu).
PS : j'ai une pensée pour mamie qui aurait été, je le crois, si fière de moi.

Minimizing communication for incomplete factorizations and low-rank approximations on large scale computers

Abstract

The impact of the communication on the performance of numerical algorithms increases with the number of cores. In the context of sparse linear systems of equations, solving $Ax = b$ on a very large computer with thousands of nodes requires the minimization of the communication to achieve very high efficiency as well as low energy cost. The high level of sequentiality in the Incomplete LU factorization (ILU) makes it difficult to parallelize. We first introduce in this manuscript a Communication-Avoiding ILU preconditioner, denoted CA-ILU(k), that factors A in parallel and then is applied at each iteration of a solver as GMRES, both steps without communication. Considering a row block of A, the key idea is to gather all the required dependencies of the block so that the factorization and the application can be done without communication. Experiments show that CA-ILU(k) preconditioner can be competitive with respect to Block Jacobi and Restricted Additive Schwarz preconditioners. We then present a low-rank algorithm named LU factorization with Column Row Tournament Pivoting (LU-CRTP). This algorithm uses a tournament pivoting strategy to select a subset of columns of A that are used to compute the block LU factorization of the permuted A as well as a good approximation of the singular values of A. Extensive parallel and sequential tests show that LU-CRTP approximates the singular values with an error close to that of the Rank Revealing QR factorization (RRQR), while the memory storage of the factors in LU-CRTP is up to 200 times lower than of the factors in RRQR. In this context, we propose an improvement of the tournament pivoting strategy that tends to reduce the number of Flops performed as well as the communication. A column of A is discarded when this column is a linear combination of other columns of A, with respect to a threshold τ. Extensive experiments show that this modification does not degrade by much the accuracy of LU-CRTP. Moreover, compared to the Communication-Avoiding variant of RRQR, our modification reduces the number of operations by a factor of up to 36.

Keywords: ilu factorization, low-rank approximation, preconditioner, reducing communication

Minimisation des commmunications lors de factorisations incomplètes et d'approximations de rang faible dans le contexte des grands supercalculateurs

Résumé

L'impact des communications sur les performances d'un code d'algèbre linéaire augmente avec le nombre de processeurs. Dans le contexte de la résolution de systèmes d'équations linéaires creux, la résolution de $Ax = b$, sur une machine composée de milliers de noeuds, nécessite la minimisation des communications dans le but d'atteindre une grande efficacité tant en terme de calcul qu'en terme d'énergie consommée. La factorisation LU, même incomplète, de la matrice A est connue pour être difficilement parallélisable. Ce manuscrit présente CA-ILU(k), un nouveau préconditionneur qui minimise les communications autant durant la phase de factorisation que durant son application à chaque itération d'un solveur tel que GMRES. L'idée est de considérer un sous-ensemble de lignes de A et de lui adjoindre des données de A tel que la factorisation du sous-ensemble, ainsi que l'application des facteurs obtenus, se fait sans communication. Les expériences réalisées montre que CA-ILU(k) rivalise avec les préconditionneurs Block Jacobi et Restricted Additive Schwarz en terme d'itérations. Nous présentons ensuite un algorithme de rang faible appelé la factorisation LU couplée à une permutation des lignes et des colonnes, LU-CRTP. Cet algorithme utilise une méthode par tournoi pour sélectionner un sous-ensemble de colonnes de A, permettant la factorisation par bloc de la matrice A permutée, ainsi qu'une approximation des valeurs singulières de A. Les test séquentiels puis parallèles ont permit de mettre en évidence que LU-CRTP retourne une approximation des valeurs singulières avec une erreur proche de celle obtenue par la factorisation QR révélant le rang de la matrice (RRQR). En outre, l'espace mémoire occupé par les facteurs de LU-CRTP est jusqu'à 200 fois plus faible que dans le cas de RRQR. Toujours dans le cadre d'une approximation de rang faible, nous proposons enfin une amélioration de la stratégie de pivotage par tournoi qui réduit le nombre d'opérations effectuées ainsi que les communications. Une colonne de A est retirée de la méthode si elle est une combinaison linéaire des autres colonnes de A, suivant un critère τ. Des tests sur un grand nombre de matrices montrent que cette modification ne dégrade pas significativement la précision de LU-CRTP. En outre, cette modification appliquée à la variante de RRQR minimisant les communications réduit par un facteur de 36 le nombre d'opérations.

Mots clés : factorisation ilu, approximation de rang faible, préconditionneur, réduction des communications

Laboratoire Jacques-Louis Lions
4 place Jussieu 75005 Paris France

Contents

Chapter 1

Introduction

Many applications from different scientific domains need accurate and efficient methods to solve a system of linear equations or least square problems on large scale computers. In astrophysics, researchers study large phenomena as the Cosmic Microwave Background (CMB) by creating experiments that produce a large amount of data. For example, the replacement of satellites COBE and WMAP by Planck increases the amount of data generated. The satellite uses concurrently multiple detectors that scan the sky and record signals hundred times per second over many years. The obtained result consists of $10^{13} - 10^{14}$ time samples which correspond to petabytes of data (10^{15}). One particular operation is the map-making step that compresses the initial data into a smaller data set of size 10^6 to 10^8. This operation involves a significant amount of memory, a large amount of computation, driven by significant data movements. Such data transformation requires the solution of a generalized least squares problem, generally performed by using the preconditioned Conjugate Gradient approach, (Szydlarski et al., 2014; Papez et al., 2018). Obviously, future satellites will generate even more data that will require much more storage and even more computation. Started in 2007, the *International Thermonuclear Experimental Reactor* (ITER) project intends to create a stable tokamak nuclear fusion reactor. In such magnetic fusion devices, the challenge goes through the understanding of the plasma turbulence. Moreover, with the quality of the plasma determining the cost of a fusion reaction, international research groups started developing codes. In particular, the Gysela 5D code, based on a semi-Lagrangian scheme, models the electrostatic branch of the Ion Temperature Gradient turbulence in tokamak plasmas. The use of more accurate models coupled with more physics is bounded by the power of current computers. The results presented in (Grandgirard et al., 2016) show that the code has a relative efficiency of 90.9% on 458k cores in the case of a weak scaling. However, the relative efficiency drops from 89% on 16k cores to 60% on 64k cores, partly due to the field solver. Therefore intensive works still require to improve the global performance of the code. Climate modeling, global warming, forecast, etc., drive the development of codes towards increasing accuracy, strongly linked to the mesh size of the grid used. In 1990, project AMIP1 was using a grid resolution of $64 \times 32 \times 10$ that required a calendar year to complete a 10 years integration. Twenty years later, the Community Climate System Model developed CCSM5, a fully coupled atmosphere-ocean-sea-ice model that was able to complete 15 years per actual day, on a grid of dimension $384 \times 320 \times 40$. In 2011, the future global tropical cyclones activity was simulated by using 5.5 million processor hours with a mesh grid of 25 kilometers. Later on, the CAM5 model predicted a mean of 50 hurricanes per year while the observations were 47, between 1980 and 2005, (Zarzycki et al., 2016). The explicit resolution of cloud systems would require

a mesh size of one kilometer, with an estimated 28 PFlops(10^{15} *Floating point operations per second*), (Wehner et al., 2011). Other current scientific applications in the domains of biology as genomic or data analytics require the solution of problems of very large size. All share the same need of more computing power.

Starting in 1993, the top500 list records every six months the 500 most powerful supercomputers in the world (sorted following the score on the benchmark Linpack). In 2009, the first petascale supercomputer, Cray Jaguar, performed 1.75 PFlops. Since then, a run towards the exascale machine (10^{18} Flops) has been driving research. However, increasing the performance of such supercomputers by three orders of magnitude is very challenging. First, the energy consumption is a real issue. The current #1 supercomputer, named Summit, has a peak energy consumption of about 15 megawatts, that is equivalent to the maximum output of two Avelia Horizon high speed train (TGV), released in July 2018. Reaching the exascale would be equivalent to 8 Summits with a consumption equivalent to 1/8 of a nuclear reactor; this is not feasible in practice. In the past ten years, the power cost of data movement did not significantly improve and still stands so that the cost in terms of energy to data movement is much higher than performing a floating point operation (Yelick, 2018). Thus, the communication has a large impact on the energy consumption as well as the global performance, and corresponds to the second issue facing the exascale computation. Increasing the number of processors unit also increases the communication cost. Although the system peak (#Flops) increases by three orders of magnitude from a PetaFlop machine to an exaFlop machine, the latency of point-to-point communication is expected to be only reduced by a factor of two.

Many scientific applications as presented above require the solution of linear systems of equations $Ax = b$, where A is a matrix of dimension $m \times n$, x and b are $n \times 1$ vectors. Depending on the application, A can be either dense or sparse, *i.e.*, with a large number of zero entries in A. To solve such linear systems, we can use two different classes of solvers. On the one hand, we have direct solvers that are based on the factorization of A as `LU` or `QR` that offer a good accuracy but require a large amount of memory and computation. On the other hand, we can use iterative methods that usually use much less memory, but may not converge to the solution. Direct solvers are not highly scalable, the iterative solvers offer better scalability. As mentioned above, the size of the computers increases to address the issues of large applications. Therefore, in the context of numerical linear algebra, related algorithms need to be redesigned to reduce the impact of communication on performance in order to achieve high performance computing. This class of algorithm is usually referred to as *CA* that stands for Communication Avoiding (J. W. Demmel, Grigori, M. Hoemmen, et al., 2008). For example, the redesign of the `LU` and `QR` factorizations are `CA-LU` and `CA-QR` algorithms, respectively.

In this manuscript, we present our contribution to the numerical linear algebra domain. Its content is split into three topics, first, a new preconditioner based on incomplete LU factorization, second a low-rank decomposition based on LU factorization, third a library that has been used to develop all parallel codes presented in this manuscript.

We start with Chapter 2 that recalls fundamental properties of some numerical algorithms as `LU` and `QR` factorizations. We also give some details on the past and current supercomputers with a brief historical overview.

Chapter 3 introduces a preconditioner named Communication Avoiding ILU(k) (CA-ILU(k)). The purpose is to factor a sparse matrix A by blocks using incomplete LU (`ILU`) factorization in parallel so that the resulted factors are similar to the sequential case. Moreover, when CA-ILU(k) is used as a preconditioner in iterative solvers as GMRES, its application is performed without communication. We compare CA-ILU(k) with Block Jacobi and Restricted Additive Schwarz preconditioners. Although the memory consumption may be an issue, we observe that CA-ILU(k) can be very competitive on large problems.

Chapter 4 presents an LU factorization with Column Row Tournament Pivoting named LU-CRTP. This novel algorithm computes a low-rank approximation of a sparse matrix A by using a block LU factorization, where at each step, k selected columns of A are permuted and then factored until the desired rank is reached. The selection of these k columns is performed by the QR factorization with Tournament Pivoting (QRTP) that minimizes communication. The results show that the method is highly parallel and is able to compute an approximation of the singular values of A with a precision close to that of the QR factorization with Column Pivoting algorithm. Our parallel implementation of QRTP shows that strong scaling on up to 2048 MPI processes can be achieve.

Chapter 5 is dedicated to the improvement of the QR factorization with Tournament Pivoting algorithm which performs unnecessary computation in the context of low-rank approximation. We show that our contribution allows to bound the norm of the error and can be used to find a better low-rank approximation at lower cost. Sequential results show the overall gain obtained in LU-CRTP as well as the Communication-Avoiding variant of the Rank Reveling QR factorization (CARRQR).

Chapter 6 presents C Parallel Linear Algebra Memory Management (CPaLAMeM), a library developed during the thesis that was used to implement the codes presented in the following chapters. This library is a collection of routines to manipulate the internal representation of mathematical objects as sparse and dense matrices, and vectors. While offering basic operations as loading data in parallel, permuting a matrix or extracting submatrices, CPaLAMeM provides a collection of parallel routines in the domain of the linear algebra as a dot product, a sparse matrix dense vector product or a tall and skinny QR factorization implementation. The purpose of CPaLAMeM is to provide a library helping to develop parallel research codes, and having some tools to track memory leaks, memory consumption, and a timer module.

We complete this manuscript with a general conclusion and an overview of the on-going and future works in Chapter 7.

This thesis led to the following publications

Journal Paper

– L. Grigori, S. Cayrols, and J. Demmel. "Low Rank Approximation of a Sparse Matrix Based on LU Factorization with Column and Row Tournament Pivoting". In: *SIAM Journal on Scientific Computing* 40.2 (2018), pp. C181–C209. eprint: `https://doi.org/10.1137/16M1074527`

To be submitted

– S. Cayrols and L. Grigori. "CA-ILU(k): a Communication-Avoiding ILU(k) preconditioner". 2019

– S. Cayrols and L. Grigori. "Tournament pivoting based on $\tau - rank$ revealing for the low-rank approximation of sparse and dense matrices". 2019

Chapter 2

Preliminaries

Outline of the current chapter

Many applications require the solution of the system

$$Ax = b, \tag{2.1}$$

where A is a square matrix of dimension n, and b and x are vectors of size n. Equation (2.1) is solved either by using direct methods and triangular solves based on LU, or by using iterative methods as GMRES (Saad and Schultz, 1986), CG (Hestenes et al., 1952). The late methods are projection methods that compute an approximate solution x_m from an affine subspace $x_0 + \mathcal{K}_m$ of dimension m, where the Petrov-Galerkin condition $b - Ax_m \perp \mathcal{L}_m$ is satisfied, with $\mathcal{L}_m$ being another subspace of dimension m. The vector x_0 is an arbitrary initial guess to the solution, in the affine subspace. These methods are referred to as Krylov subspace methods when the subspace $\mathcal{K}_m$ is the Krylov subspace given by

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\}, \tag{2.2}$$

where $r_0 = b - Ax_0$. The Krylov subspace methods differ depending on the choice of $\mathcal{L}_m$. As presented in (Saad, 2003), GMRES and CG are based on $\mathcal{L}_m = A\mathcal{K}_m$, considering the Krylov subspace defined as $\mathcal{K}_m(A, r_0/\|r_0\|_2)$, where $\|r_0\|_2$ denotes the 2-norm of the residual. Some others can use the transpose of A to span a Krylov subspace.

Iterative methods build an approximate solution by computing at each iteration a new vector of the Krylov basis. However, the methods may not converge to the solution, depending on the matrix. That is, a matrix is considered as ill-conditioned, *i.e.* difficult to invert, if the ratio of its largest singular value to its smallest, named its condition number, is large. This value is one of the aspects that leads a Krylov subspace method to not converge (or to have a slow rate of convergence). In order to improve the convergence, the system (2.1) is multiplied by a matrix M, named a preconditioner. There are three methods to precondition a system :

1. Left preconditioning $M^{-1}Ax = M^{-1}b$,
2. Right preconditioning $AM^{-1}y = b$, with $y = Mx$,
3. Split preconditioning $M_1^{-1}AM_2^{-1}y = M_1^{-1}b$, with $y = M_2x$.

In each case, M is chosen so that the 2-norm of the product $M^{-1}A$ is close to 1. In other words, M is a good approximation of A. In the case of right preconditioning, the Krylov subspace basis presented in (2.2) is therefore rewritten as

$$\mathcal{K}_m(M, A, r_0) = span\{r_0, M^{-1}Ar_0, (M^{-1}A)^2r_0, \ldots, (M^{-1}A)^{m-1}r_0\}. \tag{2.3}$$

The construction of a preconditioner must be cheap in terms of computation and communication. Otherwise, the gain obtained from the reduction of the number of iterations may not balance the overcost of its construction and its application. The cheapest and most easy preconditioner to build is the diagonal of A, well-known as Jacobi preconditioner. Along with it, we can list Gauss-Seidel, successive over-relaxation (SOR), and Symmetric successive over-relaxation (SSOR). They are based on a decomposition of A into $A = D - E - F$, where E and F are strictly lower and upper triangular matrices, respectively, and D is a diagonal matrix. These are defined as

- Jacobi preconditioner: $M = D$,
- Gauss-Seidel: $M = D - E$ or $M = D - F$,
- SOR: $M = \frac{1}{\Omega}D - E$.

Another type of preconditioners are based on an approximation of the matrix A. These preconditioners compute an incomplete factorization of A, as Incomplete LU (ILU) for general matrices or Incomplete Cholesky (IC) for symmetric positive definite matrices. Both preconditioners are built such that the residual matrix $R = A - M$ has a 2-norm small enough to use M as a preconditioner and the application of M is easy enough. The IC preconditioner is obtained by factoring $M = LL^T$ using the incomplete Cholesky factorization, where the L factor is a lower triangular matrix. This preconditioner takes advantage of the symmetry of the system by storing only L in memory. On the other hand, the ILU preconditioner is obtained by computing the incomplete LU factorization of A such that $A = LU + R$ and $M = LU$, where the L factor is a lower triangular matrix with diagonal of 1, and the U factor is an upper triangular matrix. Note that when the 2-norm of R is 0, then the factorization is complete, and this corresponds to a direct method. We give more details of the ILU preconditioner in Section 2.1.2.

The next class of preconditioners is based on a decomposition of the matrix A into submatrices, where each submatrix is considered as a subsystem of (2.1). This approach is well-known as the Schwarz Method, where the domain of the problem is decomposed into smaller subdomains. The subdomains are processed concurrently and the computation of the global solution is ensured by a partition of unity. Two types of Additive Schwarz preconditioners are based on this

approach: the Additive Schwarz Method (ASM) and the Restricted Additive Schwarz (RAS). We give more details of both preconditioners in Section 2.1.1

The last type of preconditioners presented in this section is known as the SParse Approximate Inverse (SPAI). The idea is to choose a matrix $T \in \mathbb{S}$, such that it minimizes $\|I - TA\|_F = \|I - A^T T^T\|_F$, where $\|.\|_F$ is the Frobenius norm, I is the identity matrix, A comes from Equation (2.1), and $\mathbb{S}$ is a set of sparse matrices. Since $\|I - AT\|_F = \sum_{i=1}^{n} \|e_i - AT(:,i)\|_2^2$, where e_i is a canonical vector, the construction of $M^{-1} = T$ corresponds to solving n least square problems,

$$\min_{T \in \mathbb{S}} (\sum_{i=1}^{n} \|e_i - AT(:,i)\|_2^2) = \sum_{i=1}^{n} (\min_{T \in \mathbb{S}} \|e_i - AT(:,i)\|_2^2). \tag{2.4}$$

There are other classes of preconditioners as algebraic multigrid preconditioners as AGMG (Notay, 2010), BoomerAMG (Henson et al., 2002), algebraic multilevel preconditioners as MSLR (Xi et al., 2016), or deflation techniques (Nabben et al., 2006; Tang et al., 2010), that are not presented in this thesis.

2.1 Algebraic preconditioners

In the following, we give details of algebraic preconditioners that are used in this thesis. We start by presenting the block Jacobi and Additive Schwarz preconditioners, and then we present the incomplete LU factorizations that are of importance in this manuscript.

As introduced above, the Jacobi preconditioner, defined as $M_J = diag(A)$, is the simplest preconditioner to build. Its application consists in computing the inverse of each coefficient of M_J. A variant of Jacobi, named Block Jacobi (BJ), corresponds to inverting the diagonal blocks of A. Considering p blocks in A, the Block Jacobi preconditioner is written as

$$M_{BJ} = \begin{pmatrix} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{pp} \end{pmatrix}, \tag{2.5}$$

where A_{ii} is the i-th diagonal block of A. The usual application of BJ preconditioner to a vector $y = Ax_0$ is to factorize each diagonal block of M_{BJ} by using an LU-type decomposition so that

$$M_{BJ}^{-1} y = \begin{pmatrix} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{pp} \end{pmatrix}^{-1} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} \tag{2.6}$$

$$= \begin{pmatrix} L_{11}U_{11} & & & \\ & L_{22}U_{22} & & \\ & & \ddots & \\ & & & L_{pp}U_{pp} \end{pmatrix}^{-1} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} = \begin{pmatrix} U_{11}^{-1}L_{11}^{-1}y_1 \\ U_{22}^{-1}L_{22}^{-1}y_2 \\ \vdots \\ U_{pp}^{-1}L_{pp}^{-1}y_p \end{pmatrix} \tag{2.7}$$

Besides its natural implementation in parallel, this preconditioner is built and is performed without communication. Unfortunately, when the number of blocks increases, the size of the blocks decreases and so the quality of the BJacobi preconditioner decreases. By quality, we

mean that the 2-norm of the residual matrix $R = A - M_{BJ}$ increases and the convergence of the iterative method deteriorates.

2.1.1 Additive Schwarz methods

Additive Schwarz Method (ASM) is a domain decomposition method where the problem is partitioned into subproblems. The method solves the local subproblems and then glues together the local solutions using a partition of unity. That is, the domain Ω associated with the problem is partitioned into N subdomains Ω_i, with $1 \leq i \leq N$. Differently from Block Jacobi preconditioner, ASM preconditioner defines overlapping subdomains along with a partition of unity. A partition of unity relies on a matrix R_i that restricts Ω to Ω_i, the matrix associated with the subdomain Ω_i, and a set of indices $\mathcal{N}$ split into N subsets such that

$$\mathcal{N} = \bigcup_{i=1}^{N} \mathcal{N}_i. \tag{2.8}$$

This partition leads to the following property

$$I = \sum_{i=1}^{N} R_i^T D_i R_i, \tag{2.9}$$

where I is the identity matrix.

An overlapping decomposition of Ω is equivalent to a partition of a set of indices. In the following, we focus our discussion on one level of overlap. Consider a partition that satisfies Equation (2.8). Suppose that $\forall i, j \in \{1, \ldots, N\}, i \neq j, \mathcal{N}_i \cap \mathcal{N}_j$. That is, two subdomains Ω_i and Ω_j are disjoint. Let $\mathcal{N}_i^{\delta=1}$ be the set of indices of the subdomain $\mathcal{N}_i$ with its direct neighbors, corresponding to one level of overlap. The method then defines R_i as a restriction matrix from $\mathcal{N}$ to $\mathcal{N}_i^{\delta=1}$. It follows that the Additive Schwarz Method used as preconditioner is defined as

$$M_{ASM}^{-1} = \sum_{i=1}^{N} R_i^T (R_i A R_i^T)^{-1} R_i. \tag{2.10}$$

However, the overlapping techniques mean by definition that some indices of $\mathcal{N}$ belong to more than one subdomain. As a consequence, gluing together the indices may perturb the solution. To solve this, the Restricted Additive Schwarz (RAS) method defines D_i as the diagonal matrix related to $\mathcal{N}_i^{\delta=1}$. The coefficients of D_i can be computed following mainly two strategies. The first one can be seen as boolean, where the indices of $\mathcal{N}_i$ in D_i are 1, and 0 otherwise. That is, when the index belongs to a neighbor for the considered subdomain, its contribution into the solution vector is discarded. The other strategy counts the number of subdomains that share the same index in $\mathcal{N}$. That is, for $j \in \mathcal{N}$, $m_j = \{\forall i \in \{1, \ldots, N\}, j \in \mathcal{N}_i\}$. This leads to defining D_i as $(D_i)_{jj} = 1/m_j$, with $j \in \mathcal{N}_i$. It follows that the RAS preconditioner is defined as

$$M_{RAS}^{-1} = \sum_{i=1}^{N} R_i^T D_i (R_i A R_i^T)^{-1} R_i. \tag{2.11}$$

In the previous explanation, $\delta = 1$ corresponds to one level of overlap *i.e.*, the direct neighbors of the subdomain. It follows that $\delta + 1$ is the overlap given by δ and the neighbors of this overlap. The performance of both versions of Additive Schwarz preconditioners is related to the size of the overlap. Thus, when δ increases, the preconditioner tends to reduce the number of iterations

to converge toward the solution. Note that the Block Jacobi preconditioner corresponds to the special case of RAS with $\delta = 0$, *i.e.* no overlap.

In the last few decades, optimized Schwarz methods (OSM) have been designed, as J. L. Lions' algorithms, where the transmission condition is more sophisticated. Instead of solving a Dirichlet boundary value problem, OSM solves a Robin boundary value problem. For more details, see (Lions et al., 1988; Dolean et al., 2015; Haferssas et al., 2017).

2.1.2 Incomplete LU preconditioners

The last type of preconditioner that we describe in this chapter is based on the Incomplete LU factorization of A. This class of preconditioners is suitable for general matrices and its memory consumption can be monitored in different ways. Therefore, variants of ILU preconditioners have been developed based on how the fill-in during the factorization is handled. We first present the LU decomposition in its most simple way, *i.e.* without permutation. Then we focus on the variants ILU(0), ILU(k), ILU_TP and iterative ILU.

The LU factorization of a square matrix A of dimension $n \times n$ is a Gaussian elimination process such that $A = LU$, where L is a unit lower triangular matrix, and U is an upper triangular matrix. As presented in Algorithm 2.1, the complete factorization of the given A into LU is processed as follows. Note that the MATLAB functions `triu`, `tril`, and `diag` (Lines 10, 11) return the upper triangular part, lower triangular part, and diagonal elements of A, respectively. The main issues in Algorithm 2.1 are the fill-in induced by the update in line 6, and the stability induced by the division of a small pivot in line 4. The choice of a good pivot is the most important aspect in LU factorization. However, in the following, we are interested in the fill-in aspect, that leads to several variants of ILU preconditioners. Nevertheless, the factorization of A is performed in-place so that the U factor is the upper triangular of the processed A, and the L factor is the lower triangular part of the processed A with diagonal elements equal to 1.

Algorithm 2.1 LU_factorization(A)

Input: $A \in \mathbb{R}^{n \times n}$ the square matrix to factor
Output: L the lower triangular matrix,
U the upper triangular matrix
1: Let n be the dimension of A
2: **for** $i = 2$ **to** n **do**
3: **for** $k = 1$ **to** $i - 1$ **do**
4: $a_{ik} = a_{ik}/a_{kk}$
5: **for** $j = k + 1$ **to** n **do**
6: $a_{ij} = a_{ij} - a_{ik}a_{kj}$
7: **end for**
8: **end for**
9: **end for**
10: $U = triu(A)$
11: $L = tril(A) - diag(A) + I$

When the matrix A is sparse, the fill-in that occurs in the factors L and U is evaluated as the ratio $nnz(L+U)/nnz(A)$, where $nnz(A)$ is the number of non-zero coefficients in the matrix A. The fill-in of the LU factorization depends on the structure of $A^T + A$. This observation can lead the factors to be so dense that they cannot fit in memory. The need of incomplete factorizations has, therefore, appeared to address this issue. The first incomplete factorization is known as the incomplete Cholesky factorization used as preconditioner in CG by J.A. Meijerink and H.A. Van Der Vorst in (Meijerink et al., 1977).

The Gaussian elimination algorithms are strongly sequential, by nature. As a consequence, Algorithm 2.1 offers poor parallelization in practice. Many types of research and reformulations tend to improve the parallelism, as SuperLU (Li, 2005). However, the communication aspect is often neglected. Indeed, the cost of communication greatly exceeds the cost of arithmetic operations on current and future architectures. Some communication avoiding algorithms address this issue. One of the most efficient is the Communication-avoiding LU (CALU) developed by L. Grigori *et. al* (Grigori, J. W. Demmel, et al., 2011), which performs the optimal amount of communication to compute an LU decomposition with partial pivoting. The method splits the matrix A into panels and iterates over them. At each iteration, the algorithm uses $TSLU$, a tall and skinny LU factorization to find k pivots. The k pivots are factored and used to update the trailing matrix. The $TSLU$ algorithm is a communication optimal algorithm that uses a flat or binary tree to perform a tournament pivoting and selects k pivots.

ILU(0) factorization

The simplest way to handle the fill-in in the factors is to discard all coefficients that were zero in the given matrix A. That is, an additional test is added in Algorithm 2.1, in the most inner loop. The update, line 6 is performed only if a_{ij} was already a non-zero in the given A. Thus, the sparsity pattern of $L + U$ is the same as A. However, the 2-norm of the residual matrix $R = A - LU$ defines how incomplete the factorization is, and shows that ILU(0) is the most basic incomplete factorization.

ILU(k) factorization

An extension of ILU(0) consists in handling the fill-in, based on a geometrical aspect. This variant is named ILU(k), where k corresponds to a distance in the graph associated with A. Let G be the graph of A, and i and j be two vertices of G, which have a unique number, usually the associated row index. During the update in Algorithm 2.1, line 6, the entry a_{ij} is added in A if there exists a path from the vertex i to the vertex j going through vertices numbered lower than both i and j. This condition replaces the test on the sparsity pattern of A introduced by ILU(0). It follows that the factorization of A using ILU(k) splits into two phases for performance. The first phase is named symbolic, where only the sparsity pattern of the L and U factors is computed. Then the second phase, named the numerical one, computes the coefficients of both factors. This offers the flexibility to determine the storage for L and U once only, based on the sparsity pattern of A. It follows that when an application generates several matrices with the same sparsity pattern, only the numerical phase is performed for each matrix. When the level of fill increases, the fill-in increases and the norm of the residual matrix $R = A - LU$ decreases. Note that when $k = n$, the returned factors correspond to the factors returned by Algorithm 2.1. Moreover, when $k = 0$, the algorithm is equivalent to the ILU(0) factorization presented above.

Dual-Threshold ILU factorization

Other variants of ILU are based on the numerical values of A. Saad introduces in (Saad, 1994) a variant referred to as ILUT(p,τ) for Dual-threshold ILU factorization. The key idea is to drop elements in L if they are less than a threshold, and to keep the p largest elements in both L and U. The update of Algorithm 2.1 is presented in Algorithm 2.2.

Compared to the ILU(k) variant, the sparsity pattern of L and U cannot be predicted or computed prior to the numerical phase. The behavior of this preconditioner depends on two parameters that can be hard to tune. In practice, the parameter τ may vary in the range $\{1e-3, 1e-6\}$. Although the 2-norm of the residual matrix $R = A - LU$ is smaller than for

ILU(k), experimental results have shown that the fill-in in the factors is so large that the time to apply the preconditioner is not negligible.

Algorithm 2.2 ILUT_factorization(A, p, τ)

Input: $A \in \mathbb{R}^{n \times n}$ the square matrix to factor,
p the maximum number of elements to store in a row of L and in a column of U,
τ the threshold to apply on the element of L
Output: L the lower triangular matrix,
U the upper triangular matrix

Let n be the dimension of A
Let w a vector of 0 of size n
for $i = 2$ **to** n **do**
  $w \leftarrow A(i, 1 : n)$
  for $k = 1$ **to** $i - 1$ **do**
    if $a_{ik} \neq 0$ **then**
      Compute $a_{ik} = a_{ik}/a_{kk}$
      Let $\tau_i = \tau * \|A(:, i)\|_2$ be the threshold of i-th row of the given A.
      if $a_{ik} \neq 0$ **then**
        $w(k + 1 : n) = w(k + 1 : n) - w(k) * U(k, k + 1 : n)$
      end if
    end if
  end for
  Keep in w the p largest coefficients
  $L(i, 1 : i - 1) = w(1 : i - 1)$
  $U(i, i : n) = w(i : n)$
  Reset w to 0
end for
$U = triu(A)$
$L = tril(A) - diag(A) + I$

Asynchronous iterative method for ILU factorization

A recent new approach to compute the Incomplete LU factorization has been developed by E. Chow in (Chow et al., 2015). The method is based on the following property, sustained by Proposition 10.4 in (Saad, 2003):

$$(LU)_{ij} = a_{ij}, (i, j) \in S, \tag{2.12}$$

where $(LU)_{ij}$ is the entry (i, j) located in row i and column j of the incomplete LU factorization, and S is a set of entries that define the sparsity pattern of the ILU factorization. The key idea is to consider the problem of factorizing a matrix as a set of equations to solve. The Equation (2.12) can be rewritten as a set of equations.

$$\sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, (i, j) \in S. \tag{2.13}$$

The Equation (2.13) leads to solving $|S|$ unknowns with $|S|$ equations, where $|S|$ denotes the cardinality of S. These last non-linear equations can be solved by using a fixed-point iteration such that, at iteration p, $x^{p+1} = G(x^p)$, where x is the unknown vector and G is the system composed of these equations. Therefore, each component of x^{p+1} can be computed in parallel.

The method is presented in Algorithm 2.3 that iterates over all entries in S, and computes the next value of the coefficients of L and U. The *sweep* corresponds to the replacement of x^p by x^{p+1} in the fixed-point iteration. This approach offers a fined-grained parallelism since the set of equations can be easily divided into subsets, well-balanced, and distributed over a large number of workers. One of the most interesting aspects of the method is that it is based on the set S. That is, the pattern of the L and U factors can be given by any methods. Thus, if the symbolic phase of the ILU(k) approach is used to form S, the returning L and U factors are expected to be similar to the ones returned by ILU(K). Furthermore, the threshold based ILU can also be applied during the iterations. In fact, the algorithm offers enough flexibility to update the solution between two sweeps. For much more details, see (Chow et al., 2015) and references therein.

Algorithm 2.3 Fined-grained Parallel Incomplete LU factorization

1: **for** $sweep = 1, 2$ **to** convergence **do**
2: **for** $(i, j) \in S$ **do**
3: **if** $i > j$ **then**
4: $l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) / u_{ij}$
5: **else**
6: $u_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right)$
7: **end if**
8: **end for**
9: **end for**

2.2 Rank revealing and low-rank approximation

From Principal Component Analysis (PCA) to Deep Neuronal Network (DDN), through preconditioners, an increasing number of applications require a low-rank factorization. In this section, we are interested in computing the rank of a matrix A as well as its low-rank approximation. The rank k of a matrix $A \in \mathbb{R}^{m \times n}$ is defined as the smallest integer such that A can be factored as

$$A = XY, \tag{2.14}$$

where $X \in \mathbb{R}^{m \times k}$, and $Y \in \mathbb{R}^{k \times n}$. From this definition, the low-rank approximation of a matrix A, with an accuracy τ, is defined as any matrix B such that

$$\|A - B\| \leq \tau, \tag{2.15}$$

where $\|\|$ can be any norm. In the special case of 2-norm, Equation (2.15) leads to that the optimal rank-k approximation of A is given by the Singular Value Decomposition (SVD), (Eckart et al., 1936). This decomposition can be written as

$$A = U \Sigma V^T = \sum_{i=1}^{n} \sigma_i u_i v_i^T, \tag{2.16}$$

where $U = \begin{bmatrix} u_1, & u_2, & \ldots, & u_n \end{bmatrix}$ and $V = \begin{bmatrix} v_1, & v_2, & \ldots, & v_n \end{bmatrix}$ have orthonormal columns, and Σ is a diagonal matrix whose coefficient, $diag(\Sigma) = \begin{bmatrix} \sigma_1, & \sigma_2, & \ldots, & \sigma_n \end{bmatrix}$, are the singular values of A sorted in a decreasing order. In the context of low-rank approximation, a truncated version of the Singular Value Decomposition can be easily extracted from Equation (2.16). Thus, the

truncated Singular Value Decomposition considers only the first k columns/rows of the factors in Equation (2.16). That is, the matrix $U_k\Sigma_k V_k^T$ is the rank-k approximation of A returned by the truncated SVD algorithm, so that

$$\|A - U_k\Sigma_k V_k^T\|_2 = \sigma_{k+1}. \tag{2.17}$$

Yet, this method is well-known to be costly and poorly parallel. Less expensive methods have then been studied as Lanczos methods (Lanczos, 1950) or Rank Revealing QR and LU algorithms. However, the Lanczos methods require too much communication. In the following, we first focus on the improvement of the SVD algorithm by using Randomized approach. Then, we give details of the Rank Revealing QR and LU algorithms.

2.2.1 Randomized algorithms

In the recent years, randomized methods were introduced to compute low-rank approximations of dense and sparse matrices. These methods are designed for high performance computing on modern architectures. The key idea is to multiply the matrix with a tall and skinny matrix so that the dimension of the resulting matrix is much smaller than the dimension of the original matrix. These methods rapidly became popular because of their ability to accelerate existing costly algorithms (Woolfe et al., 2008; Rokhlin et al., 2010). In that context, the most accurate algorithm to compute a low-rank approximation is the Singular Value Decomposition. The proposed solution is to mix randomized methods with the truncated SVD. This corresponds to generating a random matrix G, and then uses it to compute AG in order to reduce the size of the problem. Then the truncated Singular Value Decomposition of this later matrix, as in Equation 2.17, is performed at a reduced cost compared to the original algorithm.

Algorithm 2.4 shows that this procedure is easy to implement, has a very efficient arithmetic intensity, with a minimum communication. The algorithm takes as input the matrix A to factorize, k the requested rank for the factorization, and p the number of additional columns to generate in G. This algorithm returns the matrix $B = (QU_k)\Sigma_k V_k^T$. Halko *et. al.* have shown in (Halko et al., 2011) that the matrix B can be obtained with a bound $\|A - B\|_2 = O(\sigma_{k+1})$, associated with a failure probability of $5p^{-p}$.

Algorithm 2.4 RandomizedSampling(A, k, p)

Input: A a sparse matrix of dimension $n \times n$,
k the required rank of the factorization,
p the number of additional columns to generate in G
Output: $B = (QU_k)\Sigma_k V_k^T$ the singular value decomposition of A
1: Generate a random matrix G of dimension $n \times (k + p)$
2: Compute $QR = AG$
3: Compute the Singular Value Decomposition of $Q^T A$
4: Compute $U_k\Sigma_k V_k^T$, the truncated matrices obtained from Equation (2.16)

A better improvement of randomized algorithms is the *Randomized power method*, detailed in (Gu, 2015). The update algorithm, presented in Algorithm 2.5, takes in addition to the parameters of Algorithm 2.4, a parameter q corresponding to the number of products AA^T that are performed, line 2.

Algorithm 2.5 RandomizedPowerMethod(A, k, p q)

Input: A a sparse matrix of dimension $n \times n$,
k the required rank of the factorization,
p the number of additional columns to generate in G,
q the power for the method
Output: $B = (QU_k)\Sigma_k V_k^T$ the singular value decomposition of A
1: Generate a random matrix G of dimension $n \times (k+p)$
2: Compute $QR = (AA^T)^q AG$
3: Compute the Singular Value Decomposition of $Q^T A$
4: Compute $U_k \Sigma_k V_k^T$, the truncated matrices obtained from Equation (2.16)

2.2.2 Rank revealing factorizations

Alternatively to the Singular Value Decomposition of a matrix A, rank revealing algorithms have been targeted. This class of algorithms computes a rank-k factorization of A with a complexity of $O(mnk) \ll mn^2$ lower than the complexity of the SVD algorithm when $k \ll n$. In the following, we give details of the most common rank revealing QR and LU factorization algorithms.

Rank Revealing QR

We first introduce the QR factorization with Column Permutations of a matrix $A \in \mathbb{R}^{m \times n}$ of the form

$$AP_c = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{2.18}$$

where $Q \in \mathbb{R}^{m \times m}$ is an orthogonal matrix, $R_{11} \in \mathbb{R}^{k \times k}$ is an upper triangular matrix, $R_{12} \in \mathbb{R}^{k \times (n-k)}$, and $R_{22} \in \mathbb{R}^{(m-k) \times (n-k)}$. This factorization is said *rank revealing* (Chandrasekaran et al., 1994; Hong et al., 1992) if the following condition is satisfied

$$\sigma_{min}(R_{11}) \leq \frac{\sigma_k(A)}{p(k,n)}, \sigma_{max}(R_{22}) \leq \sigma_{k+j}(A) p(n,k), \tag{2.19}$$

where $p(n,k)$ is a low degree polynomial in k and n. This algorithm can reveal the rank of A but may not be stable enough because of the elements of $R_{11}^{-1} R_{12}$. To address this problem, a strong Rank Revealing QR factorization has been developed by M. Gu in (Gu and Eisenstat, 1996).

Theorem 1. *(Gu and Eisenstat (Gu and Eisenstat, 1996)) Let A be an $m \times n$ matrix and let $1 \leq k \leq \min(m,n)$. For any given parameter $f > 1$, there exists a permutation P_c such that*

$$AP_c = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{2.20}$$

where R_{11} is $k \times k$ and

$$(R_{11}^{-1} R_{12})_{i,j}^2 + \omega_i^2(R_{11}) \chi_j^2(R_{22}) \leq f^2, \tag{2.21}$$

for any $1 \leq i \leq k$ and $1 \leq j \leq n-k$, where $\chi_j(R_{22})$ denotes the 2-norm of the j-th column of R_{22}, and $\omega_i(R_{11})$ denotes the 2-norm of the i-th row of R_{11}^{-1}.

The inequality (2.21) bounds the singular values of R_{11} and R_{22} as in a rank revealing factorization, while it also bounds the absolute values of $R_{11}^{-1} R_{12}$.

Theorem 2. *(Gu and Eisenstat (Gu and Eisenstat, 1996)) Let the factorization in Theorem 1 satisfy inequality (2.21). Then*

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_{11})}, \frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq \sqrt{1 + f^2 k(n-k)}, \tag{2.22}$$

for any $1 \leq i \leq k$ *and* $1 \leq j \leq \min(m,n) - k$.

In the parallel case, both algorithms are not communication optimal. Thus, the communication avoiding Rank Revealing QR factorization (CARRQR) has been introduced in (J. Demmel, Grigori, Gu, et al., 2013) to compute a rank revealing factorization of a matrix A. This algorithm is a block algorithm that selects k columns from A, permutes them to the leading positions, and computes k steps of a QR factorization without pivoting. Then the algorithm iterates on the trailing matrix. The k columns are selected by using a tournament pivoting called the QR factorization with Tournament Pivoting (QRTP). Note that the communication optimality is handled similarly as in CALU algorithm. The CARRQR factorization satisfies, for $j = \{1, \ldots, n-k\}$, the following bound

$$\chi_j^2(R_{11}^{-1} R_{12}) + (\chi_j(R_{22})/\sigma_{min}(R_{11}))^2 \leq F_{TP}^2, \tag{2.23}$$

where χ_j is the norm of the j-th row of the given matrix. In this, F_{TP} depends on k, f, n, the type of the tree used in QRTP, and the number of iterations of CARRQR.

This leads to the following theorem.

Theorem 3. *Assume that there exists a permutation* P_c *for which the QR factorization*

$$AP_c = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{2.24}$$

where R_{11} *is* $k \times k$ *and satisfies (2.23). Then*

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_{11})}, \frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq \sqrt{1 + F_{TP}^2(n-k)}, \tag{2.25}$$

for any $1 \leq i \leq k$ *and* $1 \leq j \leq \min(m,n) - k$.

Rank Revealing LU

Factorizing a matrix A using the QR factorization is not efficient in terms of computation and memory consumption in the sparse case. Since R is the Cholesky factor of $A^T A$, it is expected to be dense and even denser than the LU factorization. Thus using the Rank Revealing QR algorithm is not suitable for sparse large matrices. In (Pan, 2000), C.-T. Pan has proved the existence of a Rank Revealing LU factorization. He shows the Schur complement factorization is equivalent to a Generalized LU(k) factorization assuming there always exist permutation matrices Γ_1 and Π_1 for any matrix $A_{11} \in \mathbb{R}^{k \times k}$ such that $\Gamma_1 A_{11} \Pi_1 = L_{11} U_{11}$. Its main result is presented in the following theorem.

Theorem 4. *(C.T Pan (Pan, 2000)) For a matrix* $A \in \mathbb{R}^{n \times n}$ *and any integer* $k (1 \leqslant k < n)$, *there exist permutation matrices* Γ *and* Π *such that*

$$\Gamma^T A \Pi \equiv \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} I_k & 0 \\ Z & I_{n-k} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ 0 & U_{22} \end{bmatrix} \tag{2.26}$$

where $Z = B_{21}B_{11}^{-1}$, $U_{22} = B_{22} - ZB_{12}$ *and*

$$\sigma_k(A) \geqslant \sigma_{min}(B_{11}) \geqslant \frac{1}{k(n-k)\mu^2 + 1}\sigma_k(A) \tag{2.27}$$

$$\sigma_{k+1}(A) \leqslant \|U_{22}\|_2 \leqslant (k(n-k)\mu^2 + 1)\sigma_{k+1}(A) \tag{2.28}$$

Similarly to the Rank Revealing QR factorization, this algorithm is not communication optimal. Thus a variant named Communication avoiding LU factorization with panel rank revealing pivoting (CALU_PRRP) has been developed in (Khabou et al., 2013). This algorithm, based on the Strong Rank Revealing QR factorization, is more stable, especially on pathological cases. Moreover, it also uses a tournament pivoting strategy to minimize communication.

2.3 Computer aspect

2.3.1 Half a century of processor improvement

Moore stated in (Moore, 1965) that the number of components in a single silicon chip, *i.e.*, a processor, will be doubling every year. A consequence of increasing the number of components in a chip was that the operating frequency could be increased while the power consumption stays the same, leading to better performance. In early 1970, vector processors were introduced to process several data at a time. The hyperthreading feature introduced by Intel 2002 tended to use a core more efficiently so that two threads were executed "quasi-"simultaneously on it. The Moore's law held until 2005. At that time, the processors reached the energy wall, meaning that increasing the frequency would correspond to a dissipative energy equivalent to a nuclear reactor. The vendors then started increasing the number of cores on a chip instead of the frequency. This brought many changes especially in the management of the memory and led to many different architectures. The memory was then accessible either with constant time, denoted by Uniform Memory Access (UMA), or by a non-uniform way, denoted by Non-Uniform Memory Access (NUMA). In the latter case, the cores were gathered into an island, also known as a NUMA node, where each core owns a small memory exclusively, and shares larger memory. Several levels of memory, known as cache, were used to improve performance so that the access to the levels 1 and 2 (L1 and L2) was faster than the access to the third level (L3) or the Random Access Memory (RAM). In 2008, Intel introduced in the Advanced Vector eXtensions (AVX) instruction set that manipulates registers of size 128 bits, where a single instruction is applied on multiple data (SIMD), and was further improved with AVX2 (256bits) and then with AVX-512. In 2011, the Fused Multiply-Add unit (FMA) was another improvement for the performance of a core since the multiplication of two floats and one addition is performed per cycle. The theoretical peak of performance, *i.e.*, the number of floating point operations per seconds, of a processor thus became a function of the frequency, the number of floats in a register, and the number of FMA units. The Intel Pentium 4, released in 2004, had a base frequency of 3.46 GHz and had implemented SSE2 instruction set extensions that yield a theoretical peak of $3.46 \times 2 = 6.92$ GFlop/s in double precision. The Intel Xeon 8176, Skylake architecture, released in 2017, consists of 2 NUMA nodes, 14 cores each, with a base frequency of 2.10 GHz (turbo frequency up to 3.8GHz) and 2 FMA units. In (Intel, 2018), the frequency of the processor depends on the number of active cores and the instructions set used. Thus, its theoretical peak performance using AVX2 instructions set equals to $1.7 \times 2 \times 8 \times 14 \times 2 = 0.76$ TFlop/s and up to 1.16 TFlop/s using AVX-512, both in double precision.

2.3.2 The bottleneck of future architectures

In the 90's, a natural way to improve the performance of a machine was to gather processors which are connected through a network. In the 2000's, the machines became more complex with the use of Graphic Processing Units (GPU) and Accelerators/co-processors. In 2007, only one supercomputer having such nodes has been recorded in the top500 list. Ten years later, more than 100 machines on the list consist of heterogeneous nodes. Although the most powerful machines in the world in June 1993 had a maximum of 2048 cores, we are now facing machines with more than 10 million cores. The current best machine, Summit hosted by Oak Ridge National Laboratory, is composed of 4 608 nodes, for a total of 2 282 544 cores, where the nodes are connected through a non-blocking fat-tree topology using dual-rail Mellanox EDR Infiniband. Several networking communication standards as Ethernet, Omnipath, Fiber channel and Infiniband are used by supercomputers to handle the communication between nodes. Third of the top 500 machines are using Gigabit Ethernet standard, mainly led by industry segment. In that case, communication cost is more important.

Although many efforts increased the Flops by a factor of 10^2 in 10 years (from $R_{peak} = 1.4$ PFlop/s in 2008 to $R_{peak} = 187$ PFlop/s in 2018), the communication is more and more a bottleneck. Table 2.1 presents the factor of improvement between the Petascale machine in 2009 and the expected exascale machine. We notice that the memory latency is slightly improved compared to the system peak. Unfortunately, the factor of improvement of the latency does not scale as the system peak. In the last 50 years, several topologies have been designed for mainly two purposes, first a low diameter and second a high path diversity. Table 2.2 presents the diameter complexity of the common topologies used. For a comparison of the performance of different topologies, see (Besta et al., 2014).

The machines becoming more and more complex with an increasing communication cost lead the HPC community to redesign algorithms to minimize the communication as well as using new paradigms of programning as task-based models.

	Petascale systems	Predicted exascale systems	Factor of improvement
System peak	2×10^{15}	10^{18}	$\sim 10^3$
Node Memory bandwidth	25 GB/s	0.4-4 TB/s	$\sim$ 10-100
Interconnect bandwidth	3.5 GB/s	100-400 GB/s	~ 100
Memory latency	100 ns	50 ns	~ 1
Interconnect latency	$1\mu s$	$0.5\mu s$	~ 1

Table 2.1 – Evolution of the principal indicators between Petascale systems and the predicted exascale systems, credit (Carson, 2017)

Topology	Linear Array	Ring	Torus	Binary tree	Hypercube	Fully connected
diameter	$N-1$	$N/2$	$\sqrt{N}$	log(N)	log(N)	1

Table 2.2 – Complexity of different topologies with respect to the diameter of the network.

Chapter 3

CA-ILU(k): a Communication-Avoiding ILU(k) preconditioner

Outline of the current chapter

Introduction

We recall that many scientific problems require the solution of sparse linear systems of the form

$$Ax = b, \tag{3.1}$$

where $A \in \mathbb{R}^{n \times n}$ is a general sparse matrix and $b \in \mathbb{R}^n$ is a vector. For very large systems, iterative methods, as Conjugate Gradient (CG) (Hestenes et al., 1952) or as Generalized Minimum Residual (GMRES) (Saad and Schultz, 1986), are widely used to solve Equation 3.1. The efficiency of iterative methods is measured by the number of iterations performed to compute the solution. The convergence of iterative methods depends on the condition number of A (J. Demmel, 1997). To accelerate the convergence, instead of solving the original system $Ax = b$, it is common to solve $M^{-1}Ax = M^{-1}b$, where $M \in \mathbb{R}^{n \times n}$ is called a preconditioner. M is considered as a good preconditioner when M is an approximation of A, $M^{-1}A$ has a smaller condition number than A, and $Mx = b$ is easy to solve. Preconditioners fall in general in two categories: geometric preconditioners that take into account properties of the problem, and algebraic preconditioners that have no assumption on the problem. Among algebraic preconditioners, incomplete factorizations of A such as Incomplete Cholesky for Symmetric Positive Definite matrices, or Incomplete LU for general matrices, are common methods to compute M. In this chapter we consider general matrices, and we focus on the incomplete LU factorization of A, denoted ILU, such that $M = LU$, where L and U are lower and upper triangular factor of A, respectively. We further refer the decomposition of the matrix $M = LU$ to as an ILU-based preconditioner. The simplest ILU is the so-called ILU(0) incomplete factorization, where L and U have the same nonzero structure as the lower and upper triangular part of A, respectively. It means that during the ILU(0) factorization of A, only existing entries of A are updated. However, this factorization can lead to a poor gain for the iterative solver. Therefore, several variants have been designed and can be grouped into two categories. The first category is composed of the incomplete LU factorizations based on a numerical threshold as ILU(τ), where computed entries smaller than τ are dropped during the factorization. The second category is composed of the incomplete LU factorizations based on the structure of A as ILU(k), where k is the level of fill-in. There also exist mixed approaches such as ILUT (Saad, 1994), an incomplete factorization which combines a limitation of the fill-in and a numerical threshold criterion. Incomplete LU factorizations based on numerical threshold are known to be more expensive than the incomplete factorizations based on the structure of A. Similarly to the classical LU factorization, the incomplete LU factorizations are difficult to parallelize and are intensively studied (Li, 2005; Amestoy et al., 2001). For distributed memory machines, communication cost and synchronization limit parallel efficiency of such factorizations. To enhance efficiency, overlapping techniques are used in Restricted Additive Schwarz (RAS) (Cai et al., 1999a) or s-step methods (Carson et al., 2013). The purpose is to duplicate some data in order to reduce communication. A class of algorithms, called Communication Avoiding, aims to redesign existing algorithms (J. W. Demmel, Grigori, Gu, et al., 2015; J. W. Demmel, Grigori, and Xiang, 2008; J. W. Demmel, Grigori, M. Hoemmen, et al., 2008). In this chapter, we propose a Communication-Avoiding ILU(k) preconditioner, denoted as CA-ILU(k). Considering a block rows of A, we duplicate additional rows of A on the processor that the block rows belong to, such that no communication occurs during the factorization of the block row. The additional rows form the overlap of CA-ILU(k). Since the size of the overlap can be large, we introduce a reordering of A to reduce the size of the overlap. We first present our algorithm to perform the ILU(k) factorization of A in parallel without communication. Then we study the algorithmic complexity of CA-ILU(k) used as a preconditioner and how its overlap can be bounded in the case of limited memory. We further show that there is a relation of inclusion between the overlap of RAS and the overlap of CA-ILU(k). In the experimental results section, the parallel runtime and the convergence of CA-ILU(k) are compared with those of block Jacobi and RAS preconditioners on several problems and for a number of processors increasing from 16 to 512. Moreover, we study the evolution of the overlap of CA-ILU(k) with respect to k.

3.1 Notations

Given a square matrix A of dimension $n \times n$, $a_{i,j}$ represents the element of A at row i and column j, $\text{nnz}(A)$ represents the number of nonzeros of the matrix A. Using MATLAB notation, $A(:, j)$ and $A(i, :)$ represent the j'th column and i'th row of A, respectively. Given a vector e of size n, $e(i)$ is the i'th entry of the vector. The concatenation of two vectors e_1 and e_2 of size n_1 and n_2, respectively, gives another vector $e = [e_1; e_2]$ of size $n_1 + n_2$. The notation $A(e_1, e_2)$ represents the matrix obtained after the permutation of the rows of A using e_1 and the permutation of the columns of A using e_2. If the size of a vector used to permute is smaller than the size of the matrix to permute, then the vector also extracts the concerned rows or columns to permute. Note that in the chapter, we use b, e, f, w, x, y and z to represent a vector. We define a set of indices $\mathcal{I} = \{i \in \mathbb{N}\}$ and two operations on $\mathcal{I}$, $\max(\mathcal{I})$ and $\min(\mathcal{I})$ that return the largest index and the smallest index in the set, respectively. The directed graph of A, denoted by $G(A)$ is defined as $(E(G(A)), V(G(A)))$, where $V(G(A)) = \{v_i \mid 1 \leq i \leq n\}$ is the set of vertices that correspond to the rows (or columns) of A, and $E(G(A))$ is a set of directed edges defined as

$$E(G(A)) = \{(v_i, v_j) \mid 1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j,\ a_{i,j} \neq 0\}.$$

Throughout the chapter, we denote by $v_i \in V(G(A))$ the vertex, associated with the i'th row (or column) in A, that belongs to the set of vertices of the graph of A. The number of elements in a set α is noted $|\alpha|$ and thus $|V(G(A))| = n$. A value is associated with each edge of the graph. In the case of the matrix A, we associate each edge (v_i, v_j) with the entry $a_{i,j}$ and so $(v_i, v_j) \equiv a_{i,j}$. Note that in the chapter, we also use t, u and v, to represent undistinct vertices that belong to $V(A)$. We use α, β, γ, δ, $\mathcal{R}$, $\mathcal{L}$ and $\mathcal{C}$ to represent a set of vertices.

We next introduce some definitions.

Definition 5. *Given a directed graph Ω, a subdomain Ω_i of Ω is a graph defined as follows*

$$V(\Omega_i) \subseteq V(\Omega)$$

and

$$E(\Omega_i) = \{(u, v) \mid u \in V(\Omega_i),\ v \in V(\Omega_i),\ (u, v) \in E(\Omega)\}.$$

Definition 6. *Given a directed graph Ω and two subdomains Ω_1 and Ω_2 of Ω, the union of the two subdomains denoted $\Omega_1 \cup \Omega_2$ is defined as follows*

$$V(\Omega_1 \cup \Omega_2) = V(\Omega_1) \cup V(\Omega_2) \subseteq V(\Omega)$$

and

$$\begin{aligned} E(\Omega_1 \cup \Omega_2) = {} & E(\Omega_1) \cup E(\Omega_2) \\ & \cup \{(u, v) \mid (u \in V(\Omega_1) \wedge v \in V(\Omega_2)) \vee (u \in V(\Omega_2) \wedge v \in V(\Omega_1)),\ (u, v) \in E(\Omega)\} \end{aligned} \tag{3.2}$$

Thus, if we have $\Omega_1 = G(A(1:10, 1:10))$ and $\Omega_2 = G(A(11:20, 11:20))$, then $\Omega_1 \cup \Omega_2 = G(A(1:20, 1:20))$.

Definition 7. *Given a directed graph Ω and p subdomains Ω_i of Ω ($i = \{0, \ldots, p-1\}$), the set $\{V(\Omega_0), \ldots, V(\Omega_{p-1})\}$ of disjoint subsets of $V(\Omega)$ is called a partition of Ω if $\forall i$, $V(\Omega_i) \neq \emptyset$ and $\bigcup_i V(\Omega_i) = V(\Omega)$.*

Definition 8. *Given a graph Ω and a set of indices $\mathcal{I}(\Omega) = \{1, \ldots, n\}$ associated with Ω, each*

vertex $u \in V(\Omega)$ *is numbered with an index of* $\mathcal{I}(\Omega)$ *such that*

$$\forall v \in V(\Omega),\ \mathcal{I}(u) \neq \mathcal{I}(v) \implies u \neq v.$$

The indices of $\mathcal{I}(\Omega)$ in the definition above are given by the row indices of A. Let $x(i)$ be the entry at position i of a vector x, and u its associated vertex in A such that $\mathcal{I}(u) = i$. The action of reordering the vertex u in Ω consists of changing the value returned by $\mathcal{I}(u)$. From the matrix point of view, it corresponds to changing the position of the entry $x(i)$ from i to j and so permuting the i'th row with the j'th row and the i'th column with the j'th column of A. As an example, let Ω be a graph of a matrix A and $\mathcal{I}(\Omega) = \{1, 2\}$. Let u and v be two vertices of $V(\Omega)$ with $\mathcal{I}(u) = 1$ and $\mathcal{I}(v) = 2$. Suppose that the reordering of vertices u and v such that u is numbered with the highest index in $\mathcal{I}(\Omega)$, we obtain $\mathcal{I}(u) = 2$ and $\mathcal{I}(v) = 1$. This reordering corresponds to a permutation vector $e = [2, 1]$ applied on the rows and columns of A, denoted also as $A(e, e)$.

Definition 9. *Given a directed graph* Ω*, partitioned into* p *subdomains* Ω_i *and the set of indices* $\mathcal{I}(\Omega)$ *associated with* Ω*,* $\mathcal{I}(\Omega)$ *is partitioned into* p *disjoint subsets such that* $\forall \Omega_i$*,* $\mathcal{I}(\Omega_i) \neq \emptyset$ *and*

$$\mathcal{I}(\Omega_0) \cup \mathcal{I}(\Omega_1) \cup \ldots \cup \mathcal{I}(\Omega_{p-1}) = \mathcal{I}(\Omega).$$

Each subset of indices $\mathcal{I}(\Omega_{i+1})$ *(*$i > 0$*) is defined recursively as*

$$\mathcal{I}(\Omega_{i+1}) = \{\max(\mathcal{I}(\Omega_i)) + 1,\ \ldots,\ \max(\mathcal{I}(\Omega_i)) + |V(\Omega_{i+1}|\}$$

where $\max(\mathcal{I}(\Omega_i))$ *is the largest index of* $\mathcal{I}(\Omega_i)$ *and the base case* $\mathcal{I}(\Omega_0) = \{1,\ \ldots,\ |V(\Omega_0)|\}$.

As a consequence of Definition 9, given two subdomains Ω_i and Ω_j with $j > i$, we have

$$\forall l \in \mathcal{I}(\Omega_j),\ \max(\mathcal{I}(\Omega_i)) < l$$

A path in a graph is a sequence of edges (v_1, v_2), (v_2, v_3), $\ldots$, (v_{m-1}, v_m), with $m \leq n$, where all edges and all vertices are distinct. The vertices v_1 and v_m are the initial vertex and the final vertex, respectively, of this path. The number of edges $m - 1$ defines the length of the path. Given a directed graph Ω and two vertices $u, v \in V(\Omega)$, vertex v is adjacent to vertex u if there exists a path from u to v in Ω of length one ($(u, v) \in E(\Omega)$). We denote the set of adjacent vertices of u in Ω by $\mathcal{N}_\Omega^1(u)$. The set of adjacent vertices of a set of vertices $\alpha \subseteq V(\Omega)$ in Ω is

$$\mathcal{N}_\Omega^1(\alpha) = \{v \in V(\Omega) \setminus \alpha \mid \forall u \in \alpha, (u, v) \in E(\Omega)\} \tag{3.3}$$

where $V(\Omega) \setminus \alpha$ is a set of vertices obtained by removing the vertices of α from $V(\Omega)$. Given a directed graph Ω and two vertices $u, v \in V(\Omega)$, the vertex v is reachable from the vertex u if there exists a path of length k in Ω from u to v, with $k > 0$.

The set of reachable vertices from u in Ω through paths of length at most k is referred to as $\mathcal{N}_\Omega^k(u)$. Similarly, the set of reachable vertices through paths of length at most k from a subset of vertices $\alpha \in V(\Omega)$ in Ω is

$$\forall k \geq 2,\ \mathcal{N}_\Omega^k(\alpha) = \mathcal{N}_\Omega^1\left(\mathcal{N}_\Omega^{k-1}(\alpha)\right) \cup \mathcal{N}_\Omega^{k-1}(\alpha). \tag{3.4}$$

It follows that we have the next relation of inclusion for $\alpha \subseteq V(\Omega)$

$$\forall k \geq 1,\ \mathcal{N}_\Omega^1(\alpha) \subseteq \mathcal{N}_\Omega^k(\alpha) \subseteq \mathcal{N}_\Omega^\infty(\alpha) \tag{3.5}$$

where $\mathcal{N}_\Omega^\infty(\alpha)$ corresponds to the set of vertices reachable through a path of any length.

Furthermore, we define the set of reachable subdomains from the vertices of a subdomain Ω_i. Given Ω the graph of a matrix A, partitioned into p subdomains, as in Definition 7, each subdomain Ω_i has a set of reachable subdomains in Ω defined as follow

$$\mathcal{N}_{\Omega}^{\infty}(\Omega_i) = \{\Omega_j \mid j \in \{0, \ldots, p-1\},\ j \neq i,\ u \in V(\Omega_i),\ v \in V(\Omega_j),\ v \in \mathcal{N}_{\Omega}^{\infty}(u)\}. \tag{3.6}$$

As a consequence, $\mathcal{N}_{\Omega}^{1}(\Omega_i)$ denotes the set of the subdomains which have at least one vertex adjacent to a vertex of $V(\Omega_i)$ in Ω.

Definition 10. *Given a directed graph Ω, it can be split in two subgraphs Ω^+ and Ω^- such that*

$$V(\Omega^+) = V(\Omega^-) = V(\Omega),$$
$$E(\Omega^+) = \{(u,v) \mid u,v \in V(\Omega),\ \mathcal{I}(u) < \mathcal{I}(v)\},$$
$$E(\Omega^-) = \{(u,v) \mid u,v \in V(\Omega),\ \mathcal{I}(u) > \mathcal{I}(v)\}.$$

Consider a matrix A of dimension $n \times n$ and its ILU(k) factorization $C = L + U - I_{n\times n}$, where $I_{n\times n}$ is the identity matrix of size $n \times n$. We denote as $\widehat{\Omega}$ the graph of the matrix C, where $\widehat{\Omega} \equiv G(C)$.

Definition 11. *Consider the graph Ω of a matrix A and the graph $\widehat{\Omega}$ of its ILU(k) factor C. Given the partition π of Ω into p subdomains as in Definition 7, $\widehat{\Omega}$ is partitioned using π such that for a subdomain $\widehat{\Omega}_i$ we have*

$$V(\widehat{\Omega}_i) = V(\Omega_i),$$
$$E(\Omega_i) \subseteq E(\widehat{\Omega}_i),$$
$$\mathcal{I}(\widehat{\Omega}_i) = \mathcal{I}(\Omega_i).$$

We denote as F_k the symbolic factor matrix returned by the symbolic ILU(k) factorization step of A. F_k stores the pattern of $L+U$. Each element f_{ij} represents the length of the shortest path in $G(A)$ from vertex v_i to vertex v_j going through vertices numbered lower than both $\mathcal{I}(v_i)$ and $\mathcal{I}(v_j)$. Thus the value associated with the edge (v_i, v_j) is equal to the shortest path from v_i to v_j. The value of each edge in $E(F_k)$ corresponds to the associated entry in F_k. From Definition 10, $G(F_k)^+$ is the graph of the upper triangular part of F_k, which corresponds to the factor U, that contains only edges (v_i, v_j) with $\mathcal{I}(v_i) < \mathcal{I}(v_j)$ and $G(F_k)^-$ is the graph of the lower triangular part of F_k, the factor L, that contains the remaining edges of $G(F_k)$.

3.2 CA-ILU(k) factorization

The purpose of CA-ILU(k) preconditioner is to compute the ILU(k) factorization of a matrix A in parallel and to apply it without communication. *LU*-typed factorizations are known to be poorly parallelizable. In the row-oriented version of the ILU(0) factorization of A, the update of row i depends on the previous rows j such that $\forall j < i$, $a_{i,j} \neq 0$.

In parallel, the computation of the entry $x(i)$ in $Ax = b$ involves communication with processors where the rows associated with the adjacent vertices of v_i belong to. This communication impacts the performance of both the factorization and the application of the preconditioner. We propose to solve this problem by gathering on the same processor the row i of A and all the

rows j that row i depends on. This raises the problem that the size of the gathered data can be arbitrarily large. As we show further on a 5-point stencil problem using a natural ordering, the size of the data can be equal to the number of vertices of A. To handle this problem, we propose to reorder A such that this size is reduced and even bounded under some criteria that we present further.

To solve $Ax = b$, GMRES computes at each iteration a new vector y_{i+1} of the Krylov basis by multiplying A with the previous vector y_i and then by orthogonalizing it against all previous vectors. This method is referred to as the matrix powers kernel $\{x, Ax, A^2x, \ldots, A^mx\}$ presented in (J. Demmel, M. Hoemmen, et al., 2008). Applying a preconditioner modifies the matrix powers kernel such that at iteration $i+1$, we compute $w = M^{-1}Ay_i$. The preconditioned matrix powers kernel is presented in Algorithm 3.1. Since the orthogonalization is not relevant here, we omit it in the algorithm.

Algorithm 3.1 preconditionedMPK(A, y_i, M)

This function computes the next Krylov basis vector y_{i+1} by computing Ay_i and then applying an ILU-based preconditioner M.

Input: $A \in \mathbb{R}^{n\times n}$,
y_i the i'th Krylov basis vector,
$M = LU$ the ILU-based preconditioner, where $L \in \mathbb{R}^{n\times n}$ and $U \in \mathbb{R}^{n\times n}$

1: Compute $f \leftarrow Ay_i$ /* *SpMv* */
2: Solve $f = Lz$ /* *Forward substitution* */
3: Solve $z = Uw$ /* *Backward substitution* */

Output: The vector $w = U^{-1}L^{-1}Ay_i$.

Here, w is obtained by a Sparse Matrix-Vector product (SpMv), a forward and a backward substitution required by the application of the preconditioner. The preconditioned matrix powers kernel is parallelizable by distributing A, L and U over p processors. Solving the entry $z(i)$ or $w(i)$ in the forward or the backward substitution, respectively, requires unknowns usually distributed among other processors. Based on Theorem I in (Parter, 1961), Gilbert et al. show in (Gilbert and Peierls, 1988) that the nonzeros of $U(:, i)$ correspond to the reachable vertices from v_i, $\mathcal{I}(v_i) = i$, in $G(A)$. This set can be found using a depth-first search or a breadth-first search starting with v_i in the graph of A. It follows that when all reachable vertices from a set of vertices are gathered on one processor, the factorization of the set can be done without communication. The parallel version involves communication in the three steps. Even if communication is overlapped with computation, synchronizations occur and the efficiency of the algorithm degrades in practice.

To call the matrix powers kernel on each subdomain without communication, we next outline the data involved in the three steps. Looking at Algorithm 3.1 from the bottom, it starts by solving $z = Uw$. To compute the entry $w(i)$, the backward substitution needs the rows of U corresponding to the reachable vertices of the vertex v_i in $\widehat{\Omega}^+$, denoted

$$\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_i).$$

The next step solves $f = Lz$. Similarly, the reachable vertices of the vertex v_i in $\widehat{\Omega}^-$ are those whose index is lower than i, equal to $\mathcal{N}^{\infty}_{\widehat{\Omega}^-}(v_i)$. In addition to the computation of $z(i)$, the algorithm has to compute the entries of the vector z that correspond to the reachable vertices of v_i in $\widehat{\Omega}^+$. Therefore, the required data that allows us to solve $w(i)$ from both $f = Lz$ and $z = Uw$ is

$$\mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(v_i \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_i)\right).$$

Definition 12. *Given a matrix A of size $n \times n$, a matrix $M = LU$, a vector y_i and the three equations of the preconditioned matrix powers kernel $f = Ay_i$ $f = Lz$ and $z = Uw$, the required data to compute $w(i)$, $i \in \{1, \ldots, n\}$, is defined as the CA-ILU(k) overlap of v_i and is equal to*

$$\mathcal{O}_{CAILU_k}(v_i) = \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_i) \bigcup \mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(v_i \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_i)\right).$$

Furthermore, the CA-ILU(k) overlap of a set of vertices $\alpha \subseteq V(A)$ is given by

$$\mathcal{O}_{CAILU_k}(\alpha) = \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(\alpha) \bigcup \mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(\alpha \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(\alpha)\right). \tag{3.7}$$

In the case of removing the communication during the product Ay_i, the entry $f(i)$ needs the adjacent vertices of v_i in Ω to be computed without communication. This means the adjacent vertices of the required data of the forward substitution step need to be added to the CA-ILU(k) overlap. It follows that the computation of v_i requires in addition to $\mathcal{O}_{\mathrm{CAILU}_k}(\alpha)$

$$\mathcal{N}^{1}_{G(A)}\left(v_i \cup \mathcal{N}^{\infty}_{G(L)}(v_i) \cup \mathcal{N}^{\infty}_{G(L)}\left(\mathcal{N}^{\infty}_{G(U)}(v_i)\right)\right).$$

Definition 13. *Given Ω the graph of a matrix A, partitioned into p subdomains, as in Definition 7, and the three equations of the preconditioned matrix powers kernel presented above, each subdomain Ω_i has a set of dependency subdomains denoted $\mathcal{D}(\Omega_i)$ and defined as*

$$\mathcal{D}(\Omega_i) = \{\Omega_j \mid u \in V(\Omega_i), v \in \mathcal{O}_{CAILU_k}(V(\Omega_i)), v \in V(\Omega_j), j \neq i\}$$

Consider the graph Ω of a matrix A and a subset of vertices $\alpha \in V(\Omega)$. Consider the LU factorization of $A(\mathcal{I}(\alpha), :)$. Gilbert shows in (Gilbert, 1994) that the required vertices to factor the i'th row of a matrix A are the reachable vertices from the vertex associated with this row in Ω^-. Therefore, to factor $A(\mathcal{I}(\alpha), :)$ requires the set of vertices given by $\omega = \mathcal{N}^{\infty}_{\Omega^-}(\alpha)$. The overlap of CA-ILU(k) as presented in Definition 12 contains the required vertices to factor $A(\mathcal{I}(\alpha), :)$ since $\mathcal{N}^{\infty}_{\Omega^-}(\omega) \subseteq \mathcal{O}_{\mathrm{CAILU}_k}(\alpha)$.

We present in Algorithm 3.2 how CA-ILU(k) factors a matrix A in parallel. It takes as input, the matrix A stored on processor 0, k the ILU parameter, π the partition of $G(A)$ (for example given by k-way partitioning) and *ordering* a boolean used to reorder each subdomain. It returns L_i and U_i the lower and upper triangular factors of A, respectively, that are stored on processor i. The algorithm is based on two main steps that allow us to perform the same factorization as in sequential. First, in line 5, each subdomain Ω_j, returned by the partition π applied on $G(A)$, is reordered using $CAILU\_reorderDomain$, Algorithm 3.5, which returns a permutation vector e_j of size equal to $|V(\Omega_j)|$. Then, the global permutation vector e, built by the concatenation of the permutation vectors of each subdomain ($e = [e_0; e_1; \ldots; e_{p-i}]$), is used to permute both rows and columns of A, line 8. The reordering of each subdomain aims to decrease the number of redundant operations performed during the numerical factorization of the subdomain. It also reduces the memory consumption on each processor. In line 9, the algorithm performs the symbolic factorization of A, in sequential and returns the symbolic factor matrix F_k. This routine predicts the pattern of the L and U factors returned by the ILU(k) algorithm. The second step calls the subroutine $CAILU\_addOverlap$ on each subdomain Ω_j which returns γ_j, the set of required vertices of $V(\Omega_j)$ used to both factor the matrix and apply the preconditioner locally without communication. We detail further in the chapter how γ_j is obtained. Then from γ_j, a larger subdomain $\widetilde{\Omega}_j$ is created such that $V(\widetilde{\Omega}_j) = \gamma_j$ and

$E(\widetilde{\Omega}_j) = \{(u,v) \in E(\Omega) \mid u \in \gamma_j, v \in \gamma_j\}$. Note that $\Omega_j \subseteq \widetilde{\Omega}_j$. The associated matrix A_i of $\widetilde{\Omega}_i$ is sent to processor i, $\forall i \in \{1, \ldots, p-1\}$. Therefore the numerical factorization of the associated matrix of $\widetilde{\Omega}_i$ is performed locally without communication (line 20). The algorithm returns L_i and U_i the lower and upper triangular factors, respectively, on processor i. Note that their size depends on the size of the subdomain Ω_i and the size of the requested data to be computed without communication.

Algorithm 3.2 CAILU_factorize (A, k, π, *ordering*)

This function computes the ILU(k) factorization of A by first reordering each subdomain Ω_i, then getting their overlap to send both to processor i and finally each processor factorizes their own local part without communication.

Input: A the whole matrix,
k the ILU parameter,
π the partitionning of A, of size p,
ordering the way that each subdomain is reordered

1: Let i be my processor rank
2: Let Ω_j, $j \in \{0, \ldots, p-1\}$ the subdomains obtained after the application of π on the graph of A
3: **if** $i = 0$ **then**
4: **for all** Ω_j **do**
5: $e_j \leftarrow$ CAILU_reorderDomain (Ω_j, Ω, k, *ordering*)
6: **end for**
7: $e \leftarrow [e_0; e_1; \ldots; e_{p-1}]$ /* *Concatenate vectors* e_j, $j = 0 : p-1$ */
8: $A \leftarrow A(e, e)$ /* *Permute the matrix with* e */
9: $F_k \leftarrow$ ILU_FactorSymbolic(A, k) /* *Sequential symbolic factorization* */
10: **for all** Ω_j **do**
11: $\gamma_j \leftarrow$ CAILU_addOverlap ($V(\Omega_j), G(F_k)$)
12: **if** $j \neq i$ **then**
13: Let A_j be the row block submatrix of A formed by the rows whose vertices belong to γ_j
14: Send A_j to processor j
15: **end if**
16: **end for**
17: **else**
18: Receive row block A_i from processor 0
19: **end if**
20: $[L_i, U_i] \leftarrow$ ILU_FactorNumeric(A_i) /* *Parallel numerical factorization* */

Output: L_i and U_i, the local lower and upper factors ,respectively, owned by processor i.

Algorithm 3.2 introduces two steps that we describe further. We first present the search of the overlap of $V(\Omega_i)$ in $G(F_k)$ and we show through an example that the size of the overlap can be as large as the whole domain. We then present a reordering algorithm that allows us to reduce the size of the overlap for each subdomain.

3.2.1 Search the dependencies of a subdomain in $\widehat{\Omega}$

In (Parter, 1961), Parter shows in Theorem I that for a linear system $Ax = b$, upon elimination of the entry $x(i)$ from the subset of equations, the new graph obtained from Ω is contained in Ω. This results from the removal of the associated vertex $v_i \in V(\Omega)$ to $x(i)$ and a pair-wise connection to all vertices which were connected to v_i in Ω (see the demonstration in the paper). Considering the case of LU factorization of A, in (Gilbert and Peierls, 1988) Gilbert *et al.* show that solving the i'th equation in the graph of U requires in addition the solution of the equations corresponding to the reachable vertices of v_i in $\widehat{\Omega}^+$, $\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_i)$. Then, solving the equations associated with the set of reachable vertices of $\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_i)$ in $\widehat{\Omega}^-$ leads to solving the equations of $\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_i)$ in the graph of L and therefore allows us to solve the i'th equation in the graph of L. Thus, all vertices reachable from a subdomain Ω_i in a graph compose its dependencies, also

referred to as overlap. Since we aim to make subdomains independent, CA-ILU(k) factorization has a dependency search step on each subdomain to be fully parallel. Algorithm 3.3 presents the procedure to get the overlap of Ω_i.

Algorithm 3.3 CAILU_addOverlap $(V(\Omega_i), \widehat{\Omega})$

This function returns the union of the vertices of Ω_i with its overlap, the data needed to apply the preconditioned matrix powers kernel algorithm without communication.

Input: $V(\Omega_i)$: the vertices of the i'th subdomain of Ω,
$\widehat{\Omega}$: the graph of C, the factored matrix A

1: $\beta_i \leftarrow \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i)) \cup V(\Omega_i)$
2: $\gamma_i \leftarrow \mathcal{N}^{\infty}_{\widehat{\Omega}^-}(\beta_i) \cup \beta_i$

Output: γ_i : the union of CA-ILU(k) overlap of Ω_i with $V(\Omega_i)$.

The reachable vertices of $V(\Omega_i)$ in $\widehat{\Omega}^+$ are $\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i))$ (line 1). Then, in the forward substitution, γ_i is the set of reachable vertices of β_i in $\widehat{\Omega}^-$ union to β_i (line 2). This construction implies $V(\Omega_i) \subseteq \beta_i \subseteq \gamma_i$.

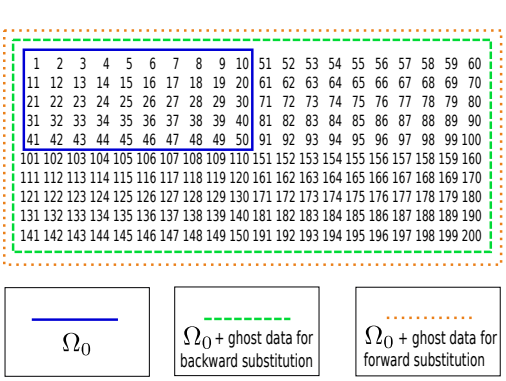
To illustrate the application of Algorithm 3.3, we use the same example as presented in (Grigori and Moufawad, 2015a). Consider the graph Ω of a 2D five-point stencil matrix A of size 200×200 displayed in Figures 3.1a and 3.1b. Ω is partitioned into 4 subdomains as in Definition 7, where each subdomain Ω_i has a set of indices $\mathcal{I}(\Omega_i)$ as in Definition 9. Each subdomain owns a quarter of the vertices of Ω and each vertex v_j is represented by its index $\mathcal{I}(v_j) \in \mathcal{I}(\Omega)$. In both figures, the edges in $E(\Omega)$ have an implicit representation. Each vertex is connected to its neighbors. For example, edges (v_1, v_2) and (v_1, v_{11}) belong to $E(\Omega)$ since vertices v_2 and v_{11} are neighbors of vertex v_1. The difference between both figures is the numbering of the vertices in each subdomain. In Figure 3.1a, the vertices of each subdomain are numbered using a natural ordering whereas the same vertices in Figure 3.1b are numbered in a different way that we present later. Figure 3.1a aims to show that the natural ordering is not appropriate and that the vertices of each subdomain have to be reordered. We focus on Figure 3.1a and its subdomain Ω_0 represented by the solid rectangular. Consider the case of no fill-in occurs during the symbolic factorization of A and so $E(\widehat{\Omega}) = E(\Omega)$. Thus we have $\widehat{\Omega} = \Omega$. Calling Algorithm 3.3 on vertex v_{50} of Ω_0 proceeds as follows for computing $\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_{50})$ in the first step:

- it starts with $\mathcal{N}^{1}_{\widehat{\Omega}^+}(v_{50})$, the search of the adjacent vertices of v_{50} in $\widehat{\Omega}^+$ and gets v_{91} and v_{110}.
- it restarts with v_{91} and v_{110} and searches their adjacent vertices in $\widehat{\Omega}^+$ that are not already visited.
- it iterates until no vertex can be added.

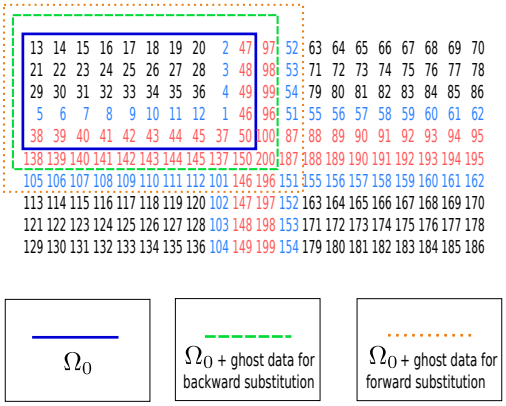
Thus, v_{50} reaches in Ω_1 vertices $\{v_{91}, v_{92}, \ldots, v_{100}\}$. Also, it reaches in Ω_3 the vertices $\{v_{110}, v_{120}, \ldots, v_{150}\}$ and all vertices in subdomain Ω_2. Finally, the vertices of Ω_0 reach the vertices of all other subdomains in $\widehat{\Omega}^+$. In this example where the vertices of each subdomain are numbered in natural ordering, we lose all gain of the parallelization.

Consider a different ordering applied on the vertices of each subdomain as in Figure 3.1b, where, in each subdomain, the vertices adjacent to other subdomains are numbered with the highest indices of the subdomain and their adjacent vertices in the subdomain are numbered with the smallest indices on the subdomain. The set of reachable vertices of v_{50} in $\widehat{\Omega}^+$ is $\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_{50}) =$

$\{100, 150, 200\}$. The hashed rectangular in Figure 3.1b represents $\beta_0 = \mathcal{N}^{\infty}_{\widehat{\Omega+}}(\Omega_0) \cup V(\Omega_0)$ whose size is much smaller than in the natural ordering case. Therefore, to solve this problem, we propose to reorder each domain, taking into account the structure and properties of U and L.



(a) Natural Ordering



(b) Reordering layers with $k = 0$

Figure 3.1 – Difference of dependency size between the original matrix (3.1a) and the reordered matrix (3.1b)

3.2.2 The reordering applied on each subdomain reduces the size of the overlap

ILU(k) algorithm applied on A introduces entries in the factor matrix F_k, i.e. edges in $E(G(F_k))$, following the condition that if there is a path of length at most $k+1$ from u to v through vertices numbered lower than both, then $f_{\mathcal{I}(u),\mathcal{I}(v)}$ becomes a nonzero element of F_k, equal to the shortest path from u to v, and $(u, v) \in E(G(F_k))$.

We introduce some notations. The vertices of a subdomain Ω_i can be split into two subsets. The first subset is composed of vertices that are adjacent to vertices of other subdomains. We call it the boundary layer of Ω_i, defined as

$$\mathcal{L}^i_0 = \{v \mid (u, v) \in E(\Omega), u \in V(\Omega) \setminus V(\Omega_i), v \in V(\Omega_i)\}, \tag{3.8}$$

where the subscript 0 represents the boundary layer. It can be rewritten as the union of adjacent sets of vertices from other subdomains in Ω_i as

$$\mathcal{L}^i_0 = \bigcup_{j \neq i,\, j=\{0,\ldots,p-1\}} \mathcal{N}^1_{\Omega_i \cup \Omega_j}(V(\Omega_j)). \tag{3.9}$$

The second subset contains the interior vertices of the subdomain. In other words, a vertex v of a subdomain Ω_i which is not in $\mathcal{L}^i_0$, can be reached from other subdomains only through its boundary layer. The adjacent set of vertices of the boundary layer of Ω_i in Ω_i forms a layer defined as $\mathcal{L}^i_1 = \mathcal{N}^1_{\Omega_i}(\mathcal{L}^i_0)$. Recursively, we define the k'th layer as

$$\forall k \geq 1, \quad \mathcal{L}^i_{k+1} = \mathcal{N}^1_{\Omega_i}(\mathcal{L}^i_k) \setminus \bigcup_{j=\{0,\ldots,k-1\}} \mathcal{L}^i_j. \tag{3.10}$$

Consider the graph Ω of a matrix A, partitioned into p subdomains as in Definition 7. Let

Ω_i and Ω_j be two subgraphs of Ω, with $i \neq j$. We note that the only way to reach a vertex $v \in V(\Omega_j)$ from a vertex $u \in V(\Omega_i)$ is through a path from u to v that traverses the vertices of the boundary layer $\mathcal{L}_0^j$, in Ω_j. Hence, we have $\mathcal{N}_{\Omega^+}^1(V(\Omega_i)) \subseteq \bigcup_{j=\{i+1,\ldots,p-1\}} \mathcal{L}_0^j$.

Lemma 14. *Consider the graph of a matrix A partitioned into p subdomains Ω_i, as in Definition 7, where each subdomain has a set of indices $\mathcal{I}(\Omega_i)$ in the sense of Definition 9 and the vertices of $\mathcal{L}_0^i$ are numbered with the highest indices in $\mathcal{I}(\Omega_i)$. Consider a vertex $u \in V(\Omega_i)$ and a vertex $v \in V(\Omega_j) \setminus \mathcal{L}_0^j$, with $j \neq i$, then there is no path from u to v in the graph of A that goes through vertices which are numbered lower than both $\mathcal{I}(u)$ and $\mathcal{I}(v)$.*

Proof. Let $u \in V(\Omega_i)$ and $v \in V(\Omega_j) \setminus \mathcal{L}_0^j$ with $j \neq i$ be two vertices. Suppose there exists a path from u to v in the graph of A. The numbering of the vertices of each boundary layer yields that v is numbered lower than all vertices of $\mathcal{L}_0^j$. From the definition of a boundary layer, the path from u to v goes through $\mathcal{L}_0^j$. Therefore, the path from u to v contains at least one vertex numbered higher than v. □

Lemma 15. *Consider the graph Ω of a matrix A partitioned into p subdomains Ω_i as in Definition 7. Each subdomain Ω_i has associated a set of indices $\mathcal{I}(\Omega_i)$ as defined in Definition 9. Consider the graph $\widehat{\Omega}$ of the factor matrix C defined as in Definition 11. If the vertices of $\mathcal{L}_0^i$ of each subdomain Ω_i are numbered with the highest indices of $\mathcal{I}(\Omega_i)$ then*

$$\forall i,\ 0 \leq i < p,\ \ \mathcal{N}_{\Omega^+}^{\infty}(V(\Omega_i)) \subseteq \bigcup_{j>i} \mathcal{L}_0^j \tag{3.11}$$

and so

$$\forall i,\ 0 \leq i < p,\ \ \mathcal{N}_{\widehat{\Omega}^+}^{\infty}(V(\Omega_i)) \subseteq \bigcup_{j>i} \mathcal{L}_0^j. \tag{3.12}$$

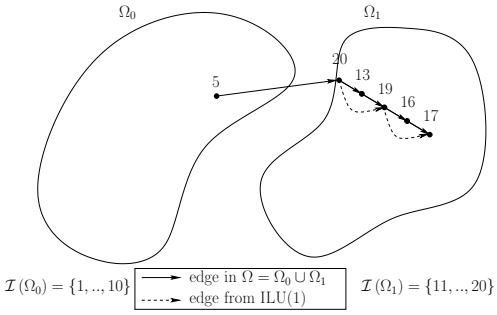
Proof. Consider a subdomain Ω_i, for which all vertices of $\mathcal{L}_0^i$ are numbered with the highest indices in $\mathcal{I}(\Omega_i)$. Therefore, in each subdomain Ω_i, the interior vertices that belong to $V(\Omega_i) \setminus \mathcal{L}_0^i$ are numbered with indices lower than $\mathcal{I}\left(\mathcal{L}_0^i\right)$.

We first prove the relation (3.11). The definition of a boundary layer implies that $\mathcal{N}_{\Omega^+}^1(V(\Omega_i)) \subseteq \bigcup_{j>i} \mathcal{L}_0^j$. To determine the set of reachable vertices from $\mathcal{L}_0^j$ in Ω^+, consider first ${\Omega_j}^+$ and then the graph defined as $\widetilde{\Omega} = (V(\Omega), E(\Omega) \setminus E(\Omega_j))$. In each subdomain Ω_j, the vertices in ${\Omega_j}^+$ are reachable from $\mathcal{L}_0^j$ if and only if they are numbered higher than at least one vertex of $\mathcal{L}_0^j$. But there is no vertex numbered from $\mathcal{I}(\Omega_j) \setminus \mathcal{I}\left(\mathcal{L}_0^j\right)$ higher than $\min(\mathcal{I}\left(\mathcal{L}_0^j\right))$. Therefore, $\mathcal{N}_{{\Omega_j}^+}^{\infty}\left(\mathcal{N}_{\Omega^+}^1(V(\Omega_i))\right) = \emptyset$. Now consider the vertices reachable from $\mathcal{L}_0^j$, $j > i$ in $\widetilde{\Omega}^+$. The numbering of the vertices of $\mathcal{L}_0^i$ with the highest indices of $\mathcal{I}(\Omega_i)$ allows us to apply Lemma 14. Therefore, the reachable vertices from $\mathcal{L}_0^j$ in $\widetilde{\Omega}^+$ are included in $\bigcup_{m>j} \mathcal{L}_0^m$. Hence, $\mathcal{N}_{\Omega^+}^{\infty}(V(\Omega_i))$, the set of reachable vertices of $V(\Omega_i)$ in Ω^+, is included in $\bigcup_{j>i} \mathcal{L}_0^j$.

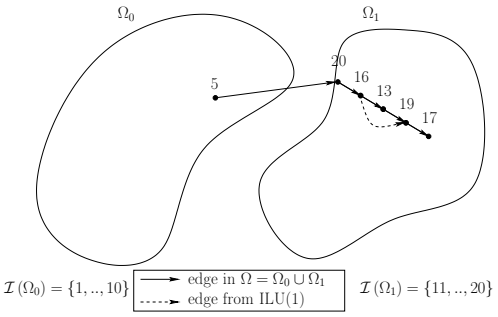
Now we prove the relation (3.12) concerning $\widehat{\Omega}$. In the definition of $\widehat{\Omega}$, we have $V(\Omega_i) = V(\widehat{\Omega}_i)$. The incomplete factorization of A does not change the definition of the boundary layer. Also, as shown in Lemma 14 it does not change the consequence that interior vertices of each subdomain are not reached from any vertices in the boundary layer in $\widehat{\Omega}^+$. Therefore, similarly to the proof of Relation (3.11), it can be shown that $\mathcal{N}_{\widehat{\Omega}^+}^{\infty}(V(\Omega_i)) \subseteq \bigcup_{j>i} \mathcal{L}_0^j$. □

When the vertices of the boundary layer of Ω_i are numbered with the highest indices of $\mathcal{I}(\Omega_i)$, the reachable vertices from $\mathcal{L}_0^i$ in Ω_i are numbered lower than the vertices that belong to $\mathcal{L}_0^i$.

The numbering of the vertices $v \in V(\Omega_i) \setminus \mathcal{L}_0^i$ can have an impact on the number of vertices reachable from $\mathcal{L}_0^i$ in Ω_i. To illustrate this, we consider a matrix of size 20 and its directed graph Ω partitioned into 2 subdomains as in Definition 7 where $|V(\Omega_0)| = |V(\Omega_1)| = 10$. Let $\mathcal{I}(\Omega_0) = \{1, .., 10\}$ and $\mathcal{I}(\Omega_1) = \{11, .., 20\}$ be the set of indices of each subdomain and the boundary layer of each subdomain is numbered with their highest indices. Consider the LU factorization of A. Assume the boundary layer of Ω_1 is the vertex 20. We study two numbering of the reachable vertices of vertex 20 in Ω_1. In Subfigure 3.2a, the vertices are numbered randomly whereas, in Subfigure 3.2b, the vertices are numbered such that the vertex at distance 3 from 20 has the highest index in $\mathcal{I}(\Omega_1) \setminus \{20\}$ and the remaining vertices are then numbered randomly.



(a) A random numbering of vertices in $V(\Omega_1) \setminus \mathcal{L}_0^1$.



(b) A numbering of vertices in $V(\Omega_1) \setminus \mathcal{L}_0^1$ such that the vertex at distance 3 from vertex v_{20} has the highest index in $\mathcal{I}(\Omega_1) \setminus \mathcal{I}\left(\mathcal{L}_0^1\right)$.

Figure 3.2 – Impact of the numbering of the vertices in subdomain Ω_1 on the CA-ILU(k) overlap with $k = 1$. A vertex is represented by a point and an oriented edge by an arrow. The number above each vertex is its index in $\mathcal{I}(\Omega)$. Consider a matrix A of size 20 and its associated graph Ω partitioned into 2 subdomains Ω_0 and Ω_1 having a set of indices $\mathcal{I}(\Omega_0)$ and $\mathcal{I}(\Omega_1)$, respectively. Subdomain Ω_1 has $\mathcal{L}_0^1 = \{v_{20}\}$ numbered with the highest number in $\mathcal{I}(\Omega_1)$. Subfigures differ in the way they number the vertex at distance 3 from the boundary layer. In 3.2a, this vertex is not numbered with the highest index of $\mathcal{I}(\Omega_1) \setminus \mathcal{I}\left(\mathcal{L}_0^1\right)$. In 3.2b, this vertex has the highest number of $\mathcal{I}(\Omega_1) \setminus \mathcal{I}\left(\mathcal{L}_0^1\right)$.

We apply in both cases Algorithm 3.3. Since only $\mathcal{L}_0^1$ is reached from $V(\Omega_0)$ in $\widehat{\Omega}^+$, β_0 is equal to $\{v_5, v_{20}\}$. The second step computes $\gamma_0 = \mathcal{N}_{\widehat{\Omega}^-}^{\infty}(\beta_0)$. For simplicity, we consider only the search in $\widehat{\Omega}_1$ which leads to computing $\alpha = \mathcal{N}_{\widehat{\Omega}_1^-}^{\infty}\left(\mathcal{L}_0^1\right)$. In Subfigure 3.2a, $\alpha = \{v_{13}, v_{19}, v_{16}, v_{17}\}$ whereas in Subfigure 3.2b, $\alpha = \{v_{16}, v_{13}\}$. The fact that the vertex at distance 3 from v_{20} in $\widehat{\Omega}_1$ is numbered with the highest index in $\mathcal{I}(\Omega_1) \setminus \mathcal{I}\left(\mathcal{L}_0^1\right)$ allows us to reduce the size of α from 4 to 2.

Lemma 16. *Consider the graph Ω of a matrix A partitioned into p subdomains Ω_i, as in Definition 7. Each subdomain Ω_i has associated a set of indices $\mathcal{I}(\Omega_i)$ as in Definition 9, and a boundary layer $\mathcal{L}_0^i$ whose vertices are numbered with the highest indices in $\mathcal{I}(\Omega_i)$. Consider the graph $\widehat{\Omega}$ of the factor matrix C of ILU(k) factorization of A defined as in Definition 11. If the vertices of the layer $\mathcal{L}_{k+2}^i$ of the subdomain Ω_i are numbered with the highest indices of $\mathcal{I}(\Omega_i) \setminus \mathcal{I}\left(\mathcal{L}_0^i\right)$, then*

$$\mathcal{N}_{\Omega_i^-}^{\infty}\left(\mathcal{L}_0^i\right) \subseteq \bigcup_{m=1}^{k+1} \mathcal{L}_m^i,$$

and

$$\mathcal{N}^{\infty}_{\widehat{\Omega}_i^-}\left(\mathcal{L}_0^i\right) \subseteq \bigcup_{m=1}^{k+1} \mathcal{L}_m^i.$$

Proof. A vertex $v \in V(\Omega_i) \setminus \mathcal{L}_0^i$ has an index lower than all vertices that belong to $\mathcal{L}_0^i$. Thus the set of adjacent vertices of $\mathcal{L}_0^i$ in ${\Omega_i}^-$ is included in $\mathcal{L}_1^i$. Suppose the vertices of the layer $\mathcal{L}_{k+2}^i$ are numbered with the highest indices in $\mathcal{I}(\Omega_i) \setminus \mathcal{I}\left(\mathcal{L}_0^i\right)$. Therefore, these vertices are not reachable from $\mathcal{L}_{k+1}^i$ in ${\Omega_i}^-$. Hence only the vertices of $\mathcal{L}_j^i$ ($j \in \{1, \ldots, k+1\}$) are reachable from $\mathcal{L}_0^i$ in ${\Omega_i}^-$.

Now we consider the matrix C. The ILU(k) factorization of A adds an edge (u, v) in $E(\widehat{\Omega})$ if there exists a path in Ω of length at most $k+1$ from a vertex $u \in V(\Omega)$ to a vertex $v \in V(\Omega)$ going through vertices numbered lower than both $\mathcal{I}(u)$ and $\mathcal{I}(v)$. Therefore, if the vertices of $\mathcal{L}_2^i$ are numbered higher than $\max(\mathcal{I}\left(\mathcal{L}_1^i\right))$ then there exists an edge $(u, v) \in E(\widehat{\Omega})$ where $u \in \mathcal{L}_0^i$ and $v \in \mathcal{L}_2^i$. Consider the layer $\mathcal{L}_{k+2}^i$ of the subdomain and its vertices, numbered with the highest indices in $\mathcal{I}(\Omega_i) \setminus \mathcal{I}\left(\mathcal{L}_0^i\right)$. The length of the shortest path from a vertex $u \in \mathcal{L}_0^i$ to a vertex $v \in \mathcal{L}_{k+2}^i$ in Ω is $k+2$, and hence, there is no edge (u, v) in $E(\widehat{\Omega})$. Hence, only the vertices of $\mathcal{L}_j^i$ ($j \in \{1, \ldots, k+1\}$) are reachable from $\mathcal{L}_0^i$ in $\widehat{\Omega}_i^-$.

□

Theorem 17. *Consider a matrix A and its ILU(k) factorization that returns $C = L + U - I_{n\times n}$, where $I_{n\times n}$ is the identity matrix of size n. Let Ω be the graph of A, partitioned into p subdomains Ω_i, as in Definition 7. Each subdomain Ω_i has associated a set of indices $\mathcal{I}(\Omega_i)$ as in Definition 9, and a boundary layer $\mathcal{L}_0^i$ whose vertices are numbered with the highest indices of $\mathcal{I}(\Omega_i)$. Let the vertices of the layer $\mathcal{L}_{k+2}^i = \mathcal{N}^1_{\Omega_i}\left(\mathcal{L}_{k+1}^i\right) \setminus \mathcal{L}_k^i$ be numbered with the highest indices in $\mathcal{I}(\Omega_i) \setminus \mathcal{I}\left(\mathcal{L}_0^i\right)$. Consider the CA-ILU(k) overlap of Ω_i, $\mathcal{O}_{CAILU_k}(V(\Omega_i)) = \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i)) \bigcup \mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i))\right)$, as in Definition 12. The set of vertices $\mathcal{O}_{CAILU_k}(\Omega_i) \cap V(\Omega_j)$, with $\Omega_j \in \mathcal{D}(\Omega_i)$ is included in $\bigcup_{m\in\{0,\ldots,k+1\}} \mathcal{L}_m^j$.*

Proof. Consider a subdomain Ω_i, its reachable subdomains $\Omega_j \in \mathcal{D}(\Omega_i)$, with $j \neq i$ and the CA-ILU(k) overlap of Ω_i, $\mathcal{O}_{\mathrm{CAILU}_k}(\Omega_i)$, as in Definition 12. Let $\widehat{\Omega}$ be the graph of $C = L + U - I_{n\times n}$, where L and U are the factors of the ILU(k) factorization of A. Lemma 14 ensures that the only way to reach a vertex $v \in V(\Omega_j)$ from a vertex $u \in V(\Omega_i)$ in Ω is through at least a vertex that belongs to $\mathcal{L}_0^j$. Therefore in each subdomain $\Omega_j \in \mathcal{D}(\Omega_i)$, at least one vertex of the boundary layer $\mathcal{L}_0^j$ is reached from a vertex that belongs to $V(\Omega_i)$. From Lemma 15, the set of reachable vertices of $V(\Omega_i)$ in $\widehat{\Omega}^+$, $\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i))$, is included in $\bigcup_{j>i} \mathcal{L}_0^j$. Thus the boundary layer $\mathcal{L}_0^j$ of each subdomain $\Omega_j \in \mathcal{D}(\Omega_i)$, $j > i$ is involved in the construction of $\mathcal{O}_{\mathrm{CAILU}_k}(\Omega_i)$.

Let $\beta_i = V(\Omega_i) \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i))$. The set of reachable vertices of β_i in $\widehat{\Omega}^-$ is $\mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i))\right)$ that can be split into $\mathcal{N}^{\infty}_{\widehat{\Omega}^-}(V(\Omega_i)) \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(\mathcal{N}^{\infty}_{\widehat{\Omega}^+}(V(\Omega_i))\right)$ which is included into $\mathcal{N}^{\infty}_{\widehat{\Omega}^-}(V(\Omega_i)) \cup \mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(\bigcup_{j>i} \mathcal{L}_0^j\right)$. By definition of the boundary layer, we have $\mathcal{N}^1_{\Omega^-}(V(\Omega_i)) = \mathcal{N}^1_{\widehat{\Omega}^-}(V(\Omega_i)) \subseteq \bigcup_{j<i} \mathcal{L}_0^j$ where $\Omega_j \in \mathcal{D}(\Omega_i)$. Therefore the set of reachable vertices of β_i in $\widehat{\Omega}^-$ is included into $\mathcal{N}^{\infty}_{\widehat{\Omega}^-}\left(\bigcup_{j\neq i} \mathcal{L}_0^j\right) \cup \bigcup_{j\neq i} \mathcal{L}_0^j$, with $\Omega_j \in \mathcal{D}(\Omega_i)$. Lemma 16 ensures that $\mathcal{N}^{\infty}_{\widehat{\Omega}_j^-}\left(\mathcal{L}_0^j\right) \subseteq \bigcup_{m=1}^{k+1} \mathcal{L}_m^j$. Therefore, a subdomain $\Omega_j \in \mathcal{D}(\Omega_i)$ has at most its first $k+2$ layers involved in the CA-ILU(k) overlap of Ω_i.

□

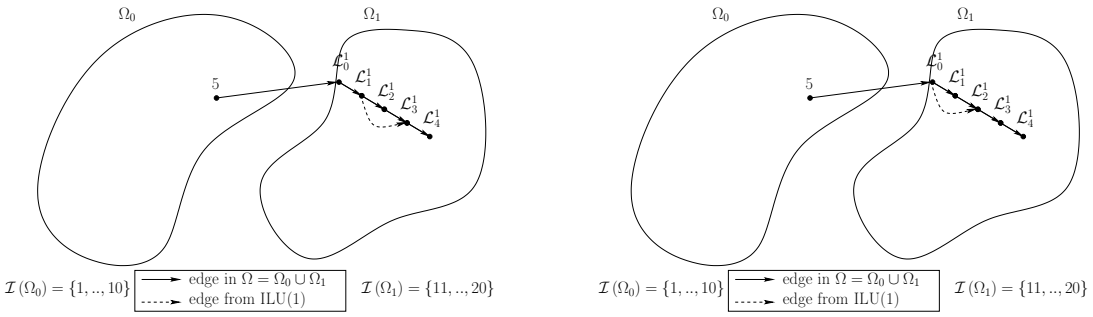
Theorem 17 shows that if the boundary layer and the layer $\mathcal{L}^i_{k+2}$ are numbered with the highest indices of Ω_i, then the size of $\mathcal{O}_{\text{CAILU}_k}(\Omega_i)$ is bounded. We denote the $k+1$ layers between these two layers as the inner layers. As discussed in the proof of Lemma 16, the indices of the vertices that belong to inner layers introduce new entries in F_k. The way these vertices are numbered is expected to impact the fill-in during the factorization.

Through an example, we next present two reordering of the vertices of the inner layers and the influence on the fill-in. Consider Ω the graph of a matrix A partitioned into 2 subdomains Ω_0 and Ω_1 as in Definition 7. Consider the layers of Ω_1 and the vertex numbered 5 in Ω_0. We discuss the case of ILU(1) factorization of A. Assume Theorem 17 is verified. We study two ways of reordering the inner layers $\mathcal{L}^1_1$ and $\mathcal{L}^1_2$ and give a general definition for any subdomain Ω_i. The first approach, called Decreasing Layers Numbering, and denoted hereafter as DLN, numbers the vertices of the inner layers in a decreasing order, such that

$$\forall j \in \{1, \ldots, k\}, \;\; \max(\mathcal{L}^i_j) > \min(\mathcal{L}^i_{j+1}). \tag{3.13}$$

The second one, called Increasing Layers Numbering, referred further as ILN, starts by numbering $\mathcal{L}^i_1$ with the lowest indices of Ω_i and then

$$\forall j \in \{1, \ldots, k\}, \;\; \max(\mathcal{L}^1_j) < \min(\mathcal{L}^1_{j+1}). \tag{3.14}$$



(a) Decreasing Layer Numbering **(b)** Increasing Layer Numbering

Figure 3.3 – Comparison of methods to reorder inner layers ($\mathcal{L}^1_1$ and $\mathcal{L}^1_2$) of subdomain Ω_1. Layers are represented by dots, edges between one vertex from a layer to a vertex from another layer by an arrow, and dot lines represent additional edges induced by ILU(1) algorithm in Ω_1.

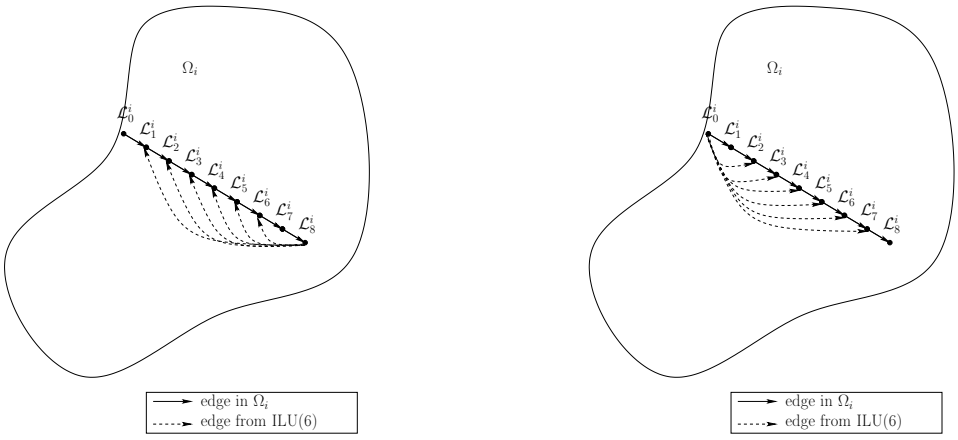
Figure 3.3 illustrates the impact of reordering proposed for $k = 1$, where, in each subfigure, Theorem 17 is verified. Layers are represented by dots, arrows are the edges from a vertex in a layer to a vertex that belongs to another layer and dot lines represent additional edges introduced by ILU(1) factorization. In each subfigure, the set of reachable vertices of Ω_0 in $\widehat{\Omega}^+_1$ is $\beta_0 = \mathcal{N}^{\infty}_{\widehat{\Omega}^+_1}(V(\Omega_0)) \subseteq \mathcal{L}^1_0$, the boundary layer of Ω_1. Then the set of reachable vertices of β_0 in $\widehat{\Omega}^-_1$ is a subset of the vertices of each inner layer of Ω_1. The difference between these two approaches is the edges added by the factorization. In Subfigure 3.3a, the DLN approach involves edges from vertices that belong to $V(\mathcal{L}^1_1)$ to vertices in $V(\mathcal{L}^1_3)$. These edges exist in $E(G(F_1))$ but not in $G(F_1)^-$. On the other hand, the ILN approach induces edges which are in $G(F_1)^-$. This leads the third layer to be part of the adjacent vertices of the boundary layer in Ω_1. Therefore, the search of the reachable vertices is expected to be faster using ILN.

Back to the 2D five-point stencil problem from Figure 3.1a, we consider $k = 0$. In that case, DLN is strictly equivalent to ILN (compare 3.3a with 3.3b by removing inner layers on both).

Therefore, we apply one of the reordering approaches. Consider the construction of the set of reachable vertices of $V(\Omega_0)$ in $\widehat{\Omega}^+$ i.e. the solid rectangular in Figure 3.1b. The reordering of the first two layers of each subdomain reduces the size of β_0 compared to the original ordering in Figure 3.1a. β_0 is a subset of the union of $V(\Omega_0)$ with $\mathcal{L}_0^1$, $\mathcal{L}_0^3$ and v_{200} which belongs to Ω_2. Then $\gamma_0 = \mathcal{N}_{\Omega^-}^{\infty}(\beta_0)$ is a subset of the union of β_0 with $\mathcal{L}_1^1$, $\mathcal{L}_1^2$ and the reachable vertices in Ω_2.

Consider a graph Ω partitioned into p subdomains as in Definition 7. Suppose each subdomain Ω_i is composed of m layers and is reordered such that Theorem 17 is verified. Therefore, the set of reachable vertices of a subdomain Ω_j in Ω^+, restrained to $V(\Omega_i)$ is $\mathcal{N}_{\Omega^+}^{\infty}(V(\Omega_j)) \cap V(\Omega_i) \subseteq \mathcal{L}_0^i$ for any value of k. Consider the DLN and ILN orderings. We aim to compare the CA-ILU(k) overlap size and especially the number of inner layers involved by each ordering. We focus the comparison on the set of reachable vertices of $\mathcal{L}_0^i$ in ${\Omega_i}^-$. First, the DLN ordering is applied to Ω_i. From Equation (3.13), there exists a vertex $u \in \mathcal{L}_l^i$ that reaches at least one vertex of $\mathcal{L}_{l+1}^i$ in ${\Omega_i}^-$, $\forall l \in \{0, \ldots, m-1\}$. Therefore, the vertices of all inner layers $\mathcal{L}_l^i$ ($l \in \{1, \ldots, m-1\}$) can be reached from the vertices of $\mathcal{L}_0^i$ and this for all values of k (even for $k = 0$). On the other hand, applying ILN ordering on these m layers implies that only the vertices of the first $k+1$ inner layers of the subdomain are reachable from the vertices of $\mathcal{L}_0^i$ in ${\Omega_i}^-$. Therefore, the size of the set of reachable vertices depends on k. Hence, ILN ordering offers more flexibility to factor the subdomain with a value of k varying from 0 to $m-2$. This ordering is particularly useful in the case of memory limitation that we discuss further.

To illustrate this comparison, we consider a subdomain Ω_i with $m = 9$ layers reordered using DLN in Subfigure 3.4a and reordered using ILN in Subfigure 3.4b. Consider the ILU(6) factorization of a matrix A associated with Ω. Edges of Ω_i are represented by arrows whereas dot arrows are the edges added by the factorization. In Subfigure 3.4a, the vertices of the 7 inner layers are reached from the boundary layer. Assume that k is reduced to 3, still, the vertices of the 7 layers are reached from the boundary layer. The only way to reduce the number of inner layers involved here is to number the 5'th layer with the highest indices in $\mathcal{I}(\Omega_i) \setminus \mathcal{I}(\mathcal{L}_0^i)$. On the other hand, in Subfigure 3.4b, when $k = 3$, only the first 4 inner layers are involved. It follows that numbering the vertices of $V(\Omega_i)$ using ILN allows us to compute the incomplete factorization of A with a parameter $0 \geq k \geq m-2$ without the need of reordering Ω_i using a different k.



(a) Decreasing Layer Numbering **(b)** Increasing Layer Numbering

Figure 3.4 – Comparison of the number of inner layers $\mathcal{L}_l^1$, $l \in \{1, \ldots, k+1\}$ involved during the search of $\mathcal{N}_{\Omega_1^-}^{\infty}(\mathcal{L}_0^1)$ for both DLN and ILN ordering methods.

Reordering inside a layer helps to reduce the size of the overlap

In Figure 3.1b, consider the lower right part of the graph denoted Ω_2. Let $k = 0$ so that $\Omega = \widehat{\Omega}$. We focus on the search of the set of reachable vertices of $V(\Omega_0)$ in $\widehat{\Omega}^+$. Let $u \in V(\Omega_2)$ be adjacent to two vertices, one belonging to Ω_1 and the other one belonging to Ω_3, and be the upper left vertex in the figure. The reordering proposed in Figure 3.1b numbers u with 200, the largest index in $\mathcal{I}(\Omega_2)$. Hence u is the only reachable vertex from $V(\Omega_0)$ in $\widehat{\Omega}^+$. However, it exists a numbering of these vertices that leads to reaching them all. Consider the pathological case in which the vertices are numbered larger than $\mathcal{I}(u)$. This ordering proceeds recursively such that the adjacent vertices $\alpha_{j+1} = \mathcal{N}_G^1(\alpha_j)$ of the set α_j in $G = (\mathcal{L}_0^2, \{(u,v) \in E(\Omega) \mid u \in \mathcal{L}_0^2,\ v \in \mathcal{L}_0^2\})$ are numbered higher than $\max(\mathcal{I}(\alpha_j))$, with $\alpha_0 = \{u\}$. Therefore, computing $\mathcal{N}^{\infty}_{G(\mathcal{L}_0^2)^+}(u)$ leads to reach all vertices of $\mathcal{L}_0^2$.

We now consider the case ILU(1). In Figure 3.1b, the lowest index in $\mathcal{I}\left(\mathcal{L}_0^2\right)$ leads to reach v_{187} during the computation of $\mathcal{N}^{\infty}_{\Omega^+}(V(\Omega_0))$. We focus on the computation of $\mathcal{N}^{\infty}_{\Omega_2^+}(v_{187})$. The vertices v_i, $i \in \{188, \ldots, 199\}$ are added to $\mathcal{N}^{\infty}_{\Omega^+}(V(\Omega_0))$. To avoid that, interchanging the index of v_{189} with v_{190} and the index of v_{198} with v_{199} allows us to bound the reachable vertices from v_{187} in $G(\mathcal{L}_0^2)$ to v_{199} and v_{190}. As a consequence, $\mathcal{N}^{\infty}_{G(\mathcal{L}_0^2)^+}(v_{187}) = \{188, 189, 196, 197, 198\}$. Note that the proposed renumbering respects the concept of reordering in Theorem 17. Hence reordering the layers with respect to each other is not sufficient. For the first $k+2$ layers of each subdomain Ω_i, we propose to reorder also the vertices inside each layer. It is not necessary to reorder the vertices of $\mathcal{L}_{k+3}^i$ since the computation of γ_0 involves only the adjacent vertices of γ_0 in $G(A)$. We first define a corner that belongs to a boundary layer and then derive the definition of an inner layer. We then present Algorithm 3.4 that reorders a layer.

Definition 18. *Given Ω the graph of a matrix A partitioned into p subdomains as in Definition 7, a corner $u \in \mathcal{L}_0^i$ of a subdomain Ω_i is an adjacent vertex of at least two vertices that belong to two different subdomains Ω_j and Ω_m with $i \neq j \neq m$.*

From Equation (3.9), a boundary layer is the set of vertices in Ω_i that are adjacent to vertices that belong to other subdomains. The corner of $\mathcal{L}_0^i$ is defined as

$$\mathcal{C}_0^i = \{u \in \mathcal{L}_0^i \mid (v,u) \in E(\Omega),\ (t,u) \in E(\Omega),\ v \in V(\Omega_j),\ t \in V(\Omega_m), i \neq j \neq m\}. \tag{3.15}$$

$\mathcal{C}_0^i$ is a subset of $\mathcal{L}_0^i$ whose vertices are connected to at least two distinct subdomains, different from Ω_i. As a consequence of this definition, a corner is potentially reachable more often than other vertices. Vertex v_{46} reaches vertices v_{96} and v_{99}. Next, v_{96} reaches v_{100} which belongs to $\mathcal{C}_0^1$ by Definition 18. Finally, v_{200} is added into $\beta_0 = \mathcal{N}^{\infty}_{\widehat{\Omega}^+}(v_{46})$. Hence the index of each vertex of $\mathcal{C}_0^i$ in $\mathcal{L}_0^i$ has to be chosen carefully.

Similarly to the boundary layer renumbering, we propose to number all vertices of $\mathcal{C}_0^i$ with the highest number of $\mathcal{I}\left(\mathcal{L}_0^i\right)$. It follows that when a corner is reached in Ω^+, it allows us to continue the search in a subdomain where one of its adjacent vertices in $\widehat{\Omega}^+$ belongs to. Also it stops the search in the boundary layer it belongs to. Analogously, we can apply Theorem 17 on the vertices of the boundary layer. Therefore, numbering the vertices of the boundary layer that are at a maximum distance of $k+2$ from a corner with the highest indices in $\mathcal{I}\left(\mathcal{L}_0^i\right) \setminus \mathcal{I}\left(\mathcal{C}_0^i\right)$ limits the search in the layer.

Going further, we can compute the corners of the layer $\mathcal{L}_l^i$ $(l > 0)$ as

$$\begin{aligned}\mathcal{C}_l^i = \{u \in \mathcal{L}_l^i \mid (v,u) \in E(\Omega_i),\ (t,u) \in E(\Omega_i),\ v \in \mathcal{N}^l_{\Omega_j \cup \Omega_i}(V(\Omega_j)) \cap \mathcal{L}_{l-1}^i,\\ t \in \mathcal{N}^l_{\Omega_m \cup \Omega_i}(V(\Omega_m)) \cap \mathcal{L}_{l-1}^i,\ i \neq j \neq m\}.\end{aligned} \tag{3.16}$$

Algorithm 3.4 presents the procedure that renumbers the vertices of a layer $\mathcal{L}_j^i$. This routine takes as input the layer to reorder $\mathcal{L}_j^i$, its subdomain Ω_i, the associated corners $\mathcal{C}_j^i$, the ILU parameter k and the ordering method (ILN or DLN). The algorithm starts by getting the adjacent vertices of the set of corners α_0 (line 2). Then, it iterates to get the adjacent vertices α_j of the union of the corners with the previous sets α_m ($m \in \{0, \ldots, j-1\}$). According to the *ordering* flag, it creates a global permutation vector e by concatenating the indices of the sets α_m ($m \in \{0, \ldots, k+1\}$) with the indices of the remaining vertices and the indices of the corners (lines 7, 9). We are using the term global permutation to make clear that the returned permutation cannot be applied directly on a subdomain without modification since the indices in Ω_i are global.

Algorithm 3.4 CAILU_reorderLayer ($\mathcal{L}_j^i$, Ω_i, $\mathcal{C}_j^i$, k, *ordering*)

Input: $\mathcal{L}_j^i$ the j'th layer to reorder which belongs to subdomain Ω_i,
Ω_i the i'th subdomain of Ω,
$\mathcal{C}_j^i$ the corners of the layer $\mathcal{L}_j^i$,
k the ILU parameter,
ordering either ILN or DLN, one of the two ordering algorithms used to number the vertices of a layer

1: Let $G \subseteq \Omega_i$ be the subgraph with $V(G) = \mathcal{L}_j^i$ and $E(G) = \{(u,v) \in E(\Omega_i) \mid u, v \in \mathcal{L}_j^i\}$
2: $\alpha_0 \leftarrow \mathcal{N}_G^1\left(\mathcal{C}_j^i\right)$
3: **for** $j = 1$ to $k+1$ **do**
4: $\quad \alpha_j \leftarrow \mathcal{N}_G^1\left(\alpha_{j-1}\right) \setminus \left(\bigcup_{m=0}^{j-1} \alpha_m \cup \mathcal{C}_j^i\right)$
5: **end for**
$\alpha_r \leftarrow \mathcal{L}_j^i \setminus \left(\bigcup_{m=0}^{k+1} \alpha_m \cup \mathcal{C}_j^i\right)$
6: **if** *ordering* $= DLN$ **then**
7: $\quad e \leftarrow \left[\mathcal{I}\left(\alpha_{k+1}\right); \mathcal{I}\left(\alpha_k\right); \ldots; \mathcal{I}\left(\alpha_0\right); \mathcal{I}\left(\alpha_r\right); \mathcal{I}\left(\mathcal{C}_j^i\right)\right]$
8: **else**
9: $\quad e \leftarrow \left[\mathcal{I}\left(\alpha_0\right); \mathcal{I}\left(\alpha_1\right); \ldots; \mathcal{I}\left(\alpha_{k+1}\right); \mathcal{I}\left(\alpha_r\right); \mathcal{I}\left(\mathcal{C}_j^i\right)\right]$
10: **end if**

Output: e the permutation vector that reorders the vertices of $\mathcal{L}_j^i$

This algorithm is equivalent to a breadth first search and its complexity depends on k and on the graph of A. In the worse case, the algorithm reorders all vertices of the layer.

Using Algorithm 3.4, we present in Algorithm 3.5 CAILU_reorderDomain routine that reorders a subdomain Ω_i in order to reduce the size of the reachable vertices during the computation of the CA-ILU(k) overlap from Definition 12. Algorithm 3.5 takes as input the subdomain Ω_i to renumber, the domain Ω, the ILU parameter k and *ordering* (either ILN or DLN), the way the layers are numbered. Lines 1 and 2, the algorithm starts by computing the boundary layer $\mathcal{L}_0^i$ of Ω_i and its associated corner set $\mathcal{C}_0^i$. Using Algorithm 3.4, it obtains e_0, the permutation vector associated with the boundary layer. Then, the algorithm iterates over j, $k+1$ times, computing $\mathcal{L}_j^i$, $\mathcal{C}_j^i$ and getting the permutation vector e_j that reorders the current layer. The algorithm returns a permutation vector p_i associated with Ω_i. It is constructed by the concatenation of the e_j permutation vectors, according to the *ordering* flag, with the indices of the remaining vertices of $V(\Omega_i)$ not already ordered and the permutation vector of the boundary layer. To obtain the boundary layer, the algorithm uses Equations (3.8) and (3.15). The cost to obtain $\mathcal{L}_0^i$ is equal to the sum of the nonzeros in the block column $A(\mathcal{I}(\Omega) \setminus \mathcal{I}(\Omega_i), \mathcal{I}(V(\Omega_i)))$. This cost corresponds to the number of edges in the graph associated with this block column. To obtain an inner layer and its associated corners, the algorithm uses Equations (3.10) and (3.16). The associated cost to get $\mathcal{L}_j^i$ and $\mathcal{C}_j^i$ is equal to the number of edges traversed from $\mathcal{L}_{j-1}^i$ in Ω_i. The global cost of

Algorithm 3.5 is equal to the sum of the cost to obtain the boundary layer, the cost to obtain the inner layers and the cost to reorder each layer.

Algorithm 3.5 CAILU_reorderDomain (Ω_i, Ω, k, *ordering*)

This function reorders the subdomain Ω_i by reordering $k+3$ layers and reordering the vertices of the first $k+2$ layers of the subdomain.

Input: Ω_i the i'th subdomain of Ω,
Ω the graph partitioned into p subdomains as in Definition 7
k the ILU parameter,
ordering the ordering algorithm used to number layers

Compute $\mathcal{L}_0^i$ as in Equation (3.8)
Compute $\mathcal{C}_0^i$ as in Equation (3.15)
$e_0 \leftarrow$ CAILU_reorderLayer $(\mathcal{L}_0^i, \Omega_i, \mathcal{C}_0^i, k, ordering)$ */* Algorithm 3.4 */*
for $j = 1$ to $k+1$ **do**
  Compute $\mathcal{L}_j^i$ as in Equation (3.10)
  Compute $\mathcal{C}_j^i$ as in Equation (3.16)
  $e_j \leftarrow$ CAILU_reorderLayer $(\mathcal{L}_j^i, \Omega_i, \mathcal{C}_j^i, k, ordering)$ */* Algorithm 3.4 */*
end for
$\alpha_r \leftarrow V(\Omega_i) \setminus \bigcup_{m=0}^{k+1} \mathcal{L}_m^i$ */* Remaining vertices not treated */*
if $ordering = DLN$ **then**
  $p_i \leftarrow [e_{k+1}; e_k; \ldots; e_1; \mathcal{I}(\alpha_r); e_0]$
else
  $p_i \leftarrow [e_1; e_2; \ldots; e_{k+1}; \mathcal{I}(\alpha_r); e_0]$
end if

Output: p_i : the permutation vector associated with the subdomain Ω_i.

3.3 CA-ILU(k) preconditioner

3.3.1 Complexity of CA-ILU(k) factorization

Here, we present the algorithmic complexity of the construction of CA-ILU(k). Given a sparse matrix A, the construction of CA-ILU(k) preconditioner is strongly related to the number of non-zeros of A. Let $G(A)$ be the graph associated with A, partitioned into p subdomains as in Definition 7 and α be a set of vertices such that $\alpha \subseteq V(G(A))$. Getting an adjacent layer and reordering inside a layer have a similar behavior. They search the adjacent vertices of a set of vertices by going through edges in the graph of A. Table 3.1 presents the complexity of the main subroutines used during the construction of the preconditioner. CAILU_reorderLayer, Algorithm 3.4, routine aims to reorder a layer. CAILU_reorderDomain, Algorithm 3.5, uses it to reorder a subdomain and CAILU_addOverlap, Algorithm 3.3, searches the overlap of a subdomain and concatenates it to its subdomain.

Algorithm	Edges visited
CAILU_reorderLayer	$Cost_{\mathcal{L}}(\mathcal{L}_j^i) = \lvert E\left(G\left(A\left(\mathcal{I}\left(\mathcal{C}_j^i \cup \bigcup_{m=0}^{k} \alpha_m\right), \mathcal{I}(\mathcal{L}_j^i)\right)\right)\right)\rvert$
CAILU_reorderDomain	$Cost_{\mathcal{D}}(\Omega_i) = \lvert E(G(A(\mathcal{I}(\Omega) \setminus \mathcal{I}(\Omega_i), \mathcal{I}(\Omega_i)))) + E\left(G\left(A\left(\mathcal{I}\left(\sum_{m=0}^{k} \mathcal{L}_m^i\right), \mathcal{I}(\Omega_i)\right)\right)\right)\rvert + \sum_{m=0}^{k+1} Cost_{\mathcal{L}}(\mathcal{L}_m^i)$
CAILU_addOverlap	$\lvert E(F_k(\mathcal{I}(\gamma_i), :))\rvert$

Table 3.1 – Complexity of main algorithms used to construct CA-ILU(k).

The first algorithm in Table 3.1 performs a breadth first search-like algorithm and starts

by getting the adjacent vertices of $\mathcal{C}_j^i$ in $G(A(\mathcal{I}\left(\mathcal{C}_j^i\right), \mathcal{I}\left(\mathcal{L}_j^i \setminus \mathcal{C}_j^i\right)))$. This graph is associated with a submatrix formed by the subset of rows of A corresponding to the indices of the vertices that belong to $\mathcal{C}_j^i$ and from the subset of columns of A corresponding to the indices of the vertices of $\mathcal{L}_j^i \setminus \mathcal{C}_j^i$. Then the algorithm iterates to get the next $k+1$ set of vertices. The total cost of this first algorithm is equal to the sum of the non-zeros of rows associated with each vertex in $\alpha_m \subseteq \mathcal{L}_j^i$ $(m \in \{0, \ldots, k\})$ and in $\mathcal{C}_j^i$, where α_m is the adjacent set of vertices of α_{m-1} in the graph $G(A(\mathcal{I}(\alpha_{m-1}), \mathcal{I}\left(\mathcal{L}_j^i\right) \setminus \mathcal{I}\left(\mathcal{C}_j^i \cup \bigcup_{l=0}^{m-1} \alpha_l\right)))$, and α_0 is the set of adjacent vertices of $\mathcal{C}_j^i$ in Ω_i. The second algorithm obtains its first $k+2$ layers and reorders them by calling CAILU_reorderLayer routine. The first term, $E(G(A(\mathcal{I}(\Omega) \setminus \mathcal{I}(\Omega_i), \mathcal{I}(\Omega_i))))$, corresponds to the vertices visited to get the boundary layer of the subdomain. The second term, $E\left(G\left(A\left(\mathcal{I}\left(\sum_{m=0}^{k} \mathcal{L}_m^i\right), \mathcal{I}(\Omega_i)\right)\right)\right)$, searches the next $k+1$ layers of the subdomain. Similarly to CAILU_reorderLayer, this algorithm starts by getting the set of adjacent vertices of $\mathcal{L}_0^i$ in the subgraph $G(A(\mathcal{I}\left(\mathcal{L}_0^i\right), \mathcal{I}(\Omega_i) \setminus \mathcal{I}\left(\mathcal{L}_0^i\right)))$. Finally, reordering this $k+2$ layers has an additional cost equal to $k+2$ times the cost to reorder a layer, $\sum_{m=0}^{k+1} Cost_{\mathcal{L}}\left(\mathcal{L}_m^i\right)$. Note that the cost to obtain the remaining vertices and the corners, as presented in Algorithm 3.5, is equal to $|V(\Omega_i)|$. The third algorithm, in Table 3.1, searches the overlap of a subdomain Ω_i and concatenates it to γ_i. Its cost is equal to the number of non-zeros of $F_k(\gamma_i, :)$, where F_k is obtained from the ILU(k) factorization of A. The search of the CA-ILU(k) overlap is similar to the breadth-first search algorithm, whose cost is related to the number of vertices reached. Therefore, the overhead cost of our algorithm induced by our reordering and the search of the overlap is equal to $\sum_{i=0}^{p-1} Cost_{\mathcal{D}}(\Omega_i)$.

After getting a subdomain and its associated overlap, each processor computes the numerical factorization of $A(\mathcal{I}(\gamma_i), \mathcal{I}(\gamma_i))$. The overlap involves an extra cost during this step. In the general scheme, the factorization of a submatrix $A(\mathcal{I}(V(\Omega_i)), \mathcal{I}(V(\Omega_i)))$ has a cost which depends on the number of non-zeros of the subdomain and the pattern of the submatrix. In our case, the overlap $\gamma_i \setminus V(\Omega_i)$ yields to do more flops during the factorization. Therefore the cost of our algorithm is the sum of the cost to reorder each subdomain, the cost to compute the symbolic factorization, the cost to get the overlap of each subdomain and the cost to factor locally $A(\mathcal{I}(\gamma_i), \mathcal{I}(\gamma_i))$. Note that the cost to obtain F_k is not presented here.

3.3.2 Overlap of CA-ILU(k) in details

Algorithm 3.2 computes the overlap of each subdomain Ω_i in order to avoid the communication during the factorization of A and the application of the ILU(k) preconditioner. To do so, this data is duplicated on each processor i that owns the associated subdomain Ω_i and leads to some redundant computation during both the numerical factorization and the application of the preconditioner. Theorem 17 ensures that the vertices of $\mathcal{L}_{k+2}^i$ are reordered such that all vertices of the subdomain not included in $\mathcal{L}_m^i$, $m \in \{0, \ldots, k+1\}$, are not reached. The first $k+2$ layers are potentially part of the overlap and thus the overlap depends on k. Here, we detail the CA-ILU(k) overlap of a subdomain Ω_i and study how it behaves with respect to p.

To study the scalability of Relation (3.7) is difficult without assumptions on the matrix. Given the case of a five-point stencil matrix as in Figure 3.1a, we can compute precisely the overlap of a subdomain Ω_i. Each subdomain has a rectangular shape of size $H \times h$ where $H \geq h$. These sizes depend on p such that the larger size H is divided in two when p is doubled. Since the shape of all subdomains is rectangular, the size of $\alpha_0^j = \mathcal{N}_{\Omega_i \cup \Omega_j}^1(V(\Omega_i))$ is equal either to

H or h, for $i \in \{0, \ldots, p-1\}$, $i \neq j$. Thus, the size of the set of adjacent vertices of α_0^j in Ω_j, $\alpha_1^j = \mathcal{N}_{\Omega_j}^1\left(\alpha_0^j\right)$, is equal to the size of α_0^j. Recursively, we obtain that the size of $\mathcal{N}_{\Omega_j}^1\left(\alpha_l^j\right)$ is equal to the size of α_l^j, either H with $l \in \{0, \ldots, h-1\}$, or h with $l \in \{0, \ldots, H-1\}$. Equation (3.7) applied on this problem gives

$$\mathcal{O}_{\text{CAILU}_k}(\Omega_i) = \sum_{j|\Omega_j \in \mathcal{N}_\Omega^1(\Omega_i)} (\alpha_0^j + \alpha_1^j + ...\alpha_{k+1}^j) + g(k) \tag{3.17}$$

$$= \sum_{j|\Omega_j \in \mathcal{N}_\Omega^1(\Omega_i)} \left((k+2)\alpha_0^j\right) + g(k), \tag{3.18}$$

where $g(k)$ contains the remaining vertices of the overlap and whose size is constant if $k < \min(H, h)$. Supposing that a subdomain has at most 4 neighboring subdomains, due to the structure of the matrix, Equation (3.18) becomes $\mathcal{O}_{\text{CAILU}_k}(\Omega_i) = 2(k+2)(H+h) + g(k)$.

When the number of subdomains is doubled, the size of the subdomain is reduced such that H is divided by two. The variation of the overlap of the subdomain Ω_i between p subdomains and $2p$ subdomains is

$$2(k+2)(H/2+h) + g(k) - 2(k+2)(H+h) - g(k) \tag{3.19}$$

$$2(k+2)(H/2+h-H-h) \tag{3.20}$$

$$-(k+2)H. \tag{3.21}$$

On general matrices, if the number of adjacent layers from one subdomain into another one is smaller than $k+2$, then the number of subdomains reached is expected to grow. Even if those conditions are verified, the size of the overlap can be so large that it does not fit in the local memory. Therefore, there exists a limit on the size of the overlap and also a limit on the scalability of CA-ILU(k).

Bounding the size of the overlap in memory

In addition to the reduction of k to fit in memory, we propose to limit the size of the overlap to η. Given p processors, a subdomain Ω_i and a parameter τ which is equal to the number of subdomains that can be duplicated in the memory of Ω_i, $\eta = \tau \times \frac{n}{p}$, where $\frac{n}{p}$ is the average number of vertices of a subdomain. Since CAILU_addOverlap algorithm traverses $\widehat{\Omega}^+$ and $\widehat{\Omega}^-$, η has to be split into two parts such that at most half of the memory is dedicated to the search in $\widehat{\Omega}^+$, whereas the remaining is dedicated to the search in $\widehat{\Omega}^-$. The size of γ_i is given by $|\gamma_i| = \eta - \min(|\beta_i|, \eta/2)$. Algorithm 3.3 is rewritten as Algorithm 3.6. It takes as input the same parameters as the original algorithm and the extra parameter η. It starts by getting the adjacent vertices, denoted α_{adj} line 2, of $V(\Omega_i)$ in $\widehat{\Omega}^+$. Then the loop, line 4, iterates until reaching either all vertices needed or the limit of $\eta/2$. The second part of the algorithm does the same as for $V(\Omega_i)$, but for β_i in $\widehat{\Omega}^-$. The stopping criterion is the same as for the first loop except the limit is now $\eta = \eta - |\beta_i|$. This algorithm uses as subroutine getBoundAdjacentLayer which takes as input the set of initial vertices, a set of vertices already visited, the graph where to search and a maximum number of vertices η to return. The subroutine returns the set of adjacent vertices of the initial vertices. We do not provide the code to compute the bounded set of adjacent vertices from a set of vertices since we further present an improved version of it (Algorithm 3.8).

Algorithm 3.6 presents a problem, it does not ensure that the overlap of CA-ILU(k) is included in the overlap of CA-ILU(k+1). In addition, experiments have shown that the performance of the preconditioner is degraded. Given a subdomain Ω_i and its overlap for $k = 0$ and $k = 1$, we

suppose that the size of the overlap for $k = 0$ is equal to η. It follows that the overlap for $k = 1$ cannot fit in memory. As presented in Algorithm 3.6, the construction of the overlap starts by getting the reachable vertices, β_i, in $\widehat{\Omega}^+$. Assuming that the size of β_i is equal to η, the part of the overlap coming from the search in $\widehat{\Omega}^-$, even for the overlap related to ILU(0), is not included in γ_i. Thus, without including the set of reachable vertices resulting from the search in $\widehat{\Omega}^-$, CA-ILU(k) does not even perform an ILU(0) factorization of Ω_i.

Algorithm 3.6 CAILU_addBoundOverlap ($V(\Omega_i)$, $G(A)$, $\widehat{\Omega}$, η)

This algorithm is a modification of Algorithm 3.3 where the size of β_i and γ_i is bounded by $\eta/2$ and $\eta - |\beta_i|$, respectively, where η a parameter.

Input: $V(\Omega_i)$ the i'th subdomain of Ω,
$G(A)$ the graph of the matrix A,
$\widehat{\Omega}$ the graph of the matrix F which stores the ILU(k) factor of A,
η the maximum number of vertices returned by the algorithm

1: Let β_i and γ_i be two empty sets of vertices
2: Compute α_{adj}, the set of adjacent vertices of $V(\Omega_i)$ in $\widehat{\Omega}^+$ whom the size is bounded by $\eta/2$
3: $\beta_i \leftarrow \beta_i \cup \alpha_{adj}$
4: **while** $|\alpha_{adj}| > 0$ and $|\beta_i| < \eta/2$ **do**
5: Compute α_{adj}, the set of adjacent vertices of the previous α, not already in $\beta_i \cup V(\Omega_i)$ in $\widehat{\Omega}^+$ whom the size is bounded by $\eta/2 - |\beta_i|$
6: $\beta_i \leftarrow \beta_i \cup \alpha_{adj}$
7: **end while**
8: $\eta \leftarrow \eta - |\beta_i|$
9: $\beta_i \leftarrow \beta_i \cup V(\Omega_i)$
10: Compute α_{adj}, the set of adjacent vertices of β_i in $\widehat{\Omega}^-$ whom the size is bounded by $\eta/2$
11: $\gamma_i \leftarrow \gamma_i \cup \alpha_{adj}$
12: **while** $|\alpha_{adj}| > 0$ and $|\gamma_i| < \eta$ **do**
13: Compute α_{adj}, the set of adjacent vertices of the previous α, not already in $\beta_i \cup \gamma_i$ in $\widehat{\Omega}^-$ whom the size is bounded by $\eta/2 - |\gamma_i|$
14: $\gamma_i \leftarrow \gamma_i \cup \alpha_{adj}$
15: **end while**
16: $\gamma_i \leftarrow \gamma_i \cup \beta_i$

Output: β_i the union of $V(\Omega_i)$ with the set of reachable vertices of $V(\Omega_i)$ in $\widehat{\Omega}^+$, γ_i the union of β_i with the set of reachable vertices of β_i in $\widehat{\Omega}^-$.

To solve this problem, we propose to update Algorithm 3.6 by adding an outer loop from $j = 0$ to k on each existing loop. The idea is to search the sets of reachable vertices of $V(\Omega_i)$ incrementally in the structure of F_j, the symbolic ILU(j) factor of A. The outer loop starts with F_0 and searches the vertices reachable through a path of length 1 in it. Using F_j corresponds to adding vertices in β_i that are reachable from $V(\Omega_i)$ through a path of length at most $j+1$. This method ensures that the reachable vertices corresponding to ILU(0) are found and added in β_i, before the others. Algorithm 3.7 presents the modifications described above.

Algorithm 3.7 has the same input as Algorithm 3.6 and the ILU parameter k. It iterates twice over j from 0 to k to find the first set of adjacent vertices. Lines 2 and 10 of Algorithm 3.6 are updated, in Algorithm 3.7, to take into account the effect of the outer loop. Now, it searches the set of adjacent vertices α_{adj} from the union of $V(\Omega_i)$ with β_i and the union of β_i with γ_i, respectively. To get α_{adj}, the algorithm calls at each iteration Algorithm 3.8, an implementation of Definition 3.3 with the modification that the size of the set of adjacent vertices is bounded. Algorithm 3.8 presents the modifications of the definition and the way the set of adjacent vertices returned is bounded by the parameter η. This variant takes the set of initial vertices α, the set

of vertices already visited $\alpha_{visited}$, G the graph of a matrix F_k whose elements correspond to the level returned by the ILU(k) factorization, and η the maximum number of vertices to return. This algorithm checks whether the number of elements in α_{adj} does not exceed η (line 2). Each adjacent vertex v is added in α_{adj} under the constraints that it is included neither in the initial set nor in the set of vertices already visited, and the value of the edge (u, v) in $E(G)$ is at most k.

Algorithm 3.7 CAILU_addBoundOrderedOverlap (Ω_i, $G(A)$, $\widehat{\Omega}$, η, k)

This algorithm is a modification of Algorithm 3.6 where the algorithm iterates over j with $0 \leq j \leq k$ and consider edges having a level at most j (in the ILU level-based sense).

Input: Ω_i the i'th subdomain of Ω,
$G(A)$ the graph of the matrix A,
$\widehat{\Omega}$ the graph of the matrix C, ILU(k) factor of A,
η the maximum number of vertices in the overlap, returned by the algorithm,
k the ILU parameter

1: Let β_i and γ_i be two empty sets of vertices
2: $j \leftarrow 0$
3: **while** $j \leq k$ and $|\beta_i| < \eta/2$ **do**
4: $\alpha_{adj} \leftarrow$ getBoundOrderedAdjacentLayer $(V(\Omega_i) \cup \beta_i, V(\Omega_i) \cup \beta_i, \widehat{\Omega}^+, \eta/2 - |\beta_i|, j)$
5: $\beta_i \leftarrow \beta_i \cup \alpha_{adj}$
6: **while** $|\alpha_{adj}| > 0$ and $|\beta_i| < \eta/2$ **do**
7: $\alpha_{adj} \leftarrow$ getBoundOrderedAdjacentLayer $(\alpha_{adj}, V(\Omega_i) \cup \beta_i, \widehat{\Omega}^+, \eta/2 - |\beta_i|, j)$
8: $\beta_i \leftarrow \beta_i \cup \alpha_{adj}$
9: **end while**
10: $j \leftarrow j + 1$
11: **end while**
12: $\eta \leftarrow \eta - |\beta_i|$
13: $\beta_i \leftarrow \beta_i \cup V(\Omega_i)$
14: $j \leftarrow 0$
15: **while** $j \leq k$ and $|\gamma_i| < \eta$ **do**
16: $\alpha_{adj} \leftarrow$ getBoundOrderedAdjacentLayer $(\beta_i \cup \gamma_i, \beta_i \cup \gamma_i, \widehat{\Omega}^-, \eta - |\gamma_i|, j)$
17: $\gamma_i \leftarrow \gamma_i \cup \alpha_{adj}$
18: **while** $|\alpha_{adj}| > 0$ and $|\gamma_i| < \eta$ **do**
19: $\alpha_{adj} \leftarrow$ getBoundOrderedAdjacentLayer $(\alpha_{adj}, \beta_i \cup \gamma_i, \widehat{\Omega}^-, \eta - |\gamma_i|, j)$
20: $\gamma_i \leftarrow \gamma_i \cup \alpha_{adj}$
21: **end while**
22: $j \leftarrow j + 1$
23: **end while**
24: $\gamma_i \leftarrow \gamma_i \cup \beta_i$

Output: β_i the union of $V(\Omega_i)$ with the set of reachable vertices of $V(\Omega_i)$ in $\widehat{\Omega}^+$, γ_i the union of β_i with the set of reachable vertices of β_i in $\widehat{\Omega}^-$.

Algorithm 3.8 getBoundOrderedAdjacentLayer (α, $\alpha_{visited}$, G, η, k)

This algorithm searches the adjacent vertices of the set of initial vertices in the graph G and returns at most η vertices. The output set of adjacent vertices α_{adj} contains vertices whose edges between two vertices, one in each subset, are added by the ILU(k) factorization of A.

Input: α the set of initial vertices,
$\alpha_{visited}$ the set of vertices already visited,
G the graph of a matrix,
η the maximum number of vertices returned by the algorithm,
k the ILU parameter

1: Let α_{adj} be an empty set of vertices
2: **for all** $v \in \mathcal{N}_G^1(\alpha)$ and $|\alpha_{adj}| < \eta$ **do**
3: **if** $(u, v) \leq k$ and $u \in \alpha$ and $v \notin (\alpha \cup \alpha_{visited})$ **then**
4: $\alpha_{adj} \leftarrow \alpha_{adj} \cup \{v\}$
5: **end if**
6: **end for**

Output: α_{adj} the set of adjacent vertices of α in G.

Thus, applying Algorithm 3.7 on a subdomain Ω_i allows to upper bound the memory consumption and does not degrade the factorization between two successive values of k. To illustrate the gain of using Algorithm 3.7, let's take as example a subdomain Ω_i of Ω and all its reachable vertices for $k = 2$ represented in Figure 3.5. Each small circle represents a vertex and the value inside corresponds to the level of fill-in, computed during the symbolic factorization, used to reach the vertex from α_i.

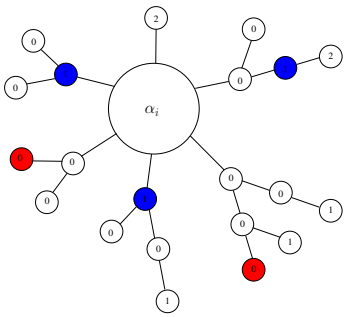


Figure 3.5 – Graph of the reachable vertices of domain Ω_i represented by the set $\alpha = V(\Omega_i)$. Each numbered circle corresponds to a vertex, where the number is the level computed during the symbolic factorization. The blue vertices selected by CAILU_addBoundOverlap with $k = 1$ and $\eta = 10$, while the red one is selected by CAILU_addBoundOverlap with $k = 0$ and $\eta = 10$.

To ease the comprehension, we assume that the graph corresponds to $\widehat{\Omega}^-$ but the following explanation can also be applied on $\widehat{\Omega}^+$. Without loss of generality, we use only for the example indistinctly the number of the vertex as the level of the edge. The description of Figure 3.5 always starts with the vertex of level 2 and we traverse the graph following the clockwise rotation.

We use as the reference, the output of the case $\eta = \infty$ for $k = 0$ and $k = 1$. We compare the output of Algorithm 3.6 with Algorithm 3.7 for both values of k and $\eta = 10$ vertices. Table 3.2 summarizes the vertices found by each call of the getBoundAdjacentLayer algorithm.

η		∞	∞	10	10
k		0	1	0	1
Algorithm					
getBoundAdjacentLayer					
	1st call	0 0 0	0 0 1 0 1	0 0 0	0 0 1 0 1
	2nd call	0 0 0 0 0	0 1 0 0 0 0 0 0 0 0	0 0 0 0 0	0 1 0 0 0
	3rd call	0	1 1 0 1	0	-
getBoundOrderedAdjacentLayer					
$j = 0$	1st call	0 0 0	0 0 0	0 0 0	0 0 0
	2nd call	0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
	3rd call	0	0	0	0
$j \leq 1$	4th call	-	1 1 1 1 1	-	1
	5th call	-	0 0 0 0	-	-
	6th call	-	1	-	-

Table 3.2 – Comparison of the set of vertices returned by each call of getBoundAdjacentLayer and getBoundOrderedAdjacentLayer. When the maximum number of adjacent vertices, η, is equal to 10 and $k = 0$, both methods return the same set. But for $k = 1$, the first method reaches 3 vertices of level 1 instead of 3 vertices of level 0, whereas the second method returns the set when $k = 0$ and only one vertex of level 1.

In the case of $k = 0$ and $\eta = 10$, both algorithms return the same output as the reference. The output of 9 vertices is not impacted by η. In the case $k = 1$, the output of Algorithm 3.6 differs from the reference. Using Figure 3.5, the algorithm adds the three blue vertices of level 1 in the set of adjacent vertices; the condition of the size being verified. As a consequence, the two red vertices of level 0 cannot be added in the set. It leads to $\gamma_i^k \not\subset \gamma_i^{k+1}$.

The interest of Algorithm 3.7 is related to the level of fill. The quality of the factorization increases in general when k increases too. On the other hand, Algorithm 3.7, with $k = 1$, searches the reachable vertices in F_0 (edges of level 0 only) and adds the 9 vertices of the reference. Since it remains one place, the value of j is incremented to search reachable vertices from the set, union of the 9 vertices with α_i, in F_1 (edges of level 0 or 1 only). It follows that the added vertex is of level 1 which is obvious since all reachable vertices of level 0 have been added previously. Finally, the output of the algorithm for $k = 1$ contains the set for $k = 0$ and one extra vertex.

Hence, if the memory bound is reached for a given k, increasing k has no impact on the set of reachable vertices. Moreover, it increases the cost of reordering each domain and the symbolic/numeric factorization. Also, reordering more layers in each subdomain is expected to decay the number of iterations of the solver.

Lemma 19. *Consider a matrix A and the graph $\widehat{\Omega}$ of C, the ILU(k) factorization of A, partitioned into p subdomains as in Definition 7. The construction of β_i and γ_i in Algorithm 3.7 ensures that for all $j \in \{1, \ldots, k\}$,*

$$\text{if } \mathcal{N}^{\infty}_{G(F_j)^+}(V(\Omega_i)) \subseteq \beta_i \text{ then } \mathcal{N}^{\infty}_{G(F_{j-1})^+}(V(\Omega_i)) \subseteq \beta_i, \tag{3.22}$$

and

$$\text{if } \mathcal{N}^{\infty}_{G(F_j)^-}(V(\Omega_i)) \subseteq \gamma_i \text{ then } \mathcal{N}^{\infty}_{G(F_{j-1})^-}(V(\Omega_i)) \subseteq \gamma_i. \tag{3.23}$$

Proof. Since the proof is similar for both Equations (3.22) and (3.23), we focus on the proof of Equation (3.22).

Suppose $|\beta_i \cup \mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))| < \eta/2$. Thus we have $\mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i)) \subseteq \beta_i$ where $\beta_i = \beta_i \cup \mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))$. The construction considers the graph of F_{k-1} before the graph of F_k.

Since the limit $\eta/2$ is not reached, we have

$$\mathcal{N}^{\infty}_{G(F_{k-1})^+}\left(V(\Omega_i)\right) \subseteq \beta_i.$$

□

3.3.3 Relation between ILU(k) and RAS

Additive Schwarz methods behave similar to CA-ILU(k) only when a partition of unity is defined. In this section, we present the relation between RAS and CA-ILU(k). We present the overlap of RAS for a set of vertices α and then we show the relation of inclusion of the overlap of RAS with a level $k+1$ in the overlap of CA-ILU(k) factorization. Consider a matrix A and its directed graph Ω. We remind that F_k is obtained from the symbolic factorization of A such that $V(G(F_k)) = V(\Omega)$ and $E(\Omega) \subseteq E(G(F_k))$. $E(G(F_k))$ is composed of the original edges of A and the additional edges introduced by the incomplete factorization. It is known that $\forall k > 0,\ \ E(G(F_{k-1})) \subseteq E(G(F_k))$.

From (Cai et al., 1999b), the overlap of RAS with a level of k, noted $\mathcal{O}_{RAS^{\delta=k}}$, can be expressed by induction as

$$\forall k \geq 1, \quad \mathcal{O}_{RAS^{\delta=k+1}}(\alpha) = \mathcal{N}^1_{\Omega}\left(\mathcal{O}_{RAS^{\delta=k}}(\alpha)\right) \cup \mathcal{O}_{RAS^{\delta=k}}(\alpha) \tag{3.24}$$

with the basic case

$$\mathcal{O}_{RAS^{\delta=1}}(\alpha) = \mathcal{N}^1_{\Omega}(\alpha). \tag{3.25}$$

Lemma 20. *Consider a matrix A and its associated graph Ω partitioned into p subdomains as in Definition 7. Let Ω_i be a subdomain of Ω. For any positive k,*

$$\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i) \subseteq \mathcal{O}_{CAILU_k}(\Omega_i). \tag{3.26}$$

Proof. We prove it by induction on k.

We recall that the CA-ILU(k) overlap as in Definition 12 is

$$\mathcal{O}_{\mathrm{CAILU}_k}(\Omega_i) = \mathcal{N}^{\infty}_{G(F_k)^+}\left(V(\Omega_i)\right) \cup \mathcal{N}^{\infty}_{G(F_k)^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{G(F_k)^+}\left(V(\Omega_i)\right)\right). \tag{3.27}$$

The first term of Equation (3.27) can be rewritten as

$$\mathcal{N}^{\infty}_{G(F_k)^+}\left(V(\Omega_i)\right) = \mathcal{N}^{1}_{G(F_k)^+}\left(V(\Omega_i)\right) \cup \mathcal{N}^{\infty}_{G(F_k)^+}\left(\mathcal{N}^{1}_{G(F_k)^+}\left(V(\Omega_i)\right)\right). \tag{3.28}$$

The second term of Equation (3.27) is split as

$$\mathcal{N}^{\infty}_{G(F_k)^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{G(F_k)^+}\left(V(\Omega_i)\right)\right) = \mathcal{N}^{\infty}_{G(F_k)^-}\left(V(\Omega_i)\right) \cup \mathcal{N}^{\infty}_{G(F_k)^-}\left(\mathcal{N}^{\infty}_{G(F_k)^+}\left(V(\Omega_i)\right)\right). \tag{3.29}$$

The first term of the union in Equation (3.29) can be rewritten as

$$\mathcal{N}^{\infty}_{G(F_k)^-}\left(V(\Omega_i)\right) = \mathcal{N}^{1}_{G(F_k)^-}\left(V(\Omega_i)\right) \cup \mathcal{N}^{\infty}_{G(F_k)^-}\left(\mathcal{N}^{1}_{G(F_k)^-}\left(V(\Omega_i)\right)\right). \tag{3.30}$$

The initial step of the induction: we prove the inclusion for $k = 0$, $\mathcal{O}_{RAS^{\delta=1}}(\Omega_i) \subseteq \mathcal{O}_{\mathrm{CAILU}_0}(\Omega_i)$.

The overlap of RAS with a level of 1 is the set of adjacent vertices of $V(\Omega_i)$ in Ω written as

$$\mathcal{O}_{RAS^{\delta=1}}(\Omega_i) = \mathcal{N}^1_{\Omega}(V(\Omega_i)), \tag{3.31}$$

$$= \mathcal{N}^1_{\Omega^+}(V(\Omega_i)) \cup \mathcal{N}^1_{\Omega^-}(V(\Omega_i)). \tag{3.32}$$

Since $k = 0$, we have that $G(F_0) = \Omega$ and $E(\Omega) = E(F_0)$ which leads to

$$\mathcal{N}^1_{\Omega^+}(V(\Omega_i)) = \mathcal{N}^1_{G(F_0)^+}(V(\Omega_i)) \tag{3.33}$$

and

$$\mathcal{N}^1_{\Omega^-}(V(\Omega_i)) = \mathcal{N}^1_{G(F_0)^-}(V(\Omega_i)). \tag{3.34}$$

Thus $\mathcal{N}^1_{\Omega^+}(V(\Omega_i)) \cup \mathcal{N}^1_{\Omega^-}(V(\Omega_i)) \subseteq \mathcal{O}_{\mathrm{CAILU}_0}(\Omega_i)$ and this proves the initial step of the proof.

We suppose that Equation (3.26) is verified for k and so we have $\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i) \subseteq \mathcal{O}_{\mathrm{CAILU}_k}(\Omega_i)$. We will prove the relation remains valid for $k+1$, that is $\mathcal{O}_{RAS^{\delta=k+2}}(\Omega_i) \subseteq \mathcal{O}_{\mathrm{CAILU}_{k+1}}(\Omega_i)$.

Since $\mathcal{O}_{RAS^{\delta=k+2}}(\Omega_i) = \mathcal{N}^1_{\Omega}(\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i)) \cup \mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i)$ and $\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i) \subseteq \mathcal{O}_{\mathrm{CAILU}_k}(\Omega_i) \subseteq \mathcal{O}_{\mathrm{CAILU}_{k+1}}(\Omega_i)$, we need to show that

$$\mathcal{N}^1_{\subseteq}(\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i))\, \mathcal{O}_{\mathrm{CAILU}_{k+1}}(\Omega_i).$$

We have that

$$\mathcal{N}^1_{\Omega}(\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i)) = \mathcal{N}^1_{\Omega^+}(\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i)) \cup \mathcal{N}^1_{\Omega^-}(\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i)). \tag{3.35}$$

Then, we prove that all adjacent vertices of any vertex included in the overlap of RAS with a level of k, that are not already in the overlap of CA-ILU(k), are in the overlap of CA-ILU(k+1). Let u be a vertex included in $\mathcal{O}_{RAS^{\delta=k+1}}(\Omega_i)$ and so in $\mathcal{O}_{\mathrm{CAILU}_k}(\Omega_i)$, by hypothesis. Let v be a vertex that belongs to the set of adjacent vertices of u in Ω such that $v \in \mathcal{N}^1_{\Omega}(u) \setminus \mathcal{O}_{RAS^{\delta=k}}(\Omega_i)$. We study two cases.

- Consider $u \in \mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))$. If $\mathcal{I}(u) > \mathcal{I}(v)$ then we have $v \in \mathcal{N}^1_{G(F_k)^-}\left(\mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))\right)$. Otherwise, if $\mathcal{I}(u) < \mathcal{I}(v)$ then $v \in \mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))$. Hence, $v \in \mathcal{O}_{\mathrm{CAILU}_k}(\Omega_i) \subseteq \mathcal{O}_{\mathrm{CAILU}_{k+1}}(\Omega_i)$.

- Consider $u \in \mathcal{N}^{\infty}_{G(F_k)^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))\right)$. If $\mathcal{I}(u) > \mathcal{I}(v)$, then v belongs to the same set of vertices as u, $\mathcal{N}^{\infty}_{G(F_k)^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))\right)$. Otherwise, it remains the case $\mathcal{I}(u) < \mathcal{I}(v)$.

Now, we focus on the case when u is reached during the computation of $\mathcal{N}^{\infty}_{G(F_k)^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{G(F_k)^+}(V(\Omega_i))\right)$ and $\mathcal{I}(u) < \mathcal{I}(v)$. Let t_h be the vertex with the highest index among the vertices that are traversed by the path from a vertex that belongs to $V(\Omega_i)$ to the vertex u, and $t_h \neq u$. Thus $u \in \mathcal{N}^{\infty}_{G(F_k)^-}(t_h)$. If $\mathcal{I}(v) > \mathcal{I}(t_h)$, then $(t_h, v) \in E(G(F_{k+1}))$. Otherwise, there exists at least a vertex, in the path from t_h to u, having an index higher than $\mathcal{I}(v)$. In this last case, we denote as t, the first vertex encounters from u to t_h, that is the reverse path from t_h to u, such that $\mathcal{I}(t) > \mathcal{I}(v)$. Thus the edge (t, v) belongs to $E(G(F_{k+1}))$. Therefore, either through the edge (t_h, v) or the edge (t, v), v is reached from t_h and so $v \in \mathcal{N}^{\infty}_{G(F_{k+1})^-}(t_h) \subseteq \mathcal{N}^{\infty}_{G(F_{k+1})^-}\left(V(\Omega_i) \cup \mathcal{N}^{\infty}_{G(F_{k+1})^+}(V(\Omega_i))\right)$. Therefore, $v \in \mathcal{O}_{\mathrm{CAILU}_{k+1}}(\Omega_i)$. Hence $\mathcal{O}_{RAS^{\delta=k+2}}(\Omega_i) \subseteq \mathcal{O}_{\mathrm{CAILU}_{k+1}}(\Omega_i)$. □

3.4 Implementation of the preconditioner in parallel

The purpose of this section is to present the starting point of CA-ILU(k) preconditioner. In (Grigori and Moufawad, 2015b), Moufawad et al. show results of iterative convergence for small matrices. CA-ILU(0) is compared with Block Jacobi preconditioner, using MATLAB code. The preliminary work is a parallel implementation of CA-ILU(0) in C. Then CA-ILU(0) is interfaced in Petsc. Further, the code is modified to consider a larger k. We compare the implementation of CA-ILU(k) preconditioner with Block Jacobi preconditioner, denoted hereafter as BJacobi, and with Restricted Additive Schwarz, denoted as RAS. We will discuss the number of iterations, the time to solve the preconditioned system, and the impact of the size of the overlap. In this section, we present the details of implementation of CA-ILU(0). We detail the distribution of the matrix A and the right-hand side b. Then we expose the construction of CA-ILU(0) preconditioner. We later show that CA-ILU(0) is interfaced with Petsc and so its application on a vector is done through Petsc.

Consider a matrix A of size $n \times n$ and the left preconditioned system to solve

$$M^{-1}Ax = M^{-1}b, \tag{3.36}$$

where $x \in \mathbb{R}^n$ is the unknown vector, $b \in \mathbb{R}^n$ is the right-hand side and M is a preconditioner. A is split into p block rows where p corresponds to the number of processors. Each block row, $A_i \in \mathbb{R}^{n_i \times n}$, is indexed by processor i, where $i \in \{0, \ldots, p-1\}$ and $\sum_i n_i = n$. The vectors x and b are split in order to respect the splitting of A. At each iteration, GMRES computes a new Krylov basis vector y_{i+1} using the previous vector y_i. In the case of CA-ILU(0), the preconditioner, $M = LU$, is built from the Incomplete LU factorization of A (no fill-in).

3.4.1 Preparation of the distribution of the matrix and the right-hand side

In parallel computation, the number of communication instances as well as the amount of data exchanged impact the performance. The classical way to reduce it is to decrease the dependencies between processors. It especially matters during the parallel computation of a matrix-vector product. To do so, we use multilevel K-way partitioning algorithm introduced by Karypis et al. in (Karypis et al., 1998). This routine takes as input the matrix and the number of partitions, and returns a partition vector that aims to reduce the number of dependencies between the blocks. The following algorithm presents the preparation of A and b in order to distribute them in a block row layout.

Algorithm 3.9 prepareData(A, b, p)

This algorithm presents the preparation of the matrix and the right-hand side.

Input: $A \in \mathbb{R}^{n \times n}$: the matrix,
b : the right-hand side,
p : the number of processors

1: Call KwayPartitioning(A, p) which returns e, the partition vector
2: Create a permutation matrix P from e
3: Compute $A \leftarrow PAP^T$
4: Compute $b \leftarrow Pb$
5: Create a vector $posB$ that stores the position of the first row of each partition in A permuted

Output: A and b permuted following Kway partitioning, $posB$ the starting row index of each partition in A permuted

From the partitioning returned by k-way algorithm, A and b are permuted such that all rows that belong to the partition i, ($i \in \{0, \ldots, p-1\}$) are gathered, starting with elements of partition 0 moved at the beginning of A and b. In order to distribute A and b in block row layout, Algorithm 3.9 creates a vector, denoted $posB$, that stores the position of the first row of each partition. Each partition of A and b can, therefore, be distributed to the corresponding processor using $posB$.

3.4.2 Implementation of the factorization of A in parallel

CA-ILU(0) aims to avoid the communication phase performed during the application of the preconditioner. The key idea is to duplicate data as it is done in s-step methods (J. Demmel, M. Hoemmen, et al., 2008) that performs s consecutive parallel Matrix-Vector product $A^s v$, starting with v, all without communication. In that case, the duplicated data corresponds to the rows associated with the vertices adjacent to the considered subdomain. In the case of ILU(0), to avoid communication is much more difficult than getting the adjacent vertices. As described in Algorithm 3.3, it corresponds to gathering the overlap of a subdomain as in Equation (3.7). The size of the overlap can reach the entire matrix. Therefore, in (Grigori and Moufawad, 2015b), it is proposed to reorder A using Alternating Min-Max Layers algorithm, denoted further as AMML. This purpose is to reduce the size of the overlap. This algorithm (or its variants) decomposes each subdomain into layers, defined in Equations (3.8) and (3.10). In the case of CA-ILU(0), we consider only the first two layers and the remaining vertices of the subdomain. In each subdomain, we compute the boundary layer, i.e., the vertices that are adjacent to the other blocks. Then, the second layer is composed of the adjacent vertices and corresponding rows of the boundary layer of the block. Finally, the remaining vertices of the subdomain are renumbered as the vertices of a layer. The vertices of these three sets are numbered alternatively with maximum and minimum indices.

The description of the implementation of CA-ILU(0) is presented in Algorithm 3.10. Note that lines 2 to 11 correspond to a sequential block of instructions. The efficiency of this section of code can be improved. In the sequential section, the root processor reorders each block A_i. It obtains the overlap of each subdomain and sends the block A_i and its overlap to processor i, and its associated part of b. Then each processor factors its local matrix A_i.

We introduce a few modifications to the original algorithm. In the paper (Grigori and Moufawad, 2015a), the vertices of each layer are also reordered by applying AMML algorithm. In our implementation, we replace this step by applying Nested Dissection algorithm (George, 1973). Although CA-ILU(0) is designed to be used with s-step GMRES, we use classical GMRES. Therefore, the computation of Ay_i performed by GMRES involves communication. This approach allows us to make a fair comparison with BJacobi and RAS, for which this communication step occurs.

Algorithm 3.10 buildCAILU0(A)

This algorithm presents the main steps in our implementation of CA-ILU(0)

Input: $A \in \mathbb{R}^{n \times n}$: The matrix to factor,

Let id be the index of the processor and p be the number of processors
if $id = 0$ **then**
  for $i = 0$ to $p - 1$ **do**
    Reorder the layers of A_i using AMML and Nested Dissection algorithm
  end for
  Apply the same reordering to b
  for $i = 0$ to $p - 1$ **do**
    Compute the dependencies of A_i in order to apply L_i and U_i locally without communication
    Distribute A_i, its overlap and the corresponding part of b to processor i
  end for
end if
Each processor factors its local block of A without communication

Output: L_i and U_i the triangular factors of A on processor i.

3.4.3 Interfacing CA-ILU(0) with Petsc

Petsc library provides an interface that allows us to add a custom preconditioner that can be called by a solver as GMRES. It also provides classical preconditioners as Block Jacobi and Restricted Additive Schwarz. To add a new preconditioner, four basic routines are required by Petsc to be called by the solver. First, we have PCCreate_CAILU0 routine that creates the environment of our preconditioner and PCDestroy_CAILU0 to undo the creation. Second, we have PCSetUp_CAILU0 routine that creates the preconditioner as in Algorithm 3.10. Petsc handles the factorization on each processor using MatILUFactorSymbolic and then MatLUFactorNumeric to get the ILU(0) factored matrix C of A. Third, we create PCApply_CAILU0 routine that is called at each iteration by GMRES. Since this last routine is of importance, we detail it in Algorithm 3.11. We also create an additional routine, PCPrepare_CAILU0, to prepare the matrix as presented in Algorithm 3.9.

During the application of the preconditioner, GMRES calls the PCApply_CAILU0 routine, presented in Algorithm 3.11. This routine takes a vector $x = Ay_i$, the result of the sparse matrix vector product. Since during the setup of the preconditioner, we compute some duplicated data, we need to get the corresponding elements in x. For that, Petsc provides a useful structure called `VecScatter` which handles the communication in order to get locally the duplicated data and the local x into x_{up}. This routine also uses the C matrix, the result of the incomplete factorization of the block. This factored matrix in the sense of Petsc is given to MatSolve routine to perform the classic backward and forward substitutions. Finally, only the non-duplicated elements of w are returned to GMRES.

Algorithm 3.11 PCApply_CAILU0(C,f)

This algorithm presents the main steps during the application of CA-ILU(0) preconditioner.

Input: $C \in \mathbb{R}^{m \times m}$: the factored matrix of A,

Input: f : the local vector, obtained by the computation of $f = Ay_i$.

Let $f_{up} \in \mathbb{R}^m$ be a larger vector than f
Gather in f_{up} the required duplicated data /* *Communication handled by Petsc* */
Solve $w_{up} \leftarrow C f_{up}$ using Petsc routine MatSolve /* *Performed without communication* */
Keep in w, the subset of w_{up} corresponding to the local part of f

Output: w the result of the application of the preconditioner

3.5 Experimental results

3.5.1 Test environment

As described above, our implementation is a parallel C and MPI code that allows us to study the scalability and to compare with two common preconditioners Block Jacobi, denoted further BJacobi, and Restrictive Additive Schwarz, referred hereafter as RAS. All the tests are made on a supercomputer called Poincare, an IBM supercomputer with 92 nodes, located at MDLS, CEA Saclay. Each node is composed of 2 Sandy Bridge E5-2670 (2.60GHz, 8 cores per processor) with 32 GB of memory. The network is an Infiniband QLogic QDR, and the whole system is managed by CentOS 6.5. This machine is used for our MPI tests. It provides the Intel compiler 15.0.090 and Intel-Mpi 5.0.1.035. The third libraries are mainly Petsc version 3.7.5 for the solving part and metis 5.1.0 for the partitioning of the data.

Table 3.3 summarizes the test matrices. All matrices are related to 2D or 3D problems starting with $145,563$ unknowns and up to $3,375,000$ unknowns with $23,490,000$ nnz for the Skycraper problem labeled as 3DSKY150P1.

Matrix	Size	nnz(A)	symmetric	2D/3D	Problem
matvf2dAD400400	160 000	798 400	no	2D	
Elasticity3D4001010	145 563	4 907 997	yes	3D	Elasticity
parabolic_fem	525 825	2 100 225	yes	3D	Diffusion-convection
3DSKY100P1	1 000 000	6 940 000	yes	3D	Skycraper
SPE10	1 094 421	7 478 141	no	3D	Reservoir
3DSKY150P1	3 375 000	23 490 000	yes	3D	Skycraper

Table 3.3 – Matrices used for the parallel test

The Elasticity3D4001010 matrix arises from the linear elasticity problem with Dirichlet and Neumann boundary conditions defined such as

$$\operatorname{div}(\sigma(u)) + f = 0 \qquad \text{on } \Omega \tag{3.37}$$

$$u = 0 \qquad \text{on } \partial\Omega_D \tag{3.38}$$

$$\sigma(u) \cdot n = 0 \qquad \text{on } \partial\Omega_N \tag{3.39}$$

where Ω is a unit square (2D) or cube (3D), $\partial\Omega_D$ is the Dirichlet boundary, $\partial\Omega_N$ is the Neumann boundary, f is some body force, u is the unknown displacement field and $\sigma(.)$ is the Cauchy stress tensor. The latter is given by Hooke's law and can be expressed in terms of Young's Modulus E and Poisson's ration ν. The discretization uses a triangular mesh with $50 \times 10 \times 10$ points on the corresponding vertices. We consider discontinuous E and ν in 3D: $(E_1, \nu_1) = (2 \times 10^{11}, 0.25)$ and $(E_2, \nu_2) = (10^7, 0.45)$. For a more detailed description of the problem see (Jolivet et al., 2013).

The SKY3D matrices come from boundary value problems of the diffusion equations:

$$-\operatorname{div}(\kappa(x)\nabla u) = f \qquad \text{on } \Omega \tag{3.40}$$

$$u = 0 \qquad \text{on } \partial\Omega_D \tag{3.41}$$

$$\frac{\partial u}{\partial n} = 0 \qquad \text{on } \partial\Omega_N \tag{3.42}$$

where Ω is a unit square (2D) or cube (3D). The tensor κ is a given coefficient of the partial differential operator set to $\partial\Omega_D = [0,1] \times \{0,1\}$ in 2D and $\partial\Omega_D = [0,1] \times \{0,1\} \times [0,1]$ in 3D. In both cases, $\partial\Omega_N$ is chosen as $\partial\Omega_N = \partial\Omega \setminus \partial\Omega_D$.

The matrices SKY3D come from skyscraper problems where the domain contains many zones of high permeability which are isolated from each other. More precisely κ is taken as:

$$\kappa(x) = 10^3 * ([10 * x_2] + 1) \qquad \text{if } [10x_i] \text{ is odd, } i = \{1, 2\} \tag{3.43}$$

$$\kappa(x) = 1 \qquad \text{otherwise,} \tag{3.44}$$

where $[x]$ is the integer value of x.

We compare the performance of the three methods on a set of matrices that are also used in (Grigori, Moufawad, and Nataf, 2016; Achdou et al., 2007; Niu et al., 2010) where they are described in more details. We compare CA-ILU(0) preconditioner with Block Jacobi and RAS preconditioner and without preconditioner to study the stability, the number of iterations, the residual and the runtime of the parallel version.

In all tests, we consider A as scaled. The scaling applied here is the common method used in order to reduce the effect of possibly very high values on the diagonal. Therefore, for each matrix $A \in \mathbb{R}^{n \times n}$, we compute DAD with

$$D = \begin{bmatrix} 1/\sqrt{\max(abs(a_{1,:}))} & \dots & 0 \\ \vdots & 1/\sqrt{\max(abs(a_{2,:}))} & \\ & & \ddots \\ 0 & & 1/\sqrt{\max(abs(a_{n,:}))} \end{bmatrix} \tag{3.45}$$

where $\max(abs(a_{i,:}))$ is the largest absolute element on the i'th row of the matrix A.

The true solution vector x_s is constructed such that $x_s = w/\|w\|_2$ with $w(i) = 1, i \in \{1, \dots, n\}$. The right-hand side b is given by the product Ax_s and then normalized. For each preconditioner, the test starts loading the matrix, then scales it. Then all preconditioners permute A by calling KwayPartitioning algorithm. Then A is distributed except in the case of CA-ILU(0) where the distribution is handled by PCSetUp_CAILU0. GMRES solver is setup using a maximum number of iterations of 1000, a restart of 200 and modified Gram-Schmidt orthogonalization; except other mention, the relative tolerance is set to $1e-6$. Finally KSPsolve routine takes as input A and b, and returns x the computed solution.

3.5.2 Parallel results of CA-ILU(0)

In this section, we present the parallel results of CA-ILU(0) compared with BJacobi and RAS with one and two levels of overlap. The number of subdomains, which is equal to the number of processors, goes from 16 to 512, except for few cases where it stops at 256 subdomains. For each problem, we first discuss the number of iterations to converge.

We first discuss the number of iterations to converge for each method that is summarized in Table 3.4. For matvf2dAD400400 problem, RAS is close to 260 iterations when the number of subdomains increases, whereas CA-ILU(0) increases from 276 to 326 iterations. Concerning the elasticity3d4001010 problem, the number of iterations of BJacobi and RAS are almost the same up to 256 subdomains. Their number of iterations starts roughly at 1300 iterations whereas CA-ILU(0) needs almost 1700 iterations. On this problem particularly, CA-ILU(0) converges with the largest number of iterations, and, this, for all subdomains. The number of iterations for all methods on Parabolic_fem problem does not change when the number of subdomains increases. BJacobi is slower with around 530 iterations whereas the others are close to 460. Note that for 512 subdomains, CA-ILU(0) converges in 407 iterations. For 3DSKY problems, the best results are obtained by RAS with 300 iterations for the 100P1 problem and 350 iterations for the 150P1

problem. BJacobi and CA-ILU(0) require a similar number of iterations, around 360 and 340, respectively, for the 100P1 problem. For the 150P1 case, CA-ILU(0) is closer to RAS and for 256 and 512 subdomains CA-ILU(0) is faster. Therefore, in almost all cases, RAS preconditioner is the fastest in terms of iterations. Except on elasticity3d4001010, CA-ILU(0) converges in fewer iterations than BJacobi.

The elasticity3d4001010 problem is the densest among those presented in Table 3.3. It suggests that the unknowns are more connected. As a consequence, the reordering of each subdomain is expected to be more important. The reordering increases the number of iterations to converge. Table A.2 presents the stability of the methods. In addition to the number of iterations, we have the relative residual of the preconditioned system returned by GMRES, the relative residual of the original system $\|b - Ax\|_2/\|b\|_2$ where x is the computed solution vector. Since we generate the exact solution x_s, we compute the relative error on x. CA-ILU(0) reaches the same accuracy as the other methods, but it requires more iterations for that. Table A.3 presents the impact of the reordering on the number of iterations for BJacobi and RAS(2). We observe that for 16 subdomains the permutation leads to an increase of 281 and 389 iterations for BJacobi and RAS(2), respectively. From 16 to 64 subdomains, RAS(2)_permuted and CA-ILU(0) perform exactly the same number of iterations to converge. For a larger number of subdomains, CA-ILU(0) needs slightly fewer iterations to converge compared to RAS(2)_permuted, clearly related to their similar overlap. Note that the overlap of RAS does not suffer from any permutation.

	Problem	matvf2dAD400400	elasticity3d4001010	parabolic_fem	3DSKY100P1	SPE10	3DSKY150P1
	Reference	1638	3000	2750	399	1683	399
np = 16	BJacobi	345	**1273**	537	364	512	440
	CAILU(0)	276	1678	463	337	550	359
	RAS(1)	264	1398	474	304	514	351
	RAS(2)	**261**	1289	**458**	**294**	**497**	**348**
np = 32	BJacobi	348	**1281**	539	363	537	402
	CAILU(0)	280	1967	461	336	572	360
	RAS(1)	266	1449	476	305	547	**352**
	RAS(2)	**260**	1526	**456**	**295**	**521**	**352**
np = 64	BJacobi	349	1589	532	363	566	439
	CAILU(0)	286	2326	463	336	534	369
	RAS(1)	268	**1550**	472	309	523	**349**
	RAS(2)	**264**	1587	**455**	**301**	**505**	350
np = 128	BJacobi	354	1786	535	362	583	398
	CAILU(0)	305	2197	**455**	342	545	372
	RAS(1)	270	**1730**	481	337	543	**347**
	RAS(2)	**265**	1874	463	**308**	**513**	350
np = 256	BJacobi	357	2105	516	372	654	399
	CAILU(0)	326	2450	462	347	567	**345**
	RAS(1)	275	**1786**	462	337	579	349
	RAS(2)	**267**	1896	**436**	**311**	**537**	348
np = 512	BJacobi		2503	535		643	395
	CAILU(0)		2521	**407**		615	**347**
	RAS(1)		1994	476		650	351
	RAS(2)		**1977**	471		**581**	348

Table 3.4 – Comparison of the number of GMRES iterations to converge using CA-ILU(0) with BJacobi and RAS on our test problems for a number of partitions increasing from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

Table 3.5 presents the evolution of the overlap for CA-ILU(0), RAS(1) and RAS(2) with respect to the number of subdomains. In addition to the size of each subdomain, the size of the overlap is discussed in three columns, which are the average, the minimum and the maximum size among all subdomains. The size of the overlap of RAS increases slowly until 64 to reach

751 and 1530 for 1 level and 2 level of overlap, respectively. Then it drops to 334 and 816, respectively, for 512 subdomains. On the other hand, CA-ILU(0) overlap size is the same in average, minimum and maximum as RAS(2) for 16 and 32 subdomains but it becomes twice the size of the overlap of RAS(2) for 64 subdomains. Further, the overlap reaches $14,000$ for 512 subdomains. For the maximum overlap, on 512 subdomains, CA-ILU(0) has an overlap 20 times larger than the largest overlap of RAS(2) (3.11×10^4 for CA-ILU(0) against 1.42×10^3 for RAS(2)). In the case of RAS(2)_permuted, the overlap is the same for 16 and 32 subdomains and close for 64 subdomains. This overlap leads to the same number of iterations. Therefore the overlap of RAS is the same as CA-ILU(0), this corresponds to the relation of inclusion in section 3.3.3.

Concerning the parabolic_fem problem, Table A.4 presents the stability of CA-ILU(0), compared with the other methods. As for elasticity3d4001010 problem, the behavior is the same. The number of iterations of BJacobi is higher than the others with 100 iterations. CA-ILU(0) is better than RAS(1) (or equal for 256 subdomains) and close to RAS(2). Moreover, CA-ILU(0) requires fewer iterations for 128 and 512 subdomains. Note that this number is particularly low for 512 subdomains (50 iterations less than the other subdomains). Focusing on the size of the overlap in Table 3.6, RAS(2) and CA-ILU(0) are close to each other. Their size decreases when the number of subdomains increases. The maximum size is 1470 for CA-ILU(0) versus 1410 for RAS(2), and drops to 432 and 328 for CA-ILU(0) and RAS(2), respectively. Note that for all subdomains, the size of RAS(2) overlap is smaller than the size of the overlap of CA-ILU(0).

		Domain size			Overlap size		
	PC	mean	min	max	mean	min	max
np = 16	CAILU(0)	9097	8838	9354	1.37e+03	7.32e+02	1.47e+03
	RAS(1)	-	-	-	6.83e+02	3.66e+02	7.35e+02
	RAS(2)	-	-	-	1.37e+03	7.32e+02	1.47e+03
np = 32	CAILU(0)	4548	4425	4683	1.42e+03	7.26e+02	1.51e+03
	RAS(1)	-	-	-	7.09e+02	3.63e+02	7.53e+02
	RAS(2)	-	-	-	1.42e+03	7.26e+02	1.51e+03
np = 64	CAILU(0)	2274	2208	2342	3.40e+03	7.74e+02	7.81e+03
	RAS(1)	-	-	-	7.51e+02	3.63e+02	8.37e+02
	RAS(2)	-	-	-	1.53e+03	7.26e+02	1.78e+03
np = 128	CAILU(0)	1137	1104	1171	9.07e+03	2.09e+03	2.22e+04
	RAS(1)	-	-	-	6.12e+02	3.42e+02	7.71e+02
	RAS(2)	-	-	-	1.33e+03	7.23e+02	1.69e+03
np = 256	CAILU(0)	568	552	585	1.09e+04	4.50e+03	2.27e+04
	RAS(1)	-	-	-	4.53e+02	2.84e+02	5.88e+02
	RAS(2)	-	-	-	1.05e+03	6.30e+02	1.34e+03
np = 512	CAILU(0)	284	276	292	1.40e+04	3.36e+03	3.11e+04
	RAS(1)	-	-	-	3.34e+02	1.77e+02	5.62e+02
	RAS(2)	-	-	-	8.16e+02	4.05e+02	1.42e+03

Table 3.5 – Comparison of the overlap of CA-ILU(0) with RAS on elasticity3D4001010 for different number of partitions.

		Domain size			Overlap size		
	PC	mean	min	max	mean	min	max
np = 16	CAILU(0)	32864	32845	32885	1.16e+03	6.43e+02	1.47e+03
	RAS(1)	-	-	-	5.66e+02	3.06e+02	7.05e+02
	RAS(2)	-	-	-	1.13e+03	6.13e+02	1.41e+03
np = 32	CAILU(0)	16432	16405	16451	9.09e+02	4.21e+02	1.23e+03
	RAS(1)	-	-	-	4.31e+02	2.03e+02	5.80e+02
	RAS(2)	-	-	-	8.65e+02	4.07e+02	1.17e+03
np = 64	CAILU(0)	8216	8176	8247	7.11e+02	2.96e+02	9.55e+02
	RAS(1)	-	-	-	3.30e+02	1.41e+02	4.53e+02
	RAS(2)	-	-	-	6.64e+02	2.83e+02	9.12e+02
np = 128	CAILU(0)	4108	4051	4152	5.40e+02	2.33e+02	7.27e+02
	RAS(1)	-	-	-	2.42e+02	1.06e+02	3.15e+02
	RAS(2)	-	-	-	4.89e+02	2.13e+02	6.36e+02
np = 256	CAILU(0)	2054	1994	2077	4.07e+02	1.70e+02	5.31e+02
	RAS(1)	-	-	-	1.74e+02	8.20e+01	2.21e+02
	RAS(2)	-	-	-	3.54e+02	1.65e+02	4.48e+02
np = 512	CAILU(0)	1027	997	1057	3.10e+02	1.04e+02	4.32e+02
	RAS(1)	-	-	-	1.25e+02	5.00e+01	1.61e+02
	RAS(2)	-	-	-	2.55e+02	1.01e+02	3.28e+02

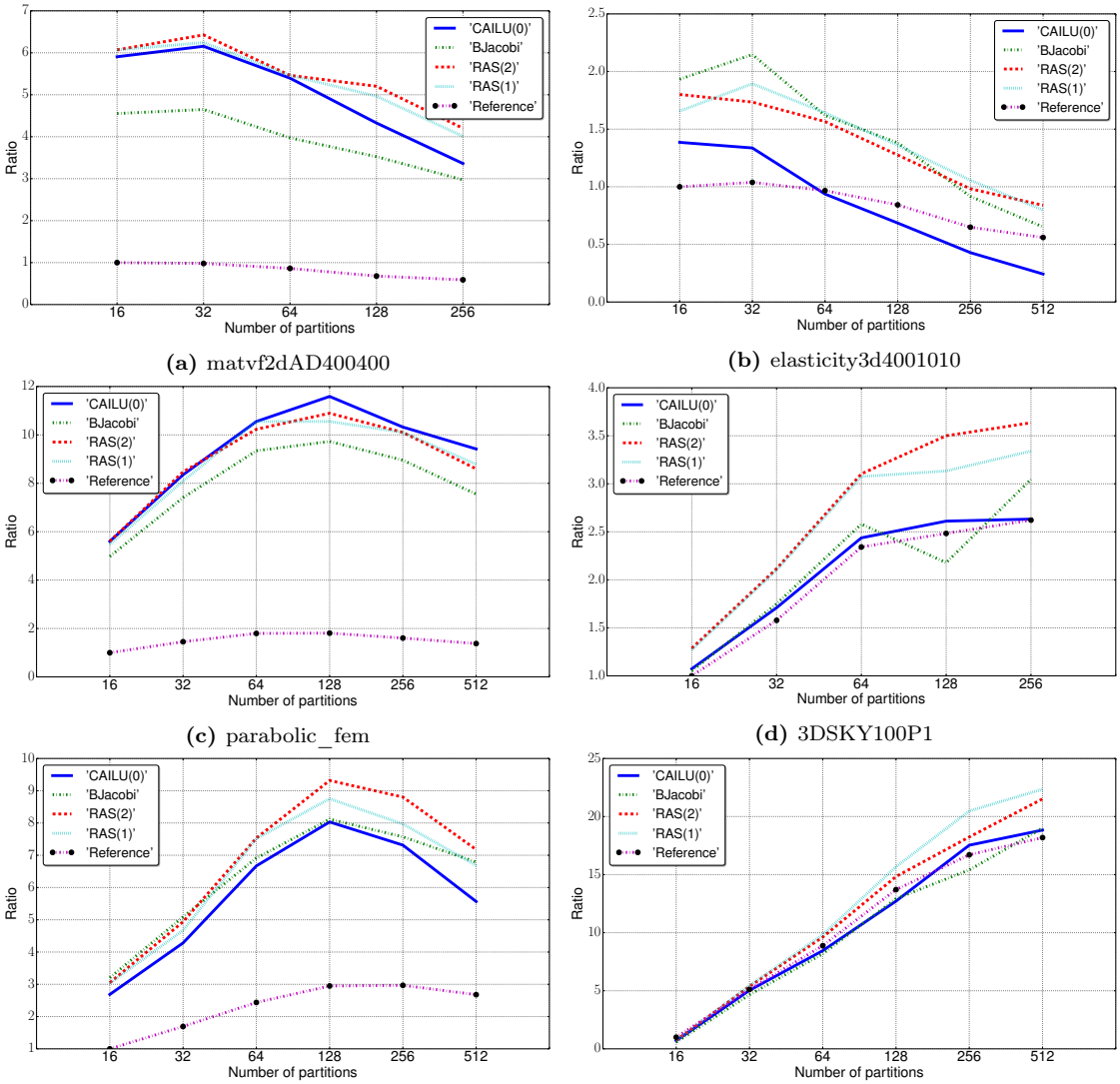
Table 3.6 – Comparison of the overlap of CA-ILU(0) with RAS on parabolic_fem for different number of partitions.

Parallel efficiency

In this section, we present the parallel efficiency of CA-ILU(0) compared with BJacobi and RAS with one and two levels of overlap. We measure the time using `MPI_Wtime` to solve the preconditioned system using GMRES of Petsc (v. 3.7.5). The number of subdomains, which is equal to the number of processors, goes from 16 to 512, except for few cases where it stops at 256 subdomains. For each problem, we first discuss the number of iterations to converge. We consider the runtime of GMRES without preconditioner to converge as a reference, and we use the runtime with 16 subdomains, *i.e.* 16 MPI processors, as a runtime reference. Then we compute the ratio for each method with respect to the reference as follows

$$Ratio(method, p) = runtime_{reference}/runtime_{method}(p) \tag{3.46}$$

where *method* corresponds to one of the preconditioners or GMRES without preconditioner, and p is the number of subdomains.



(a) matvf2dAD400400 **(b)** elasticity3d4001010

(c) parabolic_fem **(d)** 3DSKY100P1

(e) SPE10 **(f)** 3DSKY150P1

Figure 3.6 – Comparison of the runtime of CA-ILU(0), BJacobi, RAS(1) and RAS(2) with the reference, GMRES without a preconditioner for the test matrices presented in Table 3.3. The number of subdomain, which is equivalent to the number of partitions, increases from 16 to 512 (256 subdomains in two cases). GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$. Higher ratio means better method.

We now focus on the runtime to compute the solution for each method. We are using as reference GMRES without preconditioner, denoted in all subfigures of Figure 3.6, as $Reference$. We choose as the reference time, the time for $Reference$ to solve the system on 16 processors. We plot in Figure 3.6 the ratio of the runtime of each method with respect to the reference time as in Relation (3.46), for each test matrix presented in Table 3.3. At first, in Subfigure 3.6a, BJacobi runtime is the largest of the preconditioned system. Its ratio of the runtime is around 4.5 for 16 subdomains whereas the other methods have a ratio of 6. CA-ILU(0) is close to the

RAS versions but it never outperforms them. When the number of subdomains increases, the performance of all methods decreases. CA-ILU(0) gets closer to BJacobi, which has a ratio of 3. The maximum efficiency for this problem is obtained for 32 subdomains. The elasticity3d4001010 problem from Subfigure 3.6b leads to a poor efficiency for all methods. BJacobi is the fastest preconditioner with a maximum ratio of 2.2 for 32 subdomains. It becomes less efficient when the number of subdomains is larger than 256. From the overlap size study presented in Table 3.5, the performance of CA-ILU(0) is expected to be deteriorated with respect to the number of subdomains. CA-ILU(0) has a maximum efficiency of 1.4. The reference has a better ratio, starting with 64 subdomains. Parabolic_fem runtimes are presented in Subfigure 3.6c. BJacobi has the lowest ratios for all subdomains, starting at 5 and reaching a maximum of 9.6 for 128 subdomains. Both RAS versions are close to CA-ILU(0) and RAS(2) is slightly faster for 32 subdomains. However, CA-ILU(0) is the fastest preconditioner with a maximum ratio of 11.5 for 128 subdomains. 3DSKY problems have a similar runtime. In the case of 3DSKY100P1, all methods have an increasing ratio of runtime up to 64 subdomains. For a number of subdomains larger than 64, the reference and CA-ILU(0) stagnate. The other methods still increase with a lower coefficient. Note that BJacobi has an unexpected behavior for $p = 128$. In the case of 3DSKY150P1, the breaking line occurs for 256 subdomains. For SPE10 problem, Subfigure 3.6e, CA-ILU(0) preconditioner is slower than the others. This is due to the size of the overlap. In Table A.9, the largest overlap of CA-ILU(0) is 2.25×10^4 for 16 subdomains and decreases to 1.69×10^4 for 512 subdomains. In comparison, RAS(2) has a size of 1.32×10^4 for $p = 16$ and drops to 2.45×10^3 for $p = 512$.

Replacement of ILU(0) by LU in each diagonal block

In some cases, it is common for BJacobi and RAS to use LU instead of ILU(0) for the factorization of each diagonal block. We study the behavior of CA-ILU(0) when each subdomain is factored using LU with partial pivoting instead of ILU(0). The construction of the preconditioner is the same as before, except for the numerical factorization. That is the reordering of each subdomain, the search of the overlap for each subdomain is identical to CA-ILU(0). We further denote this modification of our preconditioner as CA-ILU(0)-LU. CA-ILU(0)-LU is compared to RAS and BJacobi also using LU to factor each subdomain, denoted RAS(*)-LU and BJacobi-LU, respectively.

We perform the same tests as for the classical CA-ILU(0) algorithm on the problems presented in Table 3.3. Previous results show that in the case of CA-ILU(0), a bigger overlap leads to more FLOPS and so a degraded performance compared to the other preconditioners. From the overlap point of view, the performance of CA-ILU(0)-LU is expected to be better and even to become better than RAS. At first, we study SPE10 problem with 16 to 512 subdomains and we summarize the convergence results in Table 3.7. We observe that BJacobi-LU needs at least 184 iterations and up to 410 iterations for 512 subdomains. CA-ILU(0)-LU and RAS(2)-LU are quite close starting with 69 and 71 iterations, respectively, for 16 subdomains and increasing until 151 and 174 iterations for 512 subdomains. For each number of subdomains, CA-ILU(0)-LU needs fewer iterations than RAS(2)-LU to converge. RAS(1)-LU needs at least 30 more iterations than the two most competitive preconditioners and the gap grows to 76 iterations for 512 cores. All these methods are as stable as others with a relative residual of $1e-6$, a relative error on the system of $1e-7$ and an error on the solution close to $1e-7$, with a slightly worse error of $1e-6$ for BJacobi. For all numbers of subdomains studied, CA-ILU(0)-LU needs fewer iterations to converge, and this is due to the size of the overlap.

In Figure 3.7, we plot the overlap used by RAS(1)-LU, RAS(2)-LU and CA-ILU(0)-LU. As a reminder, the size of the overlap is the number of vertices, i.e., the number of rows, composing

the overlap. The replacement of ILU by LU factorization on each diagonal block does not impact the size of the overlap. Hence, the overlap size of RAS(1)-LU is strictly the same as RAS(1). In Figure 3.7, bars correspond to the average overlap size of each method and for different subdomains. Associated with each bar, the lower and higher lines represent the minimum and maximum, respectively, of the overlap size. For a small number of subdomains, CA-ILU(0)-LU overlap size is 1.5 times larger than RAS(2)-LU and almost 3 times the size of RAS(1)-LU overlap. This gap should lead our method to have a better runtime than both versions of RAS. Considering the size of each subdomain, the overlaps of all methods involve negligible number of FLOPS until 128 subdomains. For 128 subdomains, the size of each subdomain is roughly $1e4$ and the size of CA-ILU(0)-LU overlap is at most 1.8×10^4. Then, for a larger number of subdomains, the maximum overlap size of CA-ILU(0)-LU is roughly identical. Thus our method stops scaling in terms of redundant computations for 128 subdomains. This behavior points out the fact that the size of the overlap and hence of redundant computation has to be related to the size of each subdomain. When the ratio of the overlap size over the size of the subdomains becomes large enough, the amount of redundant computation becomes too important and the method stops scaling.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	1683	9.99e-07	9.99e-07	1.95e-05
16	BJacobi-LU	184	3.47e-06	7.25e-07	5.66e-07
	CAILU(0)-LU	**69**	3.32e-06	3.90e-07	2.79e-08
	RAS(1)-LU	113	3.58e-06	4.68e-07	2.81e-07
	RAS(2)-LU	71	3.37e-06	5.01e-07	2.92e-08
32	BJacobi-LU	280	3.12e-06	1.01e-06	1.54e-06
	CAILU(0)-LU	**107**	3.09e-06	4.20e-07	4.39e-08
	RAS(1)-LU	140	3.20e-06	4.00e-07	1.76e-07
	RAS(2)-LU	110	3.38e-06	4.65e-07	6.46e-08
64	BJacobi-LU	301	3.02e-06	5.82e-07	1.49e-06
	CAILU(0)-LU	**108**	2.91e-06	4.48e-07	4.09e-08
	RAS(1)-LU	149	3.16e-06	3.85e-07	2.16e-07
	RAS(2)-LU	116	3.29e-06	5.15e-07	5.54e-08
128	BJacobi-LU	342	2.85e-06	6.65e-07	1.38e-06
	CAILU(0)-LU	**115**	3.25e-06	3.68e-07	1.54e-07
	RAS(1)-LU	171	3.11e-06	4.40e-07	2.55e-07
	RAS(2)-LU	121	3.24e-06	4.24e-07	1.61e-07
256	BJacobi-LU	367	2.68e-06	6.68e-07	1.30e-06
	CAILU(0)-LU	**128**	3.09e-06	4.09e-07	1.30e-07
	RAS(1)-LU	199	2.79e-06	4.97e-07	1.52e-07
	RAS(2)-LU	142	2.84e-06	4.93e-07	7.15e-08
512	BJacobi-LU	410	2.54e-06	5.85e-07	2.58e-06
	CAILU(0)-LU	**151**	2.89e-06	3.89e-07	1.35e-07
	RAS(1)-LU	250	2.82e-06	4.27e-07	5.32e-07
	RAS(2)-LU	174	2.93e-06	4.45e-07	1.84e-07

Table 3.7 – Comparison of CA-ILU(0)-LU with BJacobi-LU and RAS-LU, using LU on each diagonal block, on the problem SPE10 for a number of subdomains increasing from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

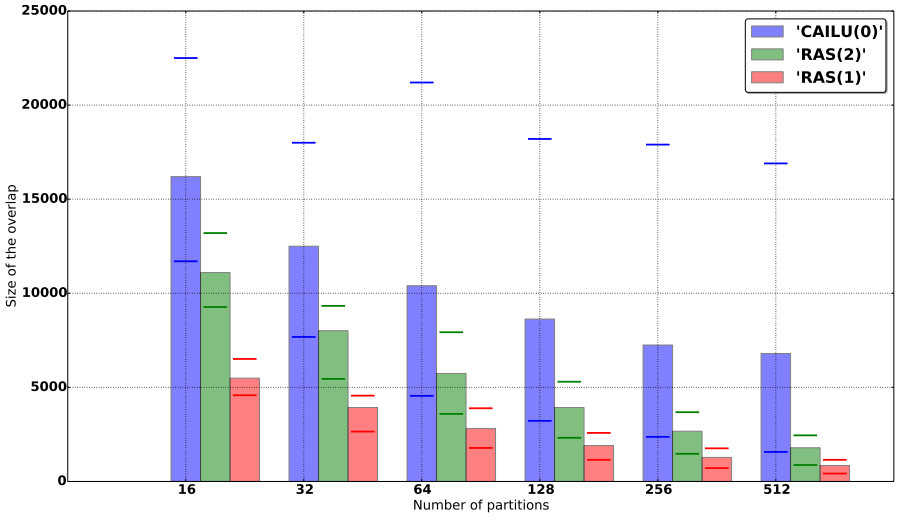


Figure 3.7 – Comparison of the overlap of CA-ILU(0) with RAS(1), RAS(2) on the problem SPE10 from 16 to 512 subdomains. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

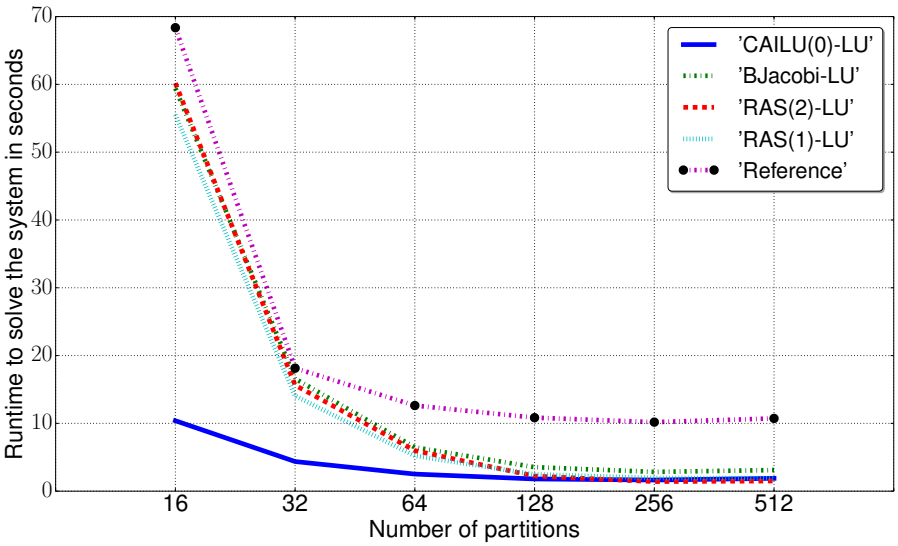


Figure 3.8 – Comparison of runtime to solve SPE10 problem with CA-ILU(0)-LU, BJacobi-LU RAS(1)-LU and RAS(2)-LU, on Poincare System. All methods use LU in each diagonal block. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$. The number of partitions is equal to the number of cores.

Now we compare the runtime to compute the solution for each preconditioner and we plot the results in Figure 3.8. At first, BJacobi-LU and RAS-LU-like compute the solution in 60 seconds for 16 subdomains, compared to CA-ILU(0)-LU which converges in 10 seconds. The runtimes decrease up to 128 subdomains. Finally, CA-ILU(0)-LU and RAS-LU-like need less than 2 seconds to converge whereas BJacobi-LU and the reference converge in 3 seconds and

10 seconds, respectively. As observed from the study of the overlap, CA-ILU(0)-LU does not scale beyond 64 subdomains. Note that CA-ILU(0)-LU is only outperformed by RAS(2)-LU when the number of subdomains is larger than 128. This result confirms that the size of the overlap limits the performance of CA-ILU(0)-LU by inducing more FLOPS at each application of the preconditioner. The efficiency of our method is strongly related to the size of the overlap compared to the size of the subdomain.

We next perform the same study on 3DSKY150P1 problem. The numerical efficiency of the three preconditioners is summarized in Table 3.8. Since the problem is larger than SPE10, the number of subdomains increases from 32 to 512. The relative residual of the preconditioned systems are slightly worse than the relative residual of the reference, varying from $1e-5$ to $1e-6$. The error of the system is close to $1e-6$ and the solution error is equal to $1e-5$. The number of iterations of RAS(2)-LU behaves similarly to the SPE10 case. RAS(2)-LU needs more iterations to converge, increasing from 103 to 307 iterations when increasing the number of subdomains. CA-ILU(0)-LU converges in 95 iterations for 32 subdomains and reaches 260 iterations for 512 subdomains. Our method is the fastest in term of iterations, except for 64 subdomains where RAS(1)-LU converges with 13 less iterations. BJacobi-LU is the preconditioner suffering the most when increasing the number of subdomains. To converge, BJacobi-LU requires at least 115 iterations and reaches 545 iterations for 256 subdomains. Moreover, comparing with BJacobi using ILU(0) presented in Table A.7, BJacobi-LU requires more iterations when the number of subdomains is 128 or higher.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	399	9.94e-07	9.94e-07	1.50e-04
32	BJacobi-LU	115	4.38e-05	7.90e-06	1.45e-04
	CAILU(0)-LU	**95**	5.18e-05	4.35e-06	6.41e-05
	RAS(1)-LU	115	4.99e-05	5.57e-06	9.92e-05
	RAS(2)-LU	103	4.81e-05	4.74e-06	6.35e-05
64	BJacobi-LU	138	2.86e-05	4.70e-06	1.50e-04
	CAILU(0)-LU	138	3.71e- 05	3.05e-06	5.17e-05
	RAS(1)-LU	**125**	3.92e-05	4.12e-06	9.05e-05
	RAS(2)-LU	142	3.71e-05	2.91e-06	5.61e-05
128	BJacobi-LU	482	4.11e-06	6.63e-07	5.81e-05
	CAILU(0)-LU	**153**	2.25e-05	2.36e-06	4.09e-05
	RAS(1)-LU	194	1.67e-05	1.72e-06	6.08e-05
	RAS(2)-LU	161	2.18e-05	2.36e-06	4.35e-05
256	BJacobi-LU	545	2.44e-06	3.82e-07	4.35e-05
	CAILU(0)-LU	**171**	1.72e-05	1.67e-06	3.53e-05
	RAS(1)-LU	326	4.06e-06	3.46e-07	1.60e-05
	RAS(2)-LU	181	1.63e-05	1.60e-06	4.03e-05
512	BJacobi-LU	530	2.12e-06	3.54e-07	4.65e-05
	CAILU(0)-LU	**260**	2.67e-06	1.84e-07	6.63e-06
	RAS(1)-LU	344	2.40e-06	2.26e-07	8.83e-06
	RAS(2)-LU	307	2.61e-06	2.34e-07	6.89e-06

Table 3.8 – Comparison of the number of iterations to converge for CA-ILU(0)-LU with BJacobi-LU and RAS-LU, using LU in each diagonal block, on the problem 3DSKY150P1 for a number of subdomains increasing from 16 to 512, and on Poincare system. GMRES is set with a maximum of 3000 iterations, a restart of 200, and a relative tolerance of $1e-6$.

In Figure 3.9, the number of iterations of BJacobi-LU drastically increases for 128 subdomains. This unexpected behavior is not directly related to the preconditioner. For 32 and 64 subdomains, BJacobi-LU converges in less than 200 iterations, which corresponds to the GMRES restart parameter. For 128 subdomains and higher, the preconditioned system requires at least one restart. Figure 3.10 shows the evolution of the relative residual returned by GMRES at each iteration for each method. Focusing on BJacobi-LU, we observe a peak at 200 iterations which corresponds to the value of GMRES restart.

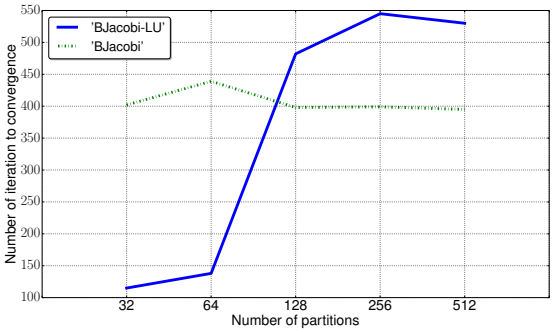


Figure 3.9 – Comparison of the number of iterations of BJacobi-ILU(0) with BJacobi-LU on 3DSKY150P1 problem ; the number of subdomains increases from 32 to 512.

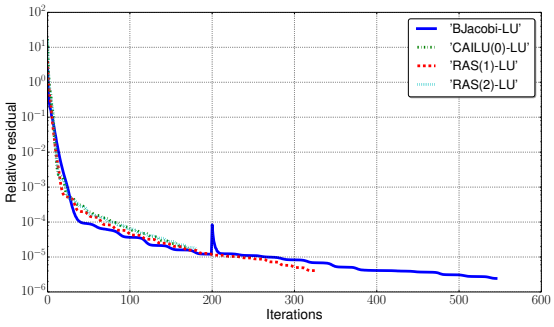


Figure 3.10 – Evolution of the relative residual returned by GMRES after each iteration for each preconditioner on 3DSKY150P1 problem, with 256 subdomains.

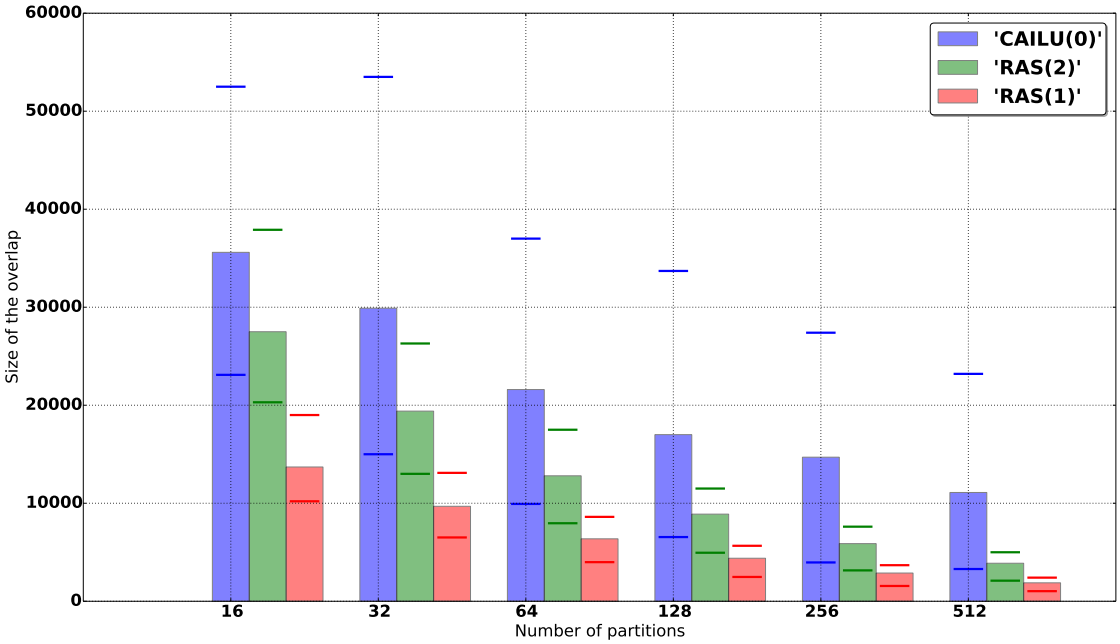


Figure 3.11 – Comparison of CA-ILU(0)-LU with RAS(1)-LU, RAS(2)-LU on the problem 3DSKY150P1 from 32 to 512 subdomains, on Poincare system. GMRES is set with a maximum of 3000 iterations, a restart of 200, and a relative tolerance of $1e-6$.

Figure 3.11 presents the size of the overlap of CA-ILU(0)-LU, RAS(1), and RAS(2). The average of the CA-ILU(0)-LU overlap size decreases when the number of subdomains increases. The maximum overlap is roughly the same for 32 and 64 subdomains, and then decreases from 5.4×10^4 to 2.3×10^4. Comparing with RAS(2), CA-ILU(0)-LU has in average 8×10^4 more vertices for all number of subdomains. Taking into account the size of each subdomain, CA-ILU(0)-LU has a maximum overlap size of 3.3×10^4, larger than the block size of 2.6×10^4, for 128 subdomains. Therefore the method factors twice the local size it should do. CA-ILU(0)-LU reaches its limit of scalability for 128 subdomains. Note that this is not the case for CA-ILU(0) since the incomplete factorization does not fill-in the subdomain. Thus, this limit is 256 subdomains as shown in Figure 3.6f. Figure 3.12 presents the runtime to solve 3DSKY150P1 problem by the three preconditioners from 32 to 512 subdomains. CA-ILU(0)-LU outperforms the other preconditioners for all studied number of subdomains, except 512. But after that, RAS becomes faster since we pay the price of a bigger overlap. As conjectured by the overlap curves, CA-ILU(0)-LU does not scale beyond 128 subdomains. We note that the fastest method to solve this problem is GMRES alone. Due to the small gap of iterations between GMRES alone and all preconditioners, the overhead cost of applying locally LU is not absorbed by the reduction of the number of iterations. Concerning CA-ILU(0)-LU, the preconditioner suffers from the reordering compared to GMRES alone.

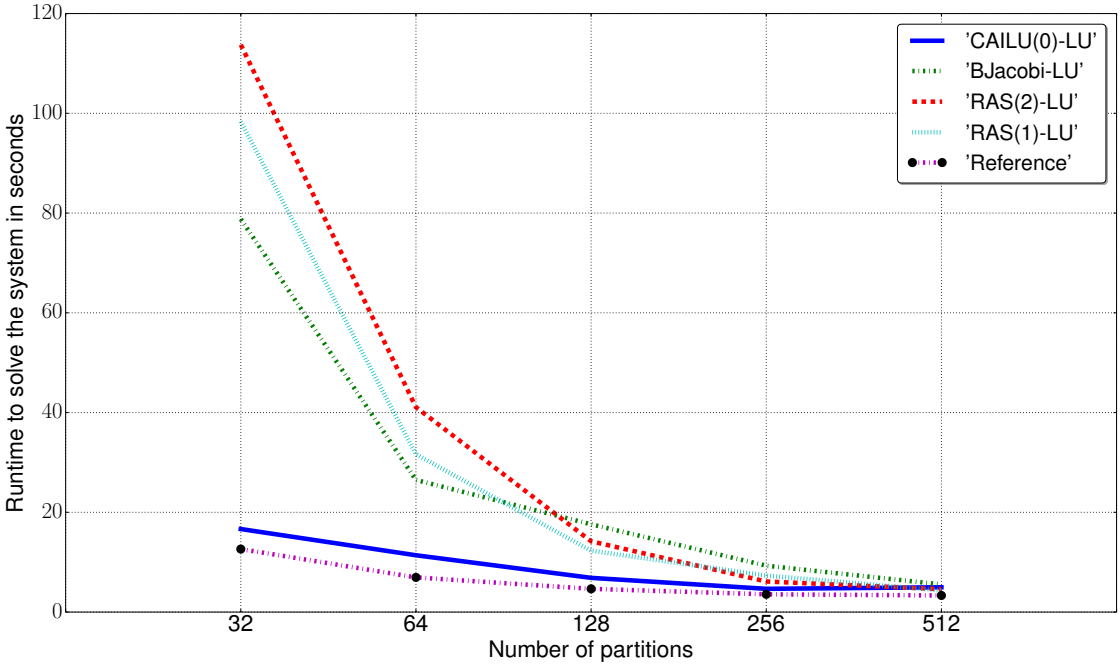
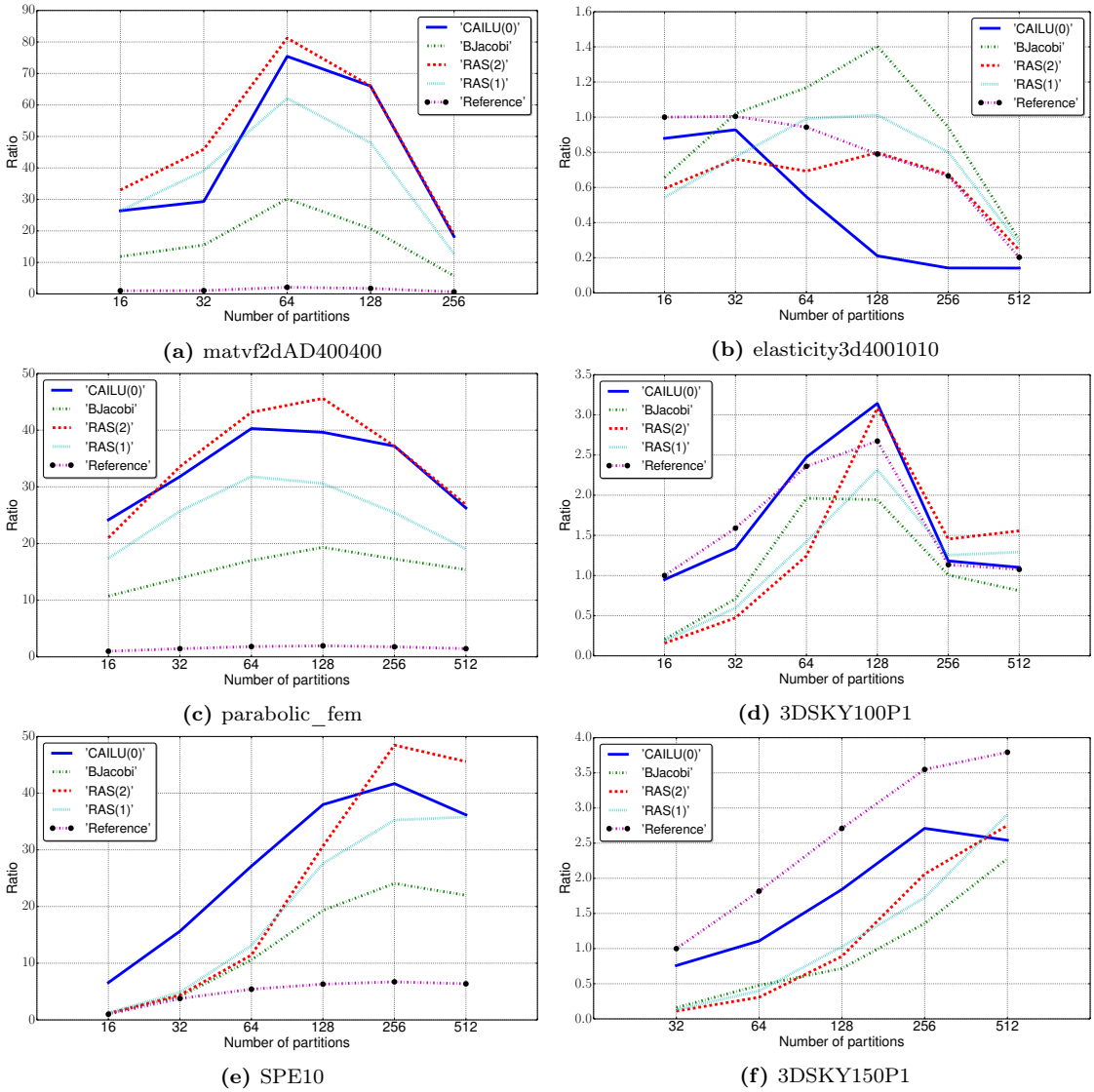


Figure 3.12 – Comparison of runtime to solve 3DSKY150P1 problem with CA-ILU(0)-LU, BJacobi-LU RAS(1)-LU and RAS(2)-LU, on Poincare system. All methods perform LU in each diagonal block. GMRES is set with a maximum 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

We next summarize in Figure 3.13 the ratio of the runtime to compute the solution of the system with respect to the reference runtime for each method and for each test matrix presented in Table 3.3. The reference corresponds to the runtime of GMRES without preconditioner, on 16 cores.



(a) matvf2dAD400400 **(b)** elasticity3d4001010

(c) parabolic_fem **(d)** 3DSKY100P1

(e) SPE10 **(f)** 3DSKY150P1

Figure 3.13 – Comparison of the runtime of CA-ILU(0)-LU, BJacobi-LU, RAS(1)-LU and RAS(2)-LU with the reference, GMRES without a preconditioner, for the test matrices presented in Table 3.3. The number of subdomain increases from 16 to 512 (256 subdomains in two cases). GMRES is set with a maximum of 3000 iterations, a restart of 200, and a relative tolerance of $1e-6$.

First, on matvf2dAD400400 problem in Subfigure 3.13a, BJacobi-LU suffers from the small size of the diagonal blocks. Compared to BJacobi-ILU(0), this version using LU goes from a ratio of 4 to at least 10. RAS(2)-LU is the fastest preconditioner for all number of subdomains studied. Its maximum ratio exceeds 80 for 64 subdomains. CA-ILU(0)-LU has a ratio of almost 30, same as RAS(1)-LU, and this ratio becomes closer to RAS(2)-LU, and it is equal for 128 and 256 subdomains. Using LU instead of ILU(0) offers better performance, especially when the number of subdomains is large. As studied before, elasticity3d4001010 problem is a challenging matrix. All methods except BJacobi suffer from having an overlap. In particular, CA-ILU(0)-LU runtime is widely degraded for 32, 64, and 128 subdomains. Its size of the overlap for 256

subdomains is close to its size of overlap of 128 subdomains, and so the ratio of CA-ILU(0)-LU does not increase significantly. On Parabolic_fem problem, in Subfigure 3.6c, CA-ILU(0) is the most efficient preconditioner. In Subfigure 3.13c, RAS(2)-LU outperforms CA-ILU(0)-LU, except for 16 subdomains. Compared to the ILU(0) version of each preconditioner, BJacobi-LU and RAS(1)-LU are far slower than the two others. Subfigure 3.13d shows the ratios of each method for 3DSKY100P1. Similar to 3DSKY150P1 problem presented above, BJacobi-LU and RAS-LU-like have a worse runtime compared to the reference. The gap is reduced when increasing the number of subdomains, but only RAS(2)-LU outperforms the reference when the number of subdomains is larger than 64. CA-ILU(0)-LU is close to the reference, outperforming it for 64 and 128 subdomains. Then our method has an overlap so large that the ratio decreases until reaching almost the runtime of GMRES alone. The case of SPE10, studied above, is presented in Subfigure 3.13e. We observe that CA-ILU(0)-LU is the fastest method to solve the system until 128 subdomains. Then RAS(2)-LU is more efficient with a maximum ratio close to 50, whereas CA-ILU(0)-LU reaches a ratio of 42. BJacobi-LU is slower. Its runtime is the same as the other methods for 16 subdomains, except for CA-ILU(0)-LU with a ratio of 7. The last problem, 3DSKY150P1, in Subfigure 3.13f, already detailed above, shows that CA-ILU(0)-LU is the preconditioner having the largest ratio compared to the other preconditioners.

The performance of CA-ILU(0) is related to the size of the overlap and the renumbering induced by the reordering of the subdomains. When the overlap becomes very large, we lose the benefit of reducing the number of iterations. A possibility is to limit the maximum overlap for CA-ILU(0). In that case, the number of iterations is expected to increase slightly, but the runtime, dominated by the cost of the application of the preconditioner, should be reduced enough such that a trade-off is reached. Another possibility is to reduce the renumbering of the unknowns due to the reordering of A. Results also show that CA-ILU(0) and CA-ILU(0)-LU do not succeed on the same problem. We next study the impact of a larger k on the efficiency of CA-ILU(k).

3.5.3 Study the impact of a larger k over CA-ILU(0)

Convergence results in sequential: comparison of CA-ILU(1) with CA-ILU(0)

In this section, we study the behavior of CA-ILU(1) compared to CA-ILU(0) in sequential. For that, we use a set of matrices of smaller dimensions, displayed in Table 3.9.

Matrix	Size	nnz(A)	symmetric	2D/3D	Problem
Elasticity3D501010	18513	618747	yes	3D	Elasticity
matvf2dNH2D200200	40000	199200	yes	2D	Non-homogenous

Table 3.9 – Matrices used for sequential test

First, we compare the stability of CA-ILU(0) with BJacobi and RAS on a challenging problem, Elasticity3D501010, by looking at the number of iterations, the relative residual returned by GMRES, the error on the solution, the condition number of the preconditioned system using `condest` routine from MATLAB, the error of the factorization and the number of iterations. The number of subdomains varies from 2 to 64. For each value, we call k-way partitioning algorithm on A to get the permutation matrix Π. This permutation is applied on A such that $A = \Pi A \Pi^\top$. Then, the obtained A is scaled using Relation (3.45). We further consider A as the permuted scaled matrix of the original A. The solution vector x_s is generated from a vector w of one ($w_i = 1, i \in [0, .., n-1]$, such that $x_s = w/\|w\|_2$. From it, we generate the right-hand

side $b = A \times x_s$ that we normalize. Finally, we solve the preconditioned system using GMRES coupled with one of the preconditioners mentioned above. We set GMRES with a maximum number of 1000 iterations without restart and a relative residual of $1e^{-8}$.

We use as reference GMRES alone, i.e., without a preconditioner, denoted *No precond*. To emphasize the impact of the reordering on the factorization, we add GMRES with ILU(0) preconditioner without permutation i.e., in a natural ordering, as a second reference. This reference is denoted further as *ILU(0)NatOrd* where the original A is scaled only. We also compare CA-ILU(0) with a sequential ILU(0) factorization on A, denoted *ILU(0)Kway* opposite to the previous reference. Since the overlap of CA-ILU(0) corresponds to at least the first two layers in each subdomain, we set RAS with 2 level of overlap. To be fair with CA-ILU(0), BJacobi and RAS also use ILU(0) in each subdomain.

Table 3.10 displays the convergence of each preconditioner and compares it with the two references, on Elasticity3D501010 problem. Solving it without preconditioner does not converge after 1000 iterations whereas all preconditioners converge. Looking at the impact of reordering on the number of iterations, we compare *ILU(0)NatOrd* with *ILU(0)Kway*. We observe that permuting the system increases the number of iterations from 266 for the reference to 267 for 2 subdomains and reaches 330 for 64 subdomains. So permuting the system degrades the number of iterations by up to 24% in the ILU(0) case. This is a well-known behavior (Duff et al., 1989). Focusing on CA-ILU(0) and *ILU(0)Kway*, we observe that the number of iterations for CA-ILU(0) increases by 20 iterations compared to *ILU(0)Kway* for 2 subdomains. This gap grows up to 91 iterations for 16 subdomains. This shows that the number of iterations increases when the unknowns are reordered with respect to the natural ordering. Moreover, when the number of subdomains increases, the size of the domains decreases and the ratio of reordered unknowns with respect to the unknowns ordered in natural order of each subdomain increases.

Definition 21. *Given Ω the graph of a matrix $A \in \mathbb{R}^{n \times n}$, we call* disorder *the ratio of the number of reordered vertices with respect to the total number of vertices.*

In other words, the number of remaining vertices computed Line 8 in Algorithm 3.5 decreases when the number of subdomains increases. Now comparing all preconditioners, the relative residual is close to $1e^{-9}$ as the relative error of the system and as the relative error of the solution in all cases. Focusing on the incomplete LU factorization, we compute the relative error of the factorization. Because of our implementation of RAS, we are not able to compute the error of the factorization for RAS. This table shows that the number of iterations is strongly related to the quality of the factorization and so the condition number of $M^{-1}A$. BJacobi has the worst factorization error and is the preconditioner that requires the most iterations to converge. This is a well-known observation on this method since it discards the dependencies between subdomains. BJacobi needs at least 302 iterations to converge for 2 subdomains and up to 573 iterations for 64 subdomains. RAS and CA-ILU(0) are close with 276 and 287 iterations, respectively, for 2 subdomains, and need at most 387 and 402 iterations respectively to converge for 64 subdomains. The number of iterations of CA-ILU(0) is between BJacobi and RAS. It is never worse than BJacobi and never better than RAS for all numbers of subdomains on this problem.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$	cond$(M^{-1}A)$	$\frac{\|\|A-LU\|\|_2}{\|\|A\|\|_2}$
References	No precond	1000	4.3e-03	4.3e-03	9.2e-04	-	-
	ILU(0) NatOrd	266	8.4e-09	1.7e-09	8.9e-10	3.1e+06	1.1e-01
2	ILU(0) Kway	**267**	9.7e-09	2.2e-09	3.0e-10	3.1e+06	1.4e-01
	CAILU(0)	287	9.5e-09	2.3e-09	4.6e-10	4.1e+06	1.9e-01
	BJacobi-ILU(0)	302	7.2e-09	2.2e-09	6.8e-10	6.8e+06	3.2e-01
	RAS(2)-ILU(0)	276	9.5e-09	2.6e-09	5.0e-10	3.1e+06	-
4	ILU(0) Kway	**277**	9.4e-09	2.1e-09	7.4e-10	4.4e+06	1.8e-01
	CAILU(0)	330	8.3e-09	2.5e-09	5.4e-10	7.2e+06	2.3e-01
	BJacobi-ILU(0)	342	8.3e-09	3.3e-09	1.7e-09	1.3e+07	3.3e-01
	RAS(2)-ILU(0)	291	8.7e-09	2.5e-09	6.4e-10	4.4e+06	-
8	ILU(0) Kway	**289**	9.2e-09	2.3e-09	4.0e-10	8.2e+06	1.7e-01
	CAILU(0)	378	7.4e-09	2.4e-09	8.2e-10	1.0e+07	2.2e-01
	BJacobi-ILU(0)	397	8.7e-09	4.1e-09	9.2e-10	1.6e+07	3.4e-01
	RAS(2)-ILU(0)	311	7.7e-09	2.6e-09	3.6e-10	8.2e+06	-
16	ILU(0) Kway	**303**	7.7e-09	1.9e-09	7.9e-10		1.9e-01
	CAILU(0)	394	9.5e-09	3.2e-09	1.8e-10	1.0e+07	2.3e-01
	BJacobi-ILU(0)	468	9.5e-09	5.0e-09	2.2e-09	2.1e+07	3.7e-01
	RAS(2)-ILU(0)	338	8.0e-09	3.0e-09	1.3e-10	9.0e+06	-
32	ILU(0) Kway	**313**	7.5e-09	2.1e-09	1.4e-10		1.8e-01
	CAILU(0)	400	9.4e-09	3.0e-09	2.1e-10	9.8e+06	2.3e-01
	BJacobi-ILU(0)	505	8.9e-09	4.8e-09	1.7e-10	1.8e+07	3.5e-01
	RAS(2)-ILU(0)	357	9.8e-09	3.8e-09	2.8e-10	6.3e+06	-
64	ILU(0) Kway	**330**	8.0e-09	2.3e-09	2.9e-10		1.9e-01
	CAILU(0)	402	9.7e-09	3.0e-09	1.5e-10	9.6e+06	2.2e-01
	BJacobi-ILU(0)	573	8.5e-09	5.0e-09	8.7e-10	9.2e+06	3.8e-01
	RAS(2)-ILU(0)	387	8.2e-09	3.2e-09	1.6e-10	5.8e+06	-

Table 3.10 – Comparison of CA-ILU(0) with Bjacobi and RAS on Elasticity3D501010 problem for a number of subdomains increasing from 2 to 64. As references, the system is solved without preconditioner and with sequential ILU(0) preconditioner. $A \in \mathbb{R}^{n\times n}$ is permuted using KwayPartitioning and scaled, x is the computed solution returned by GMRES and x_s is the real solution. GMRES is set with a maximum of 1000 iterations, no restart and a relative tolerance of $1e-8$.

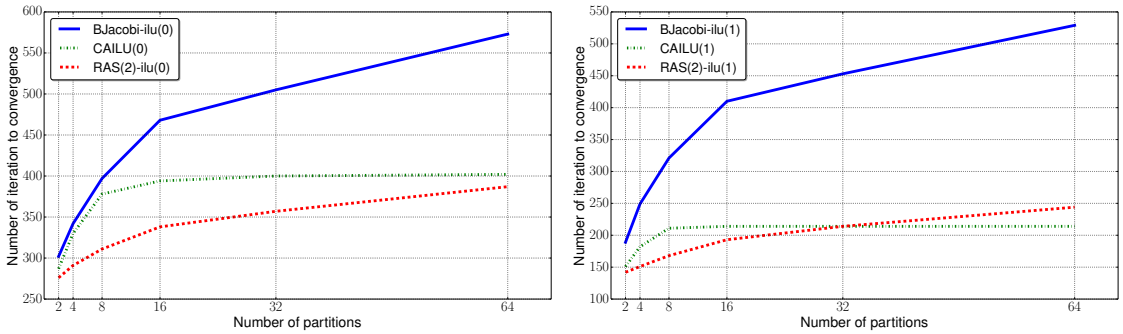
Considering the case $k = 1$, we perform the same tests as for $k = 0$ and we gather the results in Table 3.11. As for $k = 0$, the application of k-way partitioning algorithm increases the number of iterations of *ILU(1)Kway* compared to *ILU(1)NatOrd* from 135 to 136 for 2 subdomains and up to 177 iterations. The relative residual and the relative solution of the system are close to 10^{-9} and the relative error of the solution varies from 10^{-9} to 10^{-11}. CA-ILU(1) needs fewer iterations than *ILU(1)Kway*, 15 for 2 subdomains and up to 64 for 8 subdomains. For 8 subdomains and more, CA-ILU(1) converges in around 214 iterations. BJacobi converges in 189 iterations for 2 subdomains and the number of iterations increases with the number of subdomains, reaching 529 iterations for 64 subdomains. RAS has a range of iterations to converge from 142 to 244.

Figure 3.14 summarizes the evolution of the iterations with respect to the number of subdomains in subfigure (3.14a) CA-ILU(0), BJacobi-ILU(0) and RAS-ILU(0) and in subfigure (3.14b) CA-ILU(1), BJacobi-ILU(1) and RAS-ILU(1). In both subfigures, BJacobi is close to RAS and CA-ILU(k) for 2 subdomains but has a significant increase in the number of iterations for larger number of subdomains compared to the others. CA-ILU(0) tends to be as good as RAS for 64 subdomains and is expected to outperform RAS since the latter keeps increasing its number of iterations to converge. In the case of CA-ILU(1), we observe that RAS needs the same number of iterations to converge as CA-ILU(k) for 32 subdomains and then outperforms RAS for 64

subdomains with 214 iterations versus 244 for RAS.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$	cond$(M^{-1}A)$	$\frac{\|\|A-LU\|\|_2}{\|\|A\|\|_2}$
References	No precond	1000	4.3e-03	4.3e-03	9.2e-04	-	-
	ILU(1) NatOrd	135	8.2e-09	1.3e-09	1.3e-10	6.4e+05	4.3e-02
2	ILU(1) Kway	**136**	6.e-09	1.1e-09	6.7e-11	1.2e+06	5.6e-02
	CAILU(1)	151	5.6e-09	1.1e-09	1.5e-10	2.1e+06	8.5e-02
	BJacobi-ILU(1)	189	8.7e-09	4.1e-09	2.3e-09	6.1e+06	3.2e-01
	RAS(2)-ILU(1)	142	6.1e-09	1.9e-09	1.5e-10	1.2e+06	-
4	ILU(1) Kway	**141**	5.7e-09	1.0e-09	1.8e-10	2.0e+06	6.9e-02
	CAILU(1)	182	9.7e-09	2.0e-09	1.2e-09	5.0e+06	1.0e-01
	BJacobi-ILU(1)	249	9.1e-09	4.4e-09	2.9e-09	1.1e+07	3.2e-01
	RAS(2)-ILU(1)	151	8.8e-09	2.9e-09	5.9e-10	2.2e+06	-
8	ILU(1) Kway	**147**	6.1e-09	1.2e-09	6.7e-11	2.6e+06	6.7e-02
	CAILU(1)	211	8.9e-09	1.9e-09	3.5e-10	7.6e+06	1.1e-01
	BJacobi-ILU(1)	321	9.1e-09	4.6e-09	1.2e-09	1.7e+07	3.4e-01
	RAS(2)-ILU(1)	168	9.5e-09	3.8e-09	9.2e-11	3.2e+06	-
16	ILU(1) Kway	**158**	8.0e-09	1.5e-09	3.0e-10	5.2e+06	8.2e-02
	CAILU(1)	214	9.3e-09	2.1e-09	5.6e-10	6.6e+06	1.1e-01
	BJacobi-ILU(1)	410	9.3e-09	5.2e-09	3.6e-10	1.9e+07	3.8e-01
	RAS(2)-ILU(1)	193	9.7e-09	4.0e-09	1.6e-10	5.1e+06	-
32	ILU(1) Kway	**164**	5.5e-09	1.2e-09	3.2e-10	4.6e+06	7.9e-02
	CAILU(1)	214	9.3e-09	2.0e-09	2.0e-10	8.5e+06	1.1e-01
	BJacobi-ILU(1)	453	8.2e-09	4.6e-09	8.3e-10	1.7e+07	3.6e-01
	RAS(2)-ILU(1)	214	6.6e-09	2.8e-09	2.0e-10	3.6e+06	-
64	ILU(1) Kway	**177**	6.7e-09	1.3e-09	1.8e-10	1.0e+07	9.5e-02
	CAILU(1)	214	7.8e-09	1.6e-09	8.2e-11	6.7e+06	1.2e-01
	BJacobi-ILU(1)	529	9.9e-09	5.8e-09	4.8e-10	9.3e+06	3.8e-01
	RAS(2)-ILU(1)	244	9.8e-09	4.2e-09	6.3e-10	5.0e+06	-

Table 3.11 – Comparison of CA-ILU(1) with Bjacobi and RAS on Elasticity3D501010 problem for a number of subdomains increasing from 2 to 64. As references, the system is solved without preconditioner and with sequential ILU(0) preconditioner. $A \in \mathbb{R}^{n \times n}$ is permuted using KwayPartitioning and scaled, x is the computed solution returned by GMRES and x_s is the real solution. GMRES is set with a maximum of 1000 iterations, no restart and a relative tolerance of $1e-8$.



(a) $k = 0$ **(b)** $k = 1$

Figure 3.14 – Comparison of the number of iterations to solve the preconditioned system for different k for CA-ILU(k), BJacobi and RAS on Elasticity3D501010 matrix with respect to the number of subdomains. The number of partitions is equal to the number of cores.

The runtime of CA-ILU(k) is strongly related to the size of the overlap. Table 3.12 summarizes the size of the overlap of CA-ILU(0) and CA-ILU(1) according to the number of subdomains ($2^i, i \in \{1, \ldots, 6\}$) for Elasticity3D501010 problem. In addition to the overlap, the size of each subdomain is given as reference. Each size is split into three categories: its mean, its minimum and its maximum over all subdomains for a given number of subdomains *nsubdomain*. The sizes presented in Table 3.12 correspond to the number of vertices that compose either a subdomain or the overlap of a subdomain.

		subdomain size			overlap size		
nsubdomain	k	mean	min	max	mean	min	max
2	0	9256	9192	9321	738	738	738
	1	-	-	-	1104	1104	1104
4	0	4628	4554	4695	1092	726	1458
	1	-	-	-	1637	1089	2184
8	0	2314	2262	2382	2313	726	4600
	1	-	-	-	7464	1691	10775
16	0	1157	1123	1191	5143	1567	9386
	1	-	-	-	7773	2978	13749
32	0	578	561	595	6056	3257	10440
	1	-	-	-	10703	5979	17129
64	0	289	280	297	7717	2199	14384
	1	-	-	-	9271	2879	15525

Table 3.12 – Evolution of the size of the CA-ILU(k) overlap in average, minimum, and maximum over all subdomains for Elasticite3D501010.

First, the size of each subdomain, resulting from k-way partitioning algorithm, does not vary much. When $k = 0$ and 2 subdomains, the average overlap size is 738. The resulting augmented subdomains (a subdomain and its overlap) are composed of 9 930 and 10 059 vertices. For 64 subdomains, the augmented subdomains are composed of 8 006 vertices on average. When $k = 1$, the average size of the augmented subdomains is 10 360 vertices for 2 subdomains and 9 560 vertices for 64 subdomains. It means that the average size of the augmented subdomains (which corresponds to the local memory consumption) is almost not decreasing when the number of subdomains is increasing. However, the average data does not provide enough information on the limitation of the method. Due to load balancing, the overall performance of the preconditioner is more likely to be bound by the maximum overlap size. Figure 3.15 shows the evolution of the minimum and maximum size of the augmented domains.

In the case of $k = 0$, the average size of the overlap increases slowly with respect to the number of subdomains but the maximum size drastically increases after 16 subdomains. The limit of scalability is related to the size of the overlap because the application of the preconditioner, even in parallel, is a backward and forward substitutions. Therefore the size of the overlap impacts the performance of the preconditioner. When the number of subdomains is larger than 16, the runtime for solving the system is expected to be degraded whereas the number of iterations remains the same. Even if the number of iterations remains almost the same, this does not lead to a decrease in the runtime when the number of subdomains increases.

Similarly, when the number of subdomains is larger than 8, the size of the overlap for CA-ILU(1) is larger than the size of the subdomains. In term of iterations, in Table 3.11, the

stagnation of the number of iterations remains almost constant due to the fact that the size of the overlap is almost the whole matrix. For 32 subdomains, the largest overlap is 17 129, which is close to 18 513, the size of the matrix. The subdomain having the largest overlap factors has an overlap which is almost the entire matrix. Therefore, at each iteration of GMRES, for this subdomain, the application of the preconditioner corresponds to solving almost the entire system. Thus, the system converges as *ILU(1)Kway* in a degraded form since A is fully reordered so that perturbs the convergence. Thus a trade-off has to be found between the gain on the number of iterations and the size of the overlap.

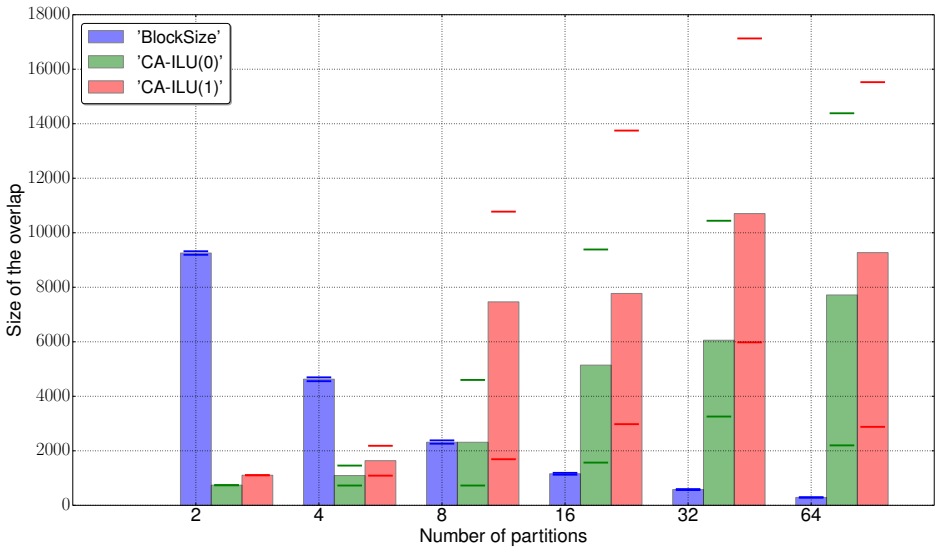


Figure 3.15 – Comparison of the evolution of the overlap size for CA-ILU(0) and CA-ILU(1) with the size of each diagonal blocks, on Elasticity3D501010 problem with respect to the number of subdomains. The *BlockSize* represents the size of the diagonal blocks of Elasticity3D501010.

Figure 3.16 shows the evolution of the relative residual at each iteration of GMRES, on Elasticity3D501010 for 64 subdomains, preconditioned either ILU(1), CA-ILU(1), BJacobi or RAS, the last two also using ILU(1) on each subdomain. The curves show a slow convergence of the residual until 10^{-1} and then the convergence becomes superlinear. The only difference between ILU(1) and CA-ILU(1) is the ordering of A. However, ILU(1) is converging in 18% fewer iterations than CA-ILU(1). Thus the ordering of A has an impact on the convergence. RAS is directly impacted by the number of subdomains that reduces the data in each domain and the size of its overlap.

Comparing $k = 0$ and $k = 1$, we observe as expected that all preconditioners using ILU(1) in each diagonal blocks outperform the preconditioners using ILU(0). A bigger k enhances the methods but with a less gain for BJacobi. For 2 subdomains, it needs 302 iterations for $k = 0$ and 189 for $k = 1$, so a gain of 38%. But for 64 subdomains, the gain falls to 8% comparing 573 and 529 iterations. On one hand, CA-ILU(1) has a gain of 47% for 2 subdomains compared to CA-ILU(0) and 46% for 64 subdomains. On the other hand, RAS has a gain of 49% for 2 subdomains and drops to 37% for 64 subdomains.

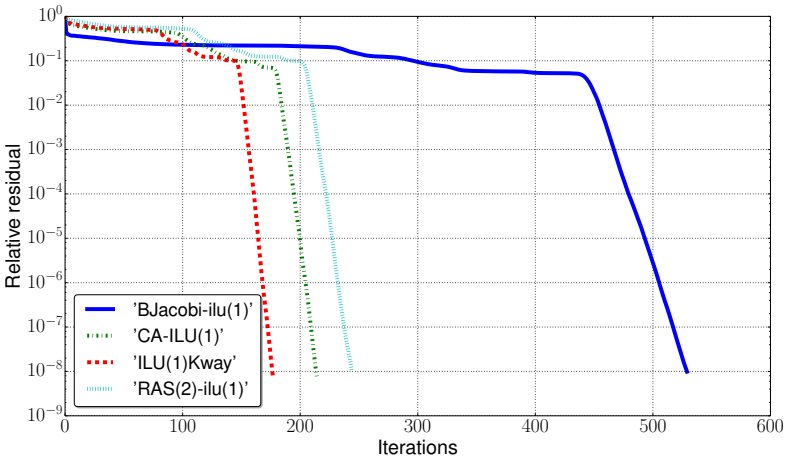


Figure 3.16 – Evolution of the relative residual during the iterations of GMRES to solve the preconditioned system for $k = 1$ with CA-ILU(1), BJacobi and RAS on Elasticity3D501010 matrix and for 64 subdomains.

Similarly, we study matvf2dNH2D200200 problem, in Table 3.13, whose the associated matrix is roughly 10 times sparser than Elasticity3D501010 problem.

		k	
nsubdomain	method	0	1
	No precond	350	350
	ILU(k)NatOrd	142	100
2	ILU(k)Kway	**161**	**109**
	CAILU(k)	180	120
	BJacobi	196	142
	RAS(2)	-	-
4	ILU(k)Kway	**156**	**108**
	CAILU(k)	170	115
	BJacobi	187	137
	RAS(2)	161	118
8	ILU(k)Kway	**166**	**108**
	CAILU(k)	181	122
	BJacobi	203	148
	RAS(2)	177	118
16	ILU(k)Kway	**160**	**113**
	CAILU(k)	185	125
	BJacobi	207	153
	RAS(2)	177	122

Table 3.13 – Comparison of stability of CA-ILU(0) and CA-ILU(1) with BJacobi and RAS on matvf2dNH200200 problem for a number of subdomains goes from 2 to 16. GMRES is set with a maximum of 1000 iterations, without restart and a relative tolerance of $1e-8$.

We first consider $k = 0$. We observe that the permutation obtained by k-way partitioning increases the number of iterations of ILU(0) by roughly 20. Compared to the elasticity problem, the offset on the number of iterations stays constant with respect to the number of subdomains. We note that for 4 subdomains, the number of iterations of *ILU(0)Kway* is lower than expected.

The permutations obtained by k-way partitioning disturb less the system and so the convergence, compared to the other number of subdomains. CA-ILU(0) decreases from 180 iterations to 170 iterations when the number of subdomains increases from 2 to 4. The reordering of Algorithm 3.5 increases the number of iterations to converge. However, this increasing is minor compared to the benefit of the partitioning. We now consider $k = 1$. As for $k = 0$, the permutation obtained by k-way partitioning increases the number of iterations by a constant number. *ILU(1)Kway* varies from 109 to 113 for 16 subdomains. BJacobi needs at least 137 iterations whereas RAS and CA-ILU(1) solves the system with a maximum of 122 and 125 iterations, respectively. For each number of subdomains, these two latter preconditioners are close with a maximum gap of 4 iterations whereas BJacobi needs at least 12 more iterations.

		subdomain size			overlap size		
nsubdomain	k	mean	min	max	mean	min	max
2	0	20000	19997	20003	431	431	431
	1	-	-	-	642	642	642
4	0	10000	9980	10012	427	399	448
	1	-	-	-	1121	635	1850
8	0	5000	4996	5004	483	283	749
	1	-	-	-	1176	720	1777
16	0	2500	2484	2527	585	243	919
	1	-	-	-	2032	395	4764

Table 3.14 – Evolution of the size of the overlap in average, minimum and maximum over all subdomains for matvf2dNH200200

In the case of CA-ILU(0), the size of the overlap is roughly stable, and varies from 431 vertices for 2 subdomains to 585 vertices (a maximum of 919) for 16 subdomains. When $k = 1$, the overlap size is tripled when the number of subdomains is multiplied by a factor of 8. The size varies from 642 vertices for 2 subdomains to 2032 vertices for 16 subdomains, with a maximum of 4764 vertices. This latter overlap size added to the largest domain gives 7291 vertices, surpassing 6783 vertices of the largest augmented domain for 8 subdomains.

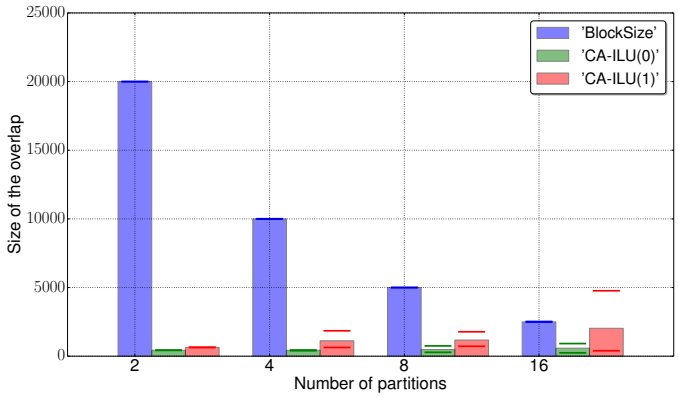


Figure 3.17 – Evolution of the overlap of CA-ILU(0) and CA-ILU(1) with respect to the number of subdomains, on matvf2dNH200200 problem. The *BlockSize* represents the size of the diagonal blocks of matvf2dNH2D200200.

Plotting the evolution of the overlap size in Figure 3.17, we observe that the size reduces with respect to the number of subdomains for $k = 0$ and for $k = 1$, except for 16 subdomains. Therefore, in the case of CA-ILU(1), the overlap size does not decrease beyond 8 subdomains.

Parallel efficiency of CA-ILU(k)

We now study the impact of a larger k in parallel. Nested Dissection algorithm used in the initial tests reorders more vertices than necessary. For that, in the remaining experimental tests, we replace the call to Nested Dissection by a call to CAILU_reorderLayer, Algorithm 3.4. Our experimental results show that it leads to a reduction of 10 iterations on average. In addition, we modify our implementation of Algorithm 3.10 and present its updated version in Algorithm 3.12. We introduce two statements, lines 7, 8. Moving to $k > 0$, we need to compute the symbolic factorization of A in order to get the dependencies of each subdomain. To avoid doing the symbolic phase on each processor, we reuse the pattern of F_k. In Line 8, we replace the pattern of A by the pattern of F_k and add a 0 for each additional entry. It means that for each edge in $(u, v) \in E(\widehat{\Omega}) \setminus E(\Omega)$, we add explicitly $a_{\mathcal{I}(u),\mathcal{I}(v)} = 0$ in the CSR format. Then, each A_i and the related overlap can be factored in-place using ILU(0).

Algorithm 3.12 buildCAILUK(A, k)

This algorithm presents the main steps in our implementation of CA-ILU(k)

Input: $A \in \mathbb{R}^{n\times n}$: the matrix to factor,
Input: k: the ILU parameter
1: Let id be the index of the processor and p be the number of processors
2: **if** $id = 0$ **then**
3: **for** $i = 0$ to $p - 1$ **do**
4: Reorder the layers of A_i using Algorithm 3.5
5: **end for**
6: Apply the same reordering to b
7: Compute F_k, the symbolic ILU(k) factorization of A
8: Replace the pattern of A by the pattern of F_k and add explicitly 0 to each additional entry.
9: **for** $i = 0$ to $p - 1$ **do**
10: Compute the dependencies of A_i in order to apply L_i and U_i locally without communication
11: Distribute A_i, its overlap and the corresponding part of b to processor i
12: **end for**
13: **end if**
14: Each processor uses ILU(0) to factor its local block of A without communication
Output: L_i and U_i the triangular factors of A on processor i.

We first focus on the SPE10 problem of dimension 10^6 and the influence of k on the size of the overlap. Figure 3.18 shows the evolution of CA-ILU(k) overlap with respect to the number of subdomains and for $k \in \{0, \ldots, 4\}$. Each bar represents the average size over all subdomains, the lower and higher horizontal lines represent the minimum and maximum size, respectively. As a reference, the size of the subdomains is given by the first bar for each group of bars. For 16 subdomains, CA-ILU(0) and CA-ILU(1) overlap sizes are close to the size of the subdomains, with a maximum of twice the size of the subdomain for CA-ILU(1). In comparison, for $k \geq 2$, the overlap size is at least 6 times larger than the size of the subdomain. Especially, CA-ILU(4) overlap reaches a size of 10^6. At that point, it corresponds to having on one processor almost the entire matrix. Combined with $k = 4$, it does not fit in memory and so CA-ILU(4) cannot be used for 16 subdomains. For all numbers of subdomains presented in the figure, the size of the overlap of CA-ILU(3) and CA-ILU(4) does not reduce when the number of subdomains

increases. Instead, the size increases until reaching the entire matrix as overlap. In the case of CA-ILU(2), its maximum overlap size is close to 4×10^5 from 16 to 64 subdomains. The size of its maximum overlap decreases by 30% for 128 subdomains and then slowly increases. We next study the number of iterations of GMRES to converge, presented in Table 3.15. As expected from the size of the overlap, CA-ILU(3) and CA-ILU(4) yield to killed jobs, caused by a problem of memory. The number of iterations of CA-ILU(0) varies from 535 for 16 subdomains to 595 for 512 subdomains, with a minimum of 518 iterations for 64 subdomains. Moving from $k = 0$ to $k = 1$, these iterations are reduced by more than 200 iterations, with a minimum of 329 iterations for 16 subdomains, and a maximum of 370 iterations for 512 subdomains. This reduction is due to the replacement of ILU(0) by ILU(1), involving a fill-in, and the size of CA-ILU(1) overlap, being twice larger than CA-ILU(0). In the case of $k = 2$, for all numbers of subdomains presented, the iterations mainly decrease by 100 iterations. In addition to the number of iterations, Figure 3.19 shows the runtime to solve the system for each value of k. CA-ILU(0) runtime decreases from 10.7 seconds to 4 seconds when the number of subdomains increases from 16 to 256. With a number of iterations smaller, a larger k and a larger overlap, CA-ILU(1) outperforms CA-ILU(0). The overhead induced by a larger overlap at each iteration is absorbed by the reduction of the number of iterations. Unlike CA-ILU(1), CA-ILU(2) does not reduce its number of iterations enough in comparison to the cost of one iteration. For 16 subdomains, the cost per iterations of CA-ILU(2) is 3 times higher than CA-ILU(0), with 0.06 and 0.02 seconds, respectively.

In conclusion, for this problem, using CA-ILU(1) is the best choice, compared to the other versions. The cost per iteration is a major factor in the efficiency of CA-ILU(k). Comparing with BJacobi-ILU(1), RAS(2)-ILU(1) and RAS(3)-ILU(1) in Figure A.7, CA-ILU(1) is better than BJacobi but slightly outperformed by both versions of RAS. Table A.14 shows that CA-ILU(1) converges with fewer iterations than other methods but the gap is not sufficient to be the fastest for this problem.

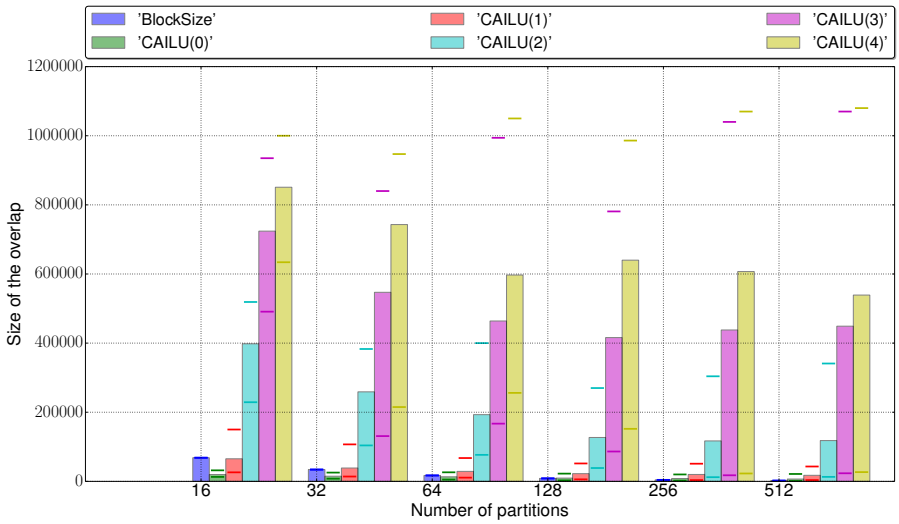


Figure 3.18 – Evolution of the overlap of CA-ILU(k) with $k \in \{0, \ldots, 4\}$ on SPE10 problem. The *BlockSize* represents the size of the diagonal blocks of matvf2dNH2D200200.

	CA-ILU(k)		
nsubdomain	0	1	2
16	535	329	227
32	552	343	245
64	518	350	266
128	529	361	269
256	546	365	284
512	595	370	305

Table 3.15 – Comparison of CA-ILU(k) for $k \in \{0, \ldots, 2\}$ on the problem of SPE10 for a number of subdomains increasing from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

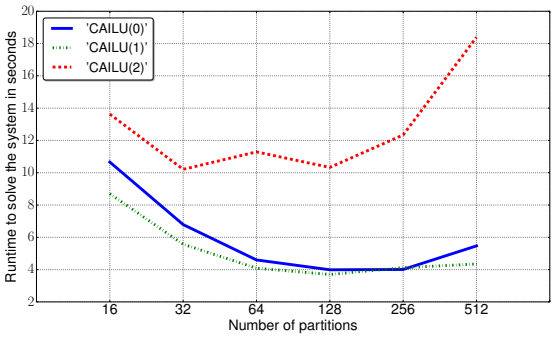


Figure 3.19 – Runtime to solve the system for CA-ILU(k) with $k \in \{0, \ldots, 2\}$ on SPE10 problem. GMRES is set with a maximum of 3000 iterations, a restart of 200, and a relative tolerance of $1e-6$.

We next summarize the behavior of CA-ILU(k) for different problems. For matvf2dAD400400, for $k \in \{0, \ldots, 4\}$ the overlap size decreases with respect to the number of subdomains. The increasing of k does not drastically increase the size of CA-ILU(k) overlap. Note that for 16 subdomains, the maximum overlap size of CA-ILU(2) is 2.5 larger than the average size and the size of the subdomain. Furthermore, the case $k = 1$ is worse compared to CA-ILU(2), whose overlap is smaller in terms of both average and maximum. It means the pattern of the system, after k-way partitioning is not in favor of the method. The associated runtimes for each value of k are presented in Figure 3.21. We observe that the time to solve the system increases when k increases, in the same time as the size of the overlap. The overhead per iteration caused by the overlap is small enough and hence the method benefits from the reduction in the number of iterations. Note that CA-ILU(3) and CA-ILU(4) have the same runtime. Therefore, using more than $k = 3$ does not offer better performance. Similarly to matvf2dAD400400, the overlap size for parabolic_fem problem decreases when the number of subdomains increases, in Figure 3.22. Figure 3.21 presents the runtime for $k = 1$ to $k = 4$. Unlike CA-ILU(4), increasing k leads to a decrease in the runtime up to 128 subdomains. Although CA-ILU(4) has a better scalability, its overlap size limits its performance and so it does not outperform the others. Now, we focus on 3DSKY100P1 problem in Figures 3.24 and 3.25. When $k \geq 3$, the overlap size is too large and is close to the size of the problem, 10^6. The same problem occurs as for SPE10; the jobs are killed by the system because of lack of memory. CA-ILU(2) also has a very large overlap and reaches almost the limit of the machine. The runtimes show that CA-ILU(2) is at least 5 times slower than CA-ILU(0) and CA-ILU(1). Comparing these last two, their runtimes are similar. Therefore, using CA-ILU(0) is a better solution since its memory consumption is smaller and its runtime is the best.

The efficiency of CA-ILU(k) is strongly related to the size of the overlap. Increasing k leads to a larger overlap and a larger overhead per iteration. If the reduction in the number of iterations to converge is not significant enough, the runtime is degraded. Moreover, the memory usage can prevent using CA-ILU(k). The best performance of CA-ILU(k) cannot be reached for a specific value of k. In addition, the scalability of the preconditioner is obtained for a maximum number of subdomains. Therefore, for a given problem, we need to study the behavior of CA-ILU(k) to select the best k and the most appropriate number of subdomains.

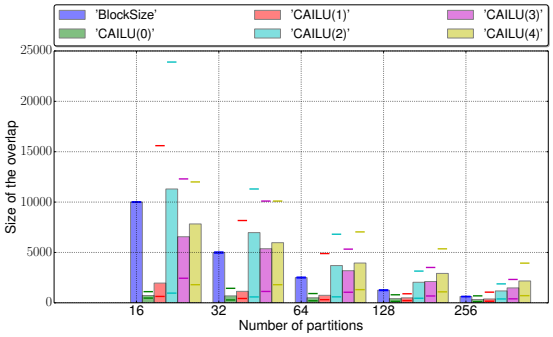


Figure 3.20 – Evolution of the overlap of CA-ILU(k) with $k \in \{0, \ldots, 4\}$ for matvf2dAD400400 problem.

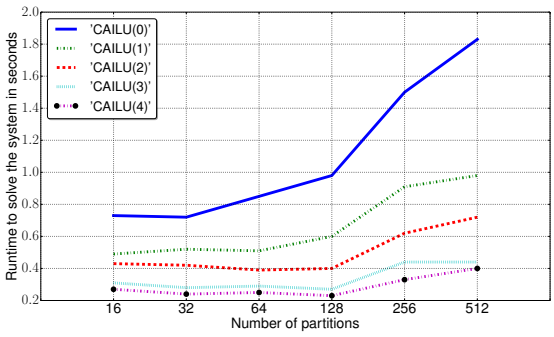


Figure 3.21 – Runtime to solve the system for CA-ILU(k) with $k \in \{1, \ldots, 4\}$ for matvf2dAD400400 problem.

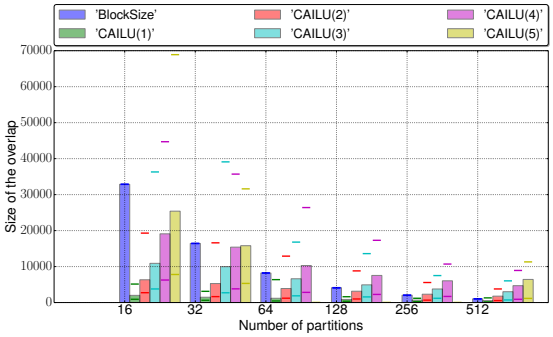


Figure 3.22 – Evolution of the overlap of CA-ILU(k) with $k \in \{1, \ldots, 5\}$ for parabolic_fem problem.

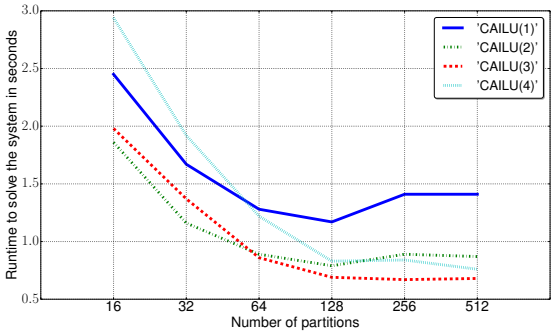


Figure 3.23 – Runtime to solve the system for CA-ILU(k) with $k \in \{1, \ldots, 4\}$ for parabolic_fem problem.

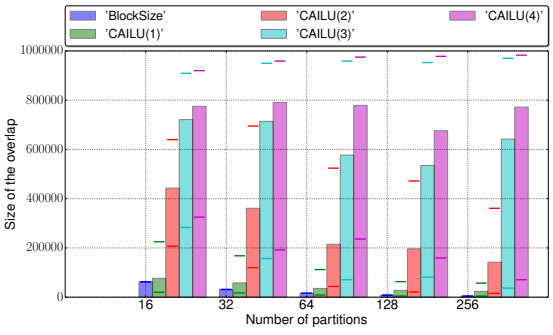


Figure 3.24 – Evolution of the overlap of CA-ILU(k) with $k \in \{1, \ldots, 4\}$ for 3DSKY100P1 problem.

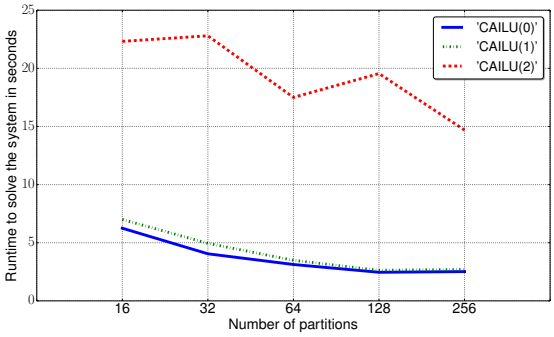
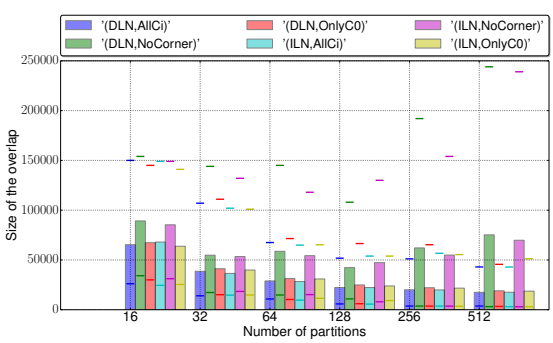


Figure 3.25 – Runtime to solve the system for CA-ILU(k) with $k \in \{0, \ldots, 2\}$ for 3DSKY100P1 problem.

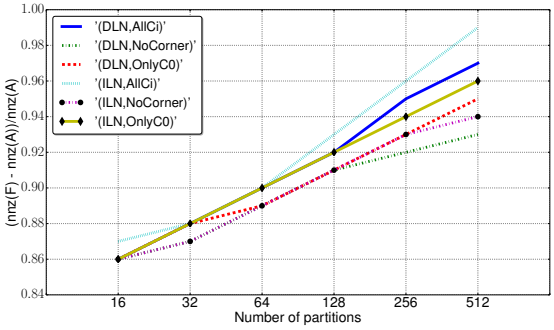
3.5.4 Reordering variants of CA-ILU(k) and overlap limitation

Impact of numberingLayer algorithms on the fill-in and on the number of iterations

Previous results show the importance of the reordering on the number of iterations. Algorithm 3.5 reorders a subdomain using either the decreasing layer numbering (DLN) or the increasing numbering layer (ILN). In addition, the reordering of the vertices in a layer starts by numbering its corners and then the remaining vertices of the layer. This reordering inside the layers aims to reduce the size of the CA-ILU(k) overlap. Therefore, we propose to study, for both numberingLayer algorithms, the three cases of corner ordering. The first case considers the corners of all layers involved in the reordering of a subdomain, denoted `AllCi` and corresponds to using all $\mathcal{C}_i^j$ of a subdomain Ω_j. The second case reorders only the boundary layer of each subdomain, referred to as `OnlyC0`. The last case ignores the reordering of the vertices in a layer, denoted `NoCorner`. We discuss the impact of these reorderings on the size of the overlap of CA-ILU(k), on the fill-in of F_k, the symbolic ILU(k) factor of A, and on the runtime to solve the system.



(a) Impact of ordering on the size of CA-ILU(1) overlap for SPE10 problem.



(b) Impact of ordering on the fill-in of F_1, the symbolic ILU(1) factorization of A computed by CA-ILU(1) for SPE10 problem.

Figure 3.26 – Impact of different orderings on CA-ILU(1) for SPE10 problem. 'DLN' and 'ILN' correspond to decreasing and increasing layer numbering, respectively, 'AllCi' considers the corners of all layer involved, 'OnlyC0' takes into account the corners of each boundary layer only and 'NoCorner' avoids reordering inside the layers.

We first study the impact of the different reorderings on SPE10 problem. Subfigure 3.26a presents the overlap size of CA-ILU(1) for each ordering and for a number of subdomains increasing from 16 to 512. We observe that for both ILN and DLN, the corner ordering `NoCorner` leads to a size of overlap larger than 5×10^4 in average, with an increasing size when the number of subdomains is greater or equals 128. In particular, the maximum overlap size for both corner orderings increases until it is close to 2.5×10^5 for 512 subdomains. In comparison, all other corner orderings decrease when the number of subdomains increases, with a maximum overlap size of 5×10^4. Comparing `OnlyC0` with `AllCi`, the overlap size is almost the same for both. This confirms the need for reordering inside a layer, considering at least the corners as in Definition 18. We now discuss the impact of these orderings on the fill-in of F_1. We plot in Subfigure 3.26b the ratio $(nnz(F_1) - nnz(A))/nnz(F_1)$. For a small number of subdomains, the fill-in is stable for all orderings. When the number of subdomains increases, the corner ordering `AllCi` fills-in more than other orderings. For 512 subdomains, `NoCorner` is the corner ordering that adds the fewest nonzeros to A. The fill-in induced by `OnlyC0` corner ordering, for both numberingLayer

algorithms, is located between the two other corner orderings. Comparing DLN and ILN, we observe that the fill-in increases faster using ILN than using DLN when the number of subdomains increases. Especially, for 512 subdomains, the fill-in coming from DLN is 2 percent point smaller than ILN. Since the complexity of our main subroutines is based on the number of non-zeros, the time to determine the overlap for each subdomain and to factor each subdomain is expected to be slightly better in the case of `OnlyC0`. In practice, we do not observe a significant overhead in terms of runtime.

Table 3.16 presents the number of iterations to converge for each numberingLayer algorithm and for all corner orderings. For most cases, `NoCorner` corner ordering needs fewer iterations to converge than the others. More precisely, the combined DLN and `NoCorner` orderings require fewer iterations when the number of subdomains is 32 or greater than 64. As expected, the fact that the vertices of the layer are not reordered leads to a smaller number of iterations to converge, for both DLN and ILN with `NoCorner`. We also note that the overhead of using the corners, in term of iterations, is only a few iterations. In particular, we observe that for 16 subdomains and DLN, the usage of corners leads to perform 232 iterations instead of 233, without taking care of the corners. In the case of ILN, considering all corners reduces the number of iterations compared to the version using `NoCorner`. Focusing on the runtime in Figure 3.27, there is no runtime difference between DLN and ILN, except when the corners are ignored. The corner ordering `NoCorner` for both DLN and ILN solves the system in more than 12 seconds for 512 subdomains, compared to the others which converge 3 times faster. This gap is due to the size of the overlap presented in Subfigure 3.26a. The cost of applying the preconditioner increases when the size of the overlap increases. Also, we remark no difference between `OnlyC0` and `AllCi` corner orderings in terms of runtime.

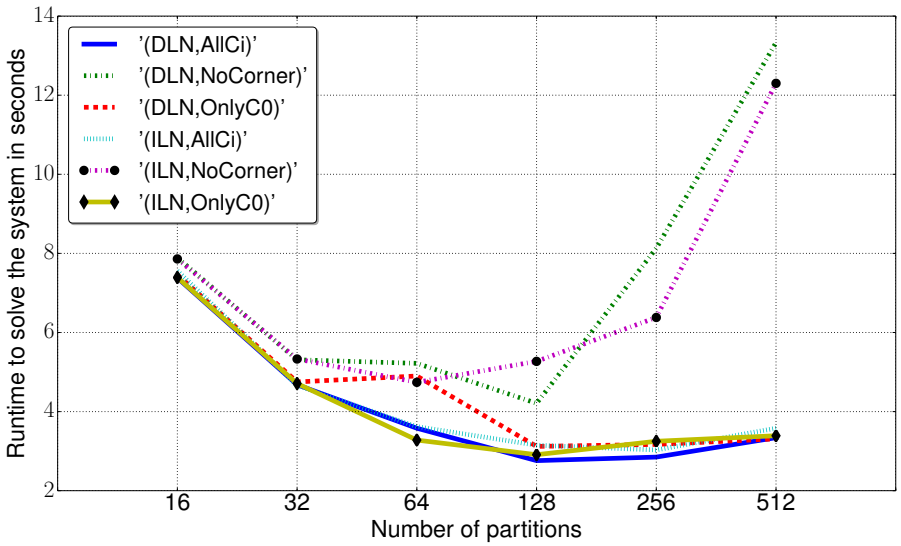
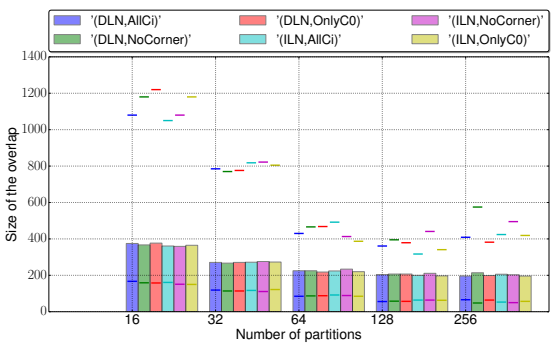


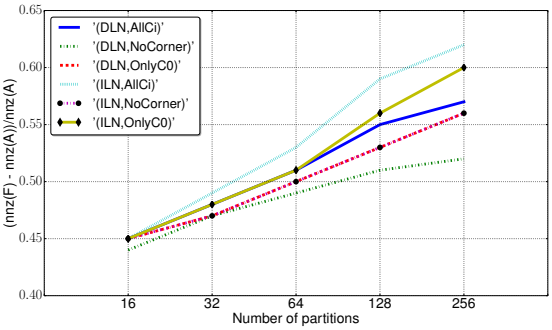
Figure 3.27 – Impact of ordering on the runtime to solve the system using CA-ILU(1) for SPE10 problem. 'DLN' and 'ILN' correspond to decreasing and increasing layer numbering, respectively, 'AllCi' considers the corners of all layers involved, 'OnlyC0' takes into account the corners of each boundary layer only, and 'NoCorner' does not reorder inside the layers.

	DLN			ILN		
nsubdomain	AllCi	OnlyC0	NoCorner	AllCi	OnlyC0	NoCorner
16	232	233	233	232	**231**	**231**
32	244	244	**243**	246	246	248
64	270	358	261	267	249	**247**
128	**258**	262	**258**	263	266	264
256	275	279	**270**	275	279	275
512	290	295	**288**	289	298	296

Table 3.16 – Comparison of CA-ILU(1) with different ordering methods on SPE10 for a number of subdomains increasing from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.



(a) Impact of ordering on the size of CA-ILU(1) overlap for matvf2dAD400400 problem.



(b) Impact of ordering on the fill-in of F_1, the symbolic ILU(1) factorization of A computed by CA-ILU(1) for matvf2dAD400400 problem.

Figure 3.28 – Impact of different orderings on CA-ILU(1) for SPE10 problem. 'DLN' and 'ILN' correspond to decreasing and increasing layer numbering, respectively, 'AllCi' considers the corners of all layer involved, 'OnlyC0' takes into account the corners of each boundary layer only and 'NoCorner' avoids reordering inside the layers.

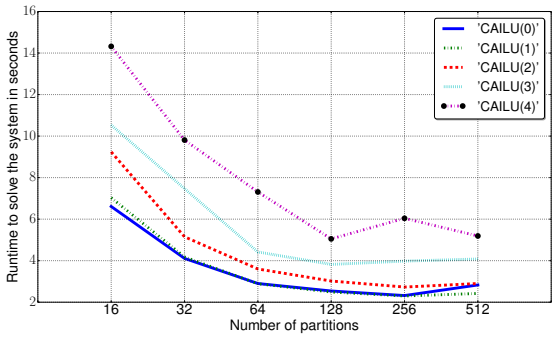
We now consider the matvf2dAD400400 problem and we also study the impact of these orderings on the overlap, the fill-in and the runtime. Subfigure 3.28a compares the variation of the size of CA-ILU(1) overlap for each ordering and for a number of subdomains increasing from 16 to 256. First, we observe that for each number of subdomains considered, reordering all corners of the layers also reduces the size of the overlap. The maximum overlap size decreases until 64 subdomains and then remains the same, except for `NoCorner` corner ordering, where the maximum overlap size reincreases for 256 subdomains. Subfigure 3.28b presents the evolution of the fill-in for each ordering when the number of subdomains increases. `OnlyC0` corner ordering induces a larger fill-in than `NoCorner` corner ordering, but a smaller fill-in than `AllCi`. DLN coupled with `NoCorner` has the most sparse F_1. As observed for the SPE10 problem, the DLN algorithm introduces fewer fill-in elements than ILN.

Studying the two numberingLayer algorithms on the test matrices presented in Table 3.3, we conclude that using the Increasing Layer Numbering yields to more fill-in and hence a denser F_k. As presented in Table 3.1, the complexity of our algorithm depends on the number of edges traversed during the search of the overlap. Also, the cost to perform the symbolic ILU(k) factorization of A is impacted by the fill-in. Furthermore, the comparison of the three corner orderings presented here shows that at least the corners of the boundary layers have to be

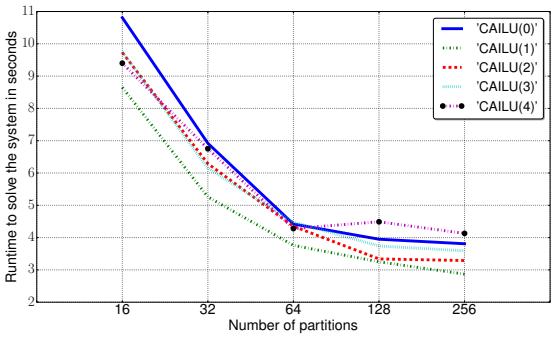
considered to avoid a too large overlap size (Subfigure 3.26a on SPE10 problem). Taking into account the fill-in induced by each corner ordering, the corner ordering `OnlyC0` should be selected, *a priori*. However, the size of the overlap leads to select `AllCi`, *a priori*. A trade-off between the fill-in and the size of the overlap leads to reducing the cost of applying CA-ILU(k).

Limitation of the memory

In the case of the problem 3DSKY100P1, the size of the overlap presented in Figure 3.24 allows to perform CA-ILU(k) with $k \in \{0, \ldots, 2\}$. For a larger k, the maximum size of the overlap implies that the size of the local memory required is too large. As discussed in subsection 3.3.2, we propose to bound the size of the overlap by a parameter $\eta = \tau \times \frac{n}{p}$, where n is the size of the matrix, p is the number of subdomains and τ is the number of subdomains that can be duplicated in memory. In this section, we present the impact of the limitation of the overlap on the number of iterations to converge and on the runtime to solve the system. To limit the size of the overlap, we apply Algorithm 3.6 instead of Algorithm 3.3. The runtimes are summarized in Subfigure 3.29a. Compared with Figure 3.25, we observe that we are able to run for $k \in \{0, \ldots, 4\}$ as expected. The runtime of CA-ILU(0) is the fastest for all number of subdomains presented here, except for 512 subdomains where CA-ILU(1) is 0.5 seconds faster. The runtime increases when the value of k increases. CA-ILU(4) is twice slower than CA-ILU(0) for 16 subdomains. For this problem, the size of the overlap is the limiting factor and increasing k does not lead to better performance. This is due to the construction of the bounded overlap. The purpose of Algorithm 3.7 is to ensure that the CA-ILU(0) overlap is added before CA-ILU(1). Therefore, when the bound is reached, the method performs an ILU(0) but with a larger overlap and a permutation of the system corresponding to a larger k. Moving to the SPE10 problem in Subfigure 3.29b, we compare the runtime of CA-ILU(k) with the same values of k. We observe that CA-ILU(1) outperforms the other methods for all numbers of subdomains. Between 16 and 64 subdomains, using $k > 0$ gives a faster runtime. CA-ILU(4) is the only method that becomes slower than CA-ILU(0) when the number of subdomains is larger than 64. Choosing an arbitrary $\tau = 2$ leads to using CA-ILU(k), $k > 2$ in comparison with the classical CA-ILU(k) in Figure 3.19.



(a) Impact of the bounded overlap size on the runtime of CA-ILU(1) for 3DSKY100P1 problem.



(b) Impact of the bounded overlap size on the runtime of CA-ILU(1) for SPE10 problem.

Figure 3.29 – Comparison of the runtime of CA-ILU(1) when the overlap size is bounded by $\eta = \tau \times \frac{n}{p}$ where n is the size of the matrix, p the number of subdomains and $\tau = 2$. The number of subdomain increases from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200, and a relative tolerance of $1e-6$.

3.6 Summary

This chapter presents CA-ILU(k) preconditioner, a communication avoiding preconditioner using an overlapping technique. This preconditioner is used in Krylov iterative methods as GMRES to solve the sparse linear system $Ax = b$. Algorithm 3.2 factors A in parallel by introducing two additional steps to the standard symbolic and numerical factorization steps. The key idea is to compute the CA-ILU(k) overlap of each subdomain that allows to factor the corresponding block row and its overlap without communication, using Algorithm 3.3. In addition, a preliminary step reorders each subdomain of A using Algorithm 3.5 in order to reduce the overlap of CA-ILU(k). Parallel tests, made on problems presented in Table 3.3, show that CA-ILU(k) is a competitive preconditioner. We first focus our experiments on CA-ILU(0) (no fill-in) compared with block Jacobi and RAS, both also computing ILU(0) on each block. CA-ILU(0) outperforms block Jacobi and is close to RAS, in term of the number of iterations to converge. Runtimes show that CA-ILU(0) is the fastest of the preconditioners for only one problem (parabolic_fem). As the size of the overlap of CA-ILU(0) is larger than for RAS, we replace ILU(0) factorization by LU factorization on each subdomain and for all preconditioners. Convergence results show that CA-ILU(0)-LU needs fewer iterations than block Jacobi and RAS to converge for all problems. However, runtimes show that CA-ILU(0)-LU has the best runtime for half of the problems considered. The size of the overlap of CA-ILU(0) coupled with LU leads to reduce the number of iterations to converge compared to the classical CA-ILU(0). For the other half of the problems studied, the cost of application of CA-ILU(0) with LU is larger than the gain on the number of iterations. Moving forward, we compare CA-ILU(k) for different values of $k \in \{0, \ldots, 4\}$. We observe that when k increases, the size of the overlap increases and that the size of the overlap can be too large so that the required memory for the factorization may not fit in memory. Algorithm 3.7 aims to bound the size of the overlap. Bounding the size of the overlap has a small impact on the number of iterations while in the meantime the runtime can be drastically reduced. Therefore, depending on the machine and the problem considered, preliminary tests are required to find the optimal value of k, the optimal number of subdomains and the maximum size of the overlap allowed per subdomain.

Chapter 4

LU-CRTP: computing the rank-k approximation of a sparse matrix: algebra and theoretical bounds

Outline of the current chapter

Introduction

An increasing number of applications require the computation of a low-rank approximation of a large sparse matrix. Ranging from data analytics problems such as Principal Component Analysis (PCA) to scientific computing problems such as fast solvers for integral equations or deep neuronal network (DNN), the low-rank approximation of a sparse matrix is a key component for reducing the cost while keeping accuracy. The best rank-k decomposition is given by the Singular Value Decomposition, that is expensive in practice. Less expensive alternatives have then been studied such as Rank Revealing QR factorization (RRQR), or Lanczos algorithm. More recently, randomized algorithms have been designed to address this problem. Those aim to

reduce the cost while obtaining accurate results with high probability. In the context of sparse matrices, the Rank Revealing QR factorization is a method returning factors that may be very dense, and even so dense that they do not fit in memory. In this chapter, we propose to use the LU factorization instead of QR to compute a truncated factorization that is effective in revealing the singular values with good accuracy, compared to the SVD. Note that this chapter corresponds to the content of (Grigori, Cayrols, et al., 2018). Since the R factor obtained from the QR factorization is the Cholesky factorization of $A^\top A$, it is expected that the factors obtained from a QR factorization are denser than the factors obtained from an LU factorization. In the following, we consider the Rank Revealing QR factorizations.

Our method aims to reduce the computation and memory consumption compared to the QR factorization, and also minimizing the communication cost. In (J. W. Demmel, Grigori, M. Hoemmen, et al., 2008), the communication cost reveals to be one of the major limitations on current and future architectures. Algorithms based on row and/or column permutation as Rank Revealing QR factorizations are known to be sub-optimal in terms of communication. Therefore, a communication avoiding Rank Revealing QR factorization, denoted CARRQR, has been introduced in (J. Demmel, Grigori, Gu, et al., 2013). Although RRQR computes a rank-k factorization with $O(k\log(p))$ messages, CARRQR requires only $O(\log(p))$ messages, modulo polylogarithmic factors, where p is the number of processors. To do so, this factorization selects k linearly independent columns using the QR factorization with Tournament Pivoting (QRTP). Experimental results presented in (J. Demmel, Grigori, Gu, et al., 2013) show that QRTP with a parameter k reveals the first k singular values of a matrix with an accuracy close to QR with Column Pivoting (QRCP).

We focus on computing a low-rank approximation of a matrix A by using a truncated LU factorization with column and row permutations, which has for a given k the form

$$\overline{A} = P_r A P_c = \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ \overline{A}_{21} & \overline{A}_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \overline{A}_{21}\overline{A}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ & S(\overline{A}_{11}) \end{bmatrix}, \tag{4.1}$$

$$S(\overline{A}_{11}) = \overline{A}_{22} - \overline{A}_{21}\overline{A}_{11}^{-1}\overline{A}_{12}, \tag{4.2}$$

where $A \in \mathbb{R}^{m\times n}$, $\overline{A}_{11} \in \mathbb{R}^{k\times k}$, $\overline{A}_{22} \in \mathbb{R}^{(m-k)\times(n-k)}$, and the rank-k approximated matrix is the following product

$$\begin{bmatrix} I \\ \overline{A}_{21}\overline{A}_{11}^{-1} \end{bmatrix} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \end{bmatrix} = \begin{bmatrix} \overline{A}_{11} \\ \overline{A}_{21} \end{bmatrix} \overline{A}_{11}^{-1} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \end{bmatrix} \tag{4.3}$$

The permutation matrices P_r and P_c are chosen so that $\overline{A}_{11}$ reveals the first k singular values of A. By revealing, we mean that the ratio of the computed singular values obtained from $\overline{A}_{11}$ to the singular values of A are bounded as in Equation 4.4. In practice, we consider that the singular values are well approximated when this ratio is on average close to one and in the worse case equal to a small constant. As shown later, the obtained factorization in Equation 4.1 satisfies the following properties

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(\overline{A}_{11})}, \frac{\sigma_j(S(\overline{A}_{11}))}{\sigma_{k+j}(A)} \leq q(m,n,k), \tag{4.4}$$

$$\rho_l(\overline{A}_{21}\overline{A}_{11}^{-1}) \leq F_{TP}, \tag{4.5}$$

for any $1 \leq l \leq m-k$, $1 \leq i \leq k$, and $1 \leq j \leq \min(m,n)-k$, where $\rho_l(B)$ denotes the 2-norm of the l-th row of B, and F_{TP} is a quantity coming from the QR factorization with Tournament Pivoting, presented in (J. Demmel, Grigori, Gu, et al., 2013).

The chapter is organized as follow. Section 4.1.1 presents the LU factorization with Column

Row Tournament Pivoting when the desired rank k is given in parameter and we detail the bounds presented above. Using this first algorithm as a building block, we present in Section 4.1.2 LU-CRTP where the rank K is unknown and determined by using a tolerance τ. We introduce in Section 4.1.3 a cheaper version named LU-CTP, where the row permutation is not computed using QRTP. Sequential and parallel experimental results presented in Section 4.2 show that LU-CRTP approximates well the singular values of a matrix A while the memory consumption to store the low-rank approximation is smaller than QRCP.

4.1 LU-CRTP: A low-rank approximation method based on Tournament Pivoting strategy

In this section we present the algebra of our block LU factorization with column and row tournament pivoting strategy. We propose to study two cases, first the rank is fixed, second the precision of the low-rank approximation is fixed. In the first case, we consider that the rank k is already known. In the second case, a tolerance τ is used as a stopping criterion, and the algorithm performs K/k iterations, where K is an overestimation of the rank. In both cases, we discuss the numerical properties of our block LU factorization, and its backward stability. In addition, we present bounds on the singular values of the obtained factors with respect to the matrix A.

Notation

Through the chapter, we use the MATLAB notation. For consistency, we use the same notations as in (Grigori, Cayrols, et al., 2018). Note that the notations can slightly differ from the previous chapters. Given a matrix A of dimension $m \times n$, the element in row i and column j is noted $A(i, j)$. Similarly, a submatrix of A formed by the rows i to j and the columns l to k is referred to as $A(i : j, l : k)$. Given two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{m\times k}$, the matrix $C \in \mathbb{R}^{m\times(n+k)}$ obtained by horizontal concatenation of A and B is referred to as $C = [A, B]$. Also, given two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{k\times n}$, the matrix $C \in \mathbb{R}^{(m+k)\times n)}$ obtained by vertical concatenation of A and B is referred to as $C = [A; B]$. When A is partitioned into blocks, we provide to the reader the dimension of a few blocks, and assume that the remaining dimensions are easily deduced from them. Therefore, if A is partitioned into $T + 1$ block columns, the partitioning given by $A = [A_{00}, \ldots, A_{T,0}]$ can be written as $A_{0:T,0}$. We note the absolute values of the matrix A as $|A|$, the max norm as $\|A\|_{max} = \max_{i,j} |A(i, j)|$. The 2-norm of the j-th row of A is denoted as $\rho_j(A)$, the 2-norm of the j-th column of A is denoted as $\chi_j(A)$, and the 2-norm of the j-th row of A^{-1} as $\omega_j(A)$.

Rank Revealing QR factorizations

We first introduce the QR factorization with column permutations of a matrix $A \in \mathbb{R}^{m\times n}$ of the form

$$AP_c = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{4.6}$$

where $Q \in \mathbb{R}^{m\times m}$ is an orthogonal matrix, $R_{11} \in \mathbb{R}^{k\times k}$ is an upper triangular matrix, $R_{12} \in \mathbb{R}^{k\times(n-k)}$, and $R_{22} \in \mathbb{R}^{(m-k)\times(n-k)}$. This factorization is said to be *rank revealing* if the following condition is satisfied, for any $1 \leq i \leq k$ and $1 \leq j \leq \min(m, n) - k$

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_{11})}, \frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq q(n, k), \tag{4.7}$$

where $q(n,k)$ is a low degree polynomial in k and n. Note that the definition given above is valid for any i and j, whereas the usual bounds as presented in (Chandrasekaran et al., 1994) and (Hong et al., 1992) concern $\sigma_{min}(R_{11})$ and $\sigma_{max}(R_{22})$. We next recall the properties of a strong Rank Revealing QR factorization, presented in (Gu and Eisenstat, 1996).

Theorem 22. *(Gu and Eisenstat (Gu and Eisenstat, 1996)) Let A be an $m \times n$ matrix and let $1 \leq k \leq \min(m,n)$. For any given parameter $f > 1$, there exists a permutation P_c such that*

$$AP_c = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{4.8}$$

where R_{11} is $k \times k$ and

$$(R_{11}^{-1}R_{12})_{i,j}^2 + \omega_i^2(R_{11})\chi_j^2(R_{22}) \leq f^2, \tag{4.9}$$

for any $1 \leq i \leq k$ and $1 \leq j \leq n-k$.

The inequality (4.9) bounds the singular values of R_{11} and R_{22} as in a rank revealing factorization, while also bounding the absolute value of $R_{11}^{-1}R_{12}$.

Theorem 23. *(Gu and Eisenstat (Gu and Eisenstat, 1996)) Let the factorization in Theorem 22 satisfy inequality (4.9). Then*

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_{11})}, \frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq \sqrt{1 + f^2 k(n-k)}, \tag{4.10}$$

for any $1 \leq i \leq k$ and $1 \leq j \leq \min(m,n) - k$.

The communication avoiding Rank Revealing QR factorization (CARRQR) introduced in (J. Demmel, Grigori, Gu, et al., 2013) computes a rank revealing factorization of a matrix A by using a block algorithm that selects k columns from A, permutes them to the leading positions, and computes k steps of a QR factorization without pivoting. Then the algorithm iterates on the trailing matrix. The k columns are selected by using a tournament pivoting called the QR factorization with Tournament Pivoting (QRTP). The CARRQR factorization satisfies

$$\chi_j^2(R_{11}^{-1}R_{12}) + (\chi(R_{22})/\sigma_{min}(R_{11}))^2 \leq F_{TP}^2, \tag{4.11}$$

for $j = \{1, \ldots, n-k\}$. In this, F_{TP} depends on k, f, n, the type of the tree used in QRTP, and the number of iterations of CARRQR. This leads to the following theorem

Theorem 24. *Assume that there exists a permutation P_c for which the QR factorization*

$$AP_c = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{4.12}$$

where R_{11} is $k \times k$ and satisfies (4.11). Then

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_{11})}, \frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq \sqrt{1 + F_{TP}^2(n-k)}, \tag{4.13}$$

for any $1 \leq i \leq k$ and $1 \leq j \leq \min(m,n) - k$.

Orthogonal matrices

We now consider the case of orthogonal matrices. Let Q be an orthogonal matrix partitioned as

$$Q = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}, \tag{4.14}$$

where $Q_{11} \in \mathbb{R}^{k \times k}$. The QR factorization with Tournament Pivoting of $[Q_{11}; Q_{21}]^\top$ leads to

$$P_r Q = \begin{bmatrix} \overline{Q}_{11} & \overline{Q}_{12} \\ \overline{Q}_{21} & \overline{Q}_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \overline{Q}_{21}\overline{Q}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{Q}_{11} & \overline{Q}_{12} \\ & S(\overline{Q}_{11}) \end{bmatrix}, \tag{4.15}$$

where $S(\overline{Q}_{11}) = \overline{Q}_{22} - \overline{Q}_{21}\overline{Q}_{11}^{-1}\overline{Q}_{12} = \overline{Q}_{22}^{-\top}$ (see (Pan, 2000), proof of Theorem 3.7). This factorization satisfies the following bounds

$$\rho_j(\overline{Q}_{21}\overline{Q}_{11}^{-1}) \leq F_{TP}, \tag{4.16}$$

$$\frac{1}{q_2(k,m)} \leq \sigma_i(\overline{Q}_{11}) \leq 1, \tag{4.17}$$

for all $1 \leq i \leq k$, $1 \leq j \leq m-k$, where $q_2(k,m) = \sqrt{1 + F_{TP}^2(m-k)}$. All these previous results lead us to present our contribution.

4.1.1 Rank-k approximation of a matrix A

Suppose the rank of an $m \times n$ matrix A is already known, and is equal to k. Thus the desired factorization of A is of the form

$$\overline{A} = P_r A P_c = \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ \overline{A}_{21} & \overline{A}_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \overline{A}_{21}\overline{A}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ & S(\overline{A}_{11}) \end{bmatrix}, \tag{4.18}$$

where

$$S(\overline{A}_{11}) = \overline{A}_{22} - \overline{A}_{21}\overline{A}_{11}^{-1}\overline{A}_{12}. \tag{4.19}$$

The permutation matrices P_r and P_c are built such that the singular values of $\overline{A}_{11}$ and $S(\overline{A}_{11})$ approximate well the largest k and the smallest $n-k$ singular values of A, respectively. The stability of the LU factorization of A depends on the ratio $||\hat{L}||_{max}||\hat{U}||_{max}/||A||_{max}$, where $\hat{L}$ and $\hat{U}$ are the triangular factors of A. We assume that the product $||\hat{L}||_{max}||\hat{U}||_{max} \approx ||L||_{max}||U||_{max}$. It means the roundoff errors do not have a large impact on the norm of both factors. We, therefore, focus on bounding the maximum norm of both factors. To do so, it suffices to bound the elements of $|\overline{A}_{21}\overline{A}_{11}^{-1}|$, since the U factor from Equation (4.18) corresponds to the first k rows of $\overline{A}$.

Our algorithm selects the first k columns of A by using the QR factorization with Tournament Pivoting on A,

$$AP_c = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{4.20}$$

where $Q \in \mathbb{R}^{m \times m}$, $Q_{11}, R_{11} \in \mathbb{R}^{k \times k}$. This factorization gives that R_{11} and R_{22} reveal the largest k and the smallest $n-k$ singular values of A. However, we aim that $\overline{A}_{11}$ and $S(\overline{A}_{11})$ reveal the singular values of A, in Equation (4.18). Thus, we select k rows from the first k columns of Q

by using the QR factorization with Tournament pivoting on $Q(:, 1:k)^\top$. This leads to

$$P_r Q = \begin{bmatrix} \overline{Q}_{11} & \overline{Q}_{12} \\ \overline{Q}_{21} & \overline{Q}_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \overline{Q}_{21}\overline{Q}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{Q}_{11} & \overline{Q}_{12} \\ & S(\overline{Q}_{11}) \end{bmatrix}, \tag{4.21}$$

where

$$S(\overline{Q}_{11}) = \overline{Q}_{22} - \overline{Q}_{21}\overline{Q}_{11}^{-1}\overline{Q}_{12} = \overline{Q}_{22}^{-\top} \tag{4.22}$$

such that $\rho_j(\overline{Q}_{21}\overline{Q}_{11}^{-1}) = \rho_j(\overline{A}_{21}\overline{A}_{11}^{-1}) \leq F_{TP}$, for all $1 \leq j \leq m-k$, is upper bounded as in Equation (4.16), and the singular values of $\overline{Q}_{11}$ and $\overline{Q}_{22}$ are bounded as in Equation (4.17).

Applying both P_c and P_r on A, we obtain

$$\begin{aligned} P_r A P_c &= \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ \overline{A}_{21} & \overline{A}_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \overline{Q}_{21}\overline{Q}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{Q}_{11} & \overline{Q}_{12} \\ & S(\overline{Q}_{11}) \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix} \\ &= \begin{bmatrix} I & \\ \overline{A}_{21}\overline{A}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ & S(\overline{A}_{11}) \end{bmatrix}, \end{aligned} \tag{4.23}$$

where

$$\overline{Q}_{21}\overline{Q}_{11}^{-1} = \overline{A}_{21}\overline{A}_{11}^{-1}, \tag{4.24}$$

$$S(\overline{A}_{11}) = S(\overline{A}_{11})R_{22} = \overline{Q}_{22}^{-\top} R_{22}. \tag{4.25}$$

Note that in (Pan, 2000), Pan uses a similar derivation to prove the existence of a rank revealing LU factorization. However, this derivation is not used in this chapter. We remark that computing P_c by using QRCP like algorithms requires the computation of R_{22}. This last factor could be therefore used instead of computing an LU factorization and its Schur complement $S(\overline{A}_{11})$. Moreover, the quantity $\|A_{21}A_{11}\|_{max}$, that is related to the numerical stability of LU, is not bounded in (Pan, 2000).

In Algorithm 4.1, we present the LU factorization with Column Row Tournament Pivoting (LU-CRTP), that takes as input an $m \times n$ matrix A, and the desired rank k. Line 1, the QR factorization with Tournament Pivoting on A selects k columns that are then moved to the leading positions of A. This factorization performs a strong Rank Revealing QR factorization of subsets of $2k$ columns. Note that, compared to Equation 4.23, the QR factorization of A is not performed until the end of the matrix. The k selected columns of A are factored by using QR. Then, similarly to the k columns, k rows are selected by the QR factorization with Tournament Pivoting on $Q(:, 1:k)^\top$, line 3. The obtained permutations are applied to A and Q, line 4.

Line 5, the term L_{21} can be computed either using Q or A. Although the equality $\overline{Q}_{21}\overline{Q}_{11}^{-1} = \overline{A}_{21}\overline{A}_{11}^{-1}$ is verified in infinite precision, it might not be the case in finite precision. Due to round-off error, using Q is more stable than using A. When A is sparse, the product $\overline{Q}_{21}\overline{Q}_{11}^{-1}$ has more nonzeros than $\overline{A}_{21}\overline{A}_{11}^{-1}$. Q being denser than A, the additional nonzeros correspond to exact cancellations and they are due to round-off errors. In our experimental results, we use a heuristic to determine whether the computation of L_{21} is performed using A or Q. First, using A, if $\|\overline{A}_{21}\overline{A}_{11}^{-1}\|_{max}\|$ is larger than $\sqrt{n}\|A\|_{max}$ (which is the growth factor observed experimentally for partial pivoting). Note that, in the case of CUR decomposition, the computation of L_{21} is not performed.

Algorithm 4.1 LU-CRTP (A, k): rank-k truncated LU factorization with Column Row Tournament Pivoting of a matrix A

Input: $A \in \mathbb{R}^{m\times n}$, k the desired rank

Output: permutation matrices P_r and P_c, rank-k truncated factorization L_kU_k, factor R_k, such that $(AP_c)(:, 1:k) = Q_kR_k$,

$$P_rAP_c = \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ \overline{A}_{21} & \overline{A}_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \overline{A}_{21}\overline{A}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ & S(\overline{A}_{11}) \end{bmatrix},$$

$$L_k = \begin{bmatrix} I \\ \overline{A}_{21}\overline{A}_{11}^{-1} \end{bmatrix} = \begin{bmatrix} I \\ L_{21} \end{bmatrix}, U_k = \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \end{bmatrix},$$

where $L_k \in \mathbb{R}^{m\times k}$, $U_k \in \mathbb{R}^{k\times n}$, $R_k \in \mathbb{R}^{k\times k}$, and the remaining matrices have the corresponding dimensions.

Note that $S(\overline{A}_{11})$ is not computed here.

1: Select k columns by using QR with tournament pivoting on A,

$$P_c \leftarrow QRTP(A, k)$$

2: Compute the thin QR factorization of the selected columns,

$$(AP_c)(:, 1:k) = Q_kR_k, \text{ where } Q_k \in \mathbb{R}^{m\times k} \text{ and } R_k \in \mathbb{R}^{k\times k}$$

3: Select k rows by using QR with tournament pivoting on $Q_k^\top$,

$$P_r \leftarrow QRTP(Q_k^\top, k)$$

4: Let

$$\overline{A} = P_rAP_c = \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ \overline{A}_{21} & \overline{A}_{22} \end{bmatrix}, \quad P_rQ_k = \begin{bmatrix} \overline{Q}_{11} \\ \overline{Q}_{21} \end{bmatrix}$$

5: Compute

$$L_{21} = \overline{Q}_{21}\overline{Q}_{11}^{-1} = \overline{A}_{21}\overline{A}_{11}^{-1} \text{ (see discussion in the text)}$$

We now prove that the factorization presented in Equation (4.18) reveals the singular values of A, while it bounds the element growth in the LU factorization.

Theorem 25. *(Grigori et al. (Grigori, Cayrols, et al., 2018)) Let A be an $m \times n$ matrix. The LU-CRTP (A,k) factorization obtained by using Algorithm 4.1,*

$$\overline{A} = P_rAP_c = \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ \overline{A}_{21} & \overline{A}_{22} \end{bmatrix} = \begin{bmatrix} I & \\ \overline{Q}_{21}\overline{Q}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ & S(\overline{A}_{11}) \end{bmatrix}, \tag{4.26}$$

where

$$S(\overline{A}_{11}) = \overline{A}_{22} - \overline{A}_{21}\overline{A}_{11}^{-1}\overline{A}_{12} = \overline{A}_{22} - \overline{Q}_{21}\overline{Q}_{11}^{-1}\overline{A}_{12}, \tag{4.27}$$

satisfies the following properties

$$\rho_l(\overline{A}_{21}\overline{A}_{11}^{-1}) = \rho_l(\overline{Q}_{21}\overline{Q}_{11}^{-1}) \leq F_{TP}, \tag{4.28}$$

$$\|S(\overline{A}_{11}\|_{max} \leq \min\left((1+F_{TP}\sqrt{k})\|A\|_{max}, F_{TP}\sqrt{1+F_{TP}^2(m-k)}\sigma_k(A)\right) \tag{4.29}$$

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(\overline{A}_{11})}, \frac{\sigma_j(S(\overline{A}_{11}))}{\sigma_{k+j}(A)} \leq q(m,n,k), \tag{4.30}$$

for any $1 \leq l \leq m-k$, $1 \leq i \leq k$, *and* $1 \leq j \leq \min(m,n)-k$. *Here* F_{TP} *is the bound obtained from QR with tournament pivoting, and* $q(m,n,k) = \sqrt{(1+F_{TP}^2(n-k))(1+F_{TP}^2(m-k))}$.

Proof. See the proof in (Grigori, Cayrols, et al., 2018). □

Considering the rank-k approximation of A from Algorithm 4.1, denoted $\tilde{A}_k$, the inverse of the ratio $\frac{\sigma_i(A)}{\sigma_i(\tilde{A}_k)}$ can be bounded as follows.

Let

$$\tilde{A}_k = \begin{bmatrix} I \\ \overline{A}_{21}\overline{A}_{11}^{-1} \end{bmatrix} \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \end{bmatrix}, \tag{4.31}$$

and

$$\overline{A} = \tilde{A}_k + \begin{bmatrix} 0_k & \\ & S(\overline{A}_{11}) \end{bmatrix}, \tag{4.32}$$

where 0_k is a zero matrix of size $k \times k$. From Equation (4.30), we obtain

$$\|\overline{A} - \tilde{A}_k\|_2 \leq q(m,n,k)\sigma_{k+1}(A). \tag{4.33}$$

Therefore, with respect to $q(m,n,k)$, $\tilde{A}_k$ is the best low-rank approximation computed by the Singular Value Decomposition. Using Corollary 8.6.2 from (G. H. Golub et al., 2013) in Equation (4.32) leads to

$$|\sigma_i(\overline{A}) - \sigma_i(\tilde{A}_k)| \leq \| \begin{bmatrix} 0_k & \\ & S(\overline{A}_{11}) \end{bmatrix} \|_2, \tag{4.34}$$

for any $1 \leq i \leq k$. Using bound in (4.33), we obtain

$$|\sigma_i(\overline{A}) - \sigma_i(\tilde{A}_k)| \leq \sigma_{k+1}(A) \cdot q(m,n,k). \tag{4.35}$$

Hence, we have the following relation between the singular values of A, $\tilde{A}_k$, and $S(\overline{A}_{11})$,

$$1 - \frac{\sigma_{k+1}(A)}{\sigma_i(A)} \cdot q(m,n,k) \leq \frac{\sigma_i(A)}{\sigma_i(\tilde{A}_k)} \leq 1 + \frac{\sigma_{k+1}(A)}{\sigma_i(A)} \cdot q(m,n,k), \tag{4.36}$$

with $1 \leq i \leq k$. This ratio is close to 1 when $\frac{\sigma_{k+1}(A)}{\sigma_i(A)} \cdot q(m,n,k)$ is small. This case appears when there is a large gap between $\sigma_k(A)$ and $\sigma_{k+1}(A)$.

4.1.2 Rank-K approximation, when K is unknown

We are now considering the case where the rank of the approximation needs to be determined during the factorization. For that, we introduce in Algorithm 4.2, LU-CRTP (A,k,τ), an updated version of LU-CRTP (A,k) which uses a stopping criterion τ. This algorithm uses Algorithm 4.1 as a subroutine and iterates until the condition using τ is satisfied. Since Algorithm 4.1 computes an approximation of the first k singular values of its input matrix, Algorithm 4.2 determines an overestimation of the rank $K = t \times k$ such that a subset of the K singular values are larger than the tolerance τ, where t is the number of iterations. Therefore, the real rank of the approximation

is between $(t-1)\times k$ and $t\times k$. One alternative is based on detecting a gap between the singular values. Although this approach is not discussed here, Algorithm 4.2 can be easily adapted.

Algorithm 4.2 proceeds as follow. The initial guess of the rank k is used by the subroutine LU-CRTP (A,k) to compute one step of the block LU factorization with Column and Row Tournament Pivoting, as described in the previous section. If the k approximated singular values computed are larger than or equal to τ, then the factorization continues recursively on the trailing matrix $S(\overline{A}_{11})$. After T iterations, we obtain the factorization of Equation (4.37)

Theorem 26. *(Grigori et al. (Grigori, Cayrols, et al., 2018)) Suppose we have computed T block steps of LU-CRTP (A,k,τ) factorization by using Algorithm 4.2, where $A\in\mathbb{R}^{m\times n}$ and $K=Tk$. We obtain the following factorization*

$$P_r A P_c = L_K U_K \tag{4.37}$$
$$= \begin{bmatrix} I & & & & \\ L_{21} & I & & & \\ \vdots & \vdots & \ddots & & \\ L_{T1} & L_{T2} & \dots & I & \\ L_{T+1,1} & L_{T+1,2} & \dots & L_{T+1,T} & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & \dots & U_{1T} & U_{1,T+1} \\ & U_{22} & \dots & U_{2T} & U_{2,T+1} \\ & & \ddots & \vdots & \vdots \\ & & & U_{TT} & U_{T,T+1} \\ & & & & U_{T+1,T+1} \end{bmatrix},$$

where $L_{i+1,j}$ and U_{ij} are $k\times k$ for $1\le i,j\le T$, and $U_{T+1,T+1}$ is $(m-Tk)\times(n-Tk)$. The following properties are satisfied:

$$\rho_l(L_{i+1,j}) \le F_{TP}, \tag{4.38}$$

$$\|U_K\|_{max} \le \min\left((1+F_{TP}\sqrt{k})^{K/k}\|A\|_{max},\ q_2(m,k)q(m,n,k)^{K/k-1}\sigma_K(A)\right), \tag{4.39}$$

$$\frac{1}{\prod_{v=0}^{t-2} q(m-vk,n-vk,k)} \le \frac{\sigma_{(t-1)k+i}(A)}{\sigma_i(U_{tt})} \le q(m-(t-1)k,n-(t-1)k,k), \tag{4.40}$$

$$1 \le \frac{\sigma_j(U_{T+1,T+1})}{\sigma_{K+j}(A)} \le \prod_{v=0}^{K/k-1} q(m-vk,n-vk,k), \tag{4.41}$$

for any $1\le l\le k$, $1\le i\le k$, $1\le t\le T$, and $1\le j\le \min(m,n)-K$. Here F_{TP} is the bound obtained from QR with Tournament Pivoting, $q_2(m,k)=\sqrt{1+F_{TP}^2(m-k)}$, and $q(m,n,k)=\sqrt{(1+F_{TP}^2(n-k))(1+F_{TP}^2(m-k))}$.

Proof. See the proof of the Theorem 3.2 in (Grigori, Cayrols, et al., 2018). □

Algorithm 4.2 presents the LU-CRTP (A,k,τ) factorization. As detailed above, this algorithm can be used when the rank of the low-rank approximation needs to be determined. Using the bounds of Theorems 25 and 26, it can easily be seen that the R factor obtained from the QR factorization with Tournament Pivoting gives a slightly better estimation of the singular values than the diagonal blocks of U_K. Hence, we use the R factor to approximate the singular values of A, and to determine the rank K of the low-rank approximation.

Algorithm 4.2 LU-CRTP (A, k, τ): rank-K truncated LU factorization with Column Row Tournament Pivoting of a matrix A, using tolerance τ to identify singular values large enough to keep in the low-rank approximation

Input: $A \in \mathbb{R}^{m\times n}$, k, tolerance τ

Output: permutation matrices P_r and P_c, rank K, truncated factorization $B = \tilde{L}_K\tilde{U}_K$ such that $\tilde{L}_K \in \mathbb{R}^{m\times K}$, $\tilde{U}_K \in \mathbb{R}^{K\times n}$ $\tilde{\sigma}_{K-k}(A) \geq \tau > \tilde{\sigma}_K(A)$, where $\tilde{\sigma}_k(A)$ is the K-th singular value of A approximated by the algorithm,

$$P_rAP_c = L_KU_K = \begin{bmatrix} I & & & & \\ L_{21} & I & & & \\ \vdots & \vdots & \ddots & & \\ L_{T1} & L_{T2} & \dots & I & \\ L_{T+1,1} & L_{T+1,2} & \dots & L_{T+1,T} & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & \dots & U_{1T} & U_{1,T+1} \\ & U_{22} & \dots & U_{2T} & U_{2,T+1} \\ & & \ddots & \vdots & \vdots \\ & & & U_{TT} & U_{T,T+1} \\ & & & & U_{T+1,T+1} \end{bmatrix},$$

$\tilde{L}_K = L_K(:, 1:K)$ $\tilde{U}_K = U_K(1:K, :)$

Note that $U_{T,T+1}$ is not updated during the last iteration.

$\overline{A} = A$
for $T = 1$ to n/k **do**
  $j = (T-1)k+1$, $K = j+k-1$
  Determine row/column permutations by using Algorithm 4.1,
  $[P_{r_k}, P_{c_k}, L_k, U_k, R_k] \leftarrow LU-CRTP(\overline{A}(j:m, j:n), k)$
  $P_{r_T} = \begin{bmatrix} I & \\ & P_{r_k} \end{bmatrix}$, $P_{c_T} = \begin{bmatrix} I & \\ & P_{c_k} \end{bmatrix}$, where I is $(j-1)\times(j-1)$
  $\overline{A} = P_rAP_c$, $L_K = P_rL_KP_c$, $U_K = P_rU_KP_c$, $P_r = P_{r_T}P_r$, $P_c = P_cP_{c_T}$
  $U_K(j:K, j:n) = U_k$, $L_K(j:m, j:K) = L_k$
  for $i = 1$, to k **do**
    $\tilde{\sigma}_{j+i-1}(A) = \sigma_i(R_k)$
  end for
  if $\tilde{\sigma}_K(A) < \tau$ **then**
    Return $\tilde{L}_K = L_K(:, 1:K)$, $\tilde{U}_K = U_K(1:K, :)$, P_r, P_c
  else
    Update the trailing matrix,
    $\overline{A}(K+1:m, K+1:n) = \overline{A}(K+1:m, K+1:n) - L_kU_k$.
  end if
end for

4.1.3 A less expensive LU factorization with Column Tournament Pivoting

In this section, we present a less expensive version of the LU factorization with Column Row Tournament Pivoting that satisfies part of the bounds from Theorem 25. We only focus on one step of the LU factorization, where the desired rank is k. However, the modifications presented here are easily applicable to the case where the rank $K > k$.

The cheaper version of LU-CRTP (A, k, τ) corresponds to avoid computing the row permutation matrix P_r by using the QR factorization with Tournament Pivoting. Therefore, this version also uses a cheaper version of LU-CRTP (A, k) in which the row permutation matrix P_r is obtained from the LU factorization with partial pivoting. We refer to this modification in Algorithm 4.1 and 4.2 as LU factorization with Column Tournament Pivoting, denoted further as LU-CTP (A, k) and LU-CTP (A, k, τ), respectively. Given an $m \times n$ matrix A, the LU factorization selects k columns by using QR factorization with Tournament Pivoting on the matrix A. The obtained

factorization is

$$AP_c = \begin{bmatrix} A_{11}^c & A_{12}^c \\ A_{21}^c & A_{22}^c \end{bmatrix} = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$$

where $A_{11}^c \in \mathbb{R}^{k\times k}$, $Q \in \mathbb{R}^{m\times m}$, $Q_{11}, R_{11} \in \mathbb{R}^{k\times k}$. Here the column permutation matrix P_c is the same as the one computed in LU-CRTP (A, k). The row permutation matrix P_r is obtained by using the LU factorization of the first k columns with partial pivoting of AP_c. Note that to reduce the communication, LU factorization with Tournament Pivoting can be used to select the k rows of $AP_c(:, 1:k)$. However, when the growth factor of L_{21} is too large, LU factorization with Tournament Pivoting is applied on the first k columns of Q (Grigori, J. W. Demmel, et al., 2011),

$$P_r \begin{bmatrix} A_{11}^c \\ A_{21}^c \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11},$$

where L_{11}, U_{11} are of dimension $k \times k$. The obtained LU factorization

$$P_r A P_c = \begin{bmatrix} \overline{A}_{11} & \overline{A}_{12} \\ \overline{A}_{21} & \overline{A}_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & S(\overline{A}_{11}) \end{bmatrix},$$

where $S(\overline{A}_{11}) = \overline{A}_{22} - L_{21}U_{12}$. Note that this factorization does not lower bound the singular values of Q_{11}. As a consequence, we cannot bound in theory the singular values of $\overline{A}_{11}$ and $S(\overline{A}_{11})$, with respect to the singular values of A. However, experimental results show that in practice this factorization approximates well the singular values of A.

4.2 Performance results

In this section, we present extensive tests made on a sequential version of LU-CRTP. We show that QRCP algorithm can be replaced by LU-CRTP to compute an approximation of the singular values of a matrix A. The singular values computed by LU-CRTP are close to the singular values returned by the Singular Value Decomposition algorithm (SVD). In addition, the factors L and U of A returned by LU-CRTP are sparser than the factors Q and R computed by QRCP. Therefore, LU-CRTP can factor matrices where QRCP requires too much memory. We study LU-CRTP through three parameters, the accuracy of the singular values, the numerical stability of the block LU factorization of A, and the fill-in of the factors. All tests are made on a local machine with MATLAB 2015a, and on an SGI UV2000 supercomputer with MATLAB 2014b, named HPC2 and managed by the LJLL at UPMC. HPC2 has 32 nodes and a total memory space of 1 TB. Each node is equipped with an Intel Xeon IvyBridge E5-4650 v2 processor (2.40GHz, 10 cores). This machine is used for running tests that require larger memory.

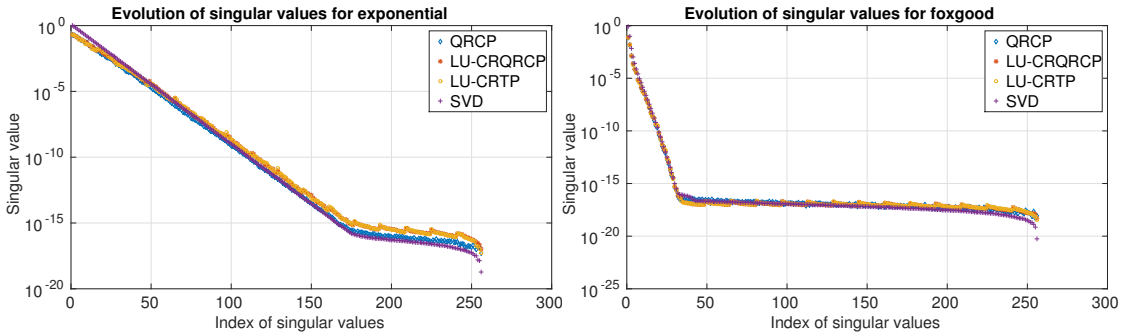
4.2.1 Accuracy of the singular values computed by LU-CRTP

To study the accuracy of the singular values returned by LU-CRTP, we use the SVD algorithm as the reference. We also compare our algorithm with the QR factorization with Column Pivoting (QRCP). In (J. Demmel, Grigori, Gu, et al., 2013), Demmel *et al.* show that the absolute diagonal elements of the R factor returned by QRCP approximate the singular values of a matrix A with a maximum error of one order of magnitude. The error on the i'th singular value is defined as the ratio $\tilde{\sigma}_i/\sigma_i$, where $\tilde{\sigma}_i$ is the approximation of the i'th singular value, and σ_i is the i'th singular value returned by the SVD algorithm. Although QRCP uses the diagonal elements of the R factor to get an approximation of the singular values, we further study different methods to compute

the singular values. We first use a set of 16 challenging matrices coming from different kind of problems, and described in Table 4.1. All matrices of the set are generated using MATLAB script, with a size of 256×256. As presented in Algorithm 4.1, our algorithm uses QRTP to select the k columns of A. Thus, to study whether QRTP has an impact on the approximation of the singular values, we introduce LU-CRQRCP, a variant of LU-CRTP, where QRTP is replaced by QRCP. We call QRCP, LU-CRQRCP and LU-CRTP algorithms on each matrix to obtain an approximation of each singular value. In the following tests, the panel size k, used as the desired rank internally, is set to 16 and the number of singular values nSV varies from 64 to 256. Since LU-CRTP is designed to return a low-rank approximation of a matrix A, we do not consider the part of the spectrum where the singular values are smaller than ϵ, the machine-precision. To avoid potential numerical issues when the singular values are smaller than ϵ, we replace them by ϵ in all tests.

No.	Matrix	Description
1	Baart	Discretization of the 1st kind Fredholm integral equation
2	Break1	Break 1 distribution, matrix with prescribed singular values
3	Break9	Break 9 distribution, matrix with prescribed singular values
4	Deriv2	Computation of second derivative
5	Devil	The devil's stairs, a matrix with gaps in its singular values
6	Exponential	Exponential distribution, $\sigma_1 = 1, \sigma_i = \alpha^{i-1} (i = 2, ..., n)$, $\alpha = 10^{\frac{-1}{11}}$
7	Foxgood	Severely ill-posed test problem of the 1st kind Fredholm integral equation used by Fox and Goodwin
8	Gravity	One-dimensional (1D) gravity surveying problem
9	Heat	Inverse heat equation
10	Phillips	Phillips' famous test problem
11	Random	Random matrix $A = 2 \times rand(n) - 1$
12	Shaw	1D image restoration model
13	Spikes	Test problem with a "spiky" solution
14	Stewart	Matrix $A = U \epsilon V^T + 0.1\sigma_m * rand(n)$, where σ_m is the smallest nonzero singular values
15	Ursell	Integral equation with no square integrable solution
16	Wing	Test problem with a discontinuous solution

Table 4.1 – Set of dense matrices



(a) Exponential problem. **(b)** Foxgood problem.

Figure 4.1 – Evolution of the singular values computed by QRCP, LU-CRQRCP, and LU-CRTP, compared to the SVD, for different problems.

We first show the behavior of each algorithm by focusing on Exponential and Foxgood problems in Figures 4.1a and 4.1b. Both figures plot the full spectrum returned by each algorithm. We observe that the spectra of LU-CRTP and LU-CRQRCP are close to the SVD, and that LU-CRTP and LU-CRQRCP return the same approximation of the singular values. We remark that after each update of the trailing matrix in LU-CRTP algorithm, the norm of few columns slightly increases. This introduces a degradation of the accuracy of our method but as we will show later, the error on the singular values does not reach two orders of magnitudes. In the context of low-rank approximation, one may request a subset of the spectrum. We next study the cases where only the first quarter or half the spectrum is needed.

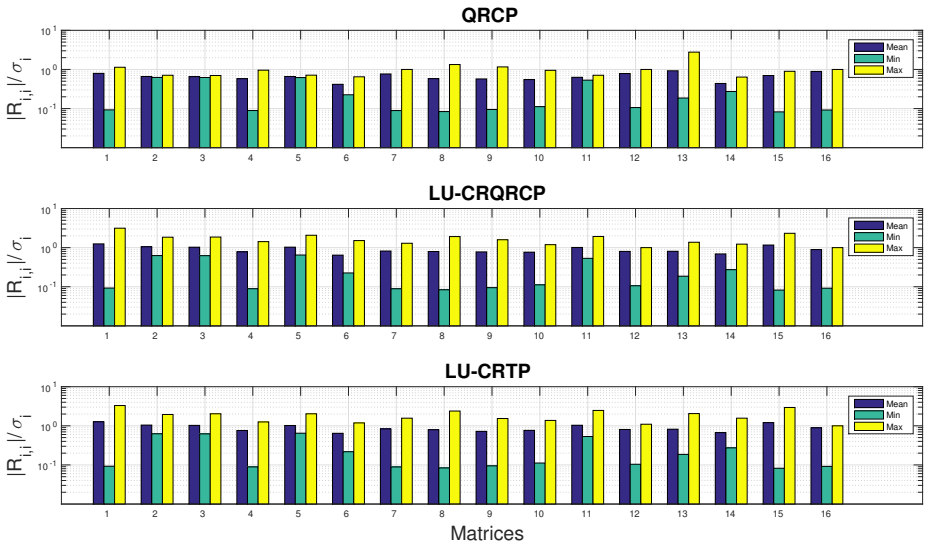


Figure 4.2 – Accuracy of the approximate singular values computed by QRCP, LU-CRQRCP and LU-CRTP compared to the real singular values. Matrix size is 256, $k = 16$ and $nSV = 64$

We consider the first $n/4$ singular values, where n is the dimension of A, and we compare the average ratio, its maximum and minimum for each method and each matrix. Figure 4.2 summarizes the accuracy of each method. The average ratio is close to 1 for all methods. Moreover, the largest maximum and smallest minimum are at most equal to one order of magnitude. The largest and smallest singular values computed by LU-CRQRCP and LU-CRTP are similar. It means that the replacement of QRCP by QRTP does not degrade the approximation of the singular values. We now study the accuracy of each method for the first half of the spectrum, and for the full spectrum, in Figure 4.3. The bars represent the ratios for the first $n/2$ singular values and the red line corresponds to the ratios for 256 singular values. When $nSV = 128$, the largest ratio of each method is still equal to one order of magnitude, at most. When $nSV = 256$, although the maximum ratios increase up to 10 for QRCP, LU-CRQRCP and LU-CRTP have a largest maximum ratio equal to 50. This increase comes from the update of the trailing matrix. Take as an example the problems Phillips and Stewart whose spectrum is plotted in Figures 4.4a and 4.4b, respectively. Focusing on the end of the spectrum, the singular values, computed by the SVD, drastically decrease. However, the last k singular values returned by LU-CRTP have a small offset and do not decrease as the real one. This explains the increase of the maximums in Figure 4.3, when the number of singular values goes from $n/2$ to n. If we remove the last

k singular values, we observe better ratios in Figure B.3, in the appendix. In summary, the approximation of the singular values returned by LU-CRTP is equivalent to the ones returned by LU-CRQRCP, and both variants are close to QRCP.

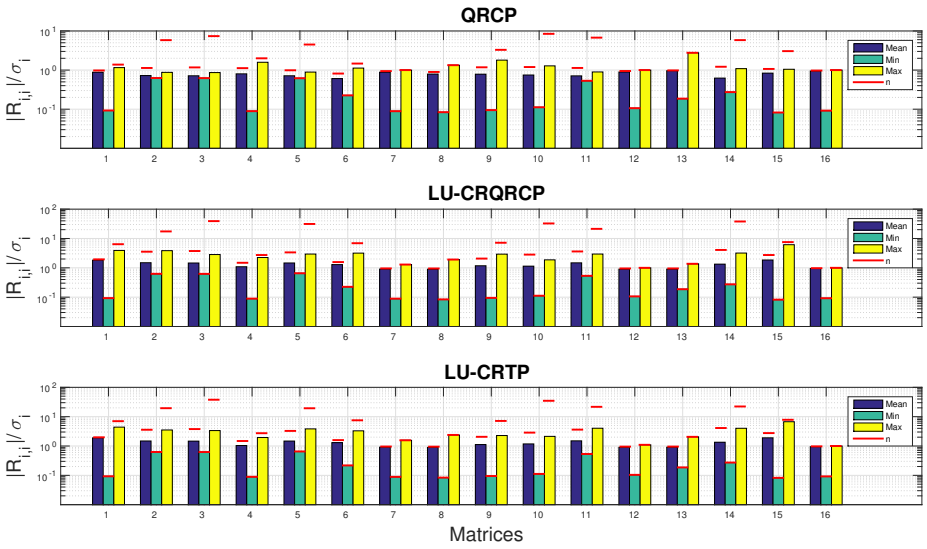
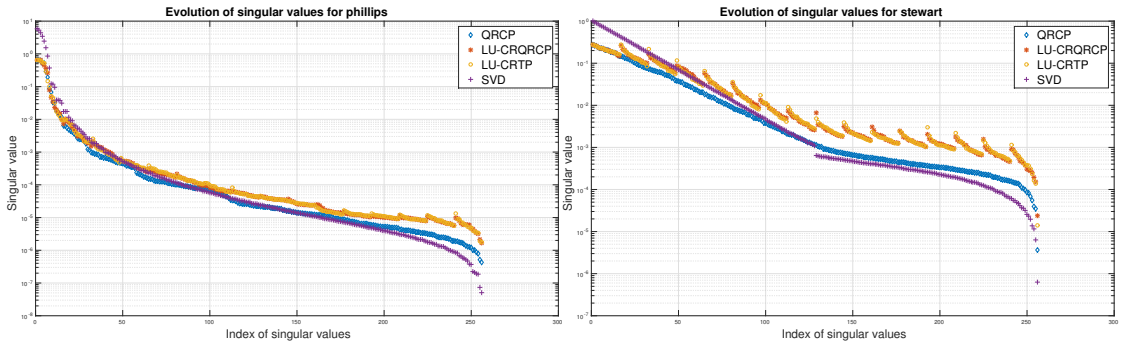


Figure 4.3 – Accuracy of the approximate singular values computed by QRCP, LU-CRQRCP and LU-CRTP, compared to the real singular values. Matrix size is 256, $k = 16$ and $nSV = [128, 256]$



(a) Phillips problem. **(b)** Stewart problem.

Figure 4.4 – Evolution of the singular values computed by QRCP, LU-CRQRCP, and LU-CRTP, compared to the SVD, for different problems.

Impact of the panel size k on the accuracy of the estimation of the singular values returned by LU-CRTP

We now discuss the impact of the parameter k, the number of columns selected by QRTP, on the accuracy of the approximated singular values returned by LU-CRTP. We use the same 16 challenging problems and request first the leading 64 singular values of the spectrum in Figure

4.5, and then the half and the full spectrum in Figure 4.6. We also consider the same ratios as presented above, and k equals to 16, 32, and 64.

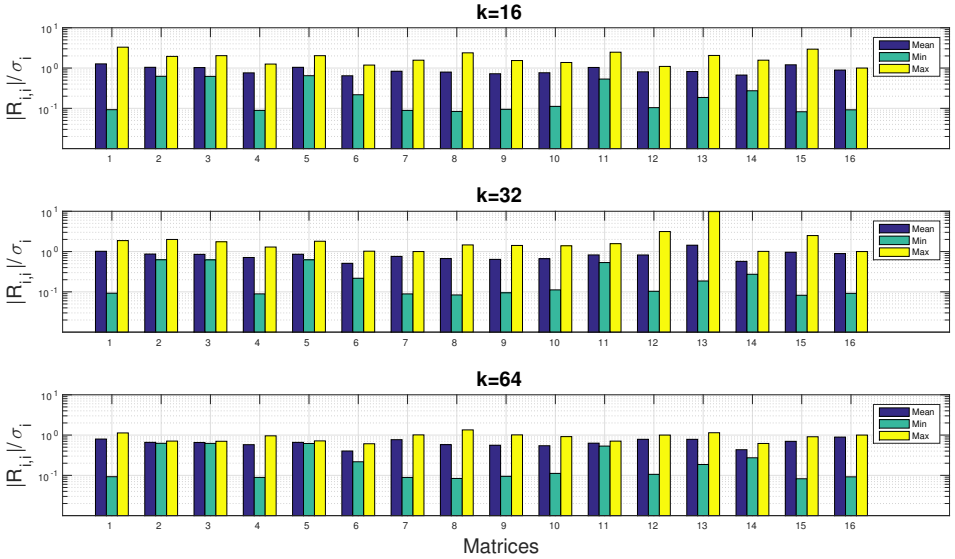


Figure 4.5 – Influence of the parameter k on the accuracy of the approximate singular values returned by LU-CRTP. Matrix size is 256 and $nSV = 64$.

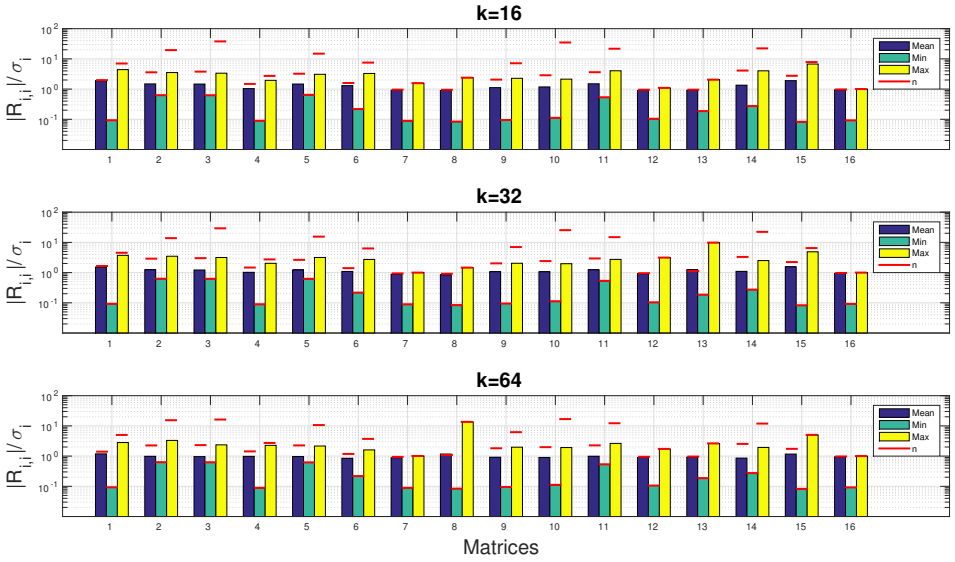
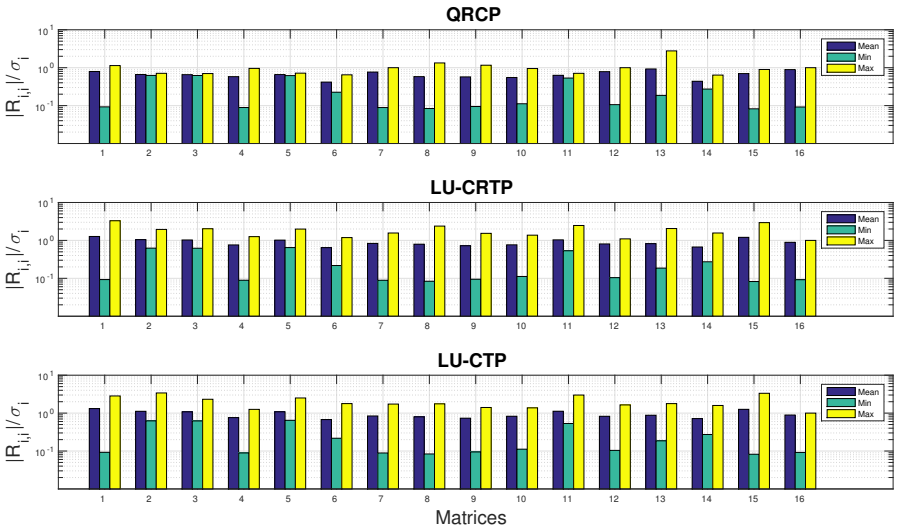


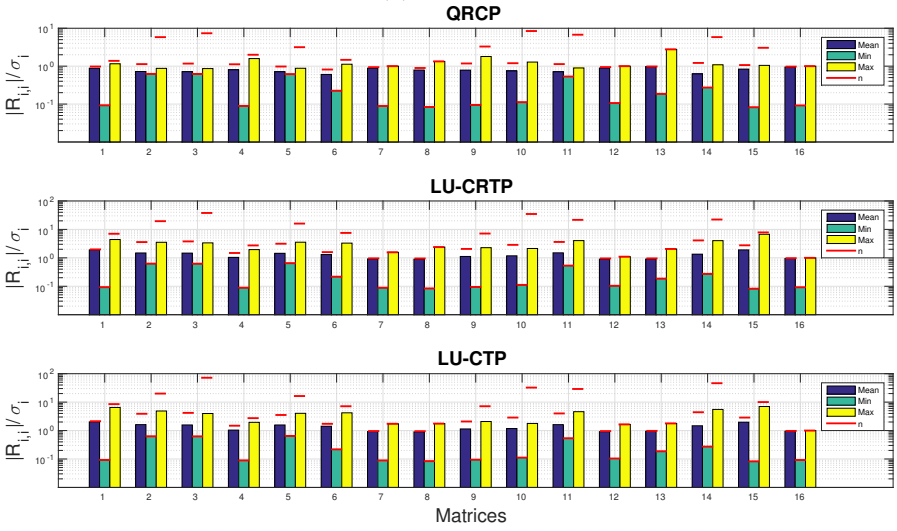
Figure 4.6 – Influence of the parameter k on the accuracy of the approximate singular values returned by LU-CRTP. Matrix size is 256 and $nSV \in \{128, 256\}$.

In Figure 4.5, when k increases, the ratios are similar, except for Spike matrix (id 13). However, when k is equal to the number of singular values requested, the ratios correspond to the results of QRCP applied on the k columns returned by QRTP. Therefore, the largest ratio

does not exceed one order of magnitude. In Figure 4.6, when the full spectrum is considered, the largest maximum ratios tend to decrease when k increases. This is explained by the fact that the number of update decreases when k increases. As discussed above, an update of the trailing matrix yields to an increase of the norm of few columns and so a small degradation of the ratio. Note that we observe few differences for matrix indices 8 and 13, which does not exceed a maximum ratio of 10. Hence, the parameter k does not impact the accuracy even if it slightly impacts the largest and smallest ratios.



(a) $nSV = 64$



(b) $nSV \in \{128, 256\}$.

Figure 4.7 – Comparison of LU-CTP with LU-CRTP, and QRCP is used as the reference. Matrix size is 256, $k = 16$.

LU-CTP, a variant of LU-CRTP with a reduced cost

We now consider LU-CTP, a modification of LU-CRTP, where only the columns of A are permuted. This variant saves the computation of calling QRTP on the rows of Q. We compare the accuracy of the singular values computed by LU-CTP with these computed by LU-CRTP. In addition, QRCP is used as the reference and the column panel size is set to 16. Figure 4.7a presents the average ratio, its maximum and minimum, of each matrix and for each method, with $nSV = 64$. The maximum and minimum ratios of LU-CTP are equivalent to the ones of LU-CRTP, except for matrix Break1 and Break9 (id 2 and 3). The maximum ratio of Break1 is larger for LU-CTP than for LU-CRTP whereas the maximum ratio of Break9 is smaller for LU-CTP than for LU-CRTP. When we consider $nSV = 128$ and $nSV = 256$, in Figure 4.7b, the maximum ratios of both LU-CTP and LU-CRTP are the same. Note that the matrix Break9, id=3, has a maximum ratio slightly larger for LU-CTP than for LU-CRTP. This variant approximates the singular values of a matrix A as well as LU-CRTP does, for the 16 challenging matrices presented above. To be more accurate, we have performed extensive tests on a larger set of matrices.

4.2.2 Extensive tests on the estimation of singular values

In order to have more accurate results about the error on the singular values returned by LU-CRTP, we have made larger tests on a bunch of 261 matrices coming from San Jose State University (Foster, 2017). Compared to the challenging matrices presented in Table 4.1, these matrices have less than 1024 rows and between 32 and 2048 columns, and are rank deficient. To get them from MATLAB, we execute

```
idx = SJget;
ids = find(idx.nrows <= 1024 &...
            idx.ncols <= 2048 & idx.ncols > 32);
[~, i] = sort(idx.numrank(ids));
ids = ids(i);
```

Since some applications require less than the real numerical rank like only half of the spectrum, tests are made on two criteria. The first one is based on the numerical rank of the matrices provided by the database. Therefore we approximate iteratively the singular values until reaching the numerical rank of A noted $rank(A)$. Figure 4.8 presents the maximum, minimum and average of the ratios for each matrix of the collection where the x-axis is the index i of the matrix (which means ids(i) in the MATLAB code). The top subgraph concerns QRCP whereas the middle one presents the ratios for LU-CRQRCP and the bottom one is for LU-CRTP. Note that the matrices are sorted by their numerical rank. So the first ratios are related to the matrix for which the numerical rank is the lowest and the last ones correspond to the highest rank. As said earlier, QRCP has a maximum of one order of magnitude and a minimum which does not drop under three orders of magnitude. LU-CRQRCP and LU-CRTP use a panel size of 16 columns. They both have roughly the same curves, except for few matrices. Thus for matrix 215, the maximum ratio goes from 32.45 for LU-CRQRCP to 48.29 for LU-CRTP, and for matrix 261, from 67.25 to 84.91. For matrix 253, the maximum ratio decreases from 80.37 to 56.12, and for matrix 258, from 64.44 to 30.62.

These two methods do not reach a maximum of two orders of magnitude and have the same lowest ratio as QRCP. Moreover we observe that when the matrix index increases the maximum ratio increases too. Again, this is due to the fact that when the numerical rank is much larger than the panel size, updating the trailing matrix using L_{21} introduces an update of the norm. As a consequence, the last matrices of the set, where the size is nearly 10^3, and rank(A) is close

to its size, are expected to have a worse maximum compared to the first ones when their rank is much smaller than their size. Therefore, the panel size has to be chosen by taking into account the number of updates of A_{22} as well as the requested rank. Few matrices like the matrix of index 80, the `heat_100`, have only one outlier value. Figure 4.9 shows the relation between the ratios (top subfigure) with its spectrum (bottom subfigure). From it, all ratios but the last one of LU-CRTP are lower than 10. Comparing with QRCP, the ratios are quite close and only the last ratio of LU-CRQRCP (37.92) and LU-CRTP (44.25) exceed the 10. Now focusing on the problem named `dwt_1007`, ids(261), we plot in Figure 4.10 the ratio evolution and the spectrum of each method. The spectrum returned by the SVD algorithm slowly decreases by a factor of 10^3 over 1000 values starting at 8.88. At first, all methods underestimate the real singular values. After roughly 200 values, LU-based methods overestimate them whereas QRCP starts to overestimating after 750 values. The LU-based methods totally cross the 10 limit after 900 values. Since the singular values of this matrix are locally quite close over 800 values, to approximate them appears to be difficult and even more at the end. Here we see the impact of the update of the trailing matrix coupled with the slow decrease of the singular values. All methods slightly overestimate the singular values and the LU-based methods are slightly worse than QRCP.

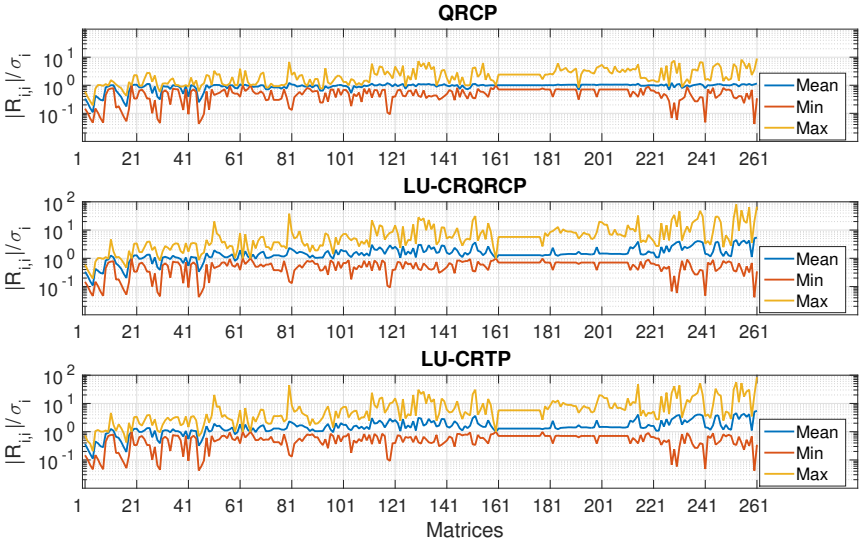


Figure 4.8 – Comparison of the ratios $|R_{i,i}|/\sigma_i$ with $1 \leq i \leq rank(A)$, where $rank(A)$ is the numerical rank, for QRCP (top), LU-CRQRCP (middle), and LU-CRTP (bottom) for a panel size of 16 on the 261 matrices. Each subgraph shows the minimum, average and maximum for each matrix where the x-axis is the index of the matrix in the subset.

Since the panel size could be too small for larger matrices, the set of matrices is divided into two. The first one is composed of the first 231 matrices whose rank is less than 500, and uses a panel size of 16, and the second one uses a panel size of 64. Figure 4.11 shows the new ratios computed with these two panel sizes. The vertical black dotted line explicitly splits the graph according to $rank(A) < 500$ at left and the remaining matrices at right. The maxima for the last matrices has drastically decreased from roughly 80 to 40. In Figure 4.12, we compare again the ratios with the spectrum of `dwt_1007` but with a panel size of 64. The largest ratio drops

from 84.9 to 45.3. Thus decreasing the number of updates has reduced the error induced by the factorization.

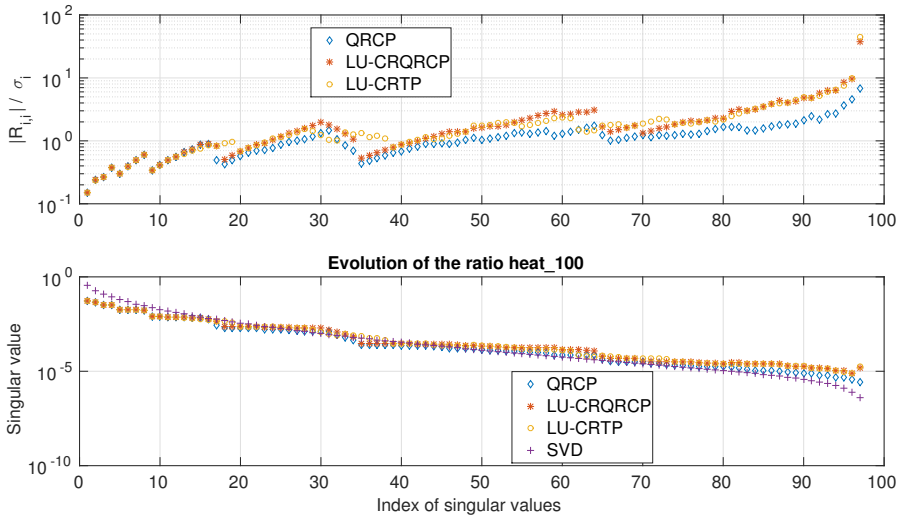


Figure 4.9 – Comparison of the spectrum and the ratio $|R_{i,i}|/\sigma_i$ for QRCP, LU-CRQRCP, and LU-CRTP for the matrix heat_100, ids(80) of size 100×100 and $rank(A) = 97$. The top subfigure plots the evolution of the ratio for each method where the ratio is $|R_{i,i}|/\sigma_i$ related to the bottom subfigure plotting the evolution of the singular values and their approximations.

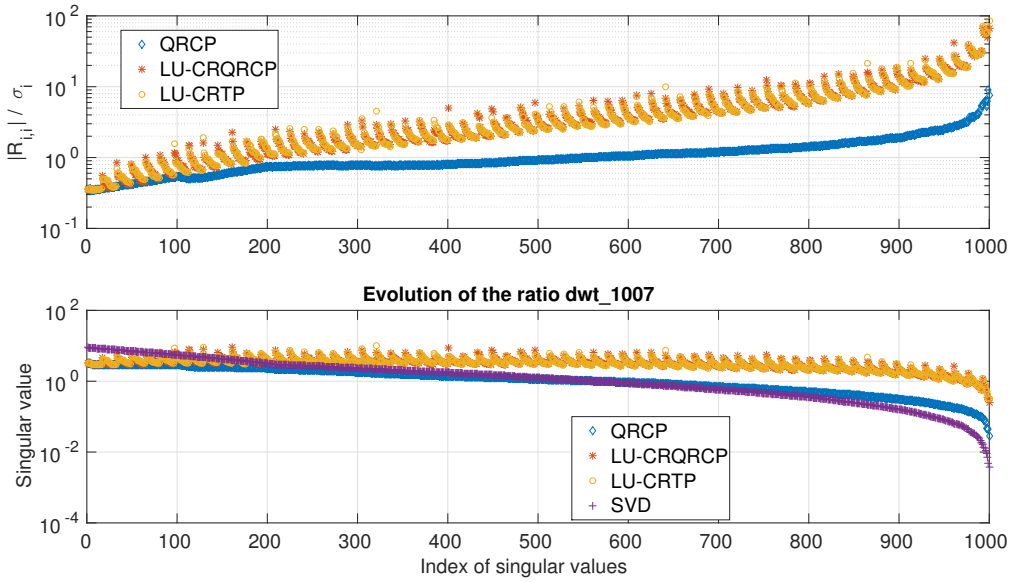


Figure 4.10 – Comparison of the spectrum and the ratio $|R_{i,i}|/\sigma_i$ for QRCP, LU-CRQRCP, and LU-CRTP for the matrix dwt_1007, ids(261) of size 1007×1007 and $rank(A) = 1000$ with a panel size of 16. The top subfigure plots the evolution of the ratio for each method where the ratio is $|R_{i,i}|/\sigma_i$ related to the bottom subfigure plotting the evolution of the singular values and their approximations.

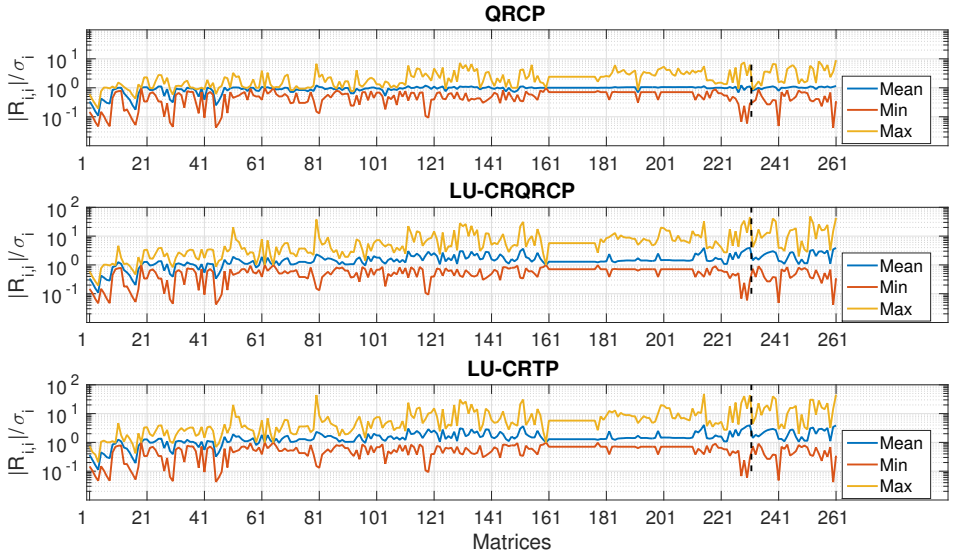


Figure 4.11 – Comparison of the ratios $|R_{i,i}|/\sigma_i$ with $1 \leq i \leq rank(A)$, where $rank(A)$ is the numerical rank, for QRCP (top), LU-CRQRCP (middle), and LU-CRTP (bottom) on 261 matrices. The vertical black dotted line splits the graph with at left the matrices having a numerical rank lower than 500, and a panel size of 16, and at right the remaining matrices with a panel size of 64. Each subgraph shows the minimum, average and maximum for each matrix where the x-axis is the index of the matrix in the subset.

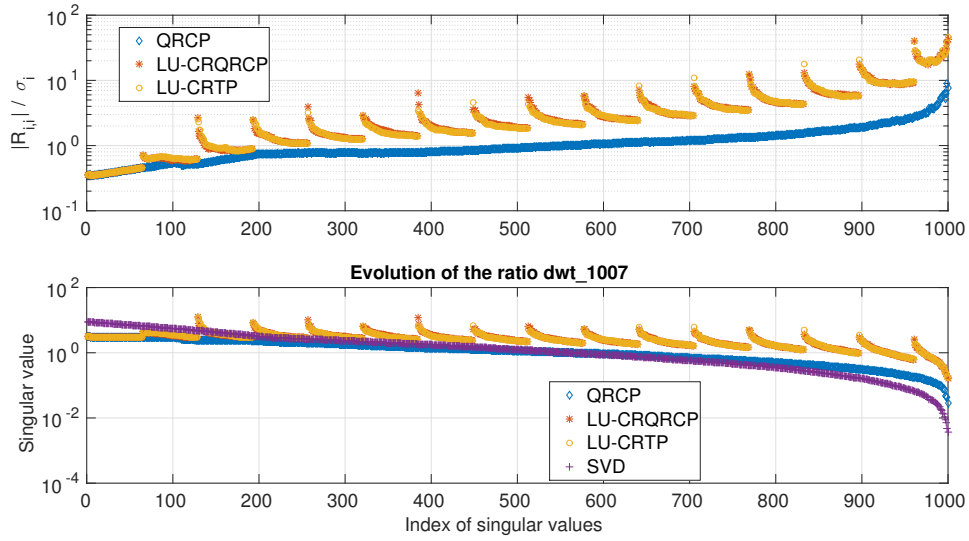


Figure 4.12 – Comparison of the spectrum and the ratio $|R_{i,i}|/\sigma_i$ for QRCP, LU-CRQRCP, and LU-CRTP for the matrix dwt_1007, ids(261) of size 1007×1007 and $rank(A) = 1000$ with a panel size of 64. The top subfigure plots the evolution of the ratio for each method where the ratio is $|R_{i,i}|/\sigma_i$ related to the bottom subfigure plotting the evolution of the singular values and their approximations.

Table 4.2 presents the largest of the maxima and the smallest of the minima observed in Figure 4.8 for each method with a panel size of 16. The first set of data concerns the full spectrum whereas the second one focuses on the rank which means the singular values index goes from 1 to rank(A). Thus we take into account the singular values until reaching the rank of each matrix. The smallest value obtained by matrix `Ursell_1000`, ids(260), is $4.169 * 10^{-2}$ for all methods. This smallest value and the largest of QRCP are equal to the data presented in (J. Demmel, Grigori, Gu, et al., 2013). The largest values of LU-CRQRCP and LU-CRTP, $8.037 * 10^{1}$ and $8.491 * 10^{1}$ respectively, are one order of magnitude worse than QRCP but do not reach two orders of magnitudes, in absolute value.

Spectrum	Method	min		max	
		matrix id	ratio	matrix id	ratio
Full spectrum	QRCP	106	3.011e-142	60	Inf
	LU_CRQRCP	229	4.250e-308	72	4.233e+02
	LU_CRTP	104	1.323e-263	95	1.209e+14
$1 \leq i \leq rank(A)$	QRCP	260	4.169e-02	261	8.957e+00
	LU_CRQRCP	260	4.169e-02	253	8.037e+01
	LU_CRTP	260	4.169e-02	261	8.491e+01

Table 4.2 – Smallest minimum and largest maximum of the ratio $|R_{i,i}|/\sigma_i$ of QRCP, LU-CRQRCP, and LU-CRTP for the full spectrum and for $1 \leq i \leq rank(A)$, where $rank(A)$ is the numerical rank provided by the database for the all 261 matrices.

Considering the two subsets of matrices, we compute the same ratios and display the smallest and largest values in Table 4.3. The first set uses a panel size of 16 and the second one of 64 as described above. For the first 231 matrices, the smallest value is slightly better than in the previous table, equal to $4.24 * 10^{-2}$. The largest ratio is 46 and 48.3 for LU-CRQRCP and LU-CRTP respectively. When the panel size is 64, the smallest ratio is equal to the one in the previous table and the largest decreases from 80.4 to 48.2 for LU-CRQRCP and from 84.9 to 45.3 for LU-CRTP. Moreover, for LU-CRTP, only 60 matrices exceed one order of magnitude.

k	Method	Min		Max	
		matrix id	ratio	matrix id	ratio
16	QRCP	45	4.24e-02	229	7.49e+00
	LU_CRQRCP	45	4.24e-02	231	4.60e+01
	LU_CRTP	45	4.24e-02	215	4.83e+01
64	QRCP	260	4.169e-02	261	8.96e+00
	LU_CRQRCP	260	4.169e-02	252	4.82e+01
	LU_CRTP	260	4.169e-02	261	4.53e+01

Table 4.3 – Smallest minimum and largest maximum of the ratio $|R_{i,i}|/\sigma_i$ of QRCP LU-CRQRCP and LU-CRTP for $1 \leq i \leq rank(A)$ where $rank(A)$ is the numerical rank provided by the database for all 261 matrices, divided into two groups. The first group is composed of matrices 1 to 231 with $k = 16$, and the second group is formed by matrices 232 to 261 with $k = 64$.

To conclude, LU-CRTP approximates the singular values of a matrix A with an error smaller than two orders of magnitude. LU-CRTP can be replaced by LU-CTP without loss of accuracy of the method. Both versions use the diagonal elements of R to approximate the singular values. This variant replaces computing the Singular Value Decomposition of R, with a smaller cost. For stability purpose, the update of the trailing matrix may use Q instead of A, depending on the matrix. Looking at the update of A_{22}, we have observed that $L_{21} = A_{21}A_{11}^{-1}$ may be unstable. If so, L_{21} is instead computed using $Q_{21}Q_{11}^{-1}$. Over these 261 matrices we have noticed that

95 matrices update the trailing matrices using Q instead. The list of these matrices is given in Table B.2 in the appendix. We next focus on the numerical stability of block LU factorization performed in LU-CRTP.

4.2.3 Numerical Stability

As mentioned above, during the extensive tests, we have noticed that the LU factorization may become unstable. For better understanding, we now focus on the numerical stability of the block LU factorization in LU-CRTP. To study it, we compute the growth factor, defined as $\frac{max_{i,j,k}|a_{i,j}^{(k)}|}{max_{i,j}|a_{i,j}|}$, where $a_{i,j}^{(k)}$ denotes the entry in position (i,j) after k iterations. In addition, we compute the 1_norm and the max norm of the factors L and U, the 1_norm of their inverse, the max norm of A, and the backward error of the block LU factorization using the Frobenius norm $\frac{||PAE-LU||_F}{||A||_F}$. The set of matrices, presented in Table 4.1, is replaced by a set of larger sparse matrices presented in Table 4.4 coming from the University of Florida sparse matrix collection (Davis and Hu, 2011). These matrices arise from different problems, where their dimension increases from a thousand to half million unknowns.

No.	Matrix	Size	nnz	Problem description
17	orani678	2 529	90 158	Economic
18	gemat11	4 929	33 108	Power network sequence
19	raefsky3	21 200	1 488 768	Computational fluid dynamics
20	wang3	26 064	177 168	Semiconductor device
21	onetone2	36 057	222 596	Frequency-domain circuit simulation
22	TSOPF_RS_b39_c30	60 098	1 079 986	Power network
23	RFdevice	74 104	365 580	Semiconductor device
24	ncvxqp3	75 000	499 964	Optimisation
25	mac_econ_fwd500	206 500	1 273 389	Economic
26	parabolic_fem	525 825	3 674 625	Computational fluid dynamics

Table 4.4 – Set of sparse matrices

For simplicity, we further denote the TSOPF_RS_b39_c30 problem as TSOPF_RS, and the mac_econ_fwd500 problem as fwd500. We consider the case where an approximation of the first K singular values of each problem is requested, and the panel size k varies (k is also the number of selected columns returned by the QR factorization with Tournament Pivoting). To do so, we call LU-CRTP (Algorithm 4.2) on each matrix of the set Table 4.4 with $K = 1024$, a tolerance $\tau = 0$, and the panel size k increasing from 16 to 128. All results are summarized in Table B.1, in the appendix. Wilkinson introduces the observation that a small growth factor leads to a more stable LU factorization. For all sparse matrices studied, the growth factor is equal to 1, while the backward error varies from 10^{-16} to 10^{-24}. Also, the value of k does not impact the stability of LU-CRTP. The block LU factorization of LU-CRTP is stable for this set of matrices. However, the extensive test has shown that a third of the 261 matrices switched the manner L_{21} is computed, replacing A by Q for better stability. The LU factorization is stable in general, and the different ways to compute L_{21} increase the stability.

4.2.4 Fill-in

One of the main reasons to develop LU-CRTP is the relation between the factorization and the fill-in induced. In the following, we compare the fill-in of LU-CRTP, LU-CTP with QRCP.

We also use LU with partial pivoting (denoted LUpp) as a reference to measure the impact of the reordering of our method relative to it. In order to apply it, we first call Approximate Minimum Degrees and Elimination tree algorithms to reduce the fill-in during the factorization. The MATLAB code is presented in Algorithm B.1, in the appendix. To compare the fill-in of each algorithm, we compare the sum of the number of non-zeros of the factors L and U, with the sum of the number of non-zeros of the Householder vectors and the number of non-zeros of the R factor. It means that the value presented in Table 4.5 for the methods LUpp, LU-CRTP and LU-CTP is equal to $nnz(L)+nnz(U)$, whereas, for QRCP, the value is equal to $nnz(Y)+nnz(R)$, where Y is the concatenation of the successive Householder vectors. All tests are made on the set of sparse matrices presented in Table 4.4, and the number of singular values varies from 128 to 1024. Table 4.5 regroups the results. For each matrix, the number of non-zeros of its first nSV columns is presented. This value corresponds to the number of non-zeros without fill-in. As expected in theory, LU-CRTP yields to a smaller fill-in than QRCP for any problem and any panel size, except for TSOPF_RS and a panel size greater than 128. We observe in particular that for the last two matrices, LU-CRTP has the smallest fill-in, even smaller than LU with partial pivoting. Comparing LU-CRTP with LU-CTP, their fill-in is close enough to say that these are similar in terms of fill-in.

Name	nSV	nnz	LUpp	QRCP	LU-CRTP	LU-CRTP
orani678	128	7 901	**8 457**	498 255	27 885	27 942
	512	55 711	**33 817**	1 851 058	299 325	295 004
	1024	71 762	**71 804**	3 843 133	790 850	827 864
gemat11	128	1 232	**1 450**	6 916	3 257	2 988
	512	4 895	**7 218**	60 622	18 559	17 903
	1024	9 583	**14 799**	549 778	47 687	49 805
raefsky3	128	7 872	**25 504**	71 783	57 584	53 472
	512	31 248	**153 312**	682 834	640 628	621 696
	1024	63 552	**254 960**	1 786 735	1 678 028	1 772 704
wang3	128	896	**4 638**	29 161	9 672	9 004
	512	3 536	**35 420**	213 916	72 546	57 474
	1024	7 120	**63 956**	230 201	79 291	64 642
onetone2	128	4 328	**1 724**	174 236	4 840	4 840
	512	9 700	**11 140**	863 686	11 748	11 748
	1024	17 150	66 530	2 108 099	**19 430**	21 246
TSOPF_RS	128	4 027	**3 328**	16 267	6 323	5 468
	512	5 563	**13 441**	27 109	32 456	29 876
	1024	7 695	**31 881**	41 545	68 018	67 338
RFdevice	128	633	**1 362**	14 323	1 430	1 523
	512	2 255	5 590	406 758	**4 924**	5 017
	1024	4 681	11 006	1 995 648	**9 631**	9 966
ncvxqp3	128	1 263	**2 081**	7 259	2 526	2 526
	512	5 067	**10 027**	35 464	10 138	10 138
	1024	10 137	36 639	73 921	**19 291**	20 278
fwd500	128	384	4 195	-	**1 408**	1 408
	512	1 535	29 448	-	**5 631**	**5 631**
	1024	5 970	115 254	-	**12 680**	14 162
parabolic_fem	128	896	3 778	-	**1 792**	**1 792**
	512	3 584	25 072	-	**7 168**	**7 168**
	1024	7 168	62 734	-	**13 952**	14 336

Table 4.5 – Comparison of fill-in for different matrices and different methods

To highlight the difference of fill-in, we compute two ratios. First, we consider the ratio of the fill-in induced by QRCP divided by the fill-in induced by LU-CRTP, noted $\frac{QRCP}{LU-CRTP}$. The ratio is at least of 1 but generally greater than 2, except for TSOPF_RS matrix. The ratio goes up to 200 for RFdevice matrix. As predicted, we observe that when the dimension of the matrix is too large, QRCP cannot be used. Actual implementations of QRCP such as the subroutine `dgeqp3` in LAPACK library requires the matrix A to be dense. In our case, the last matrices in Table 4.4 are too large to be stored in dense format (parabolic_fem would require 2.2TB of memory to be stored). The second ratio shows the impact of the perturbation of our reordering on the fill-in compared to the classical LU with partial pivoting. The ratio $\frac{LU-CRTP}{LUpp}$ shows that for small matrices, LUpp has the least fill-in for any number of singular values requested. When the size of the matrix becomes large, we observe that LU-CRTP has a smaller fill-in than LUpp, for any number of singular values required. In these cases, our algorithm leads to less fill-in than the LU with partial pivoting factorization. We assume that the permutation obtained through QRTP is better in term of fill-in than the classical combination of AMD plus elimination tree algorithms. To check this, a simple test is performed and its results are presented in Table 4.7. We apply the column permutation matrix P_c returned by LU-CRTP to A such that $A_p = AP_c$, and then we call LUpp on A_p. Results confirm that the column permutation vector impacts the fill-in of the LU with partial pivoting factorization. Finally, LU-CRTP and LU-CTP have a similar fill-in. Therefore, LU-CTP should be used as an initial method but when the LU factorization becomes unstable, LU-CRTP has to be used instead without increasing the fill-in.

Name	nSV	$\frac{QRCP}{LU-CRTP}$	$\frac{LU-CRTP}{LUpp}$
orani678	128	17.87	3.30
	512	6.18	8.85
	1024	4.86	11.01
gemat11	128	2.12	2.25
	512	3.27	2.57
	1024	11.53	3.22
raefsky3	128	1.25	2.26
	512	1.07	4.18
	1024	1.06	6.58
wang3	128	3.01	2.08
	512	2.95	2.05
	1024	2.90	1.24
onetone2	128	36	2.81
	512	73.51	1.05
	1024	108.5	**0.29**
TSOPF_RS	128	2.57	1.90
	512	0.83	2.41
	1024	0.61	2.13
RFdevice	128	10.02	1.05
	512	82.61	**0.88**
	1024	207.21	**0.88**
ncvxqp3	128	2.87	1.21
	512	3.50	1.01
	1024	3.83	**0.53**
fwd500	128	-	**0.34**
	512	-	**0.19**
	1024	-	**0.11**
parabolic_fem	128	-	**0.47**
	512	-	**0.29**
	1024	-	**0.22**

Table 4.6 – Comparison of fill-in of partial LU, QRCP, LU-CRTP and LU-CTP

Name	nSV	nnz(A(:, 1:nSV))	LUpp (A)	LUpp (A_p)
mac_econ_fwd500	128	384	4 195	1 408
	512	1 535	29 448	5 631
	1024	5 970	115 254	14 162
parabolic_fem	128	896	3 778	1 792
	512	3 584	25 072	7 168
	1024	7 168	62 734	14 336

Table 4.7 – Comparison of the fill-in induced by LU with Partial Pivoting applied on A with LU with Partial Pivoting applied on A_p, where A_p is the matrix A permuted with the column permutation matrix P_c obtained from LU-CRTP.

4.2.5 Parallel results using a 1D distribution of the matrix

In this section, we present the performance of our parallel implementation of the QR with Tournament Pivoting algorithm (QRTP). As detailed in Algorithm 4.1 and in its description, QRTP is one of the two costly steps of LU-CRTP (the second one being the update of the trailing matrix). We first present the parallel implementation and then the experimental results.

Parallel implementation of QRTP

Our implementation of QRTP is presented in Algorithm 4.3. This takes as input the matrix A, and k the number of columns to select from A. Since it is more convenient to have a 1D column distribution of the data, the input matrix of dimension $m \times n$ is split into p column panels of dimension $m \times n/p$. For the purpose of the test, we are not using a cyclic distribution, but for an optimal implementation, a 2D cyclic distribution may be considered. Also, we consider in our tests that the input matrix is permuted using COLAMD and Elimination tree.

Algorithm 4.3 QRTP (A, k): returns a column permutation matrix of A where the selected k columns are moved to the leading positions.

Input: $A \in \mathbb{R}^{m \times n}$,
k the number of columns moved to the leading position of A
Output: P_c the column permutation matrix

1: Let p be the number of processes
2: Let A_i be the column panel of size $m/p \times n$, owned by process i, where $i \in \{1, \ldots, p\}$
3: Select k columns from A_i without communication
4: Perform a global reduction on the selected k columns of A_i from the previous step, using a binary tree. At each level of the tree, k columns from the concatenation of two matrices of dimension $m \times k$ are selected until it remains only k columns.
5: Create the column permutation matrix P_c so that the k selected columns are moved to the leading position of A.

Algorithm 4.3 is divided into two steps, the local step and the global step. The local step corresponds to process i selecting k columns from A_i, with $i \in \{1, \ldots, p\}$. To do so, there are several approaches. The first idea is to apply the QR factorization with Column Pivoting on A_i so that the first k columns of $A_i\Pi_i = Q_iR_i$ are selected for the next step, where Π_i is the column permutation matrix. This approach needs the use of `dgeqp3` with the constraint that A_i is dense. Yet, in practice, it is not possible on large matrices to do it. The second approach consists in factoring A_i using a sparse algorithm (as SPQR from SuiteSparse (Davis, 2011b), or qr_mumps

(Buttari, 2013)) so that an intermediate $\tilde{R}_i = \tilde{Q}_i^\top A_i$ is computed, turned into a dense storage, and finally factored using the first approach $\tilde{R}_i \Pi_i = Q_i R_i$. This method presents the interest of reducing the memory usage that stores the dense matrix to factor from mn/p to $(n/p)^2$. On the other hand, this needs a sparse factorization of A_i which is costly, mainly due to the update of the entire trailing matrix. The last approach that we are using in our implementation of QRTP is derived from the previous approach that reduces the impact of the update. To do so, we split A_i into $n/(p \times 2k)$ column panels of size $m \times 2k$, and we perform a reduction tree on them, what leads to the k desired columns. Note that we assume for simplicity that n is a multiple of $2k$ and p, but our explanation can be easily adapted to a general n. Each column panel is processed by using a sparse QR factorization. The size of the trailing matrix is at most $2k - 1$. We call this third approach a local tournament pivoting that selects the k columns from A_i. This approach presents two main interests. First, the memory consumption is reduced to $2k \times 2k$ dense blocks, and second, its parallelism is expected to be better compared to the sparse factorization of A_i. Although the global reduction involves communication, the local tournament corresponds to no communication between processes.

The global reduction based on a binary tree can be detailed as follow. The reduction starts with the k selected columns from each A_i. At each level of the tree, k selected columns are sent from one process to another. Then, the sparse QR factorization of a local matrix of dimension $m \times 2k$ leads to creating a dense triangular factor R. This is factored using `dgeqp3` so that the first k columns of $R\Pi$ are selected for the next level. The cost of one level of the tree can be estimated as $O(Send)+O(sparse\_qr)+O(QRCP)$, where $O(Send) = \alpha+\beta*nnz((R\Pi)(:, 1:k))$ is the communication cost with α the latency of the network and β its bandwidth, $O(sparse\_qr)$ is the sparse QR factorization cost, and $O(QRCP)$ is the cost to select the k columns. The non-constant terms in this estimation are the number of values to send, and the cost of the sparse QR factorization. As we will further this last term can greatly impact the global performance of QRTP. When a good load balancing is achieved, we can consider the global cost as roughly $log(p)$ times the cost of one level of the tree.

Our environment of compilation is as follow :

Communication	Intel MPI library	5.0.2.044
Sparse operation	Sparse aspect handled by CPaLAMeM	0.1
Sparse operation	Metis	4.0
	Sparse kernel from SuiteSparse	4.4.6
Dense operation	MKL from Intel Composer XE	2015.1.133
	LAPACK	3.6.0

Parallel results of QRTP

In the following, we study the scalability of QRTP, and give more details on the inner steps of the algorithm. The parallel tests are made on a supercomputer named Edison, managed by NERSC. This machine has 5586 nodes. Each node is equipped with two 12-core Intel "Ivy Bridge" processors at 2.4 GHz, and has 64 GB DDR3 1866 MHz of memory. The tests are made on a set of larger sparse matrices coming from the Sparse Matrix Collection (Davis and Hu, 2011), and presented in Table 4.8. We first study the strong scalability of QRTP by increasing the number of MPI processes from 32 to 2048. We measure the execution time to select 256 columns from each matrix, and present the results in Table 4.9. As expected, the results show that the runtime decreases when the number of MPI processes increases. This is due to the reduction of the number of columns of A_i, the local part of A. Since the number of MPI processes is doubled, the number of columns is divided by a factor 2. The amount of computation during the local

tournament is also expected to be reduced by 2. On the other hand, the global reduction tree has one more level. Therefore, the variation of the cost between p and $2p$ MPI processes is composed of the reduction of the local work, and the extra level in the reduction tree. However, in Table 4.9 we remark that the last value of rajat31 for 2048 processors is greater than for 1024. This is due to the amount of work to perform the sparse QR factorization. The number of FLOPs returned by SPQR for process 2041 is $1e10$, whereas this value varies from $1e7$ to $1e9$ for the others. The factorization is strongly related to the pattern of the matrix to factor, and greatly impacts the runtime in that case.

No	Name	Size	nnz	problem
25	mac_econ_fwd500	206 500	1 273 389	economic problem
26	parabolic_fem	525 825	3 674 625	fluid dynamics
27	atmosmodd	1 270 432	8 814 880	fluid dynamics
28	circuit5M_dc	3 523 317	14 865 409	circuit simulation
29	rajat31	4 690 002	20 316 253	circuit simulation

Table 4.8 – Set of larger sparse matrices coming from SuiteSparse matrix collection.

Matrix	#MPI process						
	32	64	128	256	512	1024	2048
mac_econ_fwd500	367	183	118	83	57	19	12
parabolic_fem	106	65	36	22	15	5	4
atmosmodd	488	269	163	83	52	29	18
circuit5M_dc	771	367	196	109	69	44	37
rajat31	-	802	489	296	210	172	175

Table 4.9 – Execution time in seconds of QRTP with $k = 256$ (SPQR + DGEQP3 and binary tree on leaf + METIS ordering and preordered by COLAMD).

Matrix Name	COLAMD + etree	Panel of 2k columns		Local sum on leaf		Local sum on node	
		SPQR	DGEQP3	SPQR	DGEQP3	SPQR	DGEQP3
mac_econ_fwd500	0.52	0.06	0.01	180.97	0.33	0.00	0.00
		3.26	**0.02**	**246.65**	**0.36**	**0.41**	**0.01**
		22.71	0.02	361.84	0.39	3.37	0.08
parabolic_fem	0.56	0.20	0.01	54.66	0.94	0.00	0.00
		0.24	**0.02**	**79.88**	**1.03**	**0.20**	**0.01**
		0.29	0.07	103.49	1.13	1.06	.0.07
atmosmodd	3.35	0.48	0.01	183.36	2.17	0.00	0.00
		0.92	**0.02**	**236.69**	**2.39**	**0.49**	**0.01**
		4.96	0.02	483.61	2.63	2.50	0.05
circuit5M_dc	4.18	1.36	0.01	623.09	6.26	0.00	0.00
		1.59	**0.02**	**664.68**	**6.85**	**1.38**	**0.02**
		1.84	0.03	749.20	7.99	7.15	0.08

Table 4.10 – Execution time in seconds of main steps of the algorithm for 32 MPI processes with k=256, SPQR + DGEQP3 and binary tree on leaf + metis ordering and preordered by colamd+etree

We now study in details the kernels used in QRTP and consider for the test 32 processors only. Table 4.10 presents the minimum, average and maximum time to factor a column panel of size $m \times 2k$. It also displays the cumulative time spent in the local part, denoted the leaf of the tree, and in the global part, denoted the node of the tree. The average time is displayed in bold.

The table also shows the time spent to reorder the matrix (COLAMD + elimination tree). We first observe that the call to `dgeqp3` on a triangular factor of size $2k \times 2k$ is negligible compared to the call to SPQR. Note that the runtimes can slightly vary due to machine usage (the runtime of `dgeqp3` varies from 0.1 to 0.7 for parabolic_fem). In the leaf step, the computation of the sparse QR factorization is at least 100 times slower than for QRCP. Moreover, the time spent in the node part is negligible compared to the time spent in the leaf part. This explains the observation made on the strong scalability. The runtime of QRTP corresponds mainly to the time spent in the sparse QR factorization during the selection of the k column of each A_i.

dgekqp3: a modified version of dgeqp3

To select the k columns from each A_i, we have detailed several approaches. One option is to call SPQR on A_i and then `dgeqp3` on the triangular factor. Suppose the dense triangular factor can fit in memory, and $k \ll n/p$. However, the entire factorization is not requested, and a large part of the computation could be avoided. We propose an alternative that is a truncated QR factorization with Column Pivoting, denoted `dgekqp3`. To do so, we modify the source code in LAPACK by adding a break in the outer loop to stop after computing $i * nb \geq k$ Householder vectors. Thus, the column permutation matrix has the k selected columns moved to the leading position. To evaluate the impact on the performance, we compare `dgekqp3` with `dgeqp3` from MKL. We take as example parabolic_fem with 32 processors and present the runtimes in Table 4.11. `dgeqp3` factors A_i in 500 seconds, compared to `dgekqp3` which needs less than 100 seconds. This modification reduces the runtime by a factor 5.

Step	`dgekqp3`	`dgeqp3`
	91.74	413.29
Leaf	**94.60**	**526.23**
	101.26	629.93

Table 4.11 – Comparison of execution times between LAPACK and MKL. Data is parabolic_fem matrix on 32 processors using a full computation on the leaves.

4.3 Conclusion

In this chapter, we have presented the LU factorization with Column Row Tournament Pivoting, denoted LU-CRTP, a new method that computes a sparse low-rank approximation of a matrix A, and returns an approximation of its singular values. Based on a truncated LU factorization, this method proceeds by block and selects k columns at each iteration by using the QR factorization with Tournament Pivoting (QRTP). These k columns are factored and used to update the trailing matrix until the condition on the low-rank approximation is satisfied. By condition, we mean that either the rank K of A or a tolerance on the low-rank is requested. To validate the method, we have done extensive tests that reveal the stability of LU-CRTP. First, we have compared LU-CRTP with the QR factorization with Column Pivoting (QRCP) and the Singular Value Decomposition (SVD). The singular values returned by LU-CRTP are close to those of QRCP, and never reach two orders of magnitude compared to the SVD. Experimental results have shown that the factors L and U are up to 200 times sparser than the factors Q and R of QRCP. Our parallel implementation of QRTP have shown a good scalability of QRTP, tested up to 2048 cores. The performance of QRTP is strongly related to the performance of the sparse QR factorization. Therefore, having an efficient truncated sparse QR factorization with Column Pivoting would give much better performance of QRTP, and so on LU-CRTP.

Chapter 5

Tournament pivoting based on τ_rank revealing for the low-rank approximation of sparse and dense matrices

Outline of the current chapter

In the last chapter, we were interested in computing the rank-k approximation of a matrix $A \in \mathbb{R}^{m \times n}$. This type of problem is named fixed-rank problem. Along with it, the fixed-precision problem exists. It aims to compute a low-rank approximation of a matrix A such that its approximation is less than a tolerance. For example, in the case of rank revealing factorizations, the tolerance is set to the precision machine, *i.e.*, ϵ. Revealing the rank of A can be performed by using the Singular Value Decomposition of A. However, the computation of the Singular Value Decomposition of A is known to be costly both in terms of memory consumption and computation cost. Cheaper approaches are used to estimate the rank of A, and to approximate the singular values of A. The randomized algorithms aim to reduce the size of the problem by a projection of A onto an orthogonal basis, formed by random columns. The smaller matrix can then be decomposed using the Singular Value Decomposition. Alternatively,

the Communication Avoiding Rank Revealing QR factorization (CARRQR), presented in (J. Demmel, Grigori, Gu, et al., 2013), approximates well the singular values of A, and addresses at the same time the problem of communication encountered with parallel architectures. Yet, the factors of the QR factorization of A are expected to be dense, and even denser than the factors of the LU decomposition. Therefore, the LU factorization with Column Row Tournament Pivoting, presented in Chapter 4 and introduced in (Grigori, Cayrols, et al., 2018), is a good alternative to CARRQR. Both algorithms are block algorithms where k pivots are chosen such that the singular values of A are well approximated. The selection of the pivots is performed by the QR factorization with Tournament Pivoting, a divide and conquer algorithm. The costly part of both algorithms is thus the update of the trailing matrix, as in block algorithms, and the selection of the pivots at each iteration.

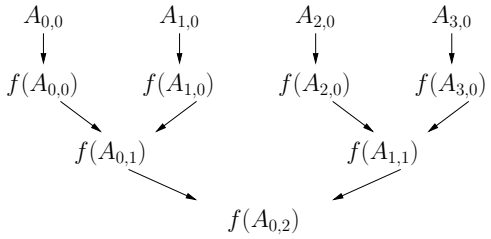
In this chapter, we focus on fixed-precision problems. We propose to detect the subset of k columns of A, denoted A_1, whose smallest singular value is close to the k-th singular value of A. These columns do not bring new information and should not be taken into account. We thus propose to modify the QR factorization with Tournament Pivoting so that these columns are detected. We will show that our modification has a small overhead ($O(k)$ FLOPS), where k is a small integer, generally smaller than 100. In addition, we propose a modification of CARRQR and LU-CRTP to save computation. The organization of the chapter is the following. After the introduction of some notations, we present how to use the properties of the QR factorization with Column Pivoting to discard some columns of A, and the integration in the QR factorization with Tournament Pivoting. In Section 5.4, we present the theoretical bounds which validate the action of discarding columns of A. In Section 5.5, experimental results will show the accuracy of the modified LU-CRTP and CARRQR to approximate the singular values of A. Moreover, the results present the gain of using the modified version of the tournament. Finally, we choose several different tolerance values and compare the obtained low-rank approximation with the theoretical bounds.

5.1 Notations

Given a matrix $A \in \mathbb{R}^{m\times n}$, $a_{i,j}$ represents the element of A at row i and column j, with $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$, and nnz(A) represents the number of nonzeros of the matrix A. Using MATLAB notation, $A(:, j)$ and $A(i, :)$ represent the j-th column and i-th row of A, respectively. Given a vector e of size n, $e(i)$ is the i-th entry of the vector. The concatenation of two vectors e_1 and e_2 of size n_1 and n_2, respectively, gives another vector $e = [e_1; e_2]$ of size $n_1 + n_2$. The notation $A(e_1, e_2)$ represents the matrix obtained after the permutation of the rows of A using e_1 and the permutation of the columns of A using e_2. If the size of a permutation vector is smaller than the size of the matrix to permute, then the vector also extracts the corresponding rows or columns to permute. The max norm is defined as $\|A\|_{max} = \max_{i,j} |a_{i,j}|$. We refer to the 2-norm of the j-th row of A as $\rho_j(A)$, the 2-norm of the j-th column of A as $\chi_j(A)$, and the 2-norm of the j-th row of A^{-1} as $\omega_j(A)$.

A k-ary tree is a rooted tree where each node has at most k children. A reduction tree aims at applying a reduction operation from the leaves to the root of the tree. A reduction operation generally reduces the size of a set of data. For example, the computation of the sum of all elements in an array of size n can be performed by using a reduction tree. The array is split into p chunks of size n/p. Let T be a k-ary tree, and `sum` be a reduction operation. Each leaf of T is set up with a chunk of the array, and the reduction operation is applied on the elements of the chunk to obtain the local sum. Each node stores the local sum of its children and then, applies the reduction operation. This reduction tree ends when the root computes the last local sum,

which is the sum of all elements in the original array. Therefore, the reduction tree performs $n-1$ operations in $log_k(p)$ steps. Throughout the chapter, we assume for simplicity that the number of columns of A is a multiple of k, and we limit our study to the special case of binary reduction tree. In that context, the matrix A is split into $n/(2k)$ subsets of columns so that $A = [A_{0,0}, A_{1,0}, A_{2,0}, \ldots, A_{n/(2k)-1,0}]$, and $A_{i,0} \in \mathbb{R}^{m \times 2k}$ with $0 \leq i < n/(2k)$. The second subscript 0 refers to the level in the reduction tree. The picture below is taken from (Grigori, Cayrols, et al., 2018), and represents the binary reduction tree of a matrix A which is partitioned into 4 subsets, and where a reduction operation f is applied on each subset and then on each node of the tree.



5.2 Reducing the cost of tournament pivoting

In (J. Demmel, Grigori, Gu, et al., 2013), the authors present the QR factorization with Tournament Pivoting algorithm, denoted QRTP. Using a reduction tree, the algorithm selects the k linearly independent columns of a matrix $A \in \mathbb{R}^{m \times n}$, and returns a column permutation matrix Π that permutes the selected columns to the leading positions of A. The algorithm proceeds as follows. The input matrix A is split into $n/(2k)$ subsets of columns. From each subset, the algorithm selects k columns by using (strong) Rank Revealing QR factorization. At each node of the tree, the selected k columns of the children of the node are adjoint next to each other. Then, the algorithm selects again k columns from the formed matrix by using (strong) Rank Revealing QR factorization. At the root of the tree, the selected k columns are permuted to the leading positions of A. As shown in Chapter 4 and in (J. Demmel, Grigori, and Cayrols, 2016), this algorithm is used as a subroutine in LU-CRTP to find the k columns that approximate well the first k singular values of A, but also to get a better stability of the truncated LU factorization. QR factorization with Tournament Pivoting is also used in the Communication Avoiding variant of the Rank Revealing QR factorization, denoted CARRQR, introduced in (J. Demmel, Grigori, Gu, et al., 2013). CARRQR algorithm calls the QR factorization with Tournament Pivoting on a matrix A, and permutes the selected k columns to the leading positions of A. Then, it factors the first k columns of the permuted A by using the QR factorization. If the rank is not revealed, the trailing matrix is updated as in block QR algorithm (G. H. Golub et al., 2013), and the updated trailing matrix is used as input for the next iteration. In both CARRQR and LU-CRTP algorithms, the runtime is mainly dominated by the call to QRTP algorithm and by the update of the trailing matrix.

For the purpose of the chapter, we introduce the notion of a τ_rank matrix, and present the implication for the columns of a τ_rank matrix A.

Definition 27. *A matrix $A \in \mathbb{R}^{m \times n}$ is a τ_rank matrix if*

$$\sigma_k(A) > \mu \geq \sigma_{k+1}(A), \tag{5.1}$$

where $\mu = \tau \|A\|_2$, and $1 \leq k < n$.

Hence, the τ_rank of A is equal to k, also noted $\tau\_rank(A) = k$.

We want to partition A into two subsets of columns so that the smallest singular value of the first subset A_1 formed by k columns of A is close to σ_k of A. As a consequence, we refer to the remaining columns of A, denoted A_2, as the discarded columns of A with respect to our τ_rank criterion. Note that, considering $\tau = 0$, the matrix A is rank deficient with a rank k if $\sigma_k(A) > 0 = \sigma_{k+1}(A)$. In the following, we consider a τ_rank matrix (or a rank deficient matrix) $A \in \mathbb{R}^{m\times n}$, and we focus on how to detect the subset A_1 of columns of A.

5.2.1 Formulation with the QR factorization with Column Pivoting

Given a matrix $A \in \mathbb{R}^{m\times n}$, with $m \geq n$, we consider the QR factorization with Column Pivoting of the form

$$A\Pi = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{5.2}$$

where $Q \in \mathbb{R}^{m\times m}$ is orthogonal, $R_{11} \in \mathbb{R}^{k\times k}$ is an upper triangular matrix, $R_{12} \in \mathbb{R}^{k\times(n-k)}$, $R_{22} \in \mathbb{R}^{(m-k)\times(n-k)}$, and Π is a column permutation matrix chosen such that it reveals the linear dependent columns of A (Gu and Eisenstat, 1996). Note that this factorization is considered as partial if R_{22} is not upper triangular. However, the factorization (5.2) is named the Rank Revealing QR factorization, denoted RRQR, if it verifies

$$\sigma_{min}(R_{11}) \geq \frac{\sigma_k(A)}{p(k,n)} \quad and \quad \sigma_{max}(R_{22}) \leq \sigma_{k+1}(A)p(k,n), \tag{5.3}$$

where $p(k,n)$ is a low-degree polynomial in k and n.

The Householder QR factorization of a matrix $A \in \mathbb{R}^{m\times n}$, presented in (G. H. Golub et al., 2013), proceeds as follow. Considering one iteration of the algorithm, it computes $PA(:,1) = \mp\|A(:,1)\|_2 e_1$, and then updates the trailing matrix $A(:,2:n)$, where $P = (I - \beta vv^\top)$ is a Householder reflection, with $\beta = \frac{2}{v^\top v}$, and v is the Householder vector. Thus, the k-th diagonal element of R in Equation (5.2) is equal to the norm of the first column of the k-th trailing matrix $A\Pi(k:m, k:n)$. The column permutation matrix Π returned by the QR factorization with Column Pivoting is built such that the column with the largest norm is moved to the leading position of the trailing matrix, and so the diagonal elements of R in absolute value in Equation (5.2) are decreasing. Let $A \in \mathbb{R}^{m\times n}$ be a rank deficient matrix, factored by using the QR factorization with Column Pivoting, as in Equation (5.2). If $r_{k,k} > 0 = r_{k+1,k+1}$, with $1 \leq k < n$, then the $(k+1)$-th corresponding column of $A\Pi$ is a linear combination of the first k columns of $A\Pi$. Furthermore, the $n-k$ last columns of $A\Pi$ are linearly dependent to the first k columns of $A\Pi$. Therefore, $\sigma_{max}(R_{22}) = 0$ and the rank of A is equal to k.

Now, let $A \in \mathbb{R}^{m\times n}$ be a τ_rank matrix as in Definition 27, factored by using the QR factorization with Column Pivoting, as in Equation (5.2).

Lemma 28. *Suppose a threshold μ and a matrix $A \in \mathbb{R}^{m\times n}$ factored using QR factorization with Column Pivoting as follows*

$$A\Pi = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{5.4}$$

where $R_{11} \in \mathbb{R}^{k\times k}$ and $R_{22} \in \mathbb{R}^{(n-k)\times(n-k)}$. If we have

$$\sigma_{min}(R_{11}) \geq \mu \quad and \quad \sigma_{max}(R_{22}) \leq \mu, \tag{5.5}$$

then A has a τ_rank equal to k.

Proof. Using the interlacing property of the singular values (G. H. Golub et al., 2013), the QRCP factorization of A satisfies

$$\sigma_i(R_{11}) \leq \sigma_i(A), \quad \sigma_{k+j}(A) \leq \sigma_j(R_{22}), \tag{5.6}$$

for $1 \leq i \leq k$ and $1 \leq j \leq n-k$. Especially, if $i = k$ and $j = 1$, we have

$$\sigma_{min}(R_{11}) \leq \sigma_k(A), \quad \sigma_{k+1}(A) \leq \sigma_{max}(R_{22}). \tag{5.7}$$

If there exists a threshold μ such that $\mu < \sigma_{min}(R_{11})$ and $\sigma_{max}(R_{22}) \leq \mu$, then A has a τ_rank equal to k.

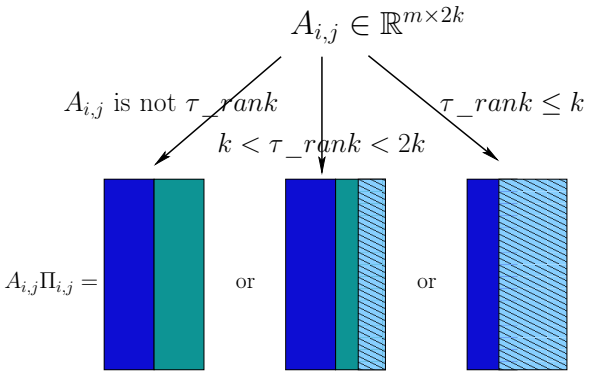
□

It follows that $A\Pi$, from Equation (5.4), can be split into two panels $A_1 = Q \begin{bmatrix} R_{11} \end{bmatrix}$, $A_1 \in \mathbb{R}^{m \times k}$ and $A_2 = Q \begin{bmatrix} R_{12} \\ R_{22} \end{bmatrix}$, $A_2 \in \mathbb{R}^{m \times n-k}$. Therefore, the QR factorization with Column Pivoting can be used to detect the columns of A that are of interest for our purpose. In (G. Golub et al., 1976), Golub *et al.* compare the Singular Value Decomposition with the QR factorization with Column Pivoting. Both approaches are able to extract independent columns of a matrix $A \in \mathbb{R}^{m \times n}$. Also, the diagonal elements of the R factor in Equation (5.4) can be used as a cheaper approach to detect the independent columns of A. This observation allows us to replace the Singular Value Decomposition of the R factor by its QR factorization with Column Pivoting. Although, our theory is based on the strong Rank Revealing QR factorization, in practice, we use the QR factorization with Column Pivoting.

5.3 Application to the QR factorization with Tournament Pivoting

For large matrices, both the Singular Value Decomposition and the QR factorization with Column Pivoting become costly to select b linearly independent columns from A. Instead, the QR factorization with Tournament Pivoting (QRTP), designed to identify b pivots, i.e., b linearly independent columns, is communication optimal and can be easily parallelized. In QRTP algorithm, if the input matrix $A \in \mathbb{R}^{m \times n}$ is rank deficient with a rank k, the last $n-k$ columns of $A\Pi$ that are linearly dependent to the first k columns of $A\Pi$ are kept in the procedure. They are updated during the update of the trailing matrix, and reused in the next call of the tournament pivoting, in both LU-CRTP and CARRQR. In case of fixed precision problems, we consider a matrix A as a τ_rank matrix in the sense of Definition 27. By fixed precision, we mean that the error of the approximation is smaller than a fixed value. The fixed precision is usually a relative tolerance that we refer hereafter to as τ. Suppose its actual rank is k, the number of columns that are of interest is smaller than k. The detection of these columns can save computation if the cost to detect them is smaller than the gain obtained from removing them during the computation of the low-rank approximation of A. We propose in Algorithm 5.1 a modification of the QR factorization with Tournament Pivoting algorithm which identifies columns of A to discard with respect to our τ_rank criterion. At each node of the tree, the modified algorithm splits the columns of the node such that the interesting columns are kept in the procedure whereas the remaining columns of the node are discarded from the procedure. We further show that the modification proposed here does not lead to significant overhead in terms of computation

whereas it can drastically reduce the cost of the QR factorization with Tournament Pivoting, and the cost of the update of the trailing matrix. Moreover, this modification does not involve extra memory consumption.



($\mathcal{S}$) Selected columns for the next level
($\mathcal{K}$) Columns kept
($\mathcal{R}$) Discarded columns of $A_{i,j}$ with respect to our τ_rank criterion

Figure 5.1 – Splitting of the columns of $A_{i,j} \in \mathbb{R}^{m \times 2k}$ into at most 3 subsets, where $A_{i,j}$ is the i-th node of the j-th level of the tree. The first subset, in blue, contains the selected columns of $A_{i,j}$ for the next level of the tree. The second subset, in cyan, contains the remaining columns of $A_{i,j}$ that are of interest. The remaining columns, in hashed light blue, are discarded with respect to our τ_rank criterion.

Our modification of QRTP algorithm corresponds to post-processing $R_{i,j} = Q_{i,j}^{\top} A_{i,j}\Pi_{i,j}$, that is obtained by the QR factorization with Column Pivoting of $A_{i,j}$ in the reduction tree, that is the i-th node of the j-th level of the tree. We determine whether $A_{i,j}$ is a τ_rank matrix as in Definition 27, by using the singular values of $R_{i,j}$. The columns of $A_{i,j}$ are split as in the original algorithm, that is the first k columns of $A_{i,j}\Pi_{i,j}$ are passed to the next level of the tree. Otherwise, the returned $\tau\_rank = r$ of $A_{i,j}$ is used to split the columns of $A_{i,j}\Pi_{i,j}$ into two subsets. The first subset contains the r columns that are of interest. The second subset contains the $(2k - r)$ remaining columns of $A_{i,j}\Pi_{i,j}$, that are discarded, with respect to our τ_rank criterion. Only the first subset of columns is of interest for the purpose of the algorithm, and the second subset should be discarded. Thus, removing the columns of the second subset saves computation. At each level of the reduction tree, the original algorithm selects k linearly independent columns from $A_{i,j}$ to be used in the next level of the tree. When r is larger than k, the first subset is again split into two parts. The first part contains the first k columns that are the most linearly independent columns of $A_{i,j}$. The second part contains the remaining $r - k$ columns. However, if the τ_rank is smaller than k, our modification of the original algorithm selects only the first r columns of $A_{i,j}\Pi_{i,j}$ that are then used in the next level of the tree. Therefore, the columns of $A_{i,j}$ are split at most into 3 subsets. All of these cases are summarized in Figure 5.1. Note that, in practice, the (strong) Rank Revealing QR factorization is replaced by the QRCP algorithm as in the QRTP algorithm presented in (J. W. Demmel, Grigori, Gu, et al., 2015). The experimental results will show that the usage of QRCP is sufficient for the purpose of this chapter.

The QR factorization with Column Pivoting can fail to isolate independent columns when the Singular Value Decomposition does. However, as discussed in (G. Golub et al., 1976), this case is an unusual phenomenon. Hence, we choose to use the diagonal elements of $R_{i,j}$ to split the columns of $A_{i,j}$. This choice reduces the cost of the modification, even if the cost to compute the Singular Value Decomposition of $R_{i,j}$ could be negligible, with respect to the size of $R_{i,j}$ (which is expected to be small). This modification of QRTP algorithm involves $\mathcal{O}(k)$ extra Flops for both the test and the splitting. Algorithm 5.1 presents QRTP_reduction, our modified version of QRTP, where some columns of a matrix $A \in \mathbb{R}^{m \times n}$ are discarded, with respect to our τ_rank criterion, at each node of the tree are removed. We consider, for simplicity, that the number of columns of A is a multiple of k. The algorithm takes as input a matrix $A \in \mathbb{R}^{m \times n}$, a panel size k, which is also the upper bound on the number of columns to select at each node, and a tolerance τ, usually smaller than 1. Since a column of A is not compared to all the others, the modification of the QR factorization with Tournament Pivoting fails to reveal the exact τ_rank of A. Hence, the algorithm returns an overestimation of the τ_rank of A, and $perm$ the column permutation vector that moves at most k columns of A to its leading positions. Compared to the original algorithm, we introduce two modifications, the first is located from lines 11 to 15, and the second is located at line 26. The first modification aims to determine the τ_rank of $A_{i,j}$ so that columns are split in at most 3 subsets. The second modification changes the construction of the column permutation vector of A. Thus, to select at most k columns of A, the matrix is split into $n/(2k)$ subsets of columns. Each submatrix $A_{i,0}$ has a size $m \times 2k$. The column index array, denoted `colInd`, stores the indices of the columns of A which are either selected from the previous level or all columns of A, that is $\{1, \ldots, n\}$. At level j of the tree, the QR factorization with Column Pivoting is applied on $A_{i,j} = A(:, panelInd)$, where $panelInd$ is the i-th subset of $2k$ indices of `colInd`. When $j \geq 1$, `colInd` is the concatenation of the selected columns from the level $j-1$ of the tree. The factorization of $A_{i,j}$ computes $A_{i,j}(:, e) = Q_{i,j}R_{i,j}$, and returns the upper triangular factor $R_{i,j}$ of size $2k \times 2k$, and the vector e of size $2k$. e is a column permutation vector so that $\Pi_{i,j} = \begin{bmatrix} I_{2k \times 2k}(:, e) \\ 0 \end{bmatrix}$ is the column permutation matrix. Hence, the array $panelInd$ is permuted with e. To split the columns of $A_{i,j}$, we apply the threshold $\mu = \tau \|A\|_2$ on the diagonal elements of $R_{i,j}$ (line 11) to get the τ_rank of $A_{i,j}$ (line 11). As discussed above, the columns of $A_{i,j}$ are therefore split at most into 3 subsets. The first subset is added into $\mathcal{S}$ that contains the indices of the columns that are selected for the next level. The second subset is added into $\mathcal{K}$ that records the indices of the columns which are potentially interesting, and not already in the first subset. The last subset is added into $\mathcal{R}$ that saves the indices of the remaining columns of A that are neither in $\mathcal{S}$ nor in $\mathcal{K}$. If the τ_rank of $A_{i,j}$ is smaller than $2k$, then the last $2k - r$ columns of $A_{i,j}(:, e)$ are added into $\mathcal{R}$. In addition, the first $b = \min(k, \tau\_rank)$ columns are added into $\mathcal{S}$, for the next level of the tree, and the $(\tau\_rank - b)$ remaining columns are added into $\mathcal{K}$, if $(\tau\_rank - b) > 0$. At the end of the while loop, the indices in $\mathcal{S}$ overwrite the content of the array `colInd`, and then $\mathcal{S}$ is reset to an empty set at the beginning of the next iteration. The algorithm returns $n - |\mathcal{R}|$, an estimation of the τ_rank of A, where n is the number of columns of A, and $|\mathcal{R}|$ is the number of discarded columns with respect to our τ_rank criterion.

Algorithm 5.1 is designed for dense matrices. However, a single modification allows us to apply it on sparse matrices. Line 9, the submatrix $A_{i,j}$ is factored using QRCP algorithm. Since $A_{i,j}$ is of dimension $m \times 2k$, with $m \gg k$, its dense representation may not fit in memory. For sparse matrices, the call to QRCP is thus replaced by two consecutive calls. First, the QR factorization of the sparse matrix $A_{i,j}$ is computed, which returns the sparse upper triangular factor, denoted $\mathbb{R}_{i,j}$. Then, QRCP algorithm is called on the dense representation of $\mathbb{R}_{i,j}$. Since the QR factorization conserves the norms of the columns, the resulted $R_{i,j}$ is the same as the upper triangular factor returned by QRCP applied on the dense representation of $A_{i,j}$. Therefore,

with this approach only the dense representation of $\mathbb{R}_{i,j}$ has to be stored, of size $2k \times 2k$.

Algorithm 5.1 QRTP_reduction (A, k, μ)

This function returns an overestimation of the τ_rank of A, and a column permutation vector that permutes the k most linearly independent columns of A to the leading positions.

Input: A the matrix of dimension $m \times n$,
k the largest number of columns to select from A,
μ the threshold used to discard columns of A

$nsubset = ceil(n/(2k))$
$\mathcal{K} = \{\emptyset\}$, $\mathcal{R} = \{\emptyset\}$
$colInd = \{1 : n\}$
$j = 0$
while $nsubset > 1$ **do**
  $\mathcal{S} = \{\emptyset\}$
  for $i = 1$ to $(nsubset - nsubset\%2)$, $step = 2$ **do**
    $panelInd = colInd(1 + (i-1)k : (i+1)k)$
    $[\tilde{}, R_{i,j}, e] = QRCP(A(:,\ panelInd))$
    $panelInd = panelInd(:, e)$
    $\tau\_rank = \arg\max_i(|r_{ii}| > \mu)$ /*where $R = (r_{ij})_{1 \le i,j \le n}$*/
    $b = \min(\tau\_rank,\ k)$
    $\mathcal{S} = [\mathcal{S},\ panelInd(1 : b)]$
    $\mathcal{K} = [\mathcal{K},\ panelInd(b + 1 : \tau\_rank)]$
    $\mathcal{R} = [\mathcal{R},\ panelInd(\tau\_rank + 1 : 2k)]$
  end for
  if $nsubset\%2 = 1$ **then**
    $colInd = [\mathcal{S},\ colInd(1 + (nsubset - 1) * k : n))]$
  else
    $colInd = \mathcal{S}$
  end if
  $n = |colInd|$
  $nsubset = ceil(n/(2k))$
  $j = j + 1$
end while
$perm = \mathcal{S} \cup \mathcal{K} \cup \mathcal{R}$

Output: $perm$ the vector of column indices where the k most linearly independent columns of A are on leading position,
$n - |\mathcal{R}|$ the estimation of the τ_rank of A

5.3.1 Computing a good threshold to detect the columns to discard

We now discuss the threshold μ used to detect the τ_rank of a matrix. In practice, to reveal the rank of a deficient matrix, a standard threshold is $\epsilon\|A\|_2$, where ϵ is the precision machine. For computing a low-rank approximation, the threshold can be set up to $1e-3\|A\|_2$, for example. In both cases, the threshold is relative to the 2-norm of A. From Equation (5.4), the error of the low-rank approximation of a matrix A is given by

$$\|A - Q \begin{bmatrix} R_{11} & R_{12} \end{bmatrix}\|_2 = \|R_{22}\|_2 \le \tau\|A\|_2, \tag{5.8}$$

where τ is a tolerance. In the previous examples, τ is either ϵ or $1e-3$. In Algorithm 5.1, the threshold μ is provided as input and is defined as $\mu = \tau\|A\|_2$. The computation of the 2-norm

of A using the Singular Value Decomposition is known to be costly. However, this norm can be approximated. We present, in the following, one approach to approximate the 2-norm of A, and we discuss the impact on the algorithm.

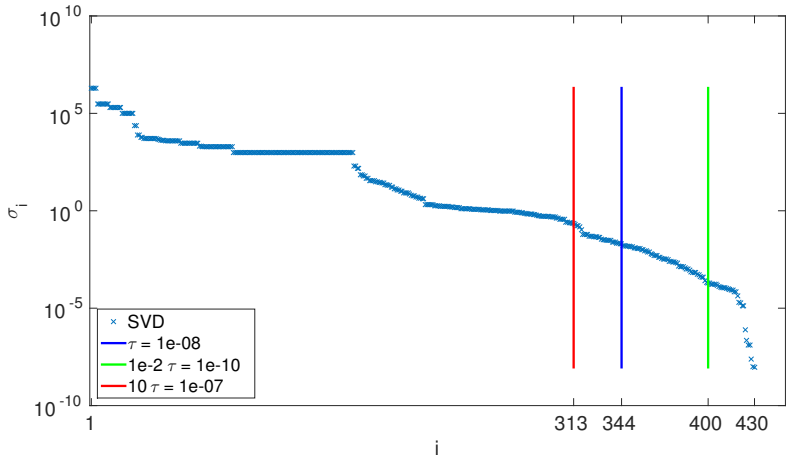


Figure 5.2 – Comparison of the number of columns of a matrix $A \in \mathbb{R}^{430\times 430}$ that are of interest, using different thresholds. A is the matrix Sandia/oscil_dcop_17, and comes from the San Jose State University database. The spectrum of A, represented by blue crosses, is cut by three vertical lines. The blue line represents the number of singular values of A larger than $\mu = \tau\|A\|_2$, where $\sigma_i > \tau\|A\|_2 > \sigma_{i+1}$. The red and green lines represent the case $\widehat{\mu} = 10\tau\|A\|_2$ and $\widehat{\mu} = 10^{-2}\tau\|A\|_2$, respectively.

The QR factorization with Column Pivoting algorithm applied on A starts by computing the maximum column 2-norm of A. Precisely, the absolute value of the first diagonal element of the R factor is equal to $\max_{1\leq i\leq n} \chi_i(A)$. Moreover, Demmel *et al.* have shown in (J. Demmel, Grigori, Gu, et al., 2013) that CARRQR approximates the singular values of A with a maximum error of one order of magnitude, and a minimum error of two orders of magnitude. Thus, the ratio of the maximum column 2-norm of A with respect to the 2-norm of A is at most equal to 10. Although the cost of the Singular Value Decomposition of $A \in \mathbb{R}^{m\times 2k}$ is $8m^2k - 32k^3/3$, the cost of this approximation is $2m^2k$. Hence, we propose a first approximation of the 2-norm of A that computes the largest 2-norm of the columns of A. This approximation may impact the number of columns discarded with respect to our τ_rank criterion. Consider the threshold μ and its approximated threshold $\widehat{\mu}$ using the maximum column 2-norm such that

$$10^{-2}\tau\|A\|_2 \leq \widehat{\mu} \leq 10\tau\|A\|_2 \tag{5.9}$$

If $\widehat{\mu} = 10\tau\|A\|_2$, then the τ_rank computed at line 11 in Algorithm 5.1 is smaller than the real τ_rank, and so the number of discarded columns increases. As a consequence, some columns are discarded, whereas they should not be. Therefore, the error of the low-rank approximation is larger than requested. If $\widehat{\mu} = 10^{-2}\tau\|A\|_2$, then the number of kept columns increases. This estimation of the threshold $\widehat{\mu}$ leads to more computation, but it does not degrade the low-rank approximation. Figure 5.2 illustrates the impact of using an approximation of the 2-norm of A in our algorithm. The figure displays the spectrum of the matrix `Sandia/oscil_dcop_17` of size 430×430, coming from the San Jose State University database (Foster, 2017). We set the tolerance $\tau = 1e-8$ so that $10^{-10} \leq \widehat{\mu}/\|A\|_2 \leq 10^{-7}$. Using the exact 2-norm of A in Algorithm 5.1 leads to having $\tau\_rank = 344$, whereas using $10\tau\|A\|_2$ or $10^{-2}\tau\|A\|_2$ leads to a τ_rank equal to 313 or 400, respectively. In the worst case, we count 31 missing columns

of A, and so that the low-rank approximation is one order of magnitude larger in this example ($\sigma_{313} = 0.2217,\ \ \sigma_{344} = 0.0204$). When the 2-norm of A is underestimated, 400 columns are kept in the algorithm. In fact, 56 columns out of 400 are not of interest. Therefore, the extra 56 columns yield to more computation and so the gain obtained by using an underestimated 2-norm of A is not maximal.

In the parallel case, A is distributed over p processors so that the local part of A is of dimension $m \times n/p$. The computation of the maximum column 2-norm of A involves a reduction operation over processors, and a broadcast of the result. A variant of our approximation considers the parallel case and computes locally the maximum column 2-norm. Each processor uses this (under)estimated maximum column 2-norm to compute μ. Since an underestimation of the 2-norm of A decreases $\widehat{\mu}$, and so increases the amount of computation, the local maximum 2-norm should be updated through the tree. The update of the approximation is performed by computing the maximum column 2-norm of the received columns and comparing with the current value. Using a butterfly communication scheme, processors exchange all selected columns so that all processors have the same selected columns of A without extra communication. At the final level of the tree, the first diagonal element of the final upper triangular factor corresponds to the largest norm over all columns of A. This value is then used when the QR factorization with Tournament Pivoting is recalled on the trailing matrix as in LU-CRTP or CARRQR. However, experimental results will show that the update of the approximated 2-norm of A through the tree is not required in practice.

5.3.2 Saving computation

The purpose of the modification of QRTP algorithm is to save unnecessary operations. To do so, we introduce a comparison step, line 11, to detect the τ_rank of $A_{i,j}$. The test leads to selecting the columns that are added to the subset $\mathcal{R}$. In the following, we show that discarding these columns saves operations in both the update of the trailing matrix and a next call of QRTP_reduction algorithm.

Consider a matrix $A \in \mathbb{R}^{m \times n}$ split into 4 subsets of columns, as in Figure 5.3, and the application of QRTP_reduction on A to select k columns of A. Suppose a tolerance τ, and a threshold μ used as input for QRTP_reduction. Figure 5.3 illustrates the application of QRTP_reduction algorithm on A. We discuss the impact of the τ_rank through the tree. In the figure, the blue rectangle represents the columns of A selected for the next level of the tree, i.e., the indices of these columns are in `colInd` in Algorithm 5.1. The cyan rectangle contains the columns of A that are added into $\mathcal{K}$, whereas the hashed rectangle contains the columns of A that are discarded, with respect to our τ_rank criterion. The first subset of columns of $A_{0,0}$ has a τ_rank equal to k. The columns of the hashed rectangle do not need to be kept in the process. It means that these columns should not be updated during the update of the trailing matrix. We observe that all the columns of $A_{0,1}$ are linearly independent. These non-selected columns can potentially be selected during a further call of QRTP_reduction. Although k columns are selected from $A_{0,2}$, $A_{0,3}$ has a τ_rank smaller than k. It follows that $A_{1,1}$ is formed by the concatenation of the selected k columns from $A_{0,2}$ with the selected columns from $A_{0,3}$. The factorization of $A_{1,1}$ is thus cheaper. Finally, the columns represented by the cyan rectangles need to be further updated, and the columns from the hashed rectangles represent the columns that are not updated and this leads to saving computation.

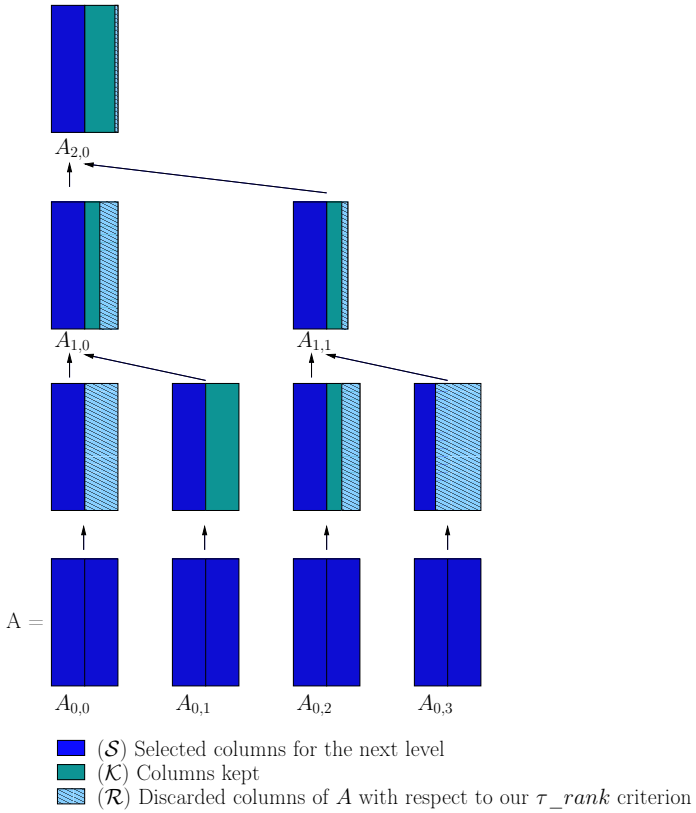


Figure 5.3 – Selection of k columns of A using QRTP_reduction Algorithm based on a binary reduction tree. The matrix A is split into 8 column panels. At each node, QRCP is applied, and the columns, represented by a hashed rectangle, are discarded, with respect to our τ_rank criterion. The selected columns, displayed by blue rectangles, form the array $colInd$ of the next level. The remaining columns, shown in cyan rectangles, are added to the subset $\mathcal{K}$.

The first gain of our modification comes from the reduction of the size of $\mathcal{S}$, compared to the original algorithm. Compared to the original version where at each node, strictly k columns are selected from $A_{i,j}$ for the next level, here only $min(\tau\_rank(A_{i,j}), k)$ columns are selected. The number of columns that participate in the tournament can be divided by more than two between two levels of the tree, and so the structure of the tree must be adapted in consequence. An adaptive tree presents some interests. Mainly, it is expected to reduce the number of nodes of the tree, and even the number of levels. Since the cost of the QRTP_reduction algorithm is related to the number of columns factored, we consider here the difference between the number of columns at a level j of the tree between the original algorithm and the modified algorithm. Therefore, the amount of saved computation is proportional to the number of columns removed. That is,

$$\sum_{i=0}^{nsubset_j} k - min(\tau\_rank(A_{i,j}), k), \tag{5.10}$$

where $0 \leq j < log(n/(2k))$.

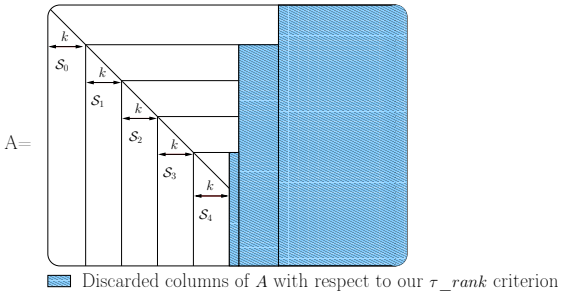


Figure 5.4 – Example of execution of a block algorithm, where QRTP_reduction is used to find the pivot columns at each iteration of the algorithm. The i'th call to QRTP_reduction returns the set $\mathcal{S}_i$ of size k and the discarded columns with respect to our τ_rank criterion, in cyan. Therefore, the update of the trailing matrix is not performed on the colored part of the trailing matrix.

Furthermore, the action of removing the columns of A, with respect to our τ_rank criterion, has an impact on the cost of updating the trailing matrix. Usually, updating the trailing matrix in a block algorithm dominates the cost. Figure 5.4 shows an example of the detected linear columns of A by QRTP_reduction, at each iteration of the block algorithm. For example, the update of the first trailing matrix is classically performed on $n-k$ columns of A, the estimated τ_rank returned by QRTP_reduction allows us to update only τ_rank-k columns of the trailing matrix.

Algorithm 5.2 LU-CRTP_adaptive (A, k, τ)

This function computes the LU decomposition of the low-rank approximation of A, and returns an approximation of the singular values of A.

Input: A the matrix to factor, of dimension $m \times n$,
k the column panel size

1: Compute μ using either $\tau\|A\|_2$, or an approximation of the 2-norm of A times τ
2: **while** $\tau\_rank > k$ **do**
3: $[e, \tau\_rank] = QRTP\_reduction(A, k, \mu)$
4: $A = A(:, e(1 : \tau\_rank))$
5: Do the same treatment as in the original LU-CRTP algorithm
6: **end while**

Output: L, U, P, E, and asv the same returned data as in the original LU-CRTP excepted for U and E which are of size $m \times \tau\_rank(A)$,

To illustrate the usage of QRTP_reduction, we present two algorithms, Algorithm 5.2 is a modification of LU-CRTP algorithm, and Algorithm 5.3 is a modified version of CARRQR algorithm. Since the modifications of LU-CRTP are equivalent as well as the explanations, we focus on Algorithm 5.2. However, the description is easily applied to Algorithm 5.3. Algorithm 5.2, named LU-CRTP_adaptive, replaces QRTP algorithm by QRTP_reduction algorithm. Along with this replacement, we add line 1 the computation of μ, and line 4 where we discard the columns of A. Firstly, the threshold μ is computed using either the 2-norm of A or an approximation of it, as discussed in 5.3.1. At the end of each iteration, the τ_rank returned by QRTP_reduction is used to check whether there are columns in A to discard. Secondly, in line 4, the columns of A are truncated according to τ_rank. In the worst case, the number of current columns of A does not decrease. However, for τ_rank matrices, we save computation during the update of the trailing matrix, since its size decreases from $\mathbb{R}^{(m-k)\times(n-k)}$ to $\mathbb{R}^{(m-k)\times(\tau\_rank-k)}$. Since the update for LU-CRTP is a matrix matrix multiplication and a matrix sum, the Flops saved by the first call to QRTP_reduction are equal to $m \times (n - \tau\_rank) \times (n + 1)$, where

$n - \tau\_rank$ is the number of discarded columns.

Algorithm 5.3 CARRQR_adaptive (A, k, τ)

This function computes the QR factorization of the low-rank approximation of A, and returns an approximation of the singular values of A.

Input: A the matrix to factor, of dimension $m \times n$,
k the column panel size

1: Computing μ using either $\tau\|A\|_2$, or an approximation of the 2-norm of A times τ
2: **while** $\tau\_rank > k$ **do**
3: $[e, \tau\_rank] = QRTP\_reduction(A, k, \mu)$
4: $A = A(:, e(1 : \tau\_rank))$
5: Do the same treatment as in the original CARRQR algorithm
6: **end while**

Output: Q, R, P, E, and asv the same returned data as in the original CARRQR excepted for R and E which are of size $m \times \tau\_rank(A)$,

5.4 Theoretical bounds

In this section, we present theoretical bounds obtained by applying the threshold μ on the singular values of $R_{i,j}$ factor for selecting at most k columns of a matrix A. We recall that the 2-norm of the j-th column of A is given by $\gamma_j(A)$, and the 2-norm of the j-th row of A^{-1} corresponds to $\omega_j(A)$. We also recall in Theorem 29 the bounds of the Strong Rank Revealing QR factorization.

Theorem 29. *(Gu and Eisenstat (Gu and Eisenstat, 1996)) Let A be an $m \times n$ matrix and let $1 \leq k \leq \min(m, n)$. For any given parameter $f > 1$, there exists a permutation Π such that*

$$A\Pi = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$$

where R_{11} is $k \times k$ and

$$(R_{11}^{-1} R_{12})_{i,j}^2 + \omega_i^2(R_{11})\gamma_j^2(R_{22}) \leq f^2 \tag{5.11}$$

The quantity $(R_{11}^{-1} R_{12})_{i,j}^2$, that is in position (i, j) of the product, can be very large without an appropriate column permutation. Therefore, in the case of a strong rank revealing factorization Equation (5.11) bounds $(R_{11}^{-1} R_{12})_{i,j}$ by a constant f. In (J. Demmel, Grigori, Gu, et al., 2013), a different bound of the Strong Rank Revealing QR factorization is derived that is used to establish a bound for the Communication Avoiding variant of the RRQR algorithm. We recall this bound in the following theorem.

Theorem 30. *Theorem 2.4 (J. Demmel, Grigori, Gu, et al., 2013) Assume that there exists a permutation Π for which the QR factorization*

$$A\Pi = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$$

where R_{11} is $k \times k$ satisfies

$$\sqrt{\gamma_j^2(R_{11}^{-1} R_{12}) + (\gamma_j(R_{22})/\sigma_{min}(R_{11}))^2} \leq F \quad for \quad j = 1, ..., n - k \tag{5.12}$$

Then

$$1 \leq \frac{\sigma_i(A)}{\sigma_i(R_{11})}, \frac{\sigma_j(R_{22})}{\sigma_{k+j}(A)} \leq \sqrt{1 + F^2(n - k)} \tag{5.13}$$

for any $1 \leq i \leq k$ *and* $1 \leq j \leq \min(m,n) - k$.

Lemma 31. *Assume that there exists a permutation* Π *for which the QR factorization*

$$A\Pi = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$$

where R_{11} *is* $k \times k$ *satisfies*

$$\sqrt{\gamma_j^2(R_{11}^{-1}R_{12}) + (\gamma_j(R_{22})/\sigma_{min}(R_{11}))^2} \leq F \quad for \quad j = \{1, \ldots, n-k\}. \tag{5.14}$$

If there exist a μ *and an integer* $r < k$ *such that*

$$\sigma_{min}(R_{11}) \leq \mu < \sigma_r(R_{11}), \tag{5.15}$$

then

$$\sqrt{\gamma_j^2(R_{11}^{-1}R_{12}) + (\gamma_j(R_{22})/\sigma_r(R_{11}))^2} < F_\mu \leq F, \tag{5.16}$$

where $F_\mu = \sqrt{\gamma_j^2(R_{11}^{-1}R_{12}) + (\gamma_j(R_{22})/\mu)^2}$.

Proof. Equation (5.16) is a direct consequence of Equation (5.15). □

As discussed earlier, the QR factorization with Tournament Pivoting selects k columns of a matrix A by using tournament pivoting. This algorithm is costly for large matrices with a small rank. Moreover, in the case of the computation of a low-rank approximation of A, the number of columns used to approximate A is even smaller than its rank. In this section, we establish the bound of one reduction step in QRTP_reduction, i.e., the concatenation of two subsets of selected columns of A from the previous level in the tree. We show that the bound of Theorem 29 can be applied for the purpose of this chapter. We also show that our bound, in the worst case, is equal to the bound presented in Lemma 2.5 of (J. Demmel, Grigori, Gu, et al., 2013) for one step of tournament pivoting. Then, we present the bound on the upper triangular factor R_{22}, in Equation 5.4.

One reduction step in QRTP_reduction

Let $\widetilde{B} = \begin{bmatrix} A_{0,l} & A_{1,l} \end{bmatrix}$ be the concatenation of $A_{0,l}$ and $A_{1,l}$ from the level l of the reduction tree, both of size $m \times 2k$, and τ be a tolerance. Note that the columns of $A_{0,l+1}$ are part of the columns of $\widetilde{B}$. Assume that the goal is to detect its τ_rank such that

$$\|A_{0,l+1} - A_{0,j+1}\Pi_{0,l+1}(:, 1:\tau\_rank)\|_2 \leq \tau \|A_{0,l+1}\|_2.$$

In the following, we use the same reasoning as in (J. Demmel, Grigori, Gu, et al., 2013) and we adapt it to τ_rank matrices. We consider that $A_{0,l}$ and $A_{1,l}$ are τ_rank matrices and their factorization satisfies Lemma 28. For simplicity, we denote their factorization as

$$A_{0,l}\Pi_{0,l} = Q_{0,l}R_{0,l} = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}, \tag{5.17}$$

and

$$A_{1,l}\Pi_{1,l} = Q_{1,l}R_{1,l} = \widehat{Q} \begin{bmatrix} \widehat{R}_{11} & \widehat{R}_{12} \\ & \widehat{R}_{22} \end{bmatrix}, \tag{5.18}$$

where $\Pi_{0,l}$ and $\Pi_{1,l}$ are permutation matrices, $R_{11} \in \mathbb{R}^{b\times b}$ and $\widehat{R}_{11} \in \mathbb{R}^{\widehat{b}\times\widehat{b}}$ are upper triangular matrices, $1 \leq b \leq k$ and $1 \leq \widehat{b} \leq k$. We also assume that the factorization of $A_{0,l}$ and $A_{1,l}$ satisfies

$$\gamma_j^2(N) + \gamma_j^2(R_{22})/\sigma_{min}^2(R_{11}) \leq F^2, \quad \gamma_j^2(\widehat{N}) + \gamma_j^2(\widehat{R}_{22})/\sigma_{min}^2(\widehat{R}_{11}) \leq \widehat{F}^2, \tag{5.19}$$

where $N = R_{11}^{-1}R_{12}$ and $\widehat{N} = \widehat{R}_{11}^{-1}\widehat{R}_{12}$. From Theorem 29, we can choose the same f for both factorizations and so $F = f\sqrt{b}$ and $\widehat{F} = f\sqrt{\widehat{b}}$.

Next, we combine $Q\begin{bmatrix}R_{11}\\ \end{bmatrix}$ from $A_{0,l}$ with $Q\begin{bmatrix}\widehat{R}_{11}\\ \end{bmatrix}$ from $A_{1,l}$ into $A_{0,l+1}$, that is factored such that

$$A_{0,l+1}\Pi_{0,l+1} \overset{def}{=} \left[Q\begin{bmatrix}R_{11}\\ \end{bmatrix}, \widehat{Q}\begin{bmatrix}\widehat{R}_{11}\\ \end{bmatrix}\right]\Pi_{0,l+1} = \widetilde{Q}\begin{bmatrix}\widetilde{R}_{11} & \bar{R}_{12}\\ & \bar{R}_{22}\end{bmatrix}, \tag{5.20}$$

with $\widetilde{R}_{11} \in \mathbb{R}^{\widetilde{b}\times\widetilde{b}}$, and where

$$(\widetilde{R}_{11}^{-1}\bar{R}_{12})_{i,j}^2 + \omega_i^2(\widetilde{R}_{11})\gamma_j^2(\bar{R}_{22}) \leq f^2 \tag{5.21}$$

for all $1 \leq i \leq \widetilde{b}$, $1 \leq j \leq b + \widehat{b} - \widetilde{b}$. Let

$$\widetilde{\Pi} = \begin{bmatrix}\Pi_{0,l} & \\ & \Pi_{1,l}\end{bmatrix}\begin{bmatrix}I & & & \\ & & I & \\ & I & & \\ & & & I\end{bmatrix}\begin{bmatrix}\Pi_{0,l+1} & \\ & I\end{bmatrix}$$

be the concatenation of all previous permutations. Then we can write $\widetilde{B}$ as

$$\widetilde{B}\widetilde{\Pi} = \left[\widetilde{Q}\begin{bmatrix}\widetilde{R}_{11} & \bar{R}_{12}\\ & \bar{R}_{22}\end{bmatrix}, \quad Q\begin{bmatrix}R_{12}\\ R_{22}\end{bmatrix}, \quad \widehat{Q}\begin{bmatrix}\widehat{R}_{12}\\ \widehat{R}_{22}\end{bmatrix}\right] \tag{5.22}$$

$$= \widetilde{Q}\left[\begin{bmatrix}\widetilde{R}_{11} & \bar{R}_{12}\\ & \bar{R}_{22}\end{bmatrix}, \quad \widetilde{Q}^TQ\begin{bmatrix}R_{12}\\ R_{22}\end{bmatrix}, \quad \widetilde{Q}^T\widehat{Q}\begin{bmatrix}\widehat{R}_{12}\\ \widehat{R}_{22}\end{bmatrix}\right] \tag{5.23}$$

$$\overset{def}{=} \widetilde{Q}\begin{bmatrix}\widetilde{R}_{11} & \widetilde{R}_{12}\\ & \widetilde{R}_{22}\end{bmatrix} \tag{5.24}$$

In the following, we establish a bound equivalent to Equation (5.12) in Theorem 30, and present Lemma 32, similar to Lemma 2.5 in (J. Demmel, Grigori, Gu, et al., 2013). To do so, we need to identify $\widetilde{R}_{12}$ and $\widetilde{R}_{22}$. Note that the following equations that identify both $\widetilde{R}_{12}$ and $\widetilde{R}_{22}$ come from (J. Demmel, Grigori, Gu, et al., 2013). We develop the following expression

$$\widetilde{Q}^TQ\begin{bmatrix}R_{12}\\ R_{22}\end{bmatrix} = \widetilde{Q}^TQ\begin{bmatrix}R_{11}N\\ R_{22}\end{bmatrix} = \widetilde{Q}^TQ\begin{bmatrix}R_{11}N\\ \end{bmatrix} + \widetilde{Q}^TQ\begin{bmatrix}\\ R_{22}\end{bmatrix}. \tag{5.25}$$

We next express the components of $Q\begin{bmatrix}R_{11}\\ \end{bmatrix}$ permuted by the factorization (5.20) such that

$$Q\begin{bmatrix}R_{11}\\ \end{bmatrix} = \widetilde{Q}\begin{bmatrix}\widetilde{R}_{11}\mathcal{N}\\ \mathcal{C}\end{bmatrix}, \tag{5.26}$$

where $\mathcal{N} \in \mathbb{R}^{\widetilde{b}\times b}$ and $\mathcal{C} \in \mathbb{R}^{(m-\widetilde{b})\times b}$. The construction of $\mathcal{N}$ and $\mathcal{C}$ is directly related to the construction of the factorization (5.20). For each column t of $Q\begin{bmatrix}R_{11}\\ \end{bmatrix}$ corresponds to a column s of the resulted factorization. Thus for each column t $(1 \leq t \leq b)$, if $s \leq \widetilde{b}$ then $\mathcal{C}(:,t) = 0$ and $\mathcal{N}(:,t) = e_s$, else
$\mathcal{C}(:,t) = \bar{R}_{22}(:, s-\widetilde{b})$ and $\mathcal{N}(:,t) = \widetilde{R}_{11}^{-1}\bar{R}_{12}(:, s-\widetilde{b})$, where e_s is a canonical vector. This leads to

$$(\mathcal{N}^2)_{i,j} + \omega_i^2(\widetilde{R}_{11})\gamma_j^2(\mathcal{C}) \leq f^2, \tag{5.27}$$

for $1 \leq i \leq \widetilde{b}$, and $1 \leq j \leq b$.

We also introduce the notation

$$\widetilde{Q}^T Q \begin{bmatrix}\\ R_{22}\end{bmatrix} \stackrel{def}{=} \begin{bmatrix}C_1\\ C_2\end{bmatrix}$$

which allows us to rewrite (5.25) such as

$$\widetilde{Q}^T Q \begin{bmatrix}R_{12}\\ R_{22}\end{bmatrix} = \begin{bmatrix}\widetilde{R}_{11}\mathcal{N}\\ \mathcal{C}\end{bmatrix} N + \widetilde{Q}^T Q \begin{bmatrix}\\ R_{22}\end{bmatrix} = \begin{bmatrix}\widetilde{R}_{11}(\mathcal{N}N + \widetilde{R}_{11}^{-1}C_1)\\ \mathcal{C}N + C_2\end{bmatrix} \tag{5.28}$$

Similarly, we can define $\widehat{\mathcal{N}}, \widehat{\mathcal{C}}, \widehat{N}, \widehat{C}_1$ and $\widehat{C}_2$ and get

$$\widetilde{Q}^T \widehat{Q} \begin{bmatrix}\widehat{R}_{12}\\ \widehat{R}_{22}\end{bmatrix} = \begin{bmatrix}\widetilde{R}_{11}\widehat{\mathcal{N}}\\ \widehat{\mathcal{C}}\end{bmatrix} \widehat{N} + \widetilde{Q}^T \widehat{Q} \begin{bmatrix}\\ \widehat{R}_{22}\end{bmatrix} = \begin{bmatrix}\widetilde{R}_{11}(\widehat{\mathcal{N}}\widehat{N} + \widetilde{R}_{11}^{-1}\widehat{C}_1)\\ \widehat{\mathcal{C}}\widehat{N} + \widehat{C}_2\end{bmatrix}$$

Finally, we can regroup these terms to identify $\widetilde{R}_{12}$ and $\widetilde{R}_{22}$

$$\widetilde{R}_{12} = \begin{bmatrix}\bar{R}_{12} & \widetilde{R}_{11}(\mathcal{N}N + \widetilde{R}_{11}^{-1}C_1) & \widetilde{R}_{11}(\widehat{\mathcal{N}}\widehat{N} + \widetilde{R}_{11}^{-1}\widehat{C}_1)\end{bmatrix} \tag{5.29}$$

$$\widetilde{R}_{22} = \begin{bmatrix}\bar{R}_{22} & \mathcal{C}N + C_2 & \widehat{\mathcal{C}}\widehat{N} + \widehat{C}_2\end{bmatrix} \tag{5.30}$$

Now we can derive the bound for Equation (5.24) studying the three terms of $\begin{bmatrix}\widetilde{R}_{12}\\ \widetilde{R}_{22}\end{bmatrix}$. First according to Theorem 29 we have inequality (5.21). Second and third terms are similar except for the number of columns selected, b and $\widehat{b}$, respectively. Therefore, it suffices to study one of the last two terms.

We have that

$$\begin{aligned}
&\gamma_j^2(\mathcal{N}N + \widetilde{R}_{11}^{-1}C_1) + \gamma_j^2(\mathcal{C}N + C_2)/\sigma_{min}^2(\widetilde{R}_{11}) && (5.31)\\
=&\gamma_j^2\left(\begin{bmatrix}\mathcal{N}N\\ \mathcal{C}N/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix} + \begin{bmatrix}\widetilde{R}_{11}^{-1}C_1\\ C_2/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix}\right)\\
\leq&2\left(\gamma_j^2\left(\begin{bmatrix}\mathcal{N}N\\ \mathcal{C}N/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix}\right) + \gamma_j^2\left(\begin{bmatrix}\widetilde{R}_{11}^{-1}C_1\\ C_2/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix}\right)\right)\\
\leq&2\left(\left|\left|\begin{bmatrix}\mathcal{N}\\ \mathcal{C}/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix}\right|\right|_F^2 \gamma_j^2(N) + \gamma_j^2\left(\begin{bmatrix}C_1\\ C_2\end{bmatrix}\right)/\sigma_{min}^2(\widetilde{R}_{11})\right) && (5.32)
\end{aligned}$$

From inequality (5.27), we have

$$\left|\left|\begin{bmatrix}\mathcal{N}\\ \mathcal{C}/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix}\right|\right|_F^2 \leq b\widetilde{b}f^2. \tag{5.33}$$

Using Equation (5.33) in (5.32), we obtain

$$2\left(\left|\left|\begin{bmatrix}\mathcal{N}\\ \mathcal{C}/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix}\right|\right|_F^2 \gamma_j^2(N) + \gamma_j^2\left(\begin{bmatrix}C_1\\ C_2\end{bmatrix}\right)/\sigma_{min}^2(\widetilde{R}_{11})\right) \tag{5.34}$$

$$\leq 2b\widetilde{b}f^2\left(\gamma_j^2(N) + \gamma_j^2\left(\begin{bmatrix}C_1\\ C_2\end{bmatrix}\right)/\sigma_{min}^2(\widetilde{R}_{11})\right). \tag{5.35}$$

Also from Equation (5.26), it follows

$$R_{11}^T R_{11} = \mathcal{N}^T \widetilde{R}_{11}^T \widetilde{R}_{11}\mathcal{N} + \mathcal{C}^T\mathcal{C}$$

which implies from Theorem 3.3.16 in (Horn et al., 1986)

$$\sigma_i^2(R_{11}) \leq \sigma_i^2(\widetilde{R}_{11})\|\mathcal{N}\|_2^2 + \|\mathcal{C}\|_2^2 \tag{5.36}$$

where $1 \leq i \leq \min(b, \widetilde{b})$. If $b \geq \widetilde{b}$, we have $\sigma_{\widetilde{b}}^2(R_{11}) \geq \sigma_b^2(R_{11}) = \sigma_{min}^2(R_{11})$ and Equation (5.36) becomes with $i = \widetilde{b}$

$$\begin{aligned}\sigma_{\widetilde{b}}^2(R_{11}) &\leq \sigma_{min}^2(\widetilde{R}_{11})\|\mathcal{N}\|_2^2 + \|\mathcal{C}\|_2^2\\ \sigma_{min}^2(R_{11}) \leq \sigma_{min}^2(\widetilde{R}_{11}) &\left(\left|\left|\begin{bmatrix}\mathcal{N}\\ \mathcal{C}/\sigma_{min}(\widetilde{R}_{11})\end{bmatrix}\right|\right|_F^2\right) \leq b\widetilde{b}f^2\sigma_{min}^2(\widetilde{R}_{11})\end{aligned} \tag{5.37}$$

Using all these bounds in (5.32), the upper bound of Equation (5.31) is now

$$\begin{aligned}\gamma_j^2(\mathcal{N}N + \widetilde{R}_{11}^{-1}C_1) + \gamma_j^2(\mathcal{C}N + C_2)/\sigma_{min}(\widetilde{R}_{11}) &\leq 2b\widetilde{b}f^2(\gamma_j^2(N) + \gamma_j^2(R_{22})/\sigma_{min}^2(R_{11}))\\ &\leq 2b\widetilde{b}f^2F^2\end{aligned}$$

Otherwise, when $b < \widetilde{b}$, by construction, we have $\sigma_{min}^2(\widetilde{R}_{11}) \geq \mu$ that leads to bounding Equation (5.35) as follow.

$$2b\widetilde{b}f^2\left(\gamma_j^2(N) + \gamma_j^2\left(\begin{bmatrix}C_1\\ C_2\end{bmatrix}\right)/\sigma_{min}^2(\widetilde{R}_{11})\right), \tag{5.38}$$

$$\leq 2b\widetilde{b}f^2\left(\gamma_j^2(N) + \gamma_j^2\left(\begin{bmatrix}C_1\\ C_2\end{bmatrix}\right)/\mu^2\right), \tag{5.39}$$

$$\leq 2b\widetilde{b}f^2F_\mu^2, \tag{5.40}$$

where, from Lemma 31, $\sqrt{\gamma_j^2(R_{11}^{-1}R_{12}) + (\gamma_j(R_{22})/\mu)^2} = F_\mu \leq F$.
Therefore, we have

$$\gamma_j^2(\mathcal{N}N + \widetilde{R}_{11}^{-1}C_1) + \gamma_j^2(\mathcal{C}N + C_2)/\sigma_{min}(\widetilde{R}_{11}) \leq 2b\widetilde{b}f^2F^2. \tag{5.41}$$

Similarly, we obtain the following upper bounds concerning the third term

$$\gamma_j^2(\widehat{\mathcal{N}}\widehat{N} + \widetilde{R}_{11}^{-1}\widehat{C}_1) + \gamma_j^2(\widehat{\mathcal{C}}\widehat{N} + \widehat{C}_2)/\sigma_{min}(\widetilde{R}_{11}) \leq 2\widetilde{b}\widetilde{b}f^2\widehat{F}^2. \tag{5.42}$$

All these bounds lead to the following lemma

Lemma 32. *Suppose given two rank revealing QR factorizations of two matrices B and $\widehat{B}$ as in (5.17) and (5.18), that reveal their τ_rank b and $\widehat{b}$, respectively. Suppose we perform another rank revealing QR factorization of the concatenation of the b selected columns of B with the $\widehat{b}$ selected columns of $\widehat{B}$. Then the $\widetilde{b}$ columns selected by this last QR factorization leads to perform a QR factorization of $\widetilde{B} = [B, \widehat{B}]$, as described in (5.24), that reveals the τ_rank of $\widetilde{B}$ with the bound*

$$\sqrt{\gamma_j^2(\widetilde{R}_{11}^{-1}\widetilde{R}_{12}) + \gamma_j^2(\widetilde{R}_{22})/\sigma_{min}^2(\widetilde{R}_{11})} \leq \sqrt{2\widetilde{b}}f\max(\sqrt{b}F, \sqrt{\widehat{b}}\widehat{F}) \tag{5.43}$$

As a consequence, Equation (5.43) is bounded by $\sqrt{2}fk\max(F, \widehat{F})$ that is the same bound as in Lemma 2.5 in (J. Demmel, Grigori, Gu, et al., 2013). The following theorem holds in our case where we select at most k columns from a panel.

Theorem 33. *((J. Demmel, Grigori, Gu, et al., 2013)) Let $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, and assume for simplicity that k divides n. Suppose that we do QR factorization with tournament pivoting on A n/k times, each time selecting k columns to pivot to the left of the remaining matrix. Then for ranks K that are multiples of k, this yields a rank revealing QR factorization of A in the sense of Theorem 30, with*

$$F = \begin{cases} \left(1 + \sqrt{k+n}c_1\right)\left(1 + \frac{1}{\sqrt{2}}\left(\sqrt{2}fk\right)^{log_2(n/k)}\right)^{n/k} & \text{for binary tree,} \\ e^4\left(1 + \sqrt{k+n}c_1\right)\left(1/\sqrt{2}\right)^{n/k}\left(\sqrt{2}fk\right)^{n/k(n/k+1)/2} & \text{for flat tree.} \end{cases}$$

5.4.1 An upper bound on the discarded columns of A

As introduced earlier, the modifications of QRTP algorithm leads to detect the columns of a matrix A that are of interest, and to remove the remaining columns of A from the algorithm. In the following, we consider that the matrix $A_{0,l+1}$, in Equation (5.20), is τ_rank, and we establish an upper bound on R_{22} from Equation (5.4) that ensures that these remaining columns are discarded. To do so, we start with Equation (5.30) obtained by identification

$$\widetilde{R}_{22} = \begin{bmatrix} \bar{R}_{22} & \mathcal{C}N + C_2 & \widehat{\mathcal{C}}\widehat{N} + \widehat{C}_2 \end{bmatrix}. \tag{5.44}$$

We can bound each term of Equation (5.44) as follows. The threshold μ applied on the upper triangular factor $R_{0,l+1}$ of $A_{0,l+1}$ leads to

$$\|\bar{R}_{22}\|_2 \leq \mu. \tag{5.45}$$

Moreover, by construction we have

$$\mathcal{C}(:,t) = \begin{cases} 0 \\ \bar{R}_{22}(:,\ s-b) \end{cases} \quad \text{and} \quad N = R_{11}^{-1}R_{12}, \tag{5.46}$$

and by definition we have

$$C_2 = \widetilde{Q}^T Q \begin{bmatrix} \\ R_{22} \end{bmatrix}. \tag{5.47}$$

Therefore, we obtain the last two bounds

$$\|C_2\|_2^2 \leq \|R_{22}\|_2^2 \leq \mu^2. \tag{5.48}$$

$$\|\mathcal{C}N\|_2^2 \leq \mu^2, \tag{5.49}$$

which lead to

$$\begin{aligned} \|\mathcal{C}N + C_2\|_2^2 &\leq \|C_2\|_2^2 + \|\mathcal{C}N\|_2^2, \\ &\leq \mu^2(F^2 + 1). \end{aligned} \tag{5.50}$$

Similarly with $\widehat{B}$, we have the bound

$$\|\widehat{\mathcal{C}}\widehat{N} + \widehat{C}_2\|_2^2 \leq \mu^2(\widehat{F}^2 + 1). \tag{5.51}$$

Combining all these relations to bound $\widetilde{R}_{22}$, we have

$$\|\widetilde{R}_{22}\|_2^2 \leq \mu^2(1 + \max(F^2, \ \widehat{F}^2)) \tag{5.52}$$

The bound (5.52) is a product of the threshold μ with a value that can be chosen to be small enough. This theoretical bound allows us to discard some the columns with respect to our τ_rank criterion.

Theorem 34. *Let A be a matrix of dimension $m \times n$ with $m \geq n$, split into n/k subsets of columns. Consider a binary reduction tree, where the leaves are $A = [A_{0,0}, A_{1,0}, \ldots, A_{n/k-1,0}]$. Suppose that we perform the strong Rank Revealing QR factorization on each node of the tree. On each node, a threshold $\mu = \tau\|A\|_2$ is applied on the $R_{i,j}$ factor to select b columns, with $b \leq k$, where $R_{i,j}$ is the upper triangular factor of the i-th subset of the j-th level of the tree. Let R_{22} be the lower right factor from the decomposition of $A\Pi = Q \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$, where Π is the column permutation returned at the end of the reduction tree, as in Algorithm 5.1. The norm of R_{22} is bounded by*

$$\|R_{22}\|_2^2 \leq 2\tau^2\|A\|_2^2 F_{max}, \tag{5.53}$$

where F_{max} is the maximum among all the F obtained during the QRTP_ reduction algorithm.

Proof. The bound presented in Equation (5.52) is equivalent to splitting A into two subsets, referred to as the basic case. Consider now A is split into four subsets, where the selected columns from the first set are $\widetilde{Q} \begin{bmatrix} \widetilde{R}_{11} \\ 0 \end{bmatrix}$, and the selected columns from the second set are $\widetilde{Q'} \begin{bmatrix} \widetilde{R'}_{11} \\ 0 \end{bmatrix}$. Let R be the upper triangular factor of the concatenation of the previous selected columns, so that R is also the upper triangular factor of A, by definition. Let F and $\widehat{F}$ be related to the first set and F' and $\widehat{F}'$ be related to the second set. Let F_{max} be the maximum over all F. The only difference with the basic case is the bound of Equation 5.48, which is rewritten using (5.52) such that

$$\|C_2\|_2^2 \leq \|R_{22}\|_2^2 \leq \mu^2(1 + \max(F^2, \widehat{F}^2)). \tag{5.54}$$

Therefore, from $R = \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$, the bound of R_{22} is

$$\|R_{22}\|_2^2 \leq \mu^2(\max(F^2,\ \widehat{F}^2) + \max(F'^2,\ \widehat{F'}^2)). \tag{5.55}$$

By induction, the relation (5.55) can be generalized to a larger reduction tree, and so we have

$$\|R_{22}\|_2^2 \leq 2\tau^2\|\widetilde{A}\|_2^2 F_{max}. \tag{5.56}$$

□

5.5 Experimental results

In this section, we study the impact of using QRTP_reduction on the approximation of the singular values returned by LU-CRTP_adaptive. Although QRTP_reduction is expected to reduce the number of operations, it should not degrade the performance of the method. We make tests on a set of 261 deficient matrices coming from San Jose State University database (Foster, 2017). For each matrix, the database provides the rank that we further use as a reference. These challenging matrices, used in (J. Demmel, Grigori, Gu, et al., 2013) have less than 1024 rows and between 32 and 2048 columns. To obtain the matrices in MATLAB, we use

```
idx = SJget;
ids = find(idx.nrows <= 1024 &...
            idx.ncols <= 2048 & idx.ncols > 32);
[~, i] = sort(idx.numrank(ids));
ids = ids(i);
```

QRTP_reduction algorithm is designed to return an estimation of the rank of the matrix. We further study whether the returned rank is a good estimation of the real one, which is provided by the database. Experimental results will show that the first call of QRTP_reduction in LU-CRTP_adaptive gives a significant gain when matrices have a small τ_rank ($A \in \mathbb{R}^{m\times n}$ and $\tau\_rank \ll n$). Unless it is mentioned, we approximate the norm of $\|A\|_2$ by using $\max_{1\leq i\leq n} \chi_i(A)$, in the computation of the threshold μ, that is applied on the diagonal elements of $R_{i,j}$ in QRTP_reduction. In addition, we focus our experiments on the impact of using different approaches for approximating the norm of $\|A\|_2$.

5.5.1 Revealing the rank of a matrix A using $\tau = \epsilon$

We consider first the case of revealing the rank of a matrix $A \in \mathbb{R}^{m\times n}$, which corresponds to setting up the tolerance to ϵ. In (J. Demmel, Grigori, Gu, et al., 2013), it is shown experimentally that QRCP algorithm computes an approximation of the singular values of A with a maximum error of one order of magnitude. As already introduced in Chapter 4, we use an intermediate algorithm denoted by LU-CRQRCP, that uses QRCP algorithm instead of QRTP_reduction to find the best columns at each iteration. Hence, QRCP is used as a reference, and LU-CRQRCP is used for comparing the impact of QRTP_reduction on LU-CRTP. Thus, since QRTP_reduction removes deficient columns of A, we compare the approximation of the singular values returned by LU-CRTP_adaptive with QRCP and LU-CRQRCP in Figure 5.5. For each method, we compute the ratio $|R_{i,i}|/\sigma_i$ where $|R_{i,i}|$ is the i'th absolute diagonal element of the R factor and σ_i is the i'th singular value of the matrix A, returned by the SVD algorithm. For each matrix and each

method, we compute the average ratio, the largest and smallest ratios which are summarized in Figure 5.5. The matrices, represented by their index, are sorted by their rank, provided by the database. The first 231 matrices have a rank smaller than 500. The accuracy of the approximations returned by LU-CRTP depends on the number of calls to tournament pivoting, that is the number of columns in a panel of the block factorization. Thus, the set of matrices is split into two groups. The first group, composed of the first 231 matrices, uses a column panel size of 16 whereas the second group uses a column panel size of 64. In Figure 5.5, the vertical black hashed line separates the two groups.

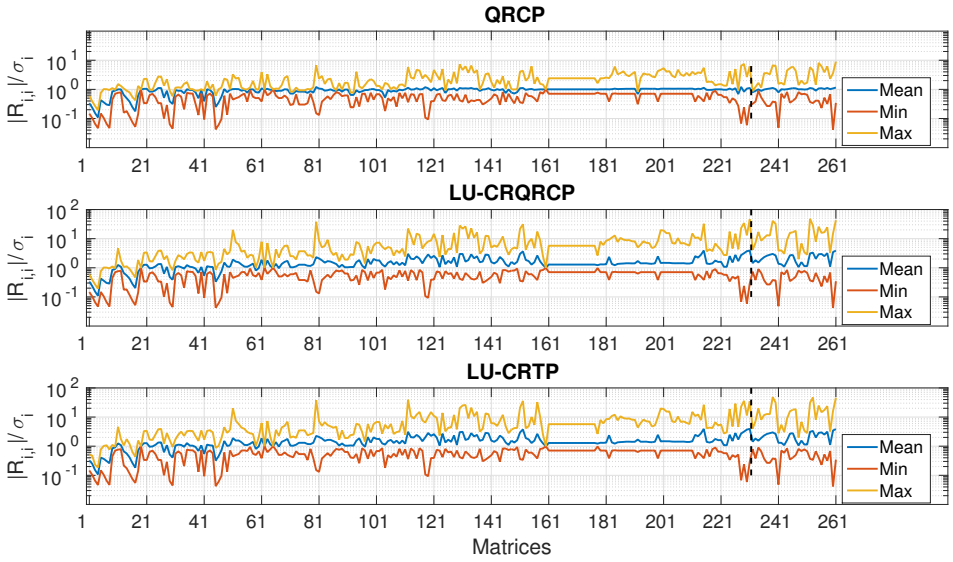


Figure 5.5 – Comparison of the largest and the smallest ratios $R_{i,i}/\sigma_i$ of LU-CRTP_adaptive with LU-CRQRCP and QRCP algorithms. The vertical dotted line splits the set such that the left part uses a panel size of 16, whereas the second part uses 64.

k	Method	Min		Max	
		matrix id	ratio	matrix id	ratio
16	QRCP	45	4.237e-02	229	7.494e+00
	LU-CRQRCP	45	4.237e-02	231	4.595e+01
	LU-CRTP_adaptive	45	4.237e-02	112	3.937e+01
64	QRCP	260	4.169e-02	261	8.957e+00
	LU-CRQRCP	260	4.169e-02	252	4.816e+01
	LU-CRTP_adaptive	260	4.169e-02	239	4.859e+01

Table 5.1 – Smallest minimum and largest maximum of the ratio $|R_{i,i}|/\sigma_i$ of QRCP, LU-CRQRCP, and LU-CRTP_adaptive, with $1 \leq i \leq rank(A)$, where $rank(A)$ is the numerical rank provided by the database. The 261 studied matrices are split into two groups. The first group is composed of matrix indices 1 to 231 and uses a column panel size of $k = 16$. The second group, i.e., matrix indices 232 to 261, uses a column panel size of $k = 64$.

We observe that the three curves of LU-CRTP_adaptive are similar to the curves of LU-

CRQRCP. The reduction of the number of columns of A, induced by QRTP_reduction during the rank revealing factorization of A does not degrade the approximation of the singular values. The maximum ratios of LU-CRTP_adaptive are slightly better than those of LU-CRQRCP for the first group of matrices. Table 5.1 presents the maximum and the minimum ratios for each method and for each group. We observe that the largest and the smallest ratios of QRCP are the same as presented in (J. Demmel, Grigori, Gu, et al., 2013). For the first group, LU-CRTP_adaptive has the largest ratio of 39, that is smaller than LU-CRQRCP, with the largest ratio of 46. Similarly to LU-CRTP, the largest ratio of LU-CRTP_adaptive does not exceed two orders of magnitude. Moreover, the second group of matrices has the largest ratio of 48, which is close to LU-CRQRCP. Note that the smallest ratio is the same for all methods.

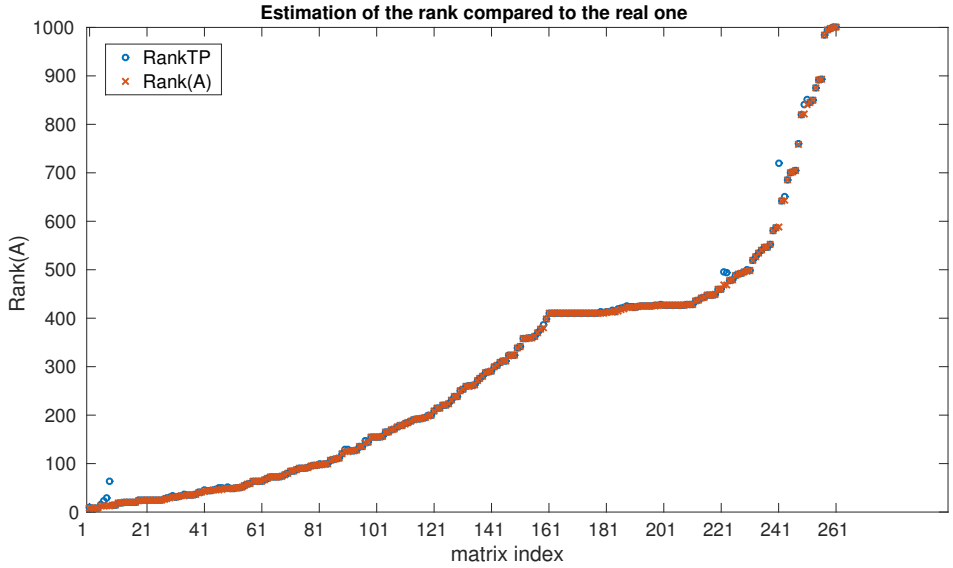


Figure 5.6 – Comparison of the estimated rank after the first call to QRTP_reduction, referred to as $rankTP$, with the rank given by the database, referred to as $rank(A)$. Here $b = 16$ for the first 231 matrices and $b = 64$ for the remaining matrices.

We now study the impact of using QRTP_reduction to estimate the rank of the 261 rank deficient matrices. We first compare the rank returned by LU-CRTP_adaptive, denoted hereafter as `rankTP`, with the rank given by the database, denoted `rank(A)`. Figure 5.6 displays both ranks and shows that the estimated rank is the same as the one given by the database, except for seven matrices (ids([5, 6, 7, 8, 222, 223, 241])). Notice that the rank returned by LU-CRTP_adaptive for the matrix `fs_760_2` ids(178) is 410 whereas the rank given by the database is 411. This is the only case where the method underestimates the rank. We focus our attention on the behavior of the methods in this case. Figure 5.7 plots both the spectrum returned by the three methods and a zoom on the end of the spectrum. We observe a maximum ratio of $\max_i(|R_{i,i}|/\sigma_i)$ of 2.64 for both LU-CRTP_adaptive and LU-CRQRCP, while for QRCP this maximum ratio is 1.63. Focusing on the end of the spectrum in Figure 5.7, we observe that the singular values and their approximations are close. Moreover, the approximation of the 411'th singular value of LU-CRTP_adaptive slightly underestimates the singular value returned by the Singular Value Decomposition. Since the estimation of the rank is based on the approximation of the singular values, this explains the difference between `rankTP` and `rank(A)`. From these observations, we

can conclude that LU-CRTP_adaptive estimates well the rank of deficient matrices.

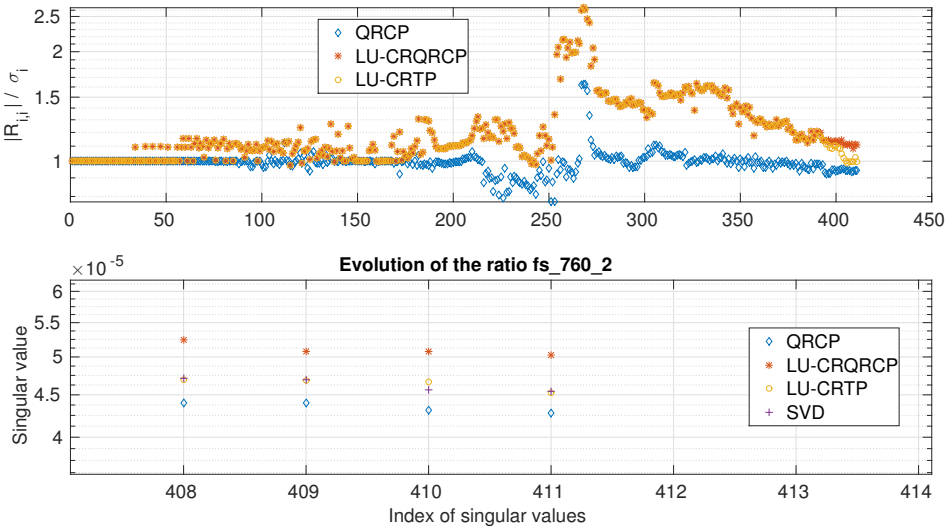


Figure 5.7 – Part of the spectrum computed by LU-CRTP_adaptive with $b = 16$ for the matrix fs_760_2 ids(178) $A \in \mathbb{R}^{760\times760}$, and the whole evolution of the ratios $\max_{1\leq i\leq 760}(|R_{i,i}|/\sigma_i)$.

We next focus on the estimation returned by the first call of QRTP_reduction in LU-CRTP_adaptive. This estimation is directly related to the number of columns that will not be updated from A. After one call of QRTP_reduction, we expect saving the most of computations. Figure 5.8 compares the estimated rank returned by QRTP_reduction with the rank of each matrix given by the database. The graph shows that the first estimation of the rank is close to the real one, for 30 matrices out of 261. Among these 231 matrices, only 9 matrices have an error on the rank larger than 100 columns and 142 matrices have an error smaller than 10 columns. This result shows that the first call of QRTP_reduction has a large impact on saving computations. This observation leads to comparing the consequences of using different methods to compute the approximation of the norm of A.

We now compare the estimated rank returned by QRTP_reduction using three different methods. The first method is the one used until now, i.e., the norm of A is approximated by the maximum column 2-norm over all columns of A. The second method, denoted as `RankTP-SVD`, uses the real norm returned by the SVD algorithm. The third one, referred to as `RankTP_L` computes the local maximum column norm of each column panel, and uses it as an initial guess for the norm of A. The estimation of the rank of all matrices and for all methods is plotted in Figure 5.8. The method `RankTP-SVD` overestimates the rank of 222 matrices. However, it also underestimates the rank of 4 matrices. As a consequence, at most two linearly independent columns of A are removed. The method `RankTP_L` overestimates the rank of 231 and returns a larger estimation of the rank than `RankTP` for 17 matrices. Especially, the matrix of index 88 has an estimated rank of 725, whereas its real rank is 117. This error is due to the local estimation of the norm. This matrix has much less linearly independent columns than estimated, and so, in this particular case, `RankTP_L` should be replaced by `RankTP`. In summary, the version using the maximum column norm is efficient enough.

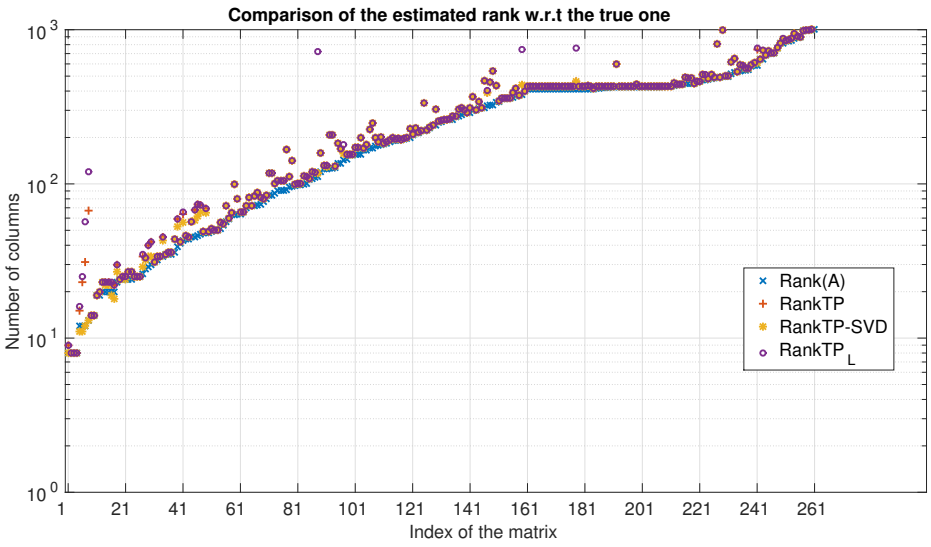
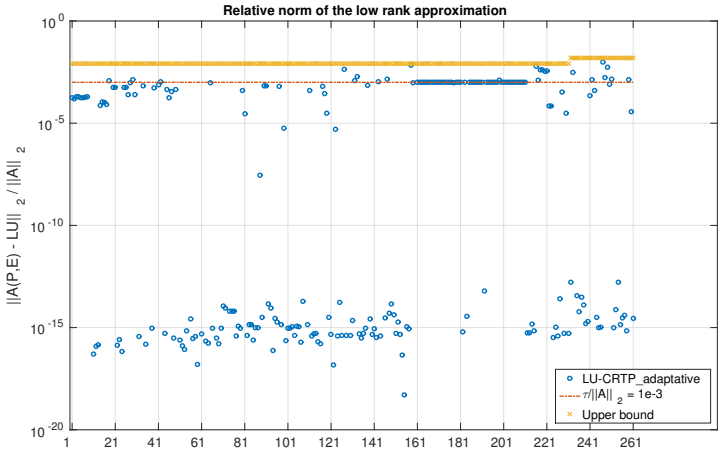


Figure 5.8 – Comparison of the estimated rank after the first call of QRTP_reduction with the rank given by the database. Here, rankTP curve uses the maximum column norm as threshold, rankTP-SVD uses the same criterium as the database and rankTP_L computes a local maximum column norm

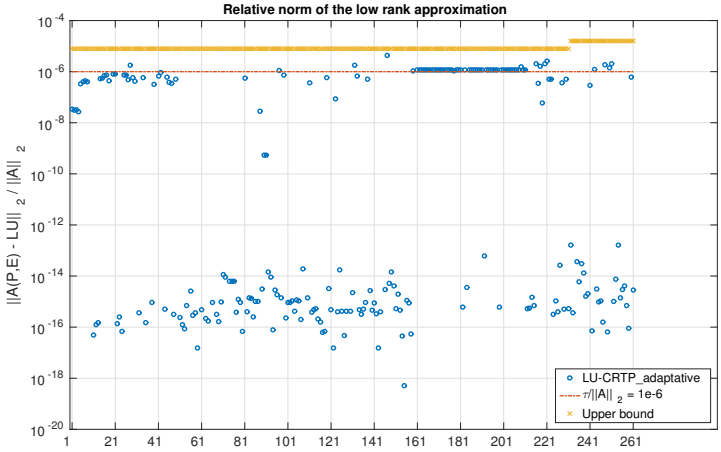
5.5.2 Low-rank approximation of a matrix A, with different tolerance

In this section, we study the accuracy of the low-rank approximation obtained with LU-CRTP_adaptive with different tolerance $\tau \in \{10^{-3}, 10^{-6}, 10^{-9}\}$. The tests are made on the 261 matrices presented above. The matrices are split into two groups according to their rank. Thus, the first 231 matrices are approximated using a column panel size of 16. The remaining matrices are approximated using a column panel size of 64. We show that the relative error on the low-rank approximation is always under the bound of Theorem 34. To study the accuracy of Algorithm 5.2, we compute the low-rank approximation of a matrix A such that $\|A(P,E) - LU\|_2 \leq \tau\|A\|_2$, where E and P are permutation matrices, and L and U are the lower and upper triangular factors of A, respectively. Then, we compare the error of the LU factorization with the upper bound $2\tau^2\|A\|_2^2 F_{max}$, presented in Theorem 34. For simplicity, we suppose that $F_{max} = \sqrt{b}$. In practice, this ideal upper bound is never reached. The database computes the rank of a matrix using the formula $max(M,N) \times \tau \times \|A\|_2$, where $A \in \mathbb{R}^{M \times N}$.

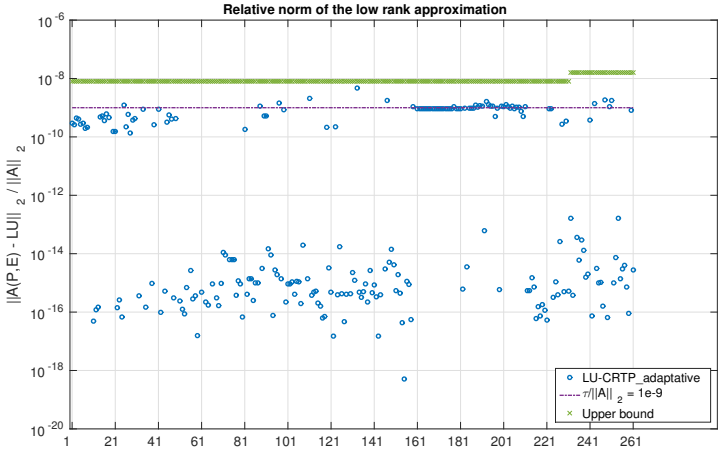
For the purpose of this section, this formula leads to a criterion that is too large, especially when τ is equal to either M or N. For example, in the dataset, the matrices of dimension 10^3 have a criterion equal to the norm of A ($10^3 \times 10^{-3}\|A\|_2$). Therefore, only the first singular value is considered and the low-rank is strictly equivalent to a rank-1 approximation. As discussed in the previous section, we approximate the norm of A by using the maximum columns 2-norm of A. Hence, the threshold used by QRTP_reduction algorithm is $\tau \times \max_{1 \leq i \leq n} \chi_i(A)$. We first set up the tolerance to 10^{-3}, and we plot the relative error on the low-rank approximation, the relative upper bound and the tolerance in Figure 5.9a. We observe that the relative error of the low-rank approximation is smaller than the tolerance, 10^{-3}, except for 22 matrices. However, these 22 matrices have a relative error that does not reach the upper bound of Theorem 34. The remove of some columns of A, with respect to our τ_rank criterion, does not degrade the low-rank approximation. Figures 5.9b and 5.9c show the same results as for 10^{-3}, which is that the upper bound is never reached. It means the removed columns do not contribute to the approximation.



(a) $\tau = 10^{-3}$.



(b) $\tau = 10^{-6}$.



(c) $\tau = 10^{-9}$.

Figure 5.9 – Comparison of the relative norm of the low-rank approximation of A returned by LU-CRTP_adaptive, for different relative thresholds. The red hashed line shows the required low-rank. The crossed symbol represents the upper bound of Theorem 34.

We now discuss the impact of using QRTP_reduction in CARRQR_adaptive algorithm on the accuracy and on the performed computations. The set of matrices is composed of the 261 matrices presented above, and splits into two groups. The first 231 matrices use a column panel size of 16 whereas the second group uses a column panel size of 64. The tolerance is set to 10^{-3}. The approximated singular values returned by CARRQR_adaptive are compared to the approximations returned by CARRQR and QRCP. To evaluate the accuracy, we compute the ratio $|R_{i,i}|/\sigma_i$, where $|R_{i,i}|$ is the absolute diagonal element of the R factor, and σ_i is the i'th singular value computed by the SVD. Figure 5.10 shows the average ratio, the minimum and maximum ratios for all matrices. We observe that the modified version of CARRQR does not degrade the accuracy compared to the original algorithm. Moreover, both communication avoiding versions have similar minimum and maximum ratios to QRCP.

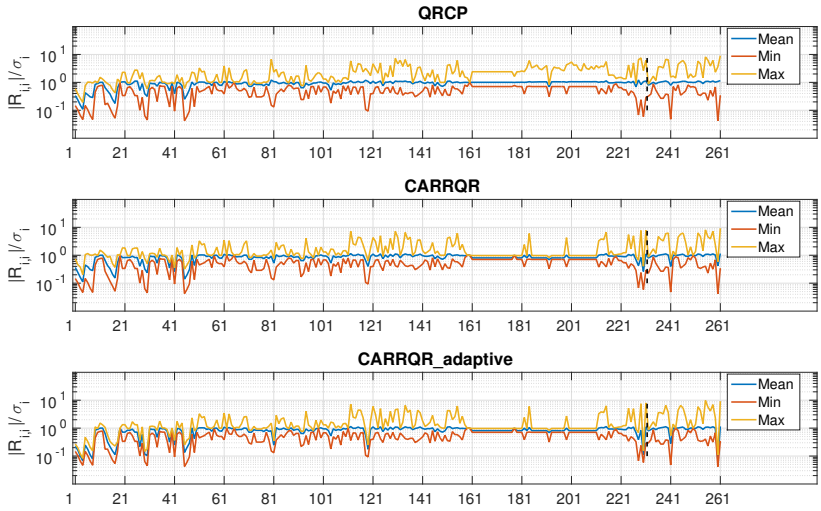


Figure 5.10 – Comparison of the error of QRCP (top), CARRQR (middle), and CARRQR_adaptive (bottom) on the singular values for 261 matrices. The set of matrices is divided into two subsets, where the first subset uses a column panel size of 16, and the second uses a column panel size of 64.

We next focus the discussion on the gain obtained by replacing QRTP with QRTP_reduction in CARRQR_adaptive. In Section 5.3.2, we explained that using QRTP_reduction is expected to reduce the number of operations during the tournament pivoting, and also reduce the cost of updating the trailing matrix. On the one hand, Figure 5.11 presents the total number of FLOPS to compute both the QR factorizations and the QR factorizations with Column Pivoting during the whole procedure, for CARRQR and CARRQR_adaptive. As expected in theory, we observe that the usage of QRTP_reduction reduces the total number of FLOPS, by a factor up to 6. On the other hand, Figure 5.12 shows the number of FLOPS performed by CARRQR_adaptive and by CARRQR, during the update of the trailing matrix. Although the gain obtained from QRTP_reduction on QR is noticeable for a few matrices, the number of FLOPS performed during the update is significantly reduced for the first half of the matrices. For the remaining matrices, the rank is closer to the dimension of the matrices so that, the number of discarded columns is fewer than with the matrices of the first half. To quantify the gain obtained from the update, we plot the ratio of the number of FLOPS performed by CARRQR to update over the number of FLOPS performed by CARRQR_adaptive to update, in Figure 5.13. We observe that the ratio is, at least, equal to one, and, a quarter of the matrices have a ratio larger than one

but smaller than 10. A few matrices have a ratio larger than 10, but we notice a maximum ratio of 32 for one matrix. These observations confirm that using QRTP_reduction has a significant impact on the number of FLOPS for the two costly steps of the algorithm. Discarding columns of A does not degrade the approximation of the singular values. Hence, with a negligible overhead, the modification of QRTP keeps the properties of accuracy of LU-CRTP and CARRQR.

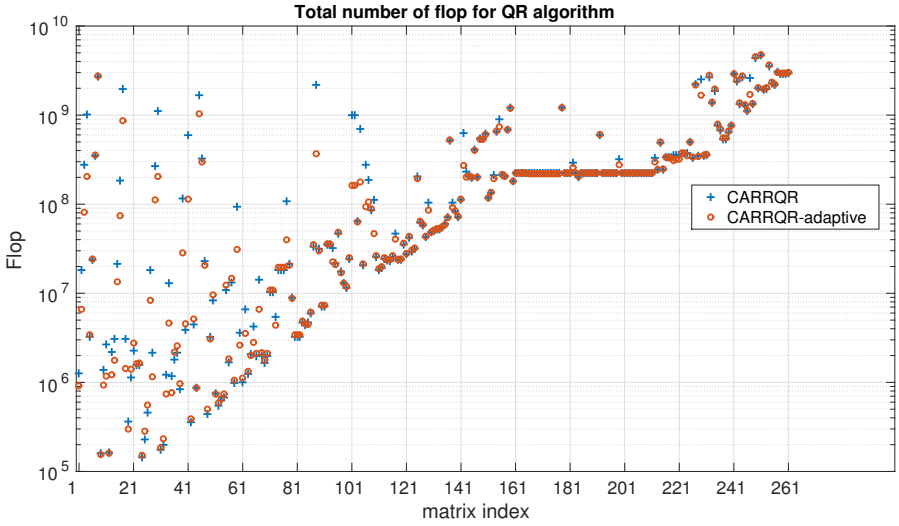


Figure 5.11 – Comparison of the total number of FLOPS performed by both QR and QRCP algorithm in the tournament pivoting algorithm, between CARRQR and CARRQR_adaptive.

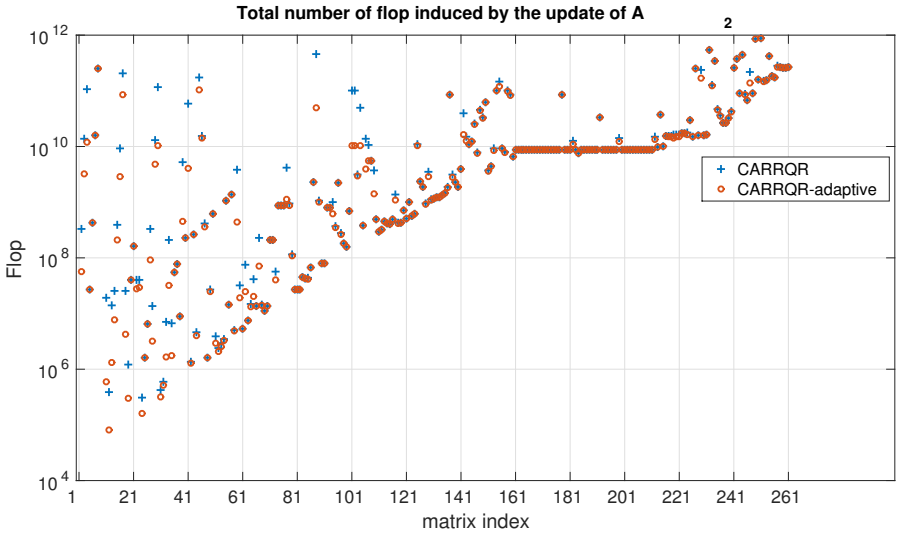


Figure 5.12 – Comparison of the total number of FLOPS performed by the update of the trailing matrix in CARRQR and CARRQR_adaptive.

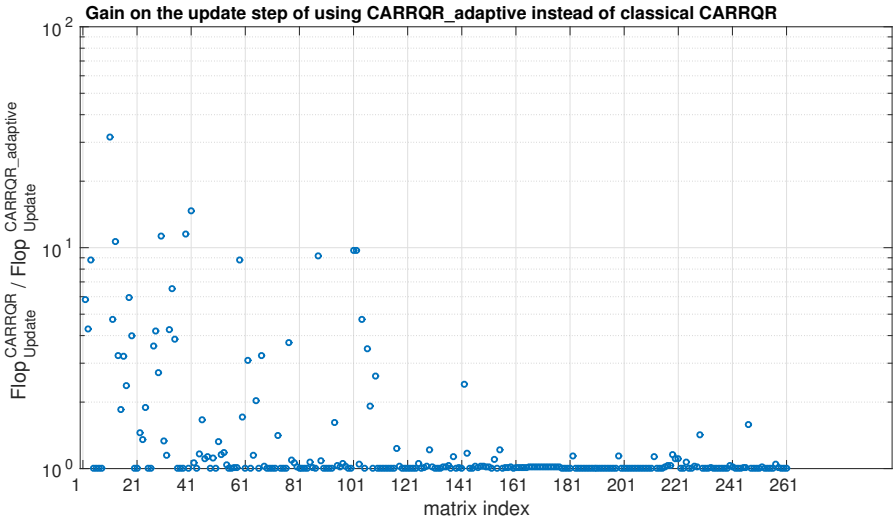


Figure 5.13 – Gain of using QRTP_reduction that removes columns of A, instead of QRTP, in CARRQR_adaptive.

5.6 Conclusion

This chapter shows that the modification of the QR factorization with Tournament Pivoting, in Algorithm 5.1, does not degrade the approximation of the singular values computed by both CARRQR and LU-CRTP. This modified QRTP algorithm is able to approximate well the τ_rank of a matrix. We note that the first call to it can even a good estimate of the τ_rank and so save computation. The purpose of these modifications was to reduce the cost to select k pivots, and to update the trailing matrix in a block algorithm. Although, its overhead is negligible, $O(k)$ FLOPS, where k is a small integer, the gain in these two steps is much larger. Experimental results, made on a set of 261 matrices, present for CARRQR a maximum gain of a factor of 6 for selecting k pivot columns by Tournament Pivoting (QRTP), and a gain of a factor up to 32 for update the trailing matrix. In addition, we have made tests where the tolerance τ varies from $1e-3$ to $1e-9$. The results show that the theoretical bound on R_{22} is never reached. All the results validate that our modification of the QR factorization with Tournament Pivoting can be reliable to compute a low-rank approximation as well as to approximate the singular values, with an unchanged error on the approximation of the singular values.

As future work, we have to make a parallel code to test on larger matrices the impact of approximating the 2-norm of A, and to obtain larger speedups. Also, the fixed-precision approach can be used to improve the performance when a fixed-rank is requested by the user. Consider that the discarded columns are sorted such that the columns that are nearly dependent are moved to the leading positions of these columns. For example, these columns could be grouped in two subsets so that the first subset contains columns that lead to have $\tau \in [10^{-1}, 10^{-3}]$, and the remaining columns in another subset, where $\tau < 10^{-3}$. Thus, the algorithm defines a first tolerance $\tau = 10^{-1}$, and factors A using it. If the fixed-rank is not reached, the tolerance is decreased so that the columns in the first subset are updated as they should have been if they were kept in the trailing matrix. Then the main algorithm resumes until the condition of fixed-rank is reached. This approach is expected to give good performance.

Chapter 6

C Parallel Linear Algebra Memory Management library

Outline of the current chapter

Introduction

In Linear Algebra, matrices and vectors are the most basics elements. In Computer Science, their representation depends on many aspects such that their purpose or their properties. For example, a sparse matrix is managed differently from a dense matrix. In the case of parallel execution, the distribution of the data impacts the performance of the execution. From a mathematical point of view, it is more convenient to use a higher level of abstraction. This allows us to be closer to the algorithm instead of taking care of some software and hardware aspects as pointers and storage. The purpose of this library is to provide a user-friendly interface that helps to develop a parallel code.

This library contains all the materials used to develop the parallel codes that are presented in this thesis. Note that the codes presented in Chapter 3 to 5 are gathered into another library named *preAlps*. The construction of the library follows some constraints. The source code is written in C and is independent of external packages, except MPI library. The installation of the library uses a bash script and generates makefiles. Therefore, the only requirement to use it is to have a C compiler, bash, Makefile, and an MPI library. The management of the memory in C is well known to easily generate errors. We have designed a small module, named MemCheck, that aims to reduce these errors. Linear algebra operations complete the purpose of the library and leads to having a *C Pa*rallel *L*inear *A*lgebra *Me*mory *M*anagement library. We present, in the first section, the datatypes that represent matrices and vectors. To deal with problems of memory allocations, the second section describes the creation of these datatypes. In addition, we present the module MemCheck. Its purpose is to register all allocations and deallocations of the memory and warns the user for unexpected behavior as a not freed block of memory, for example. Then, we detail basic routines that manipulate the mathematical objects that we consider in the library. For a specific purpose, we design a customized collective communication module that is used in our implementation of TSQR. In addition, linear algebra routines, already well optimized in LAPACK or MKL, are wrapped and presented in the next section. These routines are considered here as kernels for developing parallel routines as TSQR or Cholesky QR for instance, but also a parallel Sparse Matrix dense Matrix multiplication. The last section presents interfaces that we use to call external packages from the library.

6.1 Representation of mathematical objects

The library proposes a basic representation of a vector and of a matrix. On the one hand, sparse and dense matrices are considered separately with their associated routines. On the other hand, vectors that contain integer are distinct from the vectors of real. These two types of matrices and two types of vectors are flexible enough to be used in parallel as well as in sequential. Except for few routines, the library considers the mathematical objects from their local point of view. It means for example that there is no global vector and so no global consistency of the size as in Petsc. Instead, we manipulate locally part of the vector. We warn the reader that we only consider real matrices, and integer or real vectors for the moment. We use the 0_based index for arrays unless otherwise mentioned and provide conversion routines for external use.

6.1.1 Mathematical object: matrix representation

In this section, we present two datatypes of matrices. At first, we focus on dense matrices where all elements are involved in the computation. Then, we present our implementation of a sparse matrix, based on Compressed Sparse Row format, denoted hereafter as CSR. By definition, a sparse matrix is a matrix where a large part of its elements are zeros. For performance, sparse

matrices have to be treated differently. In that case, sparse matrix representation saves memory consumption and floating point operations. For the purpose of this section, we consider a general real matrix $A \in \mathbb{R}^{m \times n}$ where a_{ij} in A corresponds to the entry at the intersection of row i and column j of the matrix.

MatDense: the dense matrix representation

To manipulate dense matrices, we define a datatype mat_dense_t that we further call MatDense. This datatype is composed of a 1D_array and a set of data that stores information relative to the matrix. The 1D array contains all entries of the represented matrix. The set of data, denoted Info_Dense_t, is a structure that stores the dimension and the number of entries in the matrix. In the parallel case, a MatDense matrix is considered as part of a distributed matrix. To record it, Info_Dense_t also contains global dimensions. Elements in the 1D array are either an ordered column concatenation that corresponds to the standard *column major*, or an ordered row concatenation which corresponds to the standard *row major*. This representation requires to provide a *leading dimension array* which allows us to reach from an entry $a_{i,j}$ either the entry in the next column $a_{i,(j+1)}$ or the entry in the next row $a_{(i+1),j}$, by adding the leading dimension to the current position of $a_{i,j}$ in the 1D array, depending on the major used. Therefore, Info_Dense_t contains, in addition, the major and its relative leading dimension.

In Listing 6.1, we present our implementation of a dense matrix. This representation allows us easily to interface with dense linear algebra libraries like LAPACK (Anderson et al., 1990).

```
typedef struct {
  double* val;
  Info_Dense_t info;
} Mat_Dense_t;

typedef struct {
  int M;                                          /*Global number of rows*/
  int N;                                          /*Global number of cols*/
  int m;                                           /*Local number of rows*/
  int n;                                           /*Local number of cols*/
  int lda;                                  /*Leading dimension of the matrix*/
  int nval;                                                        /*m*n*/
  storage_type_t stor_type;                /*Row Major or Column Major storage*/
} Info_Dense_t;
```

Listing 6.1 – Extract from mat_dense.h

MatCSR: the CSR sparse matrix representation

When a matrix has many zeros, it is more efficient to consider a sparse storage. For that, several sparse storages exist. Among all existing storages, we present the three most common. The first storage considers the coordinate of the nonzero entry in A. This format, denoted *COO*, for coordinate, stores each nonzero as a tuple (i, j, v), where i is the row index, j is the column index and v is the value of the entry a_{ij} in A. The second storage is the Compressed Sparse Row, denoted *CSR*. It stores only the nonzero entries and compresses the information on the row. This storage is composed of three arrays: *rowPtr*, *colInd* and *val*. The array rowPtr, of size $m + 1$, indexes the position of the first nonzero entry of each row in both arrays colInd and val. Thus, the first element in the array rowPtr is 0, since we use 0_based indexing, and the last element of the array rowPtr is equal to the number of non-zeros in A. The array colInd stores the column index of each nonzero whereas the array val stores the corresponding entry, both of size

nnz. The last storage is the Compressed Sparse Column, denoted CSC. Its purpose is similar to the CSR format. Although the CSR format compresses the row, the CSC format compresses the columns. The three arrays representing the matrix are *colPtr*, *rowInd* and *val*. Both compressed storage aims to remove redundant indexes in rowPtr and colPtr, respectively. All these storages present advantages and disadvantages depending on which operation we would perform. Among these storages, the library only considers the CSR format. Each nonzero entry in a matrix A is added into the CSR format such that for each entry in row i, its associated column index is added into the array ColInd, its value is added into the array val and the $(i+1)$'th entry in rowPtr is equal to the i'th entry plus the number of non-zeros entry in row i.

In Listing 6.2, we present our implementation of the CSR storage. We define a datatype mat_CSR_t, denoted further as MatCSR, that contains the three arrays detailed above and a set of data, similar to mat_dense_t and denoted Info_t. This last set contains the dimension of the matrix $m \times n$ and its number of non-zeros $lnnz$. Similar to our representation of a dense matrix, we record the global dimension of the global matrix, $M \times N$, its total number of nonzero, nnz. In addition, we keep in the structure whether the matrix is symmetric or not, and if the matrix is stored in block CSR, denoted further as BCSR, then the size of the block. Since our representation is standard, we can easily interface it in third-party libraries like SPQR from SuiteSparse (Davis, 2011a), Petsc (Balay et al., 2016), or even MKL (with few modifications like 1_based index if needed).

```
typedef struct {
  Info_t info;
  int* rowPtr;                          /*A pointer to an array of size M+1 or m+1*/
  int* colInd;                         /*A pointer to an array of size nnz or lnnz*/
  double* val;                         /*A pointer to an array of size nnz or lnnz*/
} Mat_CSR_t;

typedef struct{
  int M;                                                    /*Global number of rows*/
  int N;                                                    /*Global number of cols*/
  int nnz;                                                /*Global non-zero entries*/
  int m;                                                      /*Local number of rows*/
  int n;                                                      /*Local number of cols*/
  int lnnz;                                                /*Local non-zero entries*/
  int blockSize;                                                /*Local block size*/
  Mat_CSR_format_t format;                    /*Local storage format :  block or not*/
  Struct_Type structure;                     /*Local symmetric or non symmetric pattern*/
} Info_t;
```

Listing 6.2 – Extract from mat_CSR.h

Remark 35. ***Block CSR.*** *Some applications have a pattern including small dense matrices (for example coupled unknowns). In that case, we can use a block representation. The matrix is still sparse but the nonzero entry in the BCSR storage corresponds to a small dense matrix. Note that we do not provide routines to be used with a block CSR storage.*

Remark 36. ***Symmetric structure*** *Concerning the symmetry of the structure of A, we manage symmetric matrices as unsymmetric. Note that in the case of the construction of the communication dependency pattern in a matrix vector product, we avoid an initial communication step. The communication dependency pattern is deduced from the local pattern.*

6.1.2 Vectors: two types of representation

When solving, for example, a linear system $Ax = b$, or computing the residual $r = Ax - b$ in iterative methods, a representation of the vectors x and b with real values is required. In

addition, permuting a matrix, getting a list of neighbors like in RAS, getting a subset of rows to distribute require a representation of vectors with integer values. From these observations, we choose to design two types of vectors: IVector_t and DVector_t for integer and double values, respectively.

```
typedef struct{
  double *val;
  int nval;
} DVector_t;
```

Listing 6.3 – Extract from DVector.h

```
typedef struct{
  int *val;
  int nval;
  int size;
} IVector_t;
```

Listing 6.4 – Extract from IVector.h

DVector: a simple representation of a real vector

Here we summarize our definition of DVector. Its structure, presented in Listing 6.3, is composed of a 1D array of doubles, referred to as *val*, and its number of elements, referred to as *nval*. This representation simplifies its use and also it aims to solve some problems of segmentation fault or buffer overflow. To ensure that, we provide a list of routines that we present in the following.

IVector: a double use of integer space

Integer arrays allow us to store different types of data as permutation matrices, list of adjacent vertices. Their use is wide enough and we propose to use it for two different purposes. Thus the IVector can be used as usual since val is a 1D array and nval the number of its elements. For a different purpose like getting adjacent vertices of several set of vertices, we offer the possibility to use IVector_t as workspace. To do that, we add a third parameter called size. The behavior is the same as a standard IVector and even DVector but we can change nval without loosing the amount of the space we have in the vector. For example, in Listing 6.5,

```
//Declaration and initialization
Mat_CSR_t A = MatCSRNULL();           //Sparse matrix
IVector_t isAdjacent  = IVectorNULL();//A boolean vector
IVector_t adjacent    = IVectorNULL();//Seen as a workspace
int nvertex = 10;

IVectorCalloc(&isAdjacent, nvertex);  //Calloc the memory space
IVectorMalloc(&adjacent, nvertex);    //Malloc the memory space

//Assuming this creates a random sparse matrix A
bar(&A, nvertex);

for(int block = 0; block < nblock; block++)
{
  //Function that fills the isAdjacent array
  foo(block, &A, &isAdjacent);

  adjacent.nval = 0;  //Here the size is still 10
  for(int i = 0; i < isAdjacent.nval; i++)
  {
    if(isAdjacent.val[i] > 0)
    {
      adjacent.val[adjacent.nval++] = i;
      isAdjacent.val[i] = 0;
    }
```

```
    }
    printf("Block %d has %d adjacent vertices in A\n",
        block,
        adjacent.nval);
    IVectorPrintf("Adjacent list :", &adjacent);
}
```

Listing 6.5 – Use IVector as workspace

the vector *adjacent* is seen as a workspace allocated only once. Then at each iteration, its number of values is reset to 0 and so it can be easily reused.

Remark 37. *Moving from a standard IVector to a workspace use, in Listing 6.5, involves only few modifications but allows better performance by obviously reducing the number of allocations. Also, for the moment there is no similar use of DVector but it can be easily done if needed.*

The four datatypes presented here represent the basic datatypes of this library. We then provide routines to allocate and manage them.

6.2 Memory management and code development

In this section, we focus on one of the two main aspects of the library: memory management. Using C language requires using mechanisms of allocation of the memory as malloc. This can also lead to some problems like out-of-bound of the memory space, memory leak, double-freed corruption. To avoid such problems, the four datatypes of the library have creation and destruction routines used to record the state of the datatypes. We first present these routines, also aiming to simplify the development of a code, and then we detail a tool called MemCheck to track down several unexpected behaviors.

6.2.1 Creation, visualization, and destruction of objects

One of the most important aspects concerns the allocation/deallocation of the memory. For that, we provide two main ways of creations of objects. The most basic routine creates an object by allocating the memory space needed and copying the content therein. To do it, we provide wrappers of each basic allocation: malloc, calloc, realloc. In addition, we also wrap the free routine. Thus the allocation of a MatDense is performed by MatDenseMalloc and the memory is freed by calling the MatDenseFree. We will show later that from these we can track part of the problems listed above. The library can manipulate data structures coming from an external library. To do so, the routines named XXXCreateFromPtr create the object XXX but does not allocate the memory. Instead, it points to the block memory space given as a parameter of the function. The only constraint is to use the same storage. For example, the storage of a CSR matrix uses three arrays. Any other storage may not be compatible with it. Point to a block leads to manipulating the data that we do not own and applying one or several of our routines on it. It offers the flexibility to avoid two copies: one to get the data to treat it and eventually another copy to get back the data. Note that in the remaining part of this chapter, we denote XXX as one of the four datatypes used in the library.

In order to visualize the content of the memory, the routines of the form XXXPrint display the data in a convenient way. For example, calling MatCSRPrint displays a MatCSR in its CSR format (each of the three arrays) whereas MatCSRPrint2D displays the content as a dense matrix by adding to the output explicit zeros. Another example is about IVector that can be seen as a classical vector of integer or as a workspace of integers. The routine IVectorPrint displays

the first *nval* elements. The routine IVectorPrintWork displays the content of the memory associated with the IVector, without taking into account the number of elements in it. Since for some applications the datasize can be large, these routines display a subset of the content of the object which is for instance the first and the last 10 values (defined at compile time). Obviously, explicit full printable versions exist like IVectorPrintFull and IVectorPrintFullWork. Moreover, since it is convenient to join a message when the variable is displayed, an 'f' version of these routines exists (MatCSRPrintf, IVectorPrintfWork, ...).

6.2.2 MemCheck: module of memory tracking

We propose a light tool to track the memory use of the library and is general enough to be used elsewhere. This tool allows us to check at the end of the execution if all memory blocks allocated by the library are freed. It also tracks an object from creation to its destruction. To do so, we define a MemBlock_t structure, denoted after as MemBlock, that stores useful information for different uses and is presented in Listing 6.6.

```
typedef struct memBlock{
  char *p;                                          /*Pointer to the memory allocated*/
  const char *varName;                                /*Name of the variable or NULL*/
  const char *file;                        /*File where the function calls the allocation*/
  int line;                              /*Line in the file where the allocation is called*/
  size_t size;                                 /*Size in bytes of the current allocation*/
  struct memBlock *next;
#ifdef MPIACTIVATE
  double time;                         /*Current time when the memory allocation is called*/
  double time_free;                          /*Current time when the memory free is called*/
#else
  time_t time;                         /*Current time when the memory allocation is called*/
  time_t time_free;                          /*Current time when the memory free is called*/
#endif
  size_t total_size;                             /*Total memory size in bytes allocated*/
  size_t total_size_free;                          /*Total memory size in bytes freed*/
  char *stackCurState;                               /*The current stack trace state*/
} MemBlock_t;
```

Listing 6.6 – Extract from cpalamem_instrumentation.h

In MemBlock, we store the pointer relative to the block and its size. We also record when and where this block is created. Along with the time of creation denoted *time*, we store the amount of memory used at that time by the library in *total_size* and from which file and at which line this creation is requested. For further use, we store the clock when this block is freed and what the current amount of memory allocated is at that time. To complete the description, we save the list of caller routines, as a stack trace in Java or Python, that leads to this call of creation/destruction, in *stackCurState*. Note that for parallel codes, we have a test of using MPI library represented by `#ifdef MPIACTIVATE`. If CPaLAMeM is compiled with MPI, which is the standard case, we use MPI_Wtime routine to get the time. We also offer the possibility to use it without MPI by replacing this routine by time from time.h.

MemBlock is constructed as a stack and we iterate over it to find a specific block referenced by the pointer address or insert a new block on top of it. All malloc and calloc operations are wrapped by a CPALAMEM_malloc and CPALAMEM_calloc, respectively. These wrappers create a new MemBlock, allocate the memory space required by the initial call and set all related information to the block.

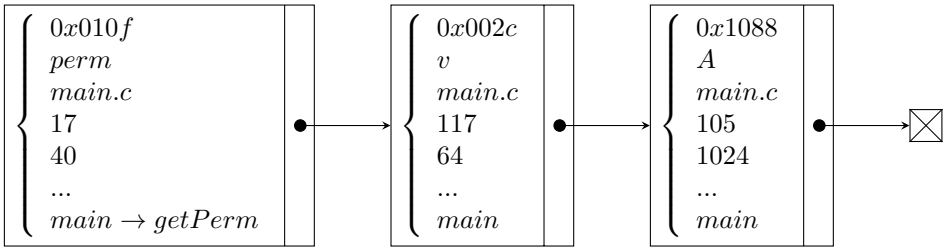


Figure 6.1 – Snapshot of the MemBlock structure related to a code that creates a variable called A of 1024 bytes at line 105, then requests a block of 64 bytes labeled v at line 117 and then goes to getPerm routine to creates at line 17 of the same file (main.c) a *perm* of size 40. Additional information in the structure is represented by the '...' line.

As shown here, there is no precision if it is a calloc or a malloc call. The only interesting aspects here are the memory use and a way to track these blocks from their creation to their destruction. When a realloc, wrapped with CPALAMEM_realloc, is requested, we choose to create a new entry in the MemBlock and considered the block to realloc as a free of it and a malloc of a new one. CPALAMEM_free wraps the classical free routine. To illustrate the internal behavior, consider the instruction `free(ptr);` written in a code. It is replaced by the routine `CPALAMEM_free(ptr, __FILE__, __LINE__, NULL);`. This routine searches *ptr* in the stack of MemBlock. If it is included, *ptr* is freed by calling the basic routine. Then the corresponding MemBlock field p and *ptr* are set to $NULL$. The amount of memory used is reduced by the $lSize$ of the MemBlock and we record the clock when free is requested. Otherwise, an error message is displayed to warn the user that he tries to free a block *ptr* in the file $\_\_FILE\_\_$ at line $\_\_LINE\_\_$ that is not recorded. Note that in the second case, the file and line information displayed are relative to the call of the free and not to the location of the creation of any block.

At the end of the execution, a routine can be called to iterate over all elements of the stack MemBlock and prints a warning message for each element of the stack whose p is not null. This mechanism catches all memory leaks relative to CPaLAMeM we can have in the code.

Remark 38. *This module is designed to be generic such that it can be used in other codes. For that, it suffices to include cpalamem_intrumentation.h header in every file where the memory has to be tracked. Note that this module is activated only if the file is compiled with the macros MEMCHECK and MEMCHECKCONSUMPTION.*

Weak pointers and workspace

As in LAPACK library, we sometimes prefer to use a workspace for better performance. A workspace is a standard memory space allocated for several different goals. It allows us to allocate only once the memory. Unfortunately, this technique may lead to overlapping two memory spaces sharing the same workspace. These memory spaces are represented by two variables, one stores the size and another one points to the first element. We introduce the definition of a shallow pointer

Definition 39. *A shallow pointer is a C pointer which does not own the memory space that it points to.*

Thus it should not be used to free the memory space pointed to. Extending this definition, we further call *shallow* any data structure that uses a shallow pointer that points to it. To illustrate, we use in our implementation of TSQR (in its rank-revealing form) several shallow pointers for

the permutation, the Householder vectors, the different tolerances associated. Therefore, the permutation vector represented by an IVector is seen internally as a shallow IVector.

To handle the overlapping problem, we develop a small module. It tracks each shallow pointer related to a workspace and checks if overlap occurs. For that, we define a recording structure ShallowElem_t of shallow pointer presented in Listing 6.7 and referred hereafter as ShallowElem where *name* labels the shallow pointer, $lSize$ stores the size of the memory space occupied, and p points to it, seen as a void pointer.

```
typedef struct{
  const char *name;
  size_t lSize;
  void **p;
}ShallowElem_t;
```

Listing 6.7 – Extract from shallow.h

To simplify the use of the module, we provide a few routines. First, we create the module environment Shallow_t with the routine `ShallowCreate` taking as parameter the initial number of ShallowElem recorded. This routine creates an array of ShallowElem. The memory space considered is set up using `ShallowInitRef` routine. This routine takes as input a name, used as a label, the size and the address of the memory space. These last two parameters define the limit of the memory space considered and are used for tests. To attach shallow pointers to the memory space, we call `ShallowAdd` routine. This routine takes as input the number of shallow pointers to add in the array and for each, three variables to set up each ShallowElem_t element in the same order as the definition given in Listing 6.7. Then the array is sorted according to the address of the shallow pointer. When the module is set with all shallow pointers, the test routines are called to check whether overlaps or out of bounds occur. Note that for general purpose, a ShallowElem uses `size_t` and `void **` types. In other words, for a shallow MatDense, lSize has to be set to `nval * sizeof(double)`. Also, each shallow pointer is cast into a `void **`. It follows that if a shallow pointer, attached into a Shallow_t, points to another memory space, we are still able to determine if it overlaps another memory space. However, we choose to not track a change of size in the memory space pointed by the shallow pointer. At that point, the environment of the module is set.

To test overlap, we are calling `ShallowIsOverlapped`. This checks for each p_i if $p_i + lSize > p_{i+1}$ and prints whether the subspaces overlap in the workspace. The sorted list involves $O(n)$ comparisons where n is the number of shallow pointers recorded. To test out of bounds, `ShallowIsOutOfBound` routine can be used. This routine checks for each shallow pointer whether the first or the last byte is outside bounds of the workspace. Also, the workspace and its shallow pointers can be displayed by using ShallowPrintState routine.

6.3 Basic routines to manipulate datatypes

In this section, we describe the basic routines that can be used to manipulate matrices and vectors. These routines offer an easy interface and intend to be optimized enough. We first focus on the MatCSR object and present routines that are grouped into three different groups. The first group is composed of routines that create objects like loading a matrix or extracting a block from a matrix. The second group contains routines that operate on the object like copying or copying. The last group addresses the communication aspects.

6.3.1 Parallel distribution with the extraction of blocks

Since a matrix is generally distributed in a 1D row layout, a 1D column layout and even a 2D (cyclic), it is important to understand how this is taken into account in the library. Note that for the following description we ignore the 2D-cyclic case and we mention that the presentation of the column layout distribution is similar to the row layout distribution, modulo a transposition. We consider the problem of distributing a matrix $A \in \mathbb{R}^{m \times n}$ over p processors using 1D row layout. We split the rows into blocks having roughly the same number of rows and store the position of the first row of each block in an array of size $p+1$, denoted *pos*, represented hereafter by an IVector. The vector *pos* has its first element equal to 0 its last element equal to m (n, in the column layout case). For a 2D distribution, we can create two IVector $posR$ and $posC$. Note that for a symmetric distribution of a matrix, only one vector, as *pos*, is required.

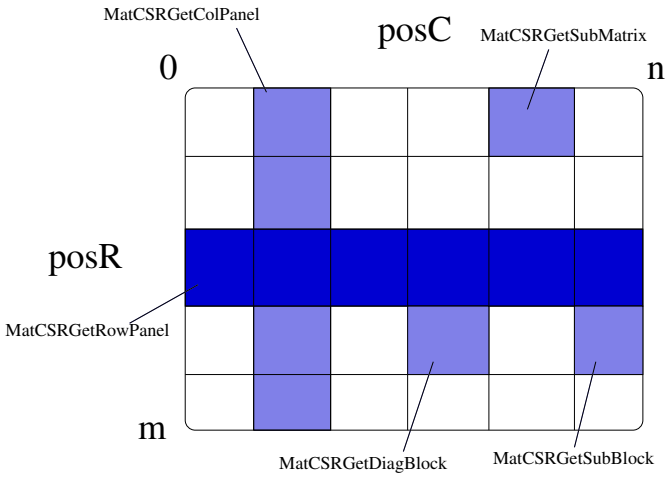


Figure 6.2 – 2D distribution of a matrix A over p processors. Here posR is a IVector indexing the beginning of each row block and posC the beginning of each column block.

Load the matrix

In this library, we provide several ways to load a sparse matrix stored in different formats. One of the common storages is the Matrix Market storage (Boisvert et al., 1996) (.mtx extension files) where only the nonzeros are stored using the COO format. As a remark, each .mtx file has a banner where it is written in particular if the matrix is symmetric or general. We do not deal with symmetric matrices. Thus a symmetric matrix is converted into a general one. We provide the following three routines to load a matrix.

- `LoadMatrixMarket` loads a matrix on a single core. It takes only the filename and returns the loaded matrix. This routine reads the file in .mtx format and converts it into a MatCSR datatype. In the case of symmetric matrices, for each element $a_{i,j}$ read from the file, the routine also adds the symmetric element $a_{j,i}$ into the structure.

- `LoadMatrixMM1D` loads a matrix in parallel and distributes it using 1D layout. It takes the filename that stores the matrix, a constant char to set the distribution ('R' for row layout or 'C' for the column layout) and a communicator. This routine does not assume the .mtx file is sorted. Therefore the file is read by a single processor and the data is split into blocks of size nnz/p. Every block is then sent to another processor. In that way, the

processor reading data stores at most two blocks. Then processors sort their own data, identify which data has to be sent and then a global communication phase exchanges data. This leads to gathering on each processor their own part of the matrix. Note that here, the distribution is first well-balanced in term of number of nonzeros and then in term of rows per processor (columns if 'C' is given as a parameter). This function returns a distributed MatCSR where A is its local representation.

- `LoadMatrixMM2D` loads a matrix in parallel with a 2D distribution scheme on a $p_r \times p_c$ grid of processors. It takes as input the filename where the matrix is stored. The number of processors is deduced from the communicator, also given as a parameter. The dimensions of the grid of processors $p_r \times p_c$ are also provided as parameters of the routine. Here the number of processors p_r for rows and p_c for columns are required. From the communicator and these two parameters, the routine first creates sub-communicators for rows and columns that are returned at the end. Secondly, the routine uses the 1D version as a subroutine with a column distribution. Finally, each column panel is split into p_r pieces and A_i, the local part of the loaded matrix is returned to processor i.

In the parallel case, each processor deduces the *pos*, *posR* or *posC* IVector which describe the distribution of A.

Extract part of the matrix

Loading a matrix is performed either in parallel or in sequential. However, it can be convenient to have a split matrix. The user may impose the splitting using a vector *pos*, as for distribution. One can call the k-way partitioning routine (Karypis et al., 1999). Our library provides an interface to call k-way partitioning that returns a partitioning vector (further detailed). This vector can be converted into a permutation vector or used to distribute the matrix. We assume that a vector *pos* that describes the splitting of the matrix is given. Based on *pos*, we provide several routines to extract part of a MatCSR. First, we consider the extraction of a row panel (a set of consecutive rows from A) using MatCSRGetRowPanel. This takes as parameter the matrix A, the vector *pos* which is the splitting of the rows of the matrix and the index of the panel to extract from (starting with 0). Note that, at least *pos* has to be an array with the first row index and the last one +1, and the panel index is 0 in that case. We also have the equivalent routine: MatCSRGetColPanel. This uses the same input as the row version plus one extra parameter referred to as $colPos$.

To improve performance, we use a vector $colPos$ variable. Its purpose is to index the beginning of each column block for each row. In other words, $colPos$ is a $rowPtr$ array, defined by the CSR format, with inserted values between the original ones. We call it a dilution of $rowPtr$. In a sense, $rowPtr$ indexes the column block 0. Let A be a 3×3 matrix split into two blocks and consider its CSR representation,

$$A = \left(\begin{array}{cc|c} 1 & 0 & 2 \\ 1 & -2 & 1 \\ 0 & 4 & 1.5 \end{array}\right) \quad A_{CSR} = \left\{\begin{array}{ll} rowPtr & = [0, 2, 5, 7] \\ colInd & = [0, 2, 0, 1, 2, 1, 2] \\ val & = [1, 2, 1, -2, 1, 4, 1.5] \end{array}\right.$$

We create two column blocks, where the first two columns form the first block and hence $pos = [0, 2, 3]$. In that case, the $colPos$ associated with it is $[0, \underline{1}, 2, \underline{4}, 5, \underline{6}, 7]$ where all underlined numbers are inserted into the original $rowPtr$ array. Thus to get the second block, the routine uses the underlined elements of the vector $colPos$ as starting index in the matrix, for each row. Thus, using $colPos$ avoids tests on the values in $colInd$ to determine the beginning and the end of the block for the current row. All this procedure is in MatCSRGetColBlockPos routine that takes

as input A, a MatCSR variable, and pos, the splitting of the columns of A, and returns $colPos$ indexing the whole A. Note that for a slightly different purpose, we also provide a partial version called MatCSRGetPartialColBlockPos, where here $colPos$ indexes only a part of A. This routine takes as input the matrix A and the vector pos, here denoted $posC$. In addition, it also needs the splitting of the rows of A and the index of the row block considered, to return a partial $colPos$. This variable is important to extract part of a MatCSR. Following the same idea of optimization, we have a routine to get an array indexing the diagonal elements of A: MatCSRGetDiagInd. Along with this routine, we have a partial version named MatCSRGetDiagIndOfPanel which takes pos, the splitting of the rows of A, $colPos$ and a row block index. This is useful for example when we have a local row panel like the blue one in 6.2 and we need to get the lower part.

In order to get a block of a MatCSR, we also have MatCSRGetDiagBlock to extract the diagonal block. And more generally, to get a particular block located in position (i, j), intersection of the i'th row panel and the j'th column panel, we have two routines MatCSRGetSubMatrix and MatCSRGetSubBlock. They only differ from the column distribution. On one hand, the first routine requires pos, $colPos$ and the indexes i and j. It also assumes that the column distribution is equal to the row distribution if $colPos$ is not set. In that case, this routine should be used for symmetric splitting. On the other hand, the second routine takes explicitly $posR$ and $posC$ and creates internally the $colPos$ associated using the partial representation and then it is destroyed. Note that these routines may need a workspace and if given in parameter, it avoids internal allocation.

Finally, the routines presented above assume the data is contiguous. We also propose routines to extract a MatCSR from sets of indices as even rows and odd columns. To do so, we provide two routines. The MatCSRGetSubFromVector routine takes as input the matrix A and an IVector as a selector of rows, and returns the row block that contains the concatenation of the row indices in the IVector. The second version, MatCSRGetSubFromVectors, takes as input the matrix A and two IVector: one for the rows and one for the columns. This routine returns the corresponding subset of rows and columns, from the indices of the two IVector given as parameter, respectively.

6.3.2 Operations on MatCSR

In this section, we consider the matrix is in memory. During development, it may be useful to compare two matrices, to extract or to check some properties of a matrix. For that purpose, we provide routines to make equality tests using either MatCSRIsEqual or MatCSRIsAbsEqual. In addition, we can check if a matrix is symmetric or at least if the pattern is symmetric. More generally, we provide a list of routines to perform some operations on A:

- `MatCSRUnsymStruct` transforms a symmetric matrix, stored with only half of the entries, into a matrix that can be treated as a general matrix.

- `MatCSRSymStruct` takes as input a non-symmetric matrix and returns $A^T + A - 2D$, where D is the diagonal of A. This routine is used for calling k-way partitioning routine, from Metis.

- `MatCSRPermute` needs as parameters a MatCSR and two vectors $rowPerm$ and $colPerm$. The first one concerns the row permutation and the second one the column permutation. It also offers the possibility to permute only the pattern in order to reduce the cost of the routine. The algorithm starts by permuting the rows using $rowPerm$, then permuting each line using the invert of $colPerm$ and finally sorts $colInd$ line by line. Note that for further performance we decide to sort the column indices, but it is not a requirement in general.

- `MatCSRAddExplicitZeros` adds explicit zeros to a matrix A. In some cases, it is more convenient to have an augmented pattern and just to perform some numerical operations as LU algorithm. This routine takes a matrix A and an augmented pattern filled-in with zeros, and copies all values of A into the new pattern.
- `MatCSRGetLUFactors` uses a diagonal index array (returned by MatCSRGetDiagInd) to extract L and U from a matrix A.
- `MatCSRCopy` copies the entire matrix into a new one.
- `MatCSRSave` saves into a .mtx file the local part of the matrix given as parameter.

Without going into details, we also have routines to convert a MatCSR into a DVector or a MatDense and vice-versa.

6.3.3 Communication

To ease the manipulation of a matrix in MatCSR format, we first define a specific datatype derived from MatCSR which is used to reduce the communication. Based on MPI, we create two MPI_datatypes: one for MatCSR and one for MatDense. Thus a matrix is sent using MatCSRSend and received through MatCSRRecv. The matrix communication scheme is split into two steps: send the information Info_t of the matrix and then send the data. The receiver reads the content and allocates using MatCSRMalloc if needed and then is ready to receive the three arrays. To illustrate, we present Listing 6.8 where the matrix A is loaded on one processor, then Kway algorithm is called on it and the matrix is distributed according to the Kway algorithm.

```
int main(int argc, char** argv)
{
  Mat_CSR_t matCSR = MatCSRNULL();
  int rank  = 0;
  int size  = 1;
  int ierr  = 0;
  char  matrixFileName[]= "../TestDir/cage4.mtx";

  CPALAMEM_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

                                                    /*Load on first process*/
  if (rank == 0)
  {

    Mat_CSR_t matCSRPermute = MatCSRNULL();
    Mat_CSR_t rowPanel      = MatCSRNULL();
    IVector_t posB   = IVectorNULL();
    IVector_t perm   = IVectorNULL();
    int nblock = size;

    ierr = LoadMatrixMarket(matrixFileName,
                            &matCSR);CHKERR(ierr);

    ierr = metisKwayOrdering(&matCSR, &perm, nblock, &posB);CHKERR(ierr);

    ierr  = MatCSRPermute(&matCSR, &matCSRPermute,
            perm.val, perm.val, PERMUTE);CHKERR(ierr);
```

```
      MatCSRFree(&matCSR);
      matCSR  = matCSRPermute;

                                                    /*Send submatrices as row panel layout*/
      for(int dest = 1; dest < size; dest++)
      {
        ierr = MatCSRGetRowPanel(&matCSR, &rowPanel, &posB, dest);CHKERR(ierr);
        ierr = MatCSRSend(&rowPanel, dest, MPI_COMM_WORLD);checkMPIERR(ierr,"Send
             rowPanel");
      }

      MatCSRFree(&rowPanel);
      IVectorFree(&posB);
      IVectorFree(&perm);

    }
    else                                /*other MPI processes received their own submatrix*/
    {
      ierr = MatCSRRecv(&matCSR,0,MPI_COMM_WORLD);CHKERR(ierr);
    }

    MatCSRPrintf2D("Local rowPanel", &matCSR);

    MatCSRFree(&matCSR);

    CPALAMEM_Finalize();

    return 0;
  }
```

Listing 6.8 – Extract from loadMatrixUsingKway.c example

For some purposes, it could be useful to have the communication dependencies, especially for a matrix multiplication operation. For that, we develop the MatCSRGetCommDep routine. This takes in parameter *colPos*, the number of rows, the number of blocks, and the index of the block concerned. In return, we have an array containing the list of dependencies for the block, which is usually the diagonal one.

6.4 Customized collective communications

In this section, we present a customized collective communication module that we are using in our implementation of TSQR for instance. At first, the idea is to provide a general tool to handle trees and to be general enough such that it can be used elsewhere. The construction of the tree is deterministic, which means that the pattern used is reproducible and so it is helpful for debugging.

We denote this module AbstractTree. Note that we also have a simpler version of it called simpleAbstractTree but we do not refer to it in this document. The module mainly manages the communication. Therefore it requires routines to send and receive and two kernel functions applied either before sending the data or after receiving it. The sketch of development is to provide both fsend and frecv routines with the signature `(void*,int,int,MPI_Comm)`. These routines take a void pointer to the data that the developer wants to manipulate, the destination/source and the tag of communication, and end with the communicator.

Remark 40. *In the current version of abstractTree, the tag parameter is all the time 0. It may be convenient to change it for further applications.*

Then the developer may provide a fkernel of signature `(void*)`. This routine takes the same void pointer to the data as fsend and frecv. With these routines, we can call any tree by just focusing on the way to send and to receive and what operations to do so.

Going into the details, we first present the internal structure of the module named TreeVoid_t in code 6.9 and then the routines to manipulate the tree. The structure is split in three parts. First, it contains the real data to manipulate, provided by the developer, here denoted as *userData*. The second part takes care of the communication aspects by storing a list of processor indexes in *src* and *dest* and their size *nlvl*. This is related to the communication scheme used like a binary tree. Then the MPI variables handle the non-blocking communication with requests and status for both send and receive aspects. The last part stores the function pointers: an fsend and an frecv function plus two kernel functions. The first kernel *kernelS* is used before sending data and the other kernel *kernelR* is called after receiving data. The last variable *commScheme* is a function pointer of prototype `(*fcommScheme) (int,int,int**,int**,int*)`. This routine takes as input the rank of the processor and the size. It returns two arrays *src* and *dest* plus their size assimilated to the number of levels in the tree *nlvl*. We provide few `fcommScheme` routines to construct a binary tree with `binCommunication` or a binary butterfly tree with `butterflyCommunication`.

```
typedef struct{
  /*DATA*/
  void  *userData;                                  /*This variable is set by the user*/

  /*DATA COMMUNICATION*/
  int *src;
  int *dest;
  int nlvl;
  int nrequestR;
  int nrequestS;
  MPI_Comm comm;
  MPI_Request *requestS;
  MPI_Request *requestR;
  MPI_Status *statusS;
  MPI_Status *statusR;

  /*FUNCTIONS*/
  fsend       send;                                                  /*To send data*/
  frecv       recv;                                                /*To receive data*/
  fkernel     kernelS;       /*Called on data at each node/leaf of the tree before send*/
  fkernel     kernelR;        /*Called on data at each node/leaf of the tree after recv*/
  fcommScheme commScheme;                  /*Defines the tree scheme of communication*/
} TreeVoid_t;
```

Listing 6.9 – Extract from AbstractTree.h

We next provide routines to handle easily the tree. Initialized, the tree is created using `TreeCallCreate()`. This routine takes as parameter the tree to create with a $fcommScheme$ function and a communicator. In case we need to change afterward the communication scheme, `TreeCallSetCommScheme` applies a new $fcommScheme$ function to the tree. Next to set up the function pointers requested by the tree, we call `TreeCallRoutines` providing a fsend, a frecv and two fkernel functions. Then to call the tree we use the `TreeCall` routine. When the tree is not required anymore, we call `TreeCallDestroy` to free the memory.

The algorithm of `TreeCall` is presented in 6.1. This algorithm iterates L times. At each level k of the tree, if the k'th destination is not the processor index, then it eventually calls *kernelS* to operate on *data* and then calls the fsend *send* routine to send information to the k'th dest. We deliberately omit the test on *kernelS* but in practice, the developer can provide a null pointer

and so the step 3 is skipped. The symmetric scheme is applied on the k'th source. Note that the algorithm does not take into account how the information is exchanged and which part of *data* is required. All this part remains to the developer. To prevent some problems, we add in the implementation at the end of the algorithm an `MPI_Waitall` routine to handle non-blocking communication routines.

Algorithm 6.1 TreeCall

This function iterates over a tree already set up and calls kernels at each level of the tree.

Input: *data* the userData,
L the level of the tree,
i the processor index,
src the list of processor indexes which send data to i,
dest the list of processor indexes which receive data from i

1: **for** $k = 0$ *to* L **do**
2: **if** $dest(k) \neq i$ **then**
3: Call the *kernelS* routine on *data*
4: Call the *send* routine to send information to $dest(k)$
5: **end if**
6: **if** $src(k) \neq i$ **then**
7: Call the *recv* routine to get information from $src(k)$ into *data*
8: Call the *kernelR* routine on *data*
9: **end if**
10: **end for**

Furthermore, one can provide its own communication scheme. The tree calls it to get the list of senders and receivers. A processor is considered as idle at iteration k if the k'th value in *src* or *dest* is equal to its index. To illustrate, assume a binary tree for 4 processors represented here by *src* and *dest* arrays which are returned by `binCommunication` routine. Then we have

processor id	src	dest	nlvl
0	[1,2]	[0,0]	2
1	[1,1]	[0,1]	2
2	[3,2]	[2,0]	2
3	[3,3]	[2,3]	2

Table 6.1 – Values of arrays *src* and *dest* returned by binCommunication routine for a binary tree representation with 4 processors.

Here, the processor 3 has nothing to receive and sends just once to the processor 2 and then is idle until the end. In the meantime, the processor 0 has nothing to send, and receives data from the processor 1, and then from the processor 2.

6.5 Performance and linear algebra routines

Linear algebra operations are highly used and hence they are well studied. Our purpose is not to reimplement linear algebra routines that are already well optimized in libraries as LAPACK or

MKL. These libraries are sequential, multi-threaded and offer simple interfaces but with many parameters. In this section we present the linear algebra part of our library. Since linear algebra libraries already exist, we first develop our own wrappers to interface our datatypes with these libraries. These wrappers, denoted hereafter as kernels, are gathered in kernels.c file. Based on our kernel routines, we develop parallel routines that are presented in this thesis.

6.5.1 Kernels of the library

We select routines that correspond to our needs, create a wrapper which interfaces our datatypes and hides the long list of parameters. For example, QR on a dense matrix is performed by dgeqrf (in its double non-blocked version) in Lapack library. Although this routine takes precisely 6 parameters, our wrapper, named MatDenseDgeqrf, takes only 2 parameters, the MatDense A and tau. With the same approach, we interface the MKL Pardiso solvers. Handling the full iparam array is done internally and it suffices to call the specific wrapper providing MatCSR and MatDense variables. We also provide two kernels for sparse matrix dense matrix multiplications (SpMM). These functions are based on `mkl_dcsrmm` routine where 15 parameters are requested. This allows us to reduce the number of parameters :

- `MatCSRKernelMatDenseMult` computes $\alpha AB + \beta C$ where A is a MatCSR and B and C are MatDense. Scalars α and $\beta \in \mathbb{R}$ are represented with double. Note that this routine is not optimized for successive calls. Therefore, this routine should be used occasionally to perform the multiplication. Indeed, the MKL routine requires a CSR matrix stored with 1_based index. Hence, to perform the multiplication, A is converted temporarily into 1_based index, then it is multiplied with B and is added to C. Finally, A is restored to its 0_based indexing.

- `MatCSRKernelGenMatDenseMult` performs the same computation as the previous routine. But this function is one level deeper. It assumes that A is already 1_based index and it is not given to the function as a MatCSR. To respect the constraint of the MKL routine, the parameters are the $colInd$ and val array, the number of rows and columns. Furthermore, it requests two $rowPtr$ arrays, denoted $rowPtrB$, indexing the beginning of each row, and $rowPtrE$, indexing the end of each row. In that way, we can multiply a block of the matrix without extraction. This routine is seen as a build-in block of a more general routine.

We next present our implementation of the parallel Sparse Matrix Matrix multiplication.

6.5.2 Cholesky QR implementation in parallel

For some purposes, using Cholesky to compute a QR factorization is sufficient in term of accuracy. Using our matrix multiplication kernel presented above, we can easily implement `MatDenseCholQR`, the Cholesky QR referred hereafter as CholeskyQR. We first describe the algebra of CholeskyQR and then present the implemented algorithm.

Let A be a matrix of dimensions $m \times n$ that we want to factorize to get Q an approximate orthogonal matrix of size $m \times n$ and an upper triangular factor R of size $n \times n$ such that $A = QR$. We first symmetrize A by computing $A^\top A$ which allows to factorize it using Cholesky algorithm. Hence $C = A^\top A$ leads to

$$C = R^\top R, \tag{6.1}$$

and so,

$$Q = AR^{-1}. \tag{6.2}$$

In parallel, the computation of C is performed by first computing, on each processor, the local part C_i and then a global communication sums all C_i. Assuming that A is distributed over p processors using block row layout A_i and A_i is stored on processor i, where $i < p$, we have

$$A^\top A = \begin{pmatrix} A_0^\top & A_1^\top & \dots & A_{p-1}^\top \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix} = \sum_{i=0}^{p-1} A_i^\top A_i = \sum_{i=}^{p-1} C_i. \tag{6.3}$$

Algorithm 6.2 presents the implementation of CholeskyQR. It starts with the computation of $C = A^\top A$, where $C \in \mathbb{R}^{n \times n}$, then its Cholesky factorization $C = R^\top R$ is computed, and finally $Q = AR^{-1}$. Note that the weakness of this algorithm is its accuracy since it relies on $A^\top A$.

Algorithm 6.2 CholeskyQR(A)

This function computes in parallel the QR factorization of a matrix $A \in \mathbb{R}^{m \times n}$ with $m >> n$ using Cholesky.

Input: $i \in \Pi$: processor index that belongs to Π, the set of processor indices,
A_i local block row of A owned by processor i

1: Compute $C_i = A_i^\top A_i$
2: Perform an all reduction communication of C_i to get $C = \sum_{j \in \Pi} C_j$
3: Decompose C using Cholesky factorization $C = R^\top R$ on each processor
4: Compute $Q_i = A_i R^{-1}$ using a triangular solve

Output: R: the R factor of A is duplicated on all processors,
Q: the orthogonal matrix, distributed as A

Its arithmetic complexity is the cost of a dense matrix dense matrix multiplication $\frac{mn^2}{p} + O(mn/p)$ flops plus an allreduction phase of cost $O(mn)$. Adding the Cholesky factorization of C leads to $n^3/3 + O(n^2)$ flops. Finally, the triangular solve involves $\frac{mn^2}{p} + O(mn/p)$ flops. Thus the complexity of CholeskyQR is $2mn^2/p + n^3/3 + O(mn/p)$.

6.5.3 Sparse and dense Tall and Skinny QR factorization and its rank revealing version

We discuss in this section the parallel implementation of the QR factorization of a sparse or dense matrix. The sequential algorithm already exists and is available for example in LAPACK for the dense case and in SuiteSparse for the sparse case. In the dense case, some parallel implementations as in scaLAPACK (Blackford et al., 1997) perform QR with $O(nlog(p))$ messages, which is suboptimal in terms of communication. Therefore, we decide to implement a parallel version of TSQR presented in (J. Demmel, Grigori, M. F. Hoemmen, et al., 2008) which is optimal in terms of communication and requires $O(logP)$ messages exchanged. We also offer the possibility to reconstruct Q from the successive Householder vectors computed during the factorization. We first present the algebra of TSQR and then we describe the parallel algorithm. Next, we provide some details of our implementation of TSQR and finally describe the modifications required for sparse matrices.

TSQR algebra

We first introduce some notations. Let $A \in \mathbb{R}^{m \times n}$ be a matrix distributed over p processors in block row layout. Thus, a block row A_i owned by processor i is of size $m/p \times n$. Its QR factorization leads to $Q_{i0}R_{i0} = qr(A_i)$. The subscript 0 denotes the stage 0 of the tree used by the algorithm.

For example, given a matrix A distributed over 4 processors

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}, \tag{6.4}$$

the factorization of each row block independently leads to

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{pmatrix}. \tag{6.5}$$

Equation (6.5) can be rewritten as

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix}. \tag{6.6}$$

The next stage groups two successive R factors such as $C_{i0} = \begin{pmatrix} R_{i0} \\ R_{(i+1)0} \end{pmatrix}$ where i is an even processor index. The factorization of each block $Q_{i1}R_{i1} = qr(C_{i0})$ leads to

$$\left(\begin{array}{c} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{array}\right) = \begin{pmatrix} Q_{01}R_{01} \\ Q_{21}R_{21} \end{pmatrix} = \left(\begin{array}{c|c} Q_{01} & \\ \hline & Q_{21} \end{array}\right) \begin{pmatrix} R_{01} \\ R_{21} \end{pmatrix}. \tag{6.7}$$

Reaching the root of the reduction tree, we have the final expression of A

$$A = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \left(\begin{array}{c|c} Q_{01} & \\ \hline & Q_{21} \end{array}\right) Q_{02}R_{02}, \tag{6.8}$$

where R_{02} is stored on processor 0 and the first three matrices are distributed over all processors. R_{02} is the R factor of $A = QR$, and the product of the first three matrices is the orthogonal matrix Q.

Focusing on the construction of the R factor, TSQR uses a binary tree to get R_{02}. It applies on each node of the tree an operation of factorization. Getting R corresponds to performing a

reduction tree as represented in (6.9).

$$\begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix} \rightarrow \begin{pmatrix} R_{01} \\ R_{21} \end{pmatrix} \rightarrow R_{02}. \tag{6.9}$$

TSQR algorithm

Algorithm 6.3 describes the parallel TSQR factorization using a general reduction tree and its application to a tall and skinny matrix $A \in \mathbb{R}^{m \times n}$, where $m \gg n$. We assume A is distributed in block row layout over p processors, and each processor stores $m/p \gg n$ rows locally. The first call to QR on the local A_i gives the first R factor of processor i denoted $R_{i,0}$ in Algorithm 6.3, line 1. Then at iteration k, all processors involved in this step exchange their local $R_{i,k-1}$ with their neighbors $\mathcal{N}_i$. On each processor i, $\forall j \neq i$ and $j \in \mathcal{N}_i$, the $R_{j,k-1}$ factors received are stacked by order of processor index with the local $R_{i,k-1}$ into a matrix $C_{i,k}$, line 5. This matrix of size $qn \times n$, where q is the number of neighbors, is factored using QR and returns the next $R_{i,k}$. Note that to avoid deadlock between send and receive operations, the algorithm uses non-blocking communication that involves a waiting step before the next factorization, line 6.

Algorithm 6.3 TSQR(A)

This function computes in parallel the QR factorization of a tall and skinny matrix $A \in \mathbb{R}^{m \times n}$ with $m \gg n$ and returns the R factor only. Here the communication scheme is q_butterfly tree of depth $L = log(p)$.

Input: $i \in \Pi$: processor index,
A_i local block row layout of A owned by processor i,

1: Compute $[Y_{i,0}, R_{i,0}] := qr(A_i)$
2: **for** $k = 1$ *to* L **do**
3: Send $R_{i,k-1}$ to its neighbors in the tree using non-blocking communication
4: Receive $R_{j,k-1}$ from its neighbors in the tree using non-blocking communication
5: Stack $R_{*,k-1}$ by order of processor index into $C_{i,k} \in \mathbb{R}^{qn \times n}$ where q is the number of neighbors
6: Wait until all send and receive operations initiated by processor i are done
7: Compute $[Y_{i,k}, R_{i,k}] := qr(C)$
8: **end for**

Output: $R_{i,L}$ the R factor of A, for $i \in \Pi$,
Q in its implicit representation: $Y_{i,k}$ where $i \in \Pi$ and $k \in \{1, 2, ..., L\}$

The Householder vectors Y are stored in memory instead of reconstructing Q. The complexity in parallel is $\frac{2mn^2}{p} + \frac{2n^3}{3}\log(p)$ flop, $log(p)$ messages, and $\frac{n^2}{2}\log(p)$ words.

This algorithm uses generally either a standard binary tree or a binary butterfly tree. Figure 6.3 shows the communication patterns for both schemes when performing TSQR on a matrix A distributed over 4 processors. At each level of the tree, on each node, the matrix C is factored using QR and the R factor is returned with the corresponding Householder vectors, denoted Y. The R factors represented by blue triangles correspond to the R factors computed and sent when the communication scheme is based on a binary tree. Processor 0 owns the final R and if required, the reconstruction of Q starts with it. This reconstruction detailed in (Ballard et al., 2013) involves communication in parallel. The general scheme starts with an identity matrix stacked on top of a 0 matrix on processor 0. It applies on it the Householder vectors denoted

$Y_{0,2}$ in the figure. Then it gets an inner Q which is split into two pieces and its lower part is sent. The communication scheme to reconstruct Q can be seen as a reverse binary tree. It follows that the lower part is sent to processor 2 in the figure. Again, both apply their Householder vectors from level 1 on this inner lower/upper Q and get a new inner Q which is also split until reaching the leaves of the tree. At that point, the local final Q is constructed using $Y_{*,0}$.

On the other hand, with the binary butterfly communication scheme referred hereafter as butterfly scheme, the final R is duplicated on all processors. Moreover, each processor has all Householder vectors needed to reconstruct its own part of Q. The mechanism is almost the same as described for the binary tree but it does not involve any communication. Therefore the butterfly scheme requires communications only for the construction of R.

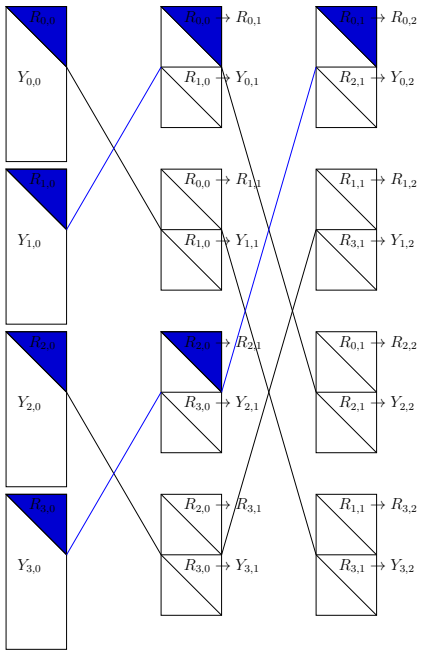


Figure 6.3 – TSQR communication scheme of the 1D block row layout distribution of A over 4 processors. The blue highlighted part of the scheme corresponds to a standard binary tree whereas the whole scheme is the binary butterfly communication scheme.

Example of use of our TSQR

Before going into the details of the implementation, we first show an example of use of TSQR in Listing 6.10. We define a TSQR_t structure which stores the environment and is set with `TSQRCreate` routine. Here, this environment needs the size of the block used by LAPACK, for example, DGEQRT routine, and the type of communication tree with the MPI communicator. If butterfly flag is set to 1 then the communication scheme is the butterfly, otherwise, the binary tree is used. At that point, the environment is set and the communication is built by following the abstractTree module. Then we set at least the matrix A and the R variable using `TSQRSetMatrices`. If Q is not null, then it is explicitly built from the Householder vectors. Also, if the parameter of permutation of RRQR is provided, i.e. eRRQR.val variable is not null, then TSQR performs $QRCP$ on the factor R, using DGEQP3 routine and returns the result in it. The last parameter is a workspace used by TSQR for optimization, with its size. Note

that, we can let TSQR allocates the required workspace and obtains it as output of the function along with its size. At this point, TSQR environment is set and we can call `TSQR()` routine to perform the TSQR algorithm. In the case of several calls of TSQR, it suffices to call successively `TSQRSetMatrices` and `TSQR()`. At the end, TSQR environment is destroyed using `TSQRDestroy`.

```
TSQR_t tsqr = TSQRNULL();

ierr = TSQRCreate(&tsqr,
                  nb,
                  butterfly,
                  MPI_COMM_WORLD);

ierr = TSQRSetMatrices( &tsqr,
                        &A,
                        &R,
                        (computeQ) ? ((inplace) ? &A : &Q) : NULL,
                        eRRQR.val,
                        &work.val,
                        &workSize
                        );CHKERR(ierr);

ierr = TSQR(&tsqr);CHKERR(ierr);

TSQRDestroy(&tsqr);
```

Listing 6.10 – Extract from callTSQR.c example

Note that in `TSQRSetMatrices`, if Q is requested, it is possible to built it inplace of A. Since the first QR factorization is inplace, A_i is destroyed and we can reuse the memory space to store Q.

Details of implementation

We are now using this example of TSQR to describe our implementation. First, we present in Listing 6.11 the structure used in TSQR environment. Besides the variables already described, two internal variables called $treeDataR$ and $treeDataQ$ of type TreeVoid_t are used. As presented above, TreeVoid_t variable refers to the abstractTree module. It stores the configuration of the communication concerning the construction of R and Q. When `TSQRCreate` is called, depending on the $butterfly$ flag, the communication pattern is built. If $butterfly$ is 1, then only $treeDataR$ is set. Otherwise, $treeDataR$ handles the binary tree and $treeDataQ$ handles the reverse binary tree. Then `TSQRSetMatrices` sets variables of TSQR and allocates the required workspace. The size of the workspace depends on the number of columns, the block size, the number of R factors received, and if Q is requested or not. This is handled internally by `TSQRGetWorkSizeNeeded` routine.

```
typedef struct{
  TreeVoid_t treeDataR;
  TreeVoid_t treeDataQ;
  Mat_Dense_t *A;
  Mat_Dense_t *Q;
  Mat_Dense_t *R;
  IVector_t e;
  double *work;
  size_t workSize;
} TSQR_t;
```

Listing 6.11 – Extract from tsqr.h

The workspace is shared between shallow pointers as presented in Figure 6.4. First, if Q is requested, we set a space $Qwork$ of size n^2 used as a workspace when Q is rebuilt. It stores the lower part of the inner Q and is sent in the binary tree case. We always reserve a space of size $nb \times n$ for τ_0 returned by the QR factorization of A_i. Then, we may compute $QRCP$ on the factor R. In that case, we store τ_{RRQR} in this space of size n. Next, we reserve the largest part of the workspace to store the matrix $C = \begin{pmatrix} R_{up} \\ R_{down} \end{pmatrix}$ and then the Householder vectors Y returned by the factorization of C matrices. Its size is n^2 times the number of R factors received. Note that since the pattern of C is already known, the Householder vectors replace the R_{down} part of C. Since all QR factorizations generate a τ along with Y, we group them in a different place represented in Figure 6.4 by the union of τ_i. Similarly, its size is $nb \times n$ times the number of R factors received. When R is computed using the butterfly communication scheme, the algorithm imposes that the $R_{*,k}$ factors are stacked ordered. Thus, R_{up} and R_{down} can be swapped. Next, if Q is requested, the rebuilt follows an ordered scheme which is the same as the one used to compute R. For this reason, we record in $swap$ whenever we swap R_{up} and R_{down}. Along with that, we index in $ptrY = \bigcup_{i=1}^{L} @Y_i$ the positions of the Householder vectors in the largest space drawn in green in Figure 6.4.

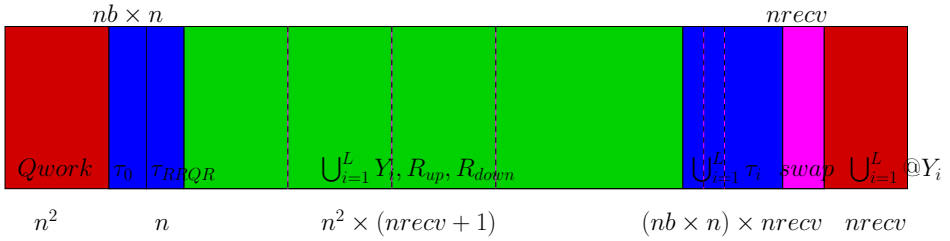


Figure 6.4 – TSQR workspace used internally to factorize a matrix A. Each name inside the rectangular shape denotes a shallow pointer used by our implementation and the outer string indicates the size of each block. The blue part corresponds to the computation of the final R whereas the explicit construction of Q involves the red and the pink parts. The green part participates in both steps.

Algorithm 6.4 Implementation of TSQR

This sketch points out the main steps of our implementation of TSQR.

Input: $i \in \Pi$, the processor index,
A_i local block row layout of A owned by processor i,
nb, the block size for BLAS3 routines,
$computeQ$, the flag set to 1 when the reconstruction of Q is necessary

1: Call `dgeqrt`(A_i, nb) returns the QR factorization inplace of A_i and fills τ_0
2: Copy of the upper triangular part of A_i, i.e. $R_{i,0}$, into R_{up}
3: Call `TSQRInner`$(R_{up}, computeQ)$ to compute the final R and to reconstruct partially Q if $computeQ$ flag is set to 1.
4: **if** $computeQ = 1$ **then**
5: Copy R_{up} into Q_i
6: Call `dgemqrt`(A_i, Q_i, τ_0) to overwrite Q_i with the final Q_i
7: **end if**

Output: $R_{i,L}$ the final R factor of A,
Q_i the local Q on processor i in its explicit form

We now present the main steps of our implementation in Algorithm 6.4. As presented above, we first compute the stage 0, i.e., $Q_{i,0}R_{i,0} = qr(A_i)$ of the algorithm to get the first R factor. The result is used as input by `TSQRInner` routine, at line 2. This inner routine computes QR on each node of the tree and reconstructs partially Q if needed. The last operation to get the final Q requires the Householder vectors computed at the leaves of the tree. As described earlier, its reconstruction takes the upper part of the inner Q and applies Y on it. This sketch uses blocked version of the routines to factorize and to get Q. We also have a non-blocked version where `dgeqrt` is replaced by `dgeqrf` and `dgemqrt` by `dormqr`.

6.5.4 Performance of TSQR compared to CholeskyQR in parallel

In this section, we compare the performance of CholeskyQR and TSQR. To compare both methods, we perform a weak scaling from 2 to 1024 cores where the local dense matrix is of size $20,000 \times 256$ and the block size used for TSQR is equal to the number of columns. Thus the global A is of size $(p \times 20,000) \times 256$.

Tests are made on an IBM supercomputer composed of 92 nodes. Each node has 32 GB of RAM and 2 Sandy Bridge E5-2670 (2.60GHz). The network is an InfiniBand QLogic QDR. The supercomputer is managed by a Linux CentOS 6.5. The library is compiled with Intel-15, Intel MPI 5.0.1 and uses MKL version provided by composer_xe_2015.0.090.

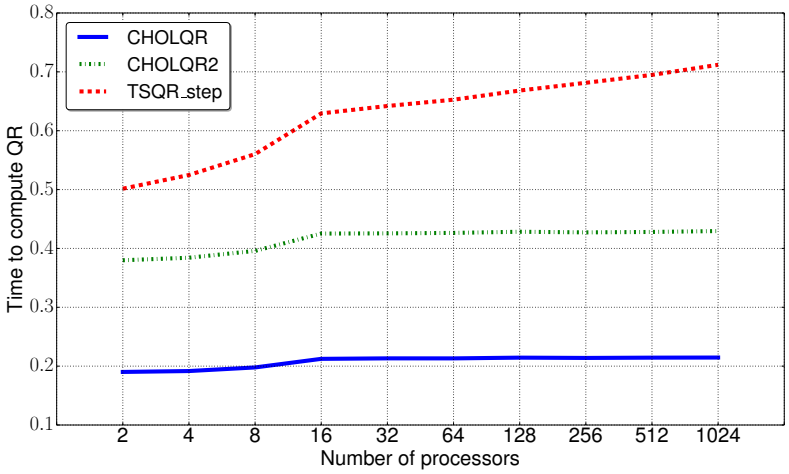


Figure 6.5 – Weak scaling

We display the performance of TSQR, CholeskyQR and CholeskyQR2 in Figure 6.5, where the number of processors p increases from 2 to 1024. The y-axis represents the time in seconds to factor A. The runtime of all methods increases from 2 to 16 processors, where the resources of the node are fully used. When the number of processors is greater than 16, the runtime of CholeskyQR and CholeskyQR2 remains almost the same, whereas the runtime of TSQR increases with the number of processors used. CholeskyQR is more than twice faster than TSQR for 2 processors and it reaches an improvement of a factor of 3.5 for 1024 processors. When we use more than one node, the communication scheme used in our implementation of TSQR degrades the performance. The use of the abstractTree module is not efficient enough in comparison with the allReduction communication scheme used by both CholeskyQR and CholeskyQR2. If we consider the case of 2 processors, the communication is negligible compared to the communication. In that case, TSQR is slower than the other methods. Especially, for better accuracy, one can

call CholeskyQR twice (corresponding to CholeskyQR2) and the runtime of this variant is still smaller than TSQR.

6.6 Interface to use external packages

To use our library as a third party library, we present some routines to convert external datatypes into our datatypes, as well as the reverse approach. We have interfaces for several external libraries. In this section we present these interfaces in details. We also interface libraries to call external routines like Kway routine in Metis (Karypis et al., 1999).

We mentioned earlier that we wrapped for example the Pardiso solver from MKL and made it available in kernels.c file. In addition, we need to interface PETSc library and SuiteSparse package. It means that we can manipulate our matrices MatCSR and MatDense and interface them with PETSc to call for example GMRes solver. SuiteSparse library is interfaced through SpTSQR. In it, we are using several cholmod_sparse matrices and some routines to concatenate or extract since this structure is a CSC format representation. The idea behind that is to keep the code consistent.

6.6.1 PETSc interface

PETSc library offers the possibility to perform many different operations. To benefit from both libraries, we design some routines to create PETSc matrices from our datatypes in sequential and in parallel. We also provide routines to convert PETSc datatypes back into our datatypes. We present a list of the main routines and the documentation of the software contains a full description of each.

- `petscCreateSeqMatFromMatCSR`, `petscCreateSeqMatFromMatDense` routines transform sequential matrices into sequential PETSc matrices, both sparse and dense.
- `petscCreateMatFromMatCSR`, `petscCreateMatFromMatDense` routines have the same functionality as the previous ones but consider parallel matrices.
- `petscCreateMatCSR` returns a MatCSR from a PETSc sparse matrix A.
- `petscPrintMatCSR` allows us to print a MatCSR using PETSc drawing interface.
- `petscGetILUFactor` factorizes a sparse matrix A using ILU(k) algorithm and returns F that contains both factors.
- `petscGetLUFactorization` factorizes a sparse matrix A using the standard LU algorithm with several parameters like τ or k that come from the PETSc interface and returns F having both factors (see PETSc documentation for more details).
- `petscGetLUFromMatCSR` does the same factorization as the previous routine but on a MatCSR.
- `petscConvertFactorToMatCSR` takes as input the F matrix that contains both L and U and returns them as MatCSR matrices.
- `petscMatLoad` loads a sparse matrix from a .mtx file into a PETSc sparse matrix, all in parallel using a communicator.
- `petscMatGetScaling` scales a matrix A using a vector Vec.

Additionally, we interface the SpMM of PETSc with `petscMatCSRMatDenseMult` taking as parameter a PETSc sparse matrix A and a MatDense B, and returns a MatDense matrix $C = A * B$.

6.6.2 SuiteSparse interface

SuiteSparse library allows us to manipulate sparse matrices stored in CSC format. Also, SuiteSparse routines are using SuiteSparse_long type. Then we provide routines to convert to SuiteSparse datatypes and from them to MatDense,MatCSR and IVector. We also wrap the spqr routine which computes a QR factorization of a sparse matrix. So `SPQR()` routines take as parameter a sparse matrix A and return Q and R plus a column permutation vector e. All sparse matrices are represented by cholmod_sparse datatype.

Moreover, since we need in SpTSQR to send and receive cholmod_sparse matrices, we have routines that wrap MPI communication. Note that these routines end with '_l', denoting the usage of long instead of integer. We first create an MPI_Datatype $MPI\_CHOLMOD$ to reduce the number of messages sent when a communication involving a cholmod_sparse occurs.

- `cholmodSendSparse_l` takes the matrix to send A_in plus the destination $dest$, a tag of communication tag and the communicator $comm$.
- `cholmodSendPartialSparse_l` is an improvement of the previous routine which can send a subset of columns of A_in of size $ncol$ starting at the column $offset$. The purpose of this routine is to avoid to extract the column panel before sending it.
- `cholmodRecvSparse_l` routine takes the source src and the tag tag and receives in A_io the sparse matrix using the communicator $comm$.

These routines allow us easily to apply the abstractTree concept on cholmod_sparse datatype.

Conclusion

In this chapter, we have presented CPaLAMeM, a library that is used to develop the codes presented in this thesis. The library offers a collection of routines that manipulate basic mathematical structures as matrices and vectors. The library is interfaced with external libraries as Petsc or SuiteSparse. CPaLAMeM has been used to develop a parallel implementation of Enlarged GMRES (Al Daas et al., 2018) and Enlarged CG (Grigori and Tissot, 2017), that scale up to 8k and 16k, respectively. The library can be downloaded at `https://github.com/cayrols/CPaLAMeM`, and is also part of the preAlps library, one of the main libraries developed by Alpines team.

Chapter 7

Conclusion

In this manuscript, we gave details of two methods that can be used in the context of a sparse linear system of equations $Ax = b$. In Chapter 3, we presented CA-ILU(k), a communication avoiding preconditioner based on the incomplete LU factorization that use an overlapping technique to factor the diagonal blocks of A, and to apply it at each GMRES iteration without communication. The parallel experiments have shown that CA-ILU(k) outperforms, in general, the Block Jacobi preconditioner, while the Restricted Additive Schwarz preconditioner is, however, the most efficient in term of number of iterations and time to solve on most of the problems. The main issue here is that the value of k impacts drastically the size of the overlap and so the overall performance of GMRES. On the other hand, the variant of CA-ILU(k) that computes the LU factorization of the augmented block rows outperforms both other preconditioners on half of the problems studied here. A future investigation is to reintroduce local communication when the size of the overlap is too large. By local communication, we mean communication with neighbors. That is, the overlapping data stored by the MPI process come from MPI processes that are located far away from it, in the network sense. This approach would be promising if the number of saved Flops is greater than the cost of this local communication. Instead of using the distance as a metric, we could consider the minimization of the communication. We believe that the best solution is using a mix of both metrics.

In Chapter 4, we introduced LU-CRTP, a low-rank approximation method based on a block LU factorization of A. The key idea is to use QRTP, the QR factorization with Tournament Pivoting that selects a subset of columns of A and $A^\top$. The experimental results enlighten that along with the LU factorization, the algorithm is able to compute an approximation of the singular values of A with a maximum relative error lower than two orders of magnitudes. Since the QRTP algorithm is also a building block of the Communication-Avoiding Rank Revealing QR factorization, we enhanced the algorithm in Chapter 5. Our purpose was to reduce the number of redundant computations, performed at each iteration of LU-CRTP and CARRQR. To do so, a mechanism selects a subset of the columns of A that are discarded with respect to our τ_rank criterion. The extensive experiments show that for different values of τ, the low-rank approximation is not degraded. Moreover, in comparison with CARRQR, our variant performs up to 36 times fewer Flops during the update of the trailing matrix than CARRQR. The next step would be to have a parallel implementation of this modification and to measure the speedup obtained in real-world applications like neural networks.

Throughout this thesis, we have developed a library named CPaLAMeM. A C parallel library that targets the development of research parallel codes in a convenient way. We observed that

the library led to the short development of two codes Enlarged GMRES and Enlarged CG, which scale up to 8k and 16k MPI processes, respectively. The library is also integrated into the preAlps library, a collection of efficient communication avoiding solvers and preconditioners in parallel.

Bibliography

[1] Y. Achdou and F. Nataf. "Low frequency tangential filtering decomposition". In: *Numerical Linear Algebra with Applications* 14.2 (2007), pp. 129–147.

[2] H. Al Daas, L. Grigori, P. Hénon, and P. Ricoux. "Enlarged GMRES for solving linear systems with one or multiple right-hand sides". In: *IMA Journal of Numerical Analysis* (2018), dry054. eprint: `/oup/backfile/content_public/journal/imajna/pap/10.1093_imanum_dry054/5/dry054.pdf`.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[4] E. Anderson et al. "LAPACK: A Portable Linear Algebra Library for High-performance Computers". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pp. 2–11.

[5] S. Balay et al. *PETSc Users Manual*. Tech. rep. ANL-95/11 - Revision 3.7. Argonne National Laboratory, 2016.

[6] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik. *Reconstructing Householder Vectors from Tall-Skinny QR*. Tech. rep. UCB/EECS-2013-175. EECS Department, University of California, Berkeley, Oct. 2013.

[7] M. Besta and T. Hoefler. "Slim Fly: A Cost Effective Low-Diameter Network Topology". In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), pp. 348–359.

[8] L. S. Blackford et al. *ScaLAPACK User's Guide*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[9] R. F. Boisvert, R. Pozo, and K. A. Remington. "The Matrix Market Exchange Formats: Initial Design". In: *NISTIR* 5935 (1996).

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

[11] X.-C. Cai and M. Sarkis. "A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems". In: *SIAM Journal on Scientific Computing* 21.2 (1999), pp. 792–797. eprint: `https://doi.org/10.1137/S106482759732678X`.

[12] X.-C. Cai and M. Sarkis. "A restricted additive Schwarz preconditioner for general sparse linear systems". In: *SIAM J. Sci. Comput.* 21.2 (1999), 792–797 (electronic).

[13] E. Carson. "High performance Krylov subspace method variants". 2017.

[14] E. Carson, N. Knight, and J. Demmel. "Avoiding Communication in Nonsymmetric Lanczos-Based Krylov Subspace Methods." In: *SIAM J. Scientific Computing* 35.5 (2013).

[15] S. Cayrols and L. Grigori. "CA-ILU(k): a Communication-Avoiding ILU(k) preconditioner". 2019.

[16] S. Cayrols and L. Grigori. "Tournament pivoting based on $\tau - rank$ revealing for the low-rank approximation of sparse and dense matrices". 2019.

[17] S. Chandrasekaran and I. C. F. Ipsen. "On Rank-Revealing Factorisations". In: *SIAM Journal on Matrix Analysis and Applications* 15.2 (1994), pp. 592–622. eprint: `https://doi.org/10.1137/S0895479891223781`.

[18] E. Chow and A. Patel. "Fine-Grained Parallel Incomplete LU Factorization". In: *SIAM Journal on Scientific Computing* 37.2 (2015), pp. C169–C193. eprint: `https://doi.org/10.1137/140968896`.

[19] T. A. Davis. "Algorithm 915: SuiteSparseQR, multifrontal multithreaded rank-revealinng sparse QR factorization". In: *ACM Trans. Math. Software* 38.1 (2011), 8:1–8:22.

[20] T. A. Davis. "Algorithm 915, SuiteSparseQR: Multifrontal Multithreaded Rank-revealing Sparse QR Factorization". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 8:1–8:22.

[21] T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25.

[22] J. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. eprint: `http://epubs.siam.org/doi/pdf/10.1137/1.9781611971446`.

[23] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang. "Comunication avoiding rank revealing QR factorization with column pivoting". In: *SIAMX* (2015).

[24] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. "Communication-optimal parallel and sequential QR and LU factorizations". In: *Technical Report No. UCB/EECS-2008-89* (2008).

[25] J. W. Demmel, L. Grigori, and H. Xiang. "Communication avoiding Gaussian elimination". In: *Proceedings of the ACM/IEEE 2008 Conference on Supercomputing, IEEE Press* (2008).

[26] J. Demmel, L. Grigori, and S. Cayrols. *Low Rank Approximation of a Sparse Matrix Based on LU Factorization with Column and Row Tournament Pivoting*. Tech. rep. UCB/EECS-2016-122. EECS Department, University of California, Berkeley, June 2016.

[27] J. Demmel, L. Grigori, M. Gu, and H. Xiang. *Communication Avoiding Rank Revealing QR Factorization with Column Pivoting*. Tech. rep. UCB/EECS-2013-46. EECS Department, University of California, Berkeley, May 2013.

[28] J. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. *Communication-optimal parallel and sequential QR and LU factorizations*. Tech. rep. UCB/EECS-2008-89. Current version available in the ArXiv at http://arxiv.org/pdf/0809.0101 Replaces EECS-2008-89 and EECS-2008-74. EECS Department, University of California, Berkeley, Aug. 2008.

[29] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. "Avoiding communication in sparse matrix computations". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE. 2008, pp. 1–12.

[30] V. Dolean, P. Jolivet, and F. Nataf. *An Introduction to Domain Decomposition Methods*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2015. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9781611974065`.

[31] I. S. Duff and G. A. Meurant. "The Effect of Ordering on Preconditioned Conjugate Gradients". In: *BIT* 29.4 (Dec. 1989), pp. 635–657.

[32] C. Eckart and G. Young. "The approximation of one matrix by another of lower rank". In: *Psychometrika* 1.3 (1936), pp. 211–218.

[33] L. Foster. *San Jose State University Singular Matrix Database*. 2017.

[34] A. George. "Nested Dissection of a Regular Finite Element Mesh". In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

[35] J. R. Gilbert. "Predicting Structure In Sparse Matrix Computations". In: *SIAM J. Matrix Anal. Appl* 15 (1994), pp. 62–79.

[36] J. R. Gilbert and T. Peierls. "Sparse Partial Pivoting in Time Proportional to Arithmetic Operations". In: *SIAM Journal on Scientific and Statistical Computing* 9.5 (1988), pp. 862–874.

[37] G. H. Golub and C. F. Van Loan. *Matrix Computations (4th Ed.)* Baltimore, MD, USA: Johns Hopkins University Press, 2013.

[38] G. Golub, V. Klema, and G. W. Stewart. *Rank Degeneracy and Least Squares Problems*. Tech. rep. 1976.

[39] V. Grandgirard et al. "A 5D gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations". In: *Computer Physics Communications* 207 (2016), pp. 35–68.

[40] L. Grigori, S. Cayrols, and J. Demmel. "Low Rank Approximation of a Sparse Matrix Based on LU Factorization with Column and Row Tournament Pivoting". In: *SIAM Journal on Scientific Computing* 40.2 (2018), pp. C181–C209. eprint: `https://doi.org/10.1137/16M1074527`.

[41] L. Grigori and O. Tissot. *Reducing the communication and computational costs of Enlarged Krylov subspaces Conjugate Gradient*. Research Report RR-9023. Inria, Feb. 2017.

[42] L. Grigori, J. W. Demmel, and H. Xiang. "CALU: A Communication Optimal LU Factorization Algorithm". In: *SIAM Journal on Matrix Analysis and Applications* 32.4 (2011), pp. 1317–1350.

[43] L. Grigori and S. Moufawad. "Communication Avoiding ILU0 Preconditioner". In: *SIAM Journal on Scientific Computing* 37.2 (2015), pp. C217–C246. eprint: `https://doi.org/10.1137/130930376`.

[44] L. Grigori and S. Moufawad. "Communication Avoiding ILU0 Preconditioner". In: *SIAM J. Scientific Computing* 37.2 (2015).

[45] L. Grigori, S. Moufawad, and F. Nataf. "Enlarged Krylov Subspace Conjugate Gradient Methods for Reducing Communication". In: *SIAM J. Matrix Analysis Applications* 37 (2016), pp. 744–773.

[46] M. Gu. "Subspace Iteration Randomization and Singular Value Problems". In: *SIAM Journal on Scientific Computing* 37.3 (2015), A1139–A1173. eprint: `https://doi.org/10.1137/130938700`.

[47] M. Gu and S. C. Eisenstat. "Efficient Algorithms for Computing a Strong Rank-revealing QR Factorization". In: *SIAM J. Sci. Comput.* 17.4 (July 1996), pp. 848–869.

[48] R. Haferssas, P. Jolivet, and F. Nataf. "An Additive Schwarz Method Type Theory for Lions's Algorithm and a Symmetrized Optimized Restricted Additive Schwarz Method". In: *SIAM Journal on Scientific Computing* 39.4 (2017), A1345–A1365.

[49] N. Halko, P. Martinsson, and J. Tropp. “Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions”. In: *SIAM Review* 53.2 (2011), pp. 217–288. eprint: `https://doi.org/10.1137/090771806`.

[50] V. E. Henson and U. M. Yang. “BoomerAMG: A parallel algebraic multigrid solver and preconditioner”. In: *Applied Numerical Mathematics* 41.1 (2002). Developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss, pp. 155–177.

[51] M. R. Hestenes and E. Stiefel. “Methods of conjugate gradients for solving linear systems”. In: *Journal of research of the National Bureau of Standards* (1952), pp. 409–436.

[52] Y. P. Hong and C.-T. Pan. “Rank-Revealing QR Factorizations and the Singular Value Decomposition”. In: *Mathematics of Computation* 58.197 (1992), pp. 213–232.

[53] R. A. Horn and C. R. Johnson, eds. *Matrix Analysis*. New York, NY, USA: Cambridge University Press, 1986.

[54] Intel. *Intel xeon specification*. Accessed: 2018-09-01. Intel, 2018.

[55] P. Jolivet, F. Hecht, F. Nataf, and C. Prud’homme. “Scalable Domain Decomposition Preconditioners For Heterogeneous Elliptic Problems”. In: *Proceedings of the 2013 International Conference on High Performance Computing, Networking, Storage and Analysis*. SC13. ACM, 2013, 80:1–80:11.

[56] G. Karypis and V. Kumar. “Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING* 48 (1998), pp. 96–129.

[57] G. Karypis and V. Kumar. “Multilevel K-way Hypergraph Partitioning”. In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. DAC ’99. New Orleans, Louisiana, USA: ACM, 1999, pp. 343–348.

[58] A. Khabou, J. W. Demmel, L. Grigori, and M. Gu. “LU Factorization with Panel Rank Revealing Pivoting and Its Communication Avoiding Version”. In: *SIAM Journal on Matrix Analysis and Applications* 34.3 (2013), pp. 1401–1429.

[59] C. Lanczos. “An iteration method for the solution of the eigenvalue problem of linear differential and integral operators”. In: *J. Res. Natl. Bur. Stand. B* 45 (1950), pp. 255–282.

[60] X. S. Li. “An Overview of SuperLU: Algorithms, Implementation, and User Interface”. In: *ACM Trans. Math. Softw.* 31.3 (Sept. 2005), pp. 302–325.

[61] P.-L. Lions, T. F. Chan, R. Glowinski, J. Périaux, and O. Widlund. “On the Schwarz alternating method,I”. In: *In First International Symposium on Domain Decomposition Methods for Partial Differential Equation* (1988).

[62] J. A. Meijerink and H. A. van der Vorst. “An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix”. In: *Mathematics of computation* 31.137 (1977), pp. 148–162.

[63] G. E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965).

[64] R. Nabben and C. Vuik. “A Comparison of Deflation and the Balancing Preconditioner”. In: *SIAM Journal on Scientific Computing* 27.5 (2006), pp. 1742–1759. eprint: `https://doi.org/10.1137/040608246`.

[65] Q. Niu, L. Grigori, P. Kumar, and F. Nataf. “Modified tangential frequency filtering decomposition and its fourier analysis”. In: *Numerische Mathematik* 116 (2010), pp. 123–148.

[66] Y. Notay. *AN AGGREGATION-BASED ALGEBRAIC MULTIGRID METHOD*. Vol. 37. Electronic Transactions on Numerical Analysis, 2010, pp. 123–146.

[67] C.-T. Pan. "On the existence and computation of rank-revealing LU factorizations". In: *Linear Algebra and its Applications* 316.1–3 (2000), pp. 199–222.

[68] J. Papez, L. Grigori, and R. Stompor. *Solving linear equations with messenger-field and conjugate gradients techniques – an application to CMB data analysis*. Research Report RR-9157. Inria Paris, Mar. 2018.

[69] S. Parter. "The Use of Linear Graphs in Gauss Elimination". In: *SIAM Review* 3.2 (1961), pp. 119–130.

[70] V. Rokhlin, A. Szlam, and M. Tygert. "A Randomized Algorithm for Principal Component Analysis". In: *SIAM Journal on Matrix Analysis and Applications* 31.3 (2010), pp. 1100–1124. eprint: `https://doi.org/10.1137/080736417`.

[71] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, 2003. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003`.

[72] Y. Saad and M. H. Schultz. "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems". In: *SIAM J. Sci. Stat. Comput.* 7.3 (July 1986), pp. 856–869.

[73] Y. Saad. "ILUT: A dual threshold incomplete LU factorization". In: *Numerical Linear Algebra with Applications* 1.4 (1994), pp. 387–402.

[74] M. Szydlarski, L. Grigori, and R. Stompor. "Accelerating Cosmic Microwave Background map-making procedure through preconditioning". In: *CoRR* abs/1408.3048 (2014).

[75] J. Tang, S. MacLachlan, R. Nabben, and C. Vuik. "A Comparison of Two-Level Preconditioners Based on Multigrid and Deflation". In: *SIAM Journal on Matrix Analysis and Applications* 31.4 (2010), pp. 1715–1739. eprint: `https://doi.org/10.1137/08072084X`.

[76] M. F. Wehner et al. "Hardware/software co-design of global cloud system resolving models". In: *Journal of Advances in Modeling Earth Systems* 3.4 (2011). eprint: `https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2011MS000073`.

[77] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert. "A fast randomized algorithm for the approximation of matrices". In: *Applied and Computational Harmonic Analysis* 25.3 (2008), pp. 335–366.

[78] Y. Xi, R. Li, and Y. Saad. "An Algebraic Multilevel Preconditioner with Low-Rank Corrections for Sparse Symmetric Matrices". In: *SIAM Journal on Matrix Analysis and Applications* 37.1 (2016), pp. 235–259. eprint: `https://doi.org/10.1137/15M1021830`.

[79] K. Yelick. "HPC for Genomic Data at Scale". 2018.

[80] C. M. Zarzycki, K. A. Reed, J. T. Bacmeister, A. P. Craig, S. C. Bates, and N. A. Rosenbloom. "Impact of surface coupling grids on tropical cyclone extremes in high-resolution atmospheric simulations". In: *Geoscientific Model Development* 9.2 (2016), pp. 779–788.

Appendix A

CA-ILU(k)

Outline of the current chapter

A.1 Additional experimental results when $k = 0$

In this section, we present additional results about the parallel efficiency of CA-ILU(0) compared with BJacobi and RAS with one and two levels of overlap.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	1638	9.98e-07	9.98e-07	1.78e-06
np = 16	BJacobi	345	2.55e-06	4.71e-07	3.54e-07
	CAILU(0)	276	2.57e-06	4.80e-07	4.79e-07
	RAS(1)	264	2.56e-06	4.67e-07	2.29e-07
	RAS(2)	261	2.58e-06	4.39e-07	2.13e-07
np = 32	BJacobi	348	2.51e-06	4.68e-07	3.60e-07
	CAILU(0)	280	2.48e-06	4.69e-07	5.25e-07
	RAS(1)	266	2.45e-06	4.58e-07	2.36e-07
	RAS(2)	260	2.49e-06	4.17e-07	2.03e-07
np = 64	BJacobi	349	2.50e-06	4.94e-07	3.27e-07
	CAILU(0)	286	2.49e-06	4.60e-07	6.78e-07
	RAS(1)	268	2.57e-06	4.79e-07	2.75e-07
	RAS(2)	264	2.48e-06	4.36e-07	2.26e-07
np = 128	BJacobi	354	2.48e-06	5.08e-07	3.73e-07
	CAILU(0)	305	2.54e-06	4.56e-07	8.06e-07
	RAS(1)	270	2.54e-06	4.81e-07	3.01e-07
	RAS(2)	265	2.50e-06	4.43e-07	2.40e-07
np = 256	BJacobi	357	2.51e-06	5.43e-07	4.37e-07
	CAILU(0)	326	2.50e-06	5.08e-07	6.52e-07
	RAS(1)	275	2.52e-06	4.99e-07	4.05e-07
	RAS(2)	267	2.51e-06	4.64e-07	2.56e-07

Table A.1 – Comparison of CA-ILU(0) with BJacobi and RAS on the problem of matvf2dAD400400 for a number of blocs goes from 16 to 256. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	3000	4.70e-06	4.70e-06	2.15e-03
np = 16	BJacobi	1273	1.48e-06	2.19e-07	9.36e-04
	CAILU(0)	1678	1.47e-06	2.11e-07	1.18e-03
	RAS(1)	1398	1.48e-06	1.51e-07	1.01e-03
	RAS(2)	1289	1.48e-06	1.46e-07	1.02e-03
np = 32	BJacobi	1281	1.45e-06	3.06e-07	8.97e-04
	CAILU(0)	1967	1.43e-06	2.36e-07	1.31e-03
	RAS(1)	1449	1.47e-06	1.55e-07	1.03e-03
	RAS(2)	1526	1.46e-06	1.55e-07	1.05e-03
np = 64	BJacobi	1589	1.41e-06	3.78e-07	9.22e-04
	CAILU(0)	2326	1.35e-06	2.58e-07	1.46e-03
	RAS(1)	1550	1.43e-06	1.98e-07	1.10e-03
	RAS(2)	1587	1.43e-06	1.66e-07	1.12e-03
np = 128	BJacobi	1786	1.38e-06	3.76e-07	1.01e-03
	CAILU(0)	2197	1.33e-06	2.73e-07	1.54e-03
	RAS(1)	1730	1.42e-06	2.14e-07	1.18e-03
	RAS(2)	1874	1.42e-06	1.74e-07	1.21e-03
np = 256	BJacobi	2105	1.35e-06	3.62e-07	1.09e-03
	CAILU(0)	2450	1.30e-06	2.67e-07	1.58e-03
	RAS(1)	1786	1.39e-06	2.45e-07	1.25e-03
	RAS(2)	1896	1.40e-06	2.01e-07	1.28e-03
np = 512	BJacobi	2503	1.33e-06	3.96e-07	1.14e-03
	CAILU(0)	2521	1.30e-06	2.68e-07	1.61e-03
	RAS(1)	1994	1.37e-06	2.08e-07	1.31e-03
	RAS(2)	1977	1.37e-06	2.12e-07	1.35e-03

Table A.2 – Comparison of CA-ILU(0) with BJacobi and RAS on the problem Elasticity3D4001010 for a number of partitions goes from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	3000	4.70e-06	4.70e-06	2.15e-03
np = 16	CAILU(0)	1678	1.47e-06	2.11e-07	1.18e-03
	BJacobi_permuted	1554	1.47e-06	3.63e-07	1.08e-03
	RAS(2)_permuted	1678	1.47e-06	2.11e-07	1.18e-03
np = 32	CAILU(0)	1967	1.43e-06	2.36e-07	1.31e-03
	BJacobi_permuted	1817	1.43e-06	3.52e-07	1.12e-03
	RAS(2)_permuted	1967	1.43e-06	2.36e-07	1.31e-03
np = 64	CAILU(0)	2326	1.35e-06	2.58e-07	1.46e-03
	BJacobi_permuted	2474	1.35e-06	3.88e-07	1.23e-03
	RAS(2)_permuted	2326	1.35e-06	2.59e-07	1.46e-03
np = 128	CAILU(0)	2197	1.33e-06	2.73e-07	1.54e-03
	BJacobi_permuted	2229	1.32e-06	4.25e-07	1.30e-03
	RAS(2)_permuted	2205	1.33e-06	2.77e-07	1.53e-03
np = 256	CAILU(0)	2450	1.30e-06	2.67e-07	1.58e-03
	BJacobi_permuted	2016	1.29e-06	3.83e-07	1.33e-03
	RAS(2)_permuted	2474	1.31e-06	2.70e-07	1.57e-03
np = 512	CAILU(0)	2521	1.30e-06	2.68e-07	1.61e-03
	BJacobi_permuted	2385	1.27e-06	5.08e-07	1.33e-03
	RAS(2)_permuted	2464	1.30e-06	2.76e-07	1.59e-03

Table A.3 – Comparison of CA-ILU(0) with BJacobi_permuted and RAS(2)_permuted, both using the permutation computed by CA-ILU(0), on the problem Elasticity3D4001010 for a number of partitions goes from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	2750	1.00e-06	1.00e-06	1.94e-06
np = 16	BJacobi	537	1.33e-06	3.67e-07	3.07e-07
	CAILU(0)	463	1.34e-06	3.59e-07	2.88e-07
	RAS(1)	474	1.35e-06	3.60e-07	2.92e-07
	RAS(2)	458	1.34e-06	3.58e-07	2.99e-07
np = 32	BJacobi	539	1.34e-06	3.70e-07	3.33e-07
	CAILU(0)	461	1.35e-06	3.63e-07	2.94e-07
	RAS(1)	476	1.34e-06	3.58e-07	2.91e-07
	RAS(2)	456	1.35e-06	3.58e-07	2.95e-07
np = 64	BJacobi	532	1.33e-06	3.74e-07	3.18e-07
	CAILU(0)	463	1.34e-06	3.56e-07	2.94e-07
	RAS(1)	472	1.34e-06	3.56e-07	2.65e-07
	RAS(2)	455	1.34e-06	3.58e-07	2.68e-07
np = 128	BJacobi	535	1.34e-06	3.83e-07	3.30e-07
	CAILU(0)	455	1.35e-06	3.57e-07	2.71e-07
	RAS(1)	481	1.34e-06	3.58e-07	3.00e-07
	RAS(2)	463	1.35e-06	3.58e-07	3.15e-07
np = 256	BJacobi	516	1.33e-06	3.83e-07	3.54e-07
	CAILU(0)	462	1.35e-06	3.51e-07	3.11e-07
	RAS(1)	462	1.34e-06	3.57e-07	2.81e-07
	RAS(2)	436	1.34e-06	3.54e-07	2.62e-07
np = 512	BJacobi	535	1.32e-06	3.96e-07	3.49e-07
	CAILU(0)	407	1.36e-06	3.46e-07	3.09e-07
	RAS(1)	476	1.34e-06	3.56e-07	3.09e-07
	RAS(2)	471	1.35e-06	3.55e-07	3.42e-07

Table A.4 – Comparison of CA-ILU(0) with BJacobi and RAS on the problem parabolic_fem for a number of partitions goes from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	1683	9.99e-07	9.99e-07	1.95e-05
np = 16	BJacobi	512	2.04e-06	3.24e-07	3.74e-06
	CAILU(0)	550	2.02e-06	3.13e-07	4.14e-06
	RAS(1)	514	2.05e-06	2.95e-07	3.64e-06
	RAS(2)	497	2.04e-06	2.84e-07	3.46e-06
np = 32	BJacobi	537	2.01e-06	3.60e-07	4.21e-06
	CAILU(0)	572	2.01e-06	3.35e-07	4.21e-06
	RAS(1)	547	2.03e-06	2.91e-07	3.94e-06
	RAS(2)	521	2.03e-06	3.00e-07	3.67e-06
np = 64	BJacobi	566	2.00e-06	3.94e-07	4.42e-06
	CAILU(0)	534	2.00e-06	3.56e-07	4.45e-06
	RAS(1)	523	2.04e-06	3.03e-07	3.83e-06
	RAS(2)	505	2.04e-06	2.95e-07	3.60e-06
np = 128	BJacobi	583	1.95e-06	4.29e-07	4.43e-06
	CAILU(0)	545	1.98e-06	3.69e-07	4.56e-06
	RAS(1)	543	1.99e-06	3.04e-07	4.00e-06
	RAS(2)	513	1.99e-06	3.01e-07	3.65e-06
np = 256	BJacobi	654	1.87e-06	4.42e-07	4.63e-06
	CAILU(0)	567	2.02e-06	3.98e-07	4.73e-06
	RAS(1)	579	1.97e-06	3.06e-07	3.82e-06
	RAS(2)	537	1.98e-06	3.06e-07	3.92e-06

Table A.5 – Comparison of CA-ILU(0) with BJacobi and RAS on the problem of SPE10 for a number of partitions goes from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	399	9.94e-07	9.94e-07	2.13e-04
np = 16	BJacobi	364	1.31e-06	2.80e-07	6.62e-05
	CAILU(0)	337	1.27e-06	2.95e-07	6.05e-05
	RAS(1)	304	1.31e-06	2.71e-07	6.37e-05
	RAS(2)	294	1.29e-06	2.69e-07	6.20e-05
np = 32	BJacobi	363	1.32e-06	2.96e-07	6.96e-05
	CAILU(0)	336	1.31e-06	3.07e-07	6.52e-05
	RAS(1)	305	1.32e-06	2.76e-07	6.47e-05
	RAS(2)	295	1.33e-06	2.76e-07	6.50e-05
np = 64	BJacobi	363	1.30e-06	2.88e-07	6.80e-05
	CAILU(0)	336	1.31e-06	2.99e-07	6.61e-05
	RAS(1)	309	1.30e-06	2.67e-07	6.19e-05
	RAS(2)	301	1.31e-06	2.73e-07	6.34e-05
np = 128	BJacobi	362	1.31e-06	3.12e-07	7.31e-05
	CAILU(0)	342	1.29e-06	3.15e-07	6.73e-05
	RAS(1)	337	1.30e-06	2.69e-07	6.08e-05
	RAS(2)	308	1.33e-06	2.71e-07	6.40e-05
np = 256	BJacobi	372	1.27e-06	3.38e-07	8.09e-05
	CAILU(0)	347	1.29e-06	3.31e-07	7.43e-05
	RAS(1)	337	1.28e-06	2.87e-07	6.20e-05
	RAS(2)	311	1.33e-06	2.81e-07	6.48e-05

Table A.6 – Comparison of CA-ILU(0) with BJacobi and RAS on the problem of 3DSKY100P1 for a number of partitions goes from 16 to 256. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	399	9.94e-07	9.94e-07	1.50e-04
np = 16	BJacobi	440	1.32e-06	2.68e-07	7.37e-05
	CAILU(0)	359	1.30e-06	2.76e-07	6.77e-05
	RAS(1)	351	1.33e-06	2.59e-07	6.68e-05
	RAS(2)	348	1.33e-06	2.47e-07	6.67e-05
np = 32	BJacobi	402	1.32e-06	2.74e-07	7.25e-05
	CAILU(0)	360	1.32e-06	2.81e-07	7.00e-05
	RAS(1)	352	1.32e-06	2.56e-07	6.69e-05
	RAS(2)	352	1.32e-06	2.52e-07	6.69e-05
np = 64	BJacobi	439	1.32e-06	2.84e-07	7.50e-05
	CAILU(0)	369	1.30e-06	2.86e-07	6.99e-05
	RAS(1)	349	1.33e-06	2.58e-07	6.75e-05
	RAS(2)	350	1.33e-06	2.53e-07	6.73e-05
np = 128	BJacobi	398	1.31e-06	2.95e-07	7.55e-05
	CAILU(0)	372	1.31e-06	2.92e-07	7.22e-05
	RAS(1)	347	1.32e-06	2.60e-07	6.79e-05
	RAS(2)	350	1.32e-06	2.54e-07	6.71e-05
np = 256	BJacobi	399	1.31e-06	3.07e-07	7.69e-05
	CAILU(0)	345	1.30e-06	3.01e-07	7.40e-05
	RAS(1)	349	1.32e-06	2.63e-07	6.86e-05
	RAS(2)	348	1.32e-06	2.58e-07	6.79e-05
np = 512	BJacobi	395	1.29e-06	3.21e-07	7.91e-05
	CAILU(0)	347	1.29e-06	3.17e-07	7.53e-05
	RAS(1)	351	1.32e-06	2.63e-07	6.96e-05
	RAS(2)	348	1.32e-06	2.60e-07	6.80e-05

Table A.7 – Comparison of CA-ILU(0) with BJacobi and RAS on the problem of 3DSKY150P1 for a number of partitions goes from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$

		Domain size			Overlap size		
	PC	mean	min	max	mean	min	max
np = 16	CAILU(0)	10000	9979	10025	7.45e+02	4.71e+02	1.12e+03
	RAS(1)	-	-	-	3.29e+02	2.00e+02	4.62e+02
	RAS(2)	-	-	-	6.48e+02	3.99e+02	9.07e+02
np = 32	CAILU(0)	5000	4939	5044	6.17e+02	2.88e+02	9.65e+02
	RAS(1)	-	-	-	2.56e+02	1.41e+02	3.65e+02
	RAS(2)	-	-	-	5.08e+02	2.80e+02	7.23e+02
np = 64	CAILU(0)	2500	2473	2547	5.02e+02	2.05e+02	8.30e+02
	RAS(1)	-	-	-	1.91e+02	1.01e+02	2.45e+02
	RAS(2)	-	-	-	3.79e+02	2.00e+02	4.88e+02
np = 128	CAILU(0)	1250	1220	1287	3.84e+02	1.47e+02	5.77e+02
	RAS(1)	-	-	-	1.38e+02	7.20e+01	1.71e+02
	RAS(2)	-	-	-	2.75e+02	1.44e+02	3.39e+02
np = 256	CAILU(0)	625	606	643	2.92e+02	1.02e+02	5.07e+02
	RAS(1)	-	-	-	9.92e+01	5.00e+01	1.30e+02
	RAS(2)	-	-	-	2.00e+02	1.00e+02	2.62e+02

Table A.8 – Comparison of the overlap of CA-ILU(0) with RAS on matvf2dAD400400 for different number of partitions.

		Domain size			Overlap size		
	PC	mean	min	max	mean	min	max
np = 16	CAILU(0)	68401	68385	68414	1.62e+04	1.17e+04	2.25e+04
	RAS(1)	-	-	-	5.49e+03	4.58e+03	6.51e+03
	RAS(2)	-	-	-	1.11e+04	9.27e+03	1.32e+04
np = 32	CAILU(0)	34200	34055	34244	1.25e+04	7.68e+03	1.80e+04
	RAS(1)	-	-	-	3.93e+03	2.65e+03	4.56e+03
	RAS(2)	-	-	-	8.01e+03	5.45e+03	9.33e+03
np = 64	CAILU(0)	17100	17021	17146	1.04e+04	4.55e+03	2.12e+04
	RAS(1)	-	-	-	2.81e+03	1.78e+03	3.89e+03
	RAS(2)	-	-	-	5.74e+03	3.59e+03	7.93e+03
np = 128	CAILU(0)	8550	8472	8665	8.63e+03	3.22e+03	1.82e+04
	RAS(1)	-	-	-	1.91e+03	1.15e+03	2.58e+03
	RAS(2)	-	-	-	3.93e+03	2.32e+03	5.30e+03
np = 256	CAILU(0)	4275	4202	4368	7.25e+03	2.37e+03	1.79e+04
	RAS(1)	-	-	-	1.28e+03	7.09e+02	1.76e+03
	RAS(2)	-	-	-	2.67e+03	1.47e+03	3.68e+03
np = 512	CAILU(0)	2137	2075	2201	6.80e+03	1.57e+03	1.69e+04
	RAS(1)	-	-	-	8.46e+02	4.21e+02	1.15e+03
	RAS(2)	-	-	-	1.79e+03	8.73e+02	2.45e+03

Table A.9 – Comparison of the overlap of CA-ILU(0) with RAS on SPE10 for different number of partitions.

		Domain size			Overlap size		
	PC	mean	min	max	mean	min	max
np = 16	CAILU(0)	62500	62487	62509	1.77e+04	9.74e+03	3.16e+04
	RAS(1)	-	-	-	6.09e+03	4.44e+03	8.11e+03
	RAS(2)	-	-	-	1.22e+04	8.92e+03	1.62e+04
np = 32	CAILU(0)	31250	31237	31268	1.57e+04	7.19e+03	2.87e+04
	RAS(1)	-	-	-	4.28e+03	2.80e+03	5.84e+03
	RAS(2)	-	-	-	8.65e+03	5.57e+03	1.19e+04
np = 64	CAILU(0)	15625	15612	15648	1.19e+04	4.66e+03	2.58e+04
	RAS(1)	-	-	-	2.80e+03	1.74e+03	3.76e+03
	RAS(2)	-	-	-	5.71e+03	3.52e+03	7.73e+03
np = 128	CAILU(0)	7812	7792	7842	1.11e+04	2.69e+03	2.48e+04
	RAS(1)	-	-	-	1.96e+03	1.16e+03	2.55e+03
	RAS(2)	-	-	-	4.01e+03	2.34e+03	5.25e+03
np = 256	CAILU(0)	3906	3881	3986	8.84e+03	2.38e+03	2.00e+04
	RAS(1)	-	-	-	1.30e+03	7.41e+02	1.68e+03
	RAS(2)	-	-	-	2.70e+03	1.51e+03	3.52e+03

Table A.10 – Comparison of the overlap of CA-ILU(0) with RAS on 3DSKY100P1 for different number of partitions.

		Domain size			Overlap size		
	PC	mean	min	max	mean	min	max
np = 16	CAILU(0)	210937	210504	217276	3.56e+04	2.31e+04	5.25e+04
	RAS(2)	-	-	-	2.75e+04	2.03e+04	3.79e+04
	RAS(1)	-	-	-	1.37e+04	1.02e+04	1.90e+04
np = 32	CAILU(0)	105468	105458	105482	2.99e+04	1.50e+04	5.35e+04
	RAS(2)	-	-	-	1.94e+04	1.30e+04	2.63e+04
	RAS(1)	-	-	-	9.69e+03	6.51e+03	1.31e+04
np = 64	CAILU(0)	52734	52715	52763	2.16e+04	9.93e+03	3.70e+04
	RAS(2)	-	-	-	1.28e+04	7.96e+03	1.75e+04
	RAS(1)	-	-	-	6.37e+03	3.99e+03	8.61e+03
np = 128	CAILU(0)	26367	26337	26387	1.70e+04	6.55e+03	3.37e+04
	RAS(2)	-	-	-	8.89e+03	4.95e+03	1.15e+04
	RAS(1)	-	-	-	4.39e+03	2.48e+03	5.66e+03
np = 256	CAILU(0)	13183	13163	13205	1.47e+04	3.96e+03	2.74e+04
	RAS(2)	-	-	-	5.88e+03	3.15e+03	7.61e+03
	RAS(1)	-	-	-	2.88e+03	1.56e+03	3.68e+03
np = 512	CAILU(0)	6591	6564	6616	1.11e+04	3.29e+03	2.32e+04
	RAS(2)	-	-	-	3.88e+03	2.10e+03	5.00e+03
	RAS(1)	-	-	-	1.88e+03	1.03e+03	2.41e+03

Table A.11 – Comparison of the overlap of CA-ILU(0) with RAS on 3DSKY150P1 for different number of partitions.

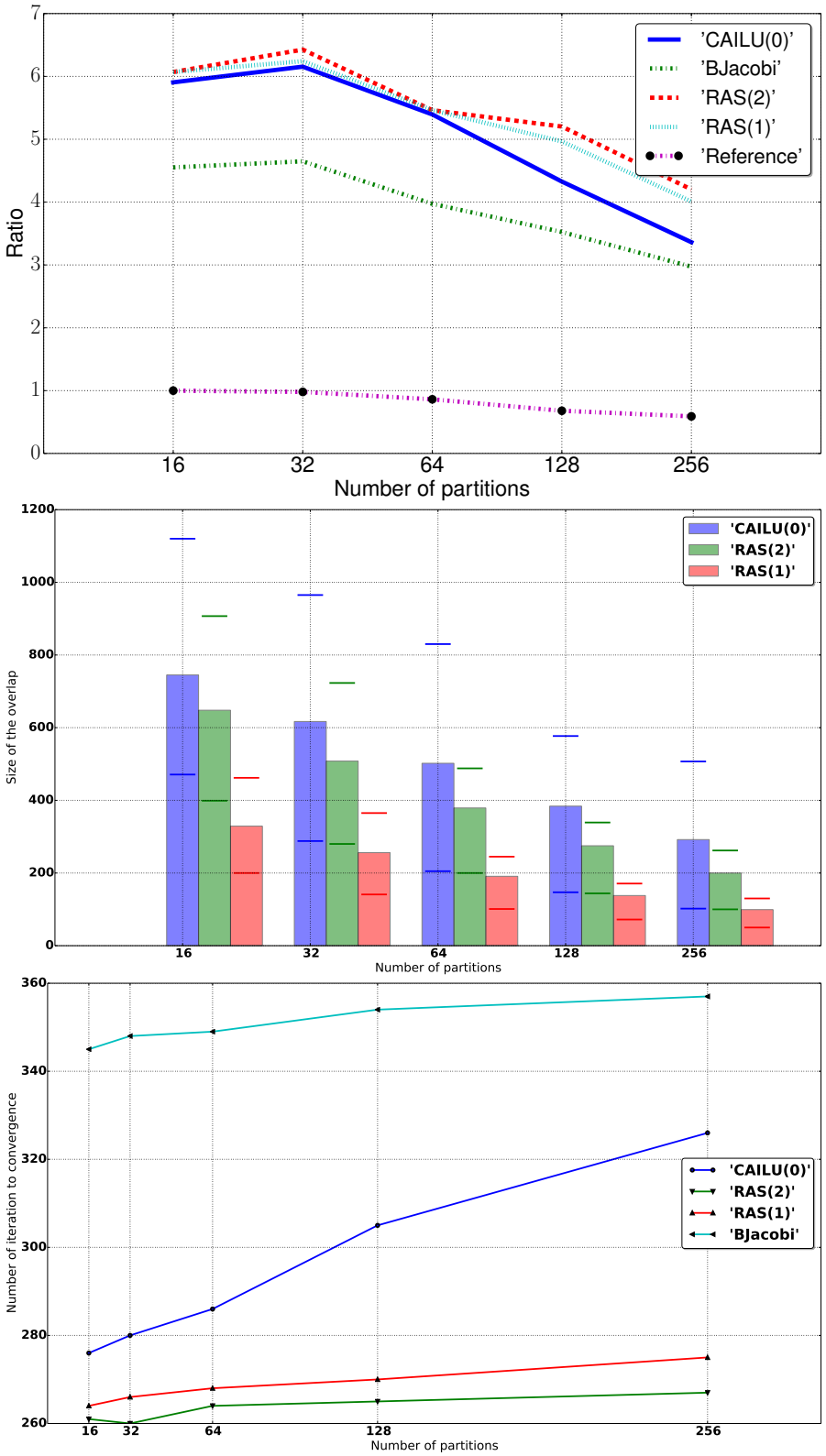


Figure A.1 – Comparison of CA-ILU(0) with BJacobi, RAS(1), RAS(2) and without a preconditioner (labeled *Reference*), on the problem matvf2dAD400400 from 16 to 256 partitions. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

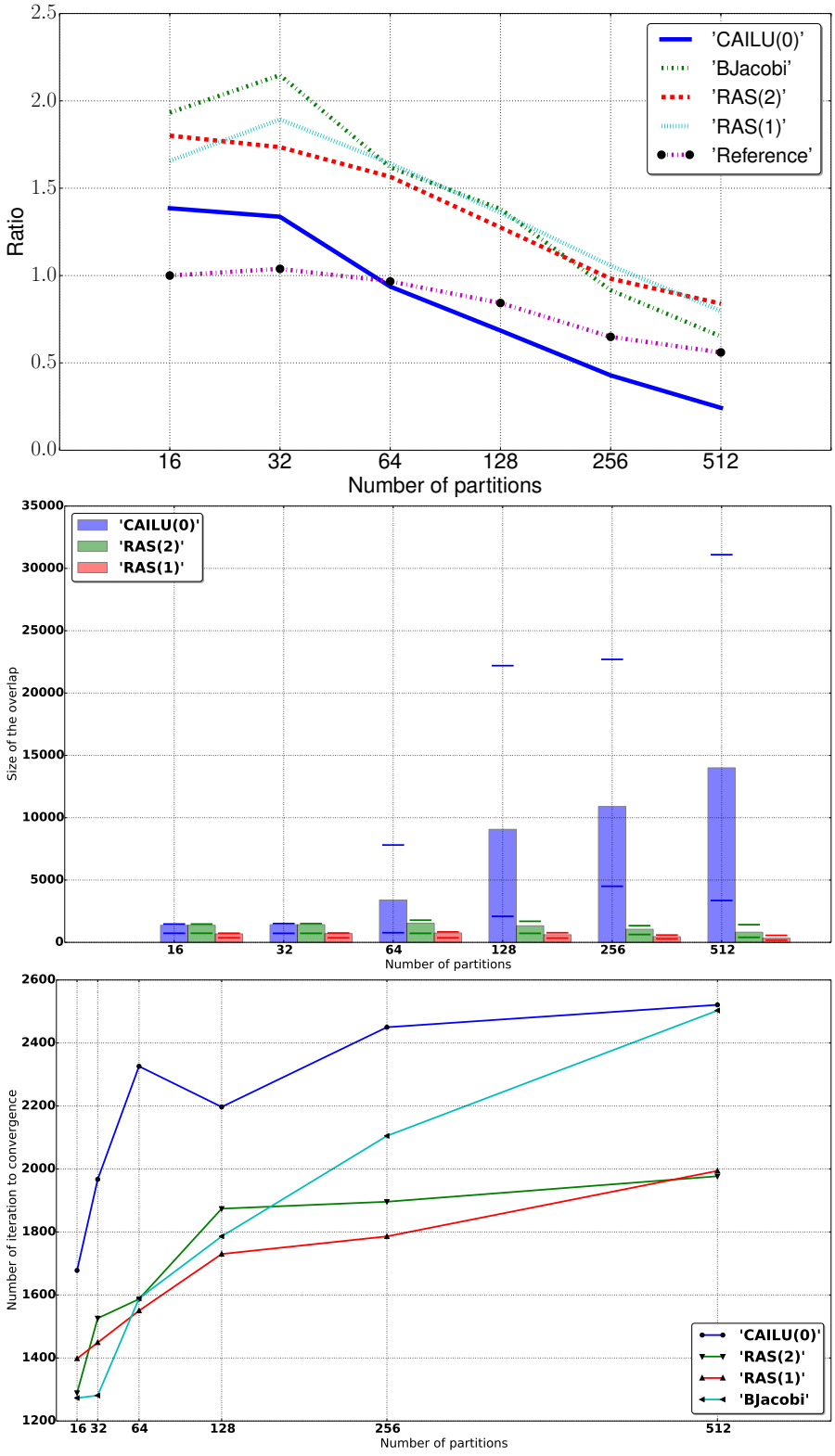


Figure A.2 – Comparison of CA-ILU(0) with BJacobi, RAS(1), RAS(2) and without a preconditioner (labeled *Reference*), on the problem Elasticity3D4001010 from 16 to 256 partitions. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

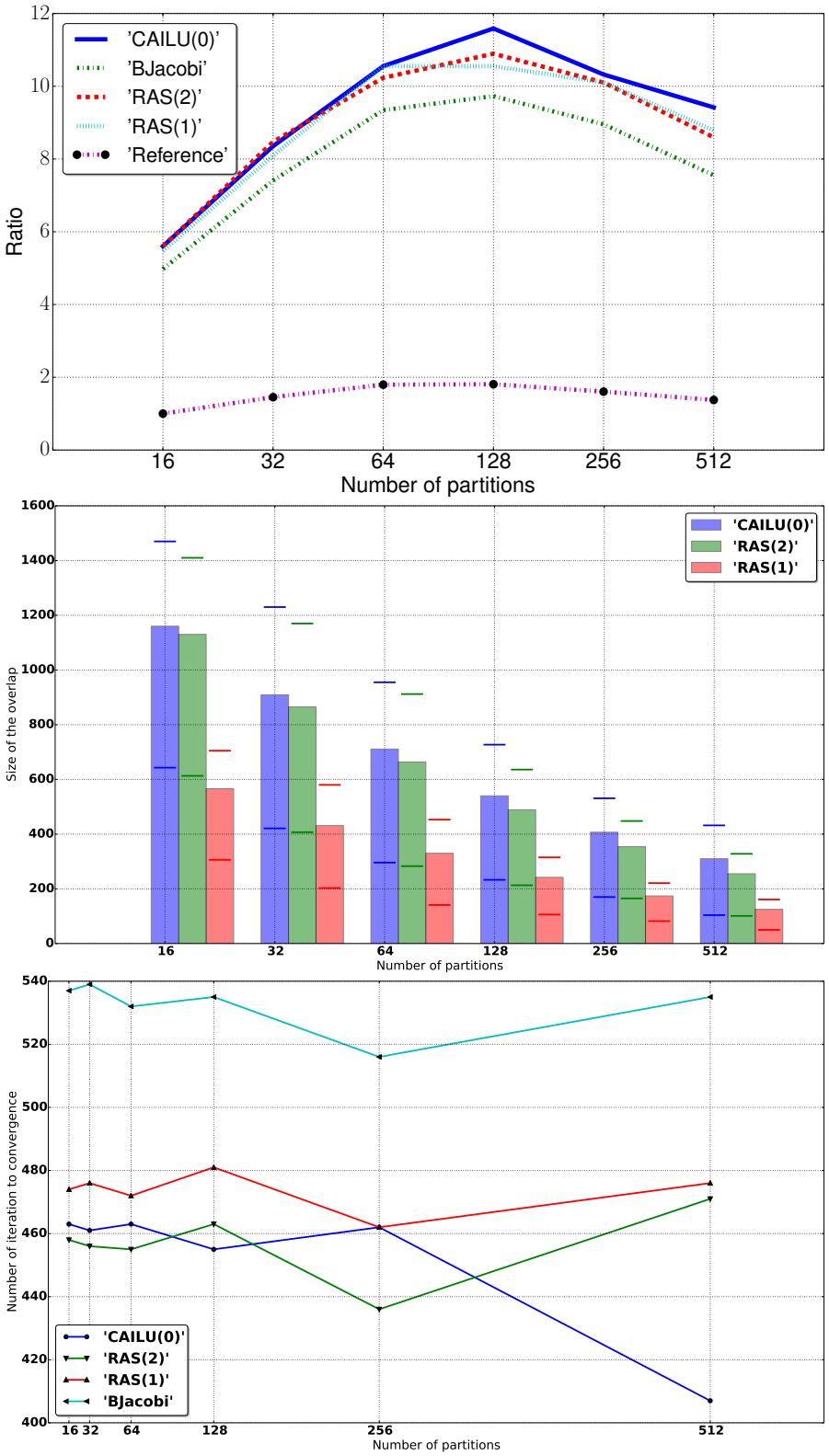


Figure A.3 – Comparison of CA-ILU(0) with BJacobi, RAS(1), RAS(2) and without a preconditioner (labeled *Reference*), on the problem parabolic_fem from 16 to 512 partitions. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

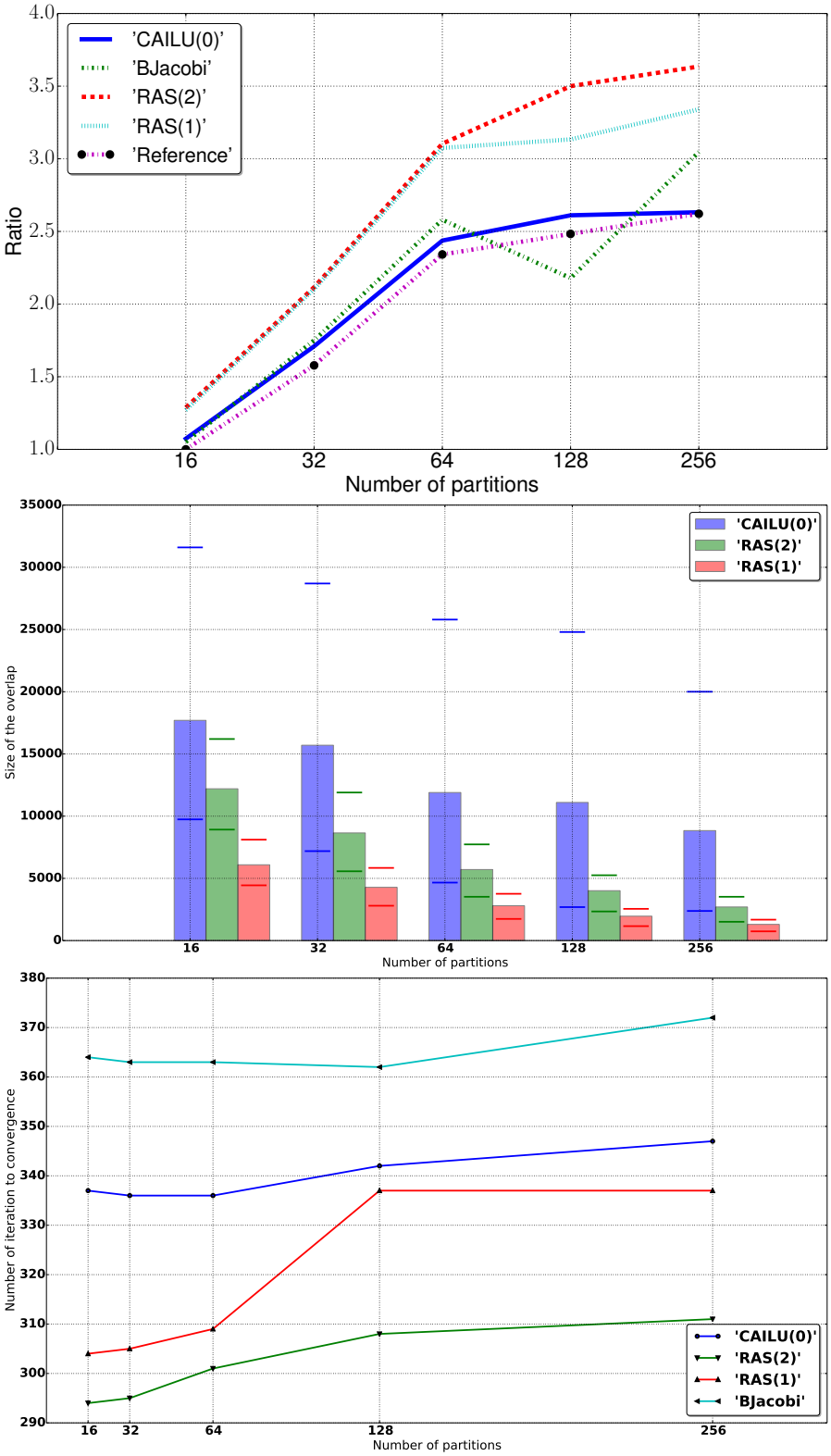


Figure A.4 – Comparison of CA-ILU(0) with BJacobi, RAS(1), RAS(2) and without a preconditioner (labeled *Reference*), on the problem 3DSKY100P1 from 16 to 512 partitions. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

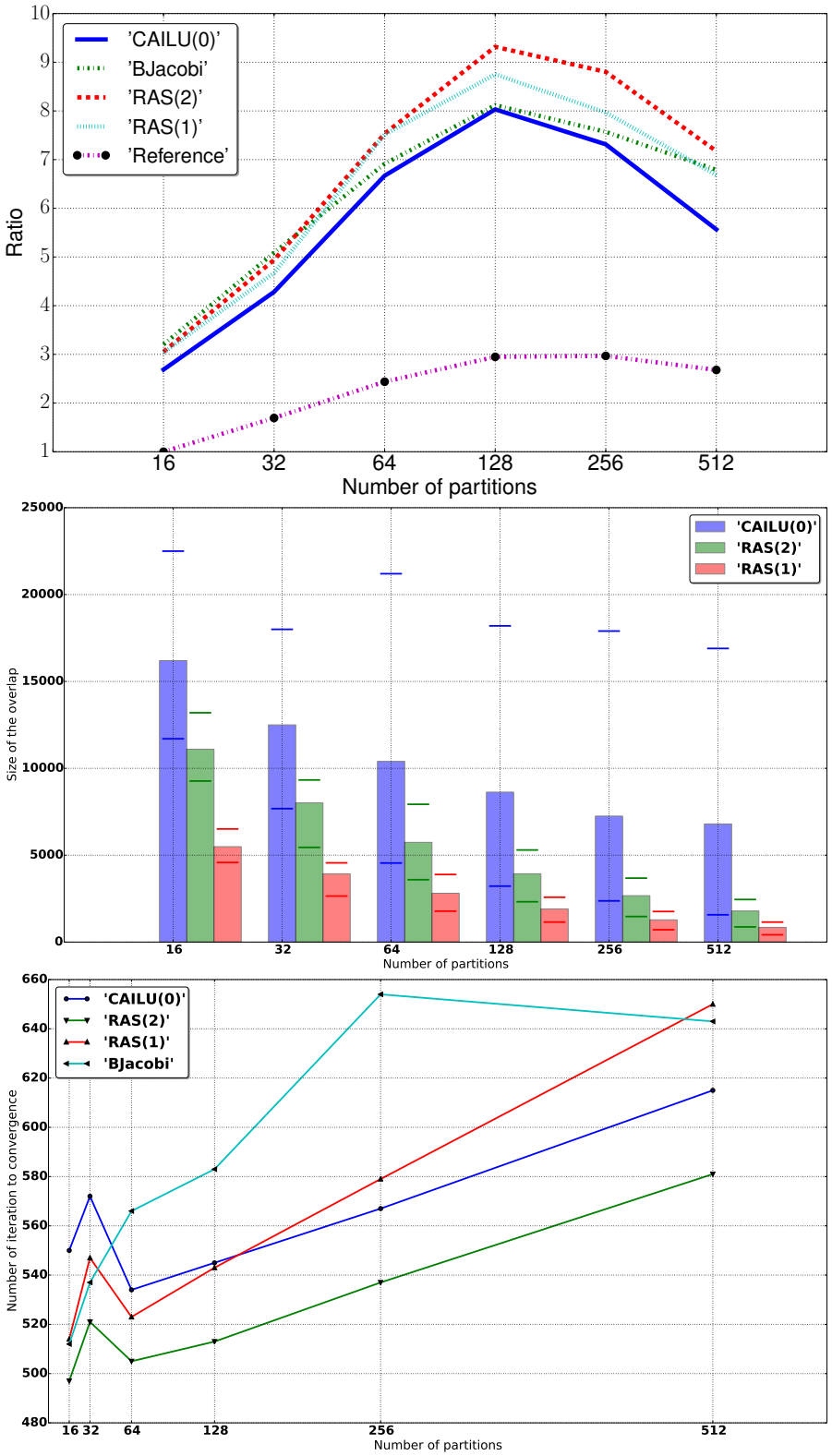


Figure A.5 – Comparison of CA-ILU(0) with BJacobi, RAS(1), RAS(2) and without a preconditioner (labeled *Reference*), on the problem SPE10 from 16 to 512 partitions. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

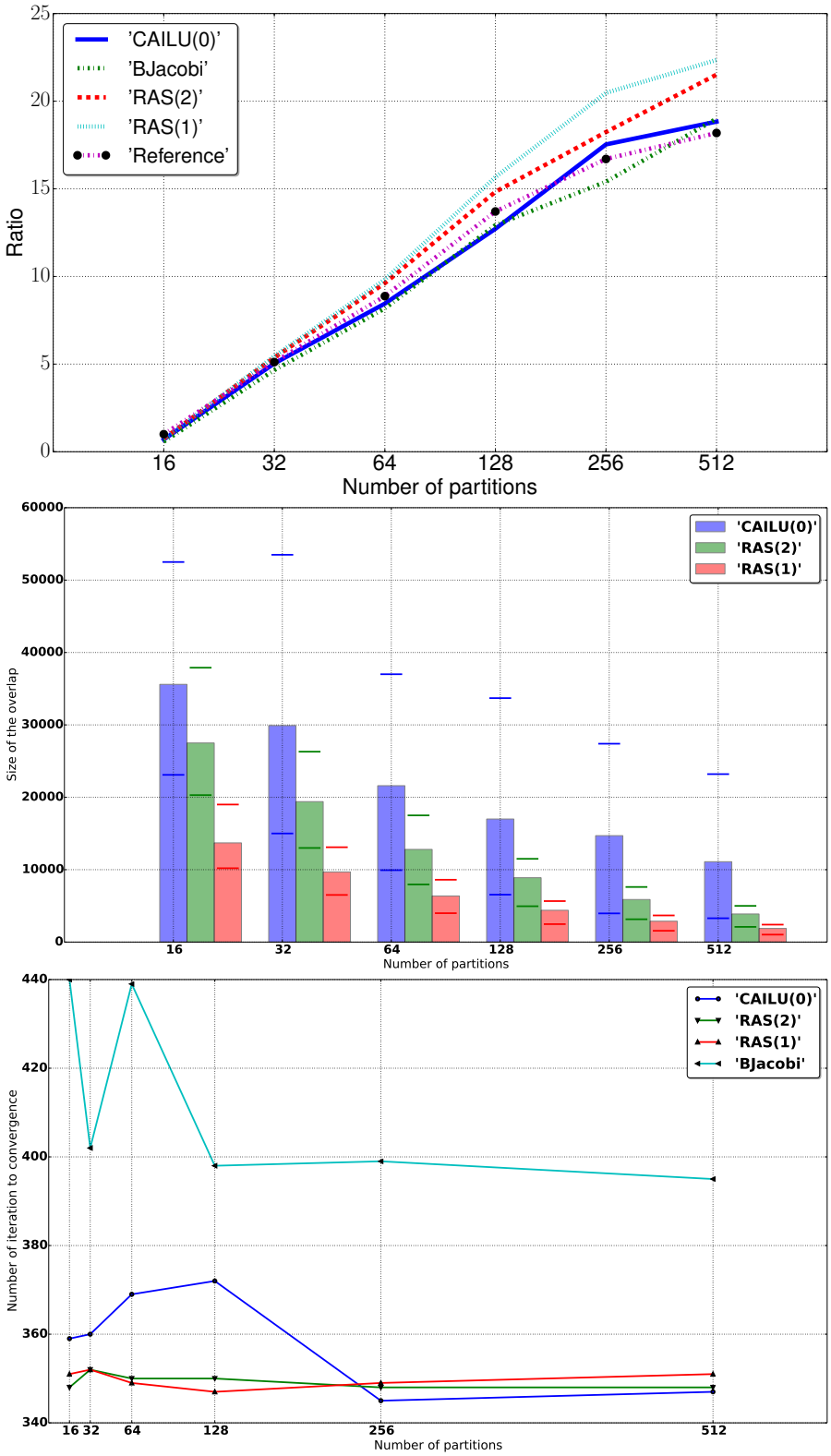


Figure A.6 – Comparison of CA-ILU(0) with BJacobi, RAS(1), RAS(2) and without a preconditioner (labeled *Reference*), on the problem 3DSKY150P1 from 16 to 512 partitions. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$	cond$(M^{-1}A)$	$\frac{\|\|A-LU\|\|_2}{\|\|A\|\|_2}$
References	No precond	350	9.8e-09	9.8e-09	3.6e-08	2.4e+04	0.0e+00
	ILU(0) NatOrder	142	9.8e-09	1.9e-09	2.1e-07	3.4e+03	7.4e-02
np = 2	ILU(0) Kway	161	1.0e-08	1.8e-09	1.6e-07	1.4e+04	1.5e-01
	CAILU(0)	180	1.0e-08	1.8e-09	3.6e-07	1.8e+04	2.0e-01
	BJacobi-ilu(0)	196	9.8e-09	1.7e-09	3.6e-07	2.2e+04	2.2e-01
np = 4	ILU(0) Kway	156	9.3e-09	1.8e-09	2.0e-07	1.3e+04	1.5e-01
	CAILU(0)	170	9.0e-09	2.0e-09	1.2e-07	2.3e+04	2.2e-01
	BJacobi-ilu(0)	187	9.7e-09	2.1e-09	1.6e-07	2.4e+04	2.2e-01
	RAS(2)-ilu(0)	161	9.8e-09	2.1e-09	2.6e-07	1.2e+04	-
np = 8	ILU(0) Kway	166	9.3e-09	2.0e-09	1.3e-07	1.6e+04	1.8e-01
	CAILU(0)	181	9.1e-09	1.9e-09	1.6e-07	2.1e+04	2.2e-01
	BJacobi-ilu(0)	203	9.8e-09	2.0e-09	1.9e-07	2.5e+04	2.6e-01
	RAS(2)-ilu(0)	177	9.5e-09	2.1e-09	1.6e-07	1.5e+04	-
np = 16	ILU(0) Kway	160	1.0e-08	1.9e-09	1.3e-07	1.1e+04	1.5e-01
	CAILU(0)	185	9.5e-09	2.0e-09	1.7e-07	2.1e+04	2.3e-01
	BJacobi-ilu(0)	207	9.5e-09	2.1e-09	1.8e-07	2.4e+04	2.5e-01
	RAS(2)-ilu(0)	177	9.3e-09	2.0e-09	1.9e-07	1.0e+04	-

Table A.12 – Comparison of the stability of CA-ILU(k) with BJacobi and RAS for $k = 0$ on the problem of Non-homogeneous 2D 200x200 for different number of partitions *np*. As references, the system is solved without preconditioner and with sequential ILU(0) preconditioner.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$	cond$(M^{-1}A)$	$\frac{\|\|A-LU\|\|_2}{\|\|A\|\|_2}$
References	No precond	350	9.8e-09	9.8e-09	3.6e-08	2.4e+04	0.0e+00
	ILU(0) NatOrder	142	9.8e-09	1.9e-09	2.1e-07	3.4e+03	7.4e-02
np = 2	ILU(1) Kway	109	9.4e-09	8.1e-10	1.2e-07	1.4e+04	7.2e-02
	CAILU(1)	120	8.6e-09	9.4e-10	8.4e-08	1.6e+04	7.7e-02
	BJacobi-ilu(1)	142	9.5e-09	1.4e-09	9.5e-08	3.4e+04	2.1e-01
np = 4	ILU(1) Kway	108	9.6e-09	8.0e-10	2.0e-07	1.5e+04	7.4e-02
	CAILU(1)	115	9.1e-09	8.6e-10	2.0e-07	1.8e+04	9.0e-02
	BJacobi-ilu(1)	137	9.5e-09	1.1e-09	2.1e-07	3.6e+04	2.1e-01
	RAS(2)-ilu(1)	118	9.8e-09	1.2e-09	1.5e-07	1.4e+04	-
np = 8	ILU(1) Kway	108	9.1e-09	1.1e-09	9.2e-08	1.3e+04	7.7e-02
	CAILU(1)	122	9.6e-09	1.1e-09	1.4e-07	1.7e+04	9.2e-02
	BJacobi-ilu(1)	148	9.4e-09	1.6e-09	9.2e-08	3.7e+04	2.5e-01
	RAS(2)-ilu(1)	118	9.5e-09	1.2e-09	1.6e-07	1.4e+04	-
np = 16	ILU(1) Kway	113	9.2e-09	1.1e-09	9.4e-08	1.5e+04	8.4e-02
	CAILU(1)	125	8.4e-09	1.1e-09	7.4e-08	1.6e+04	9.5e-02
	BJacobi-ilu(1)	153	9.5e-09	1.7e-09	9.8e-08	3.6e+04	2.3e-01
	RAS(2)-ilu(1)	122	8.1e-09	1.3e-09	8.6e-08	1.4e+04	-

Table A.13 – Comparison of the stability of CA-ILU(k) with BJacobi and RAS for $k = 1$ on the problem of Non-homogeneous 2D 200x200 for different number of partitions *np*. As references, the system is solved without preconditioner and with sequential ILU(1) preconditioner.

nsubdomain	preconditioner	niter	relative residual	$\frac{\|\|b-Ax\|\|_2}{\|\|b\|\|_2}$	$\frac{\|\|x-x_s\|\|_2}{\|\|x_s\|\|_2}$
	Reference	1683	9.99e-07	9.99e-07	1.95e-05
n = 16	BJacobi	352	2.59e-06	3.23e-07	9.46e-07
	CAILU(1)	329	2.64e-06	2.26e-07	1.08e-06
	RAS(2)	321	2.63e-06	2.01e-07	1.15e-06
	RAS(3)	**316**	2.56e-06	1.73e-07	1.21e-06
n = 32	BJacobi	366	2.53e-06	3.48e-07	9.81e-07
	CAILU(1)	343	2.48e-06	2.44e-07	8.98e-07
	RAS(2)	328	2.55e-06	2.24e-07	9.73e-07
	RAS(3)	**323**	2.57e-06	2.02e-07	1.08e-06
n = 64	BJacobi	373	2.50e-06	4.05e-07	1.13e-06
	CAILU(1)	350	2.63e-06	2.83e-07	9.42e-07
	RAS(2)	335	2.56e-06	2.55e-07	8.99e-07
	RAS(3)	**327**	2.56e-06	2.10e-07	1.00e-06
n = 128	BJacobi	382	2.46e-06	5.09e-07	2.01e-06
	CAILU(1)	361	2.44e-06	2.99e-07	9.38e-07
	RAS(2)	340	2.59e-06	2.61e-07	8.88e-07
	RAS(3)	**335**	2.58e-06	2.28e-07	9.49e-07
n = 256	BJacobi	435	2.34e-06	4.39e-07	1.93e-06
	CAILU(1)	365	2.41e-06	2.76e-07	8.79e-07
	RAS(2)	350	2.56e-06	2.90e-07	8.25e-07
	RAS(3)	**340**	2.52e-06	2.37e-07	9.08e-07

Table A.14 – Comparison of CA-ILU(1) with BJacobi and RAS on the problem of SPE10 for a number of partitions increasing from 16 to 512. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

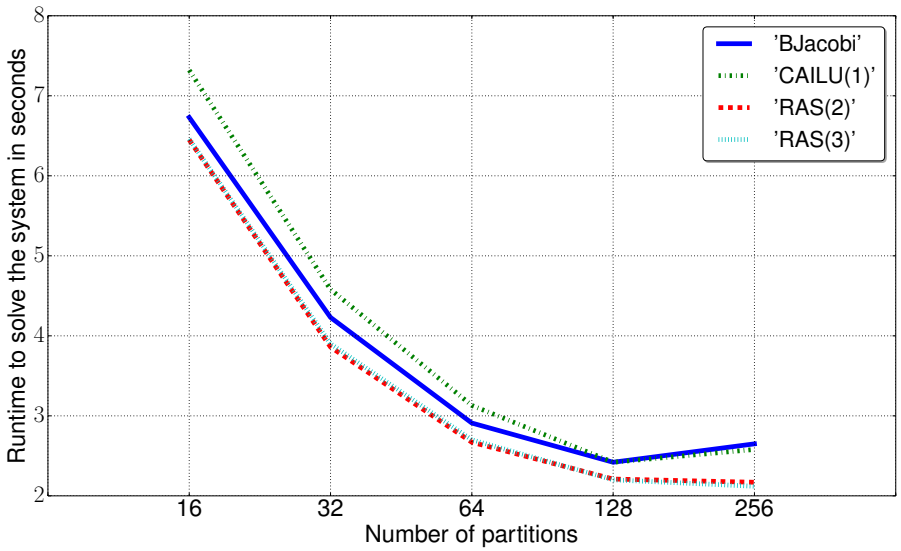


Figure A.7 – Comparison of CA-ILU(0) with BJacobi-ilu(1), RAS(2)-ilu(1) and RAS(3)-ilu(1) on the problem SPE10 from 16 to 512 partitions. GMRES is set with a maximum of 3000 iterations, a restart of 200 and a relative tolerance of $1e-6$.

Appendix B

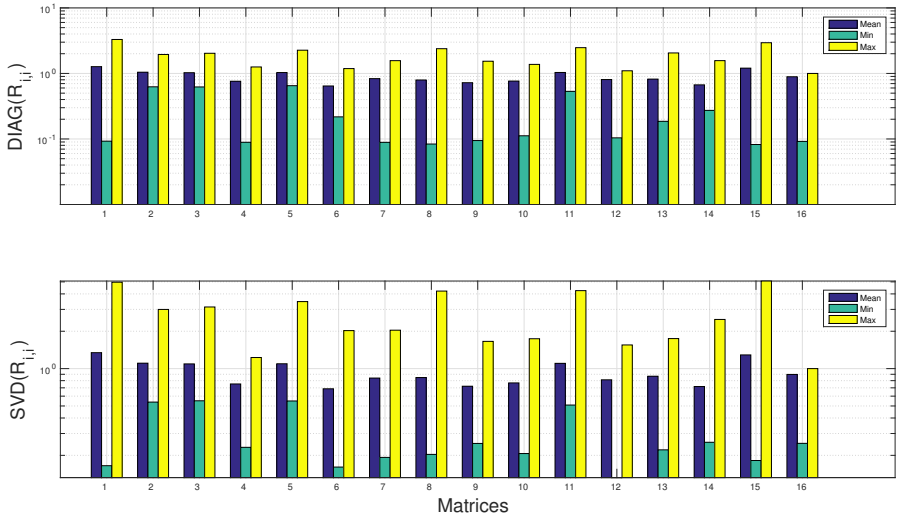
LU-CRTP

Outline of the current chapter

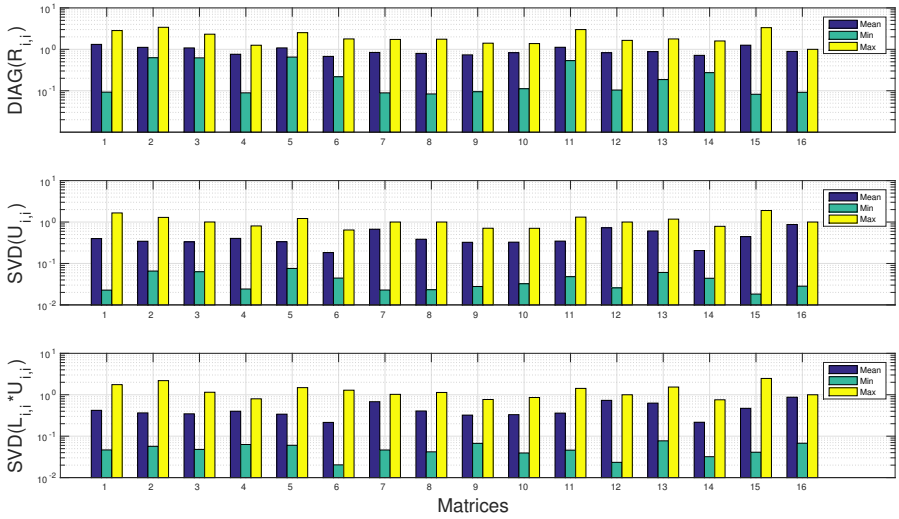
B.1 Appendix

Impact of using the diagonal elements of R instead of computing the SVD of R

To obtain an approximation of the singular values of a matrix A, Algorithm 4.1 uses the absolute diagonal elements of $R_{i,i}$, where $R_{i,i}$ is the triangular factor returned by QRCP applied on the k selected columns, at iteration i of the algorithm. In the following, we compare the accuracy of the approximate singular values of LU-CRTP with variants of LU-CRTP that use a different way to compute them. The first variant computes the singular values of $R_{i,i}$ using the SVD algorithm, and is denoted as $SVD(R_{i,i})$. The second variant computes the Singular Value Decomposition of $U_{i,i}$ whereas the third variant computes the Singular Value Decomposition of $L_{i,i}U_{i,i}$. The last two variants are referred to as $SVD(U_{i,i})$ and $SVD(L_{i,i}U_{i,i})$, respectively. Figures B.1a and B.1b compare the average ratios, its minimum and maximum, of each variant and for each matrix, with LU-CRTP with $DIAG(R_{i,i})$, with $k = 16$ and $nSV = 64$. First, the computation of the singular values of $R_{i,i}$ reduces the error on the minimum ratios, compared to the minimum ratios of LU-CRTP with $DIAG(R_{i,i})$. However, the maximum ratios are equivalent for both methods. When considering $U_{i,i}$ and $L_{i,i}U_{i,i}$ instead of $R_{i,i}$, we observe that the minimum ratios are degraded and are smaller than 10^{-1}, and the maximum ratios are not enhanced. Figures B.2a and B.2b present the results on the same tests as above, with $nSV = 128$ and $nSV = 256$. As for $nSV = 64$, the smallest minimum ratios for $SVD(R_{i,i})$ do not reach 10^{-1} whereas, for few matrices, LU-CRTP has an error that reaches this limit. However, the largest maximum ratios for $SVD(R_{i,i})$ are worse than for LU-CRTP, for several matrices (id 5, 6, 11, 14). Again, the smallest minimum ratios for $SVD(U_{i,i})$ and $SVD(L_{i,i}U_{i,i})$ are smaller than 10^{-1} and even closer to 10^{-2}, whereas these ratios are at most of 10^{-1} for LU-CRTP with $DIAG(R_{i,i})$. The largest maximum ratios for these two last variants are the same as LU-CRTP. Therefore, LU-CRTP using the diagonal elements of $R_{i,i}$ to approximate the singular values does not degrade the accuracy of the maximum ratios, in comparison with $SVD(R_{i,i})$. Moreover, the cost to obtain them is just a copy of the diagonal elements instead of computing an SVD.

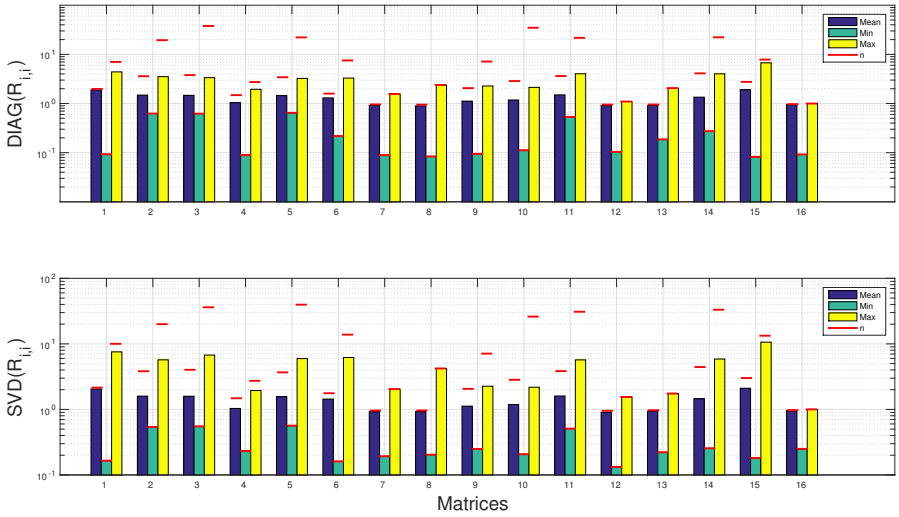


(a) Diagonal of R versus SVD of R.

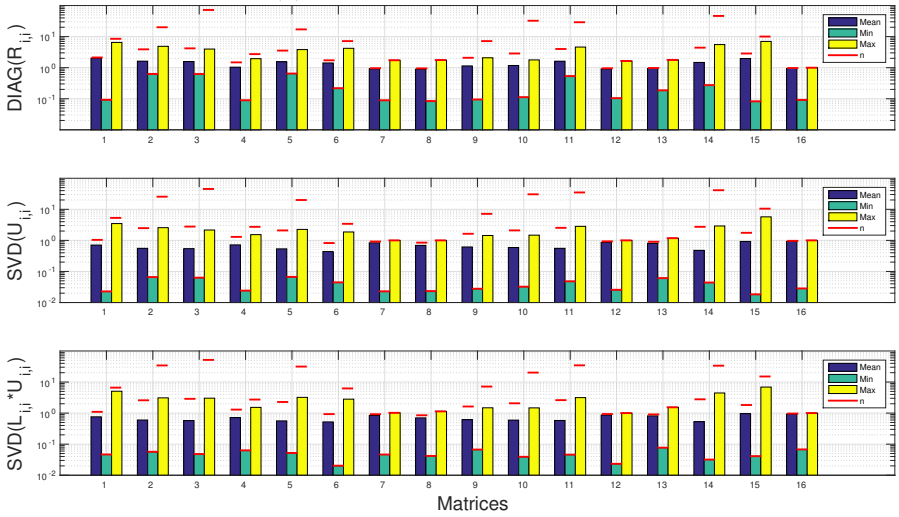


(b) Diagonal of R versus SVD of $U_{i,i}$ versus SVD of $L_{i,i}U_{i,i}$.

Figure B.1 – Comparison of different methods used by LU-CRTP to compute the ASV. Matrix size is 256, $k = 16$ and $nSV = 64$.



(a) Diagonal of R versus SVD of R.



(b) Diagonal of R versus SVD of $U_{i,i}$ versus SVD of $L_{i,i}U_{i,i}$.

Figure B.2 – Comparison of different methods used by LU-CRTP to compute the ASV. Matrix size is 256, $k = 16$ and $nSV \in \{128, 256\}$.

Matrix	k	growth factor	$\|\|L\|\|_1$	$\|\|L^{-1}\|\|_1$	$\|\|L\|\|_{max}$	$\|\|U\|\|_1$	$\|\|U^{-1}\|\|_1$	$\|\|U\|\|_{max}$	$\|\|A\|\|_{max}$	$\frac{\|\|PAE-LU\|\|_F}{\|\|A\|\|_F}$
orani678	16	1.00e+00	1.43e+02	1.12e+02	1.49e+00	5.52e+01	1.40e+02	4.09e+00	4.09e+00	2.17e-16
	32	1.00e+00	1.42e+02	1.16e+02	1.01e+00	7.65e+01	1.93e+02	4.09e+00	4.09e+00	1.91e-16
	64	1.00e+00	1.31e+02	1.18e+02	1.00e+00	9.73e+01	2.22e+02	4.09e+00	4.09e+00	1.56e-16
	128	1.00e+00	1.36e+02	1.58e+02	1.00e+00	1.45e+02	2.88e+02	4.09e+00	4.09e+00	1.51e-16
gemat11	16	1.00e+00	5.22e+00	5.32e+00	1.00e+00	6.51e+02	3.76e+00	6.23e+02	6.23e+02	4.67e-17
	32	1.00e+00	5.22e+00	5.32e+00	1.00e+00	6.51e+02	5.84e+00	6.23e+02	6.23e+02	4.77e-17
	64	1.00e+00	5.22e+00	5.32e+00	1.00e+00	6.51e+02	5.84e+00	6.23e+02	6.23e+02	4.67e-17
	128	1.00e+00	5.22e+00	5.32e+00	1.00e+00	6.51e+02	6.01e+00	6.23e+02	6.23e+02	4.64e-17
raefsky3	16	1.00e+00	1.00e+00	1.00e+00	1.00e+00	7.99e+00	1.14e+00	7.99e+00	7.99e+00	6.25e-17
	32	1.00e+00	1.00e+00	1.00e+00	1.00e+00	7.99e+00	1.14e+00	7.99e+00	7.99e+00	6.07e-17
	64	1.00e+00	1.00e+00	1.00e+00	1.00e+00	7.99e+00	1.14e+00	7.99e+00	7.99e+00	4.92e-17
	128	1.00e+00	1.00e+00	1.00e+00	1.00e+00	7.99e+00	1.14e+00	7.99e+00	7.99e+00	3.04e-17
wang3	16	1.00e+00	2.00e+00	2.29e+00	1.00e+00	1.06e+00	1.91e+01	1.69e-01	1.69e-01	3.29e-17
	32	1.00e+00	2.00e+00	2.28e+00	1.00e+00	1.06e+00	1.91e+01	1.69e-01	1.69e-01	3.04e-17
	64	1.00e+00	2.00e+00	2.27e+00	1.00e+00	1.06e+00	1.91e+01	1.69e-01	1.69e-01	2.41e-17
	128	1.00e+00	2.00e+00	2.14e+00	1.00e+00	1.06e+00	1.91e+01	1.69e-01	1.69e-01	2.47e-17
onetone2	16	1.00e+00	4.04e+00	4.04e+00	1.00e+00	2.01e+03	1.50e+00	2.01e+03	2.01e+03	2.13e-17
	32	1.00e+00	4.04e+00	4.04e+00	1.00e+00	2.01e+03	1.50e+00	2.01e+03	2.01e+03	2.13e-17
	64	1.00e+00	4.04e+00	4.04e+00	1.00e+00	2.01e+03	1.50e+00	2.01e+03	2.01e+03	2.13e-17
	128	1.00e+00	4.04e+00	4.04e+00	1.00e+00	2.01e+03	1.50e+00	2.01e+03	2.01e+03	2.12e-17
TSOPF_RS_b39_c30	16	1.00e+00	9.19e+01	9.68e+01	1.00e+00	1.11e+03	1.19e+02	6.46e+02	6.46e+02	7.07e-17
	32	1.00e+00	9.90e+01	9.90e+01	1.00e+00	1.24e+03	1.01e+02	6.46e+02	6.46e+02	5.41e-17
	64	1.00e+00	9.90e+01	9.90e+01	1.00e+00	1.39e+03	1.00e+02	6.46e+02	6.46e+02	4.39e-17
	128	1.00e+00	1.74e+01	1.98e+01	1.00e+00	1.39e+03	3.60e+01	6.46e+02	6.46e+02	4.42e-17
RFdevice	16	1.00e+00	1.00e+00	1.00e+00	1.00e+00	5.71e+08	1.00e+00	1.50e+08	1.50e+08	3.98e-24
	32	1.00e+00	1.00e+00	1.00e+00	1.00e+00	5.71e+08	1.00e+00	1.50e+08	1.50e+08	4.06e-24
	64	1.00e+00	1.00e+00	1.00e+00	1.00e+00	5.71e+08	1.00e+00	1.50e+08	1.50e+08	3.90e-24
	128	1.00e+00	1.00e+00	1.00e+00	1.00e+00	5.71e+08	1.00e+00	1.50e+08	1.50e+08	4.18e-24
ncvxqp3	16	1.00e+00	3.80e+00	3.80e+00	1.00e+00	4.25e+05	2.21e+00	4.25e+05	4.25e+05	1.94e-17
	32	1.00e+00	3.80e+00	3.80e+00	1.00e+00	4.25e+05	2.21e+00	4.25e+05	4.25e+05	1.93e-17
	64	1.00e+00	3.80e+00	3.80e+00	1.00e+00	4.25e+05	2.21e+00	4.25e+05	4.25e+05	1.94e-17
	128	1.00e+00	3.80e+00	3.80e+00	1.00e+00	4.25e+05	2.21e+00	4.25e+05	4.25e+05	1.93e-17
mac_econ_fwd500	16	1.00e+00	3.17e+00	3.17e+00	1.00e+00	1.45e+05	1.11e+00	1.45e+05	1.45e+05	1.40e-18
	32	1.00e+00	3.17e+00	3.17e+00	1.00e+00	1.45e+05	1.11e+00	1.45e+05	1.45e+05	1.40e-18
	64	1.00e+00	3.17e+00	3.17e+00	1.00e+00	1.45e+05	1.11e+00	1.45e+05	1.45e+05	1.40e-18
	128	1.00e+00	3.17e+00	3.17e+00	1.00e+00	1.45e+05	1.11e+00	1.45e+05	1.45e+05	1.40e-18
parabolic_fem	16	1.00e+00	2.00e+00	2.00e+00	1.00e+00	1.20e+00	2.50e+00	4.00e-01	4.00e-01	2.07e-18
	32	1.00e+00	2.00e+00	2.00e+00	1.00e+00	1.20e+00	2.50e+00	4.00e-01	4.00e-01	2.08e-18
	64	1.00e+00	2.00e+00	2.00e+00	1.00e+00	1.20e+00	2.50e+00	4.00e-01	4.00e-01	2.08e-18
	128	1.00e+00	2.00e+00	2.00e+00	1.00e+00	1.20e+00	2.50e+00	4.00e-01	4.00e-01	2.07e-18

Table B.1 – Comparison of the numerical stability of LU-CRTP for different values of k and on different matrices.

Algorithm B.1 LUp(A,nSV)

This function computes the partial LU reducing
the fill-in of the factors

Input: A the matrix
Input: nSV The number of Singular values as reference
Output: fI The fill-in induced the factorization

p $\leftarrow$ amd(A)
C $\leftarrow$ spones(A)+spones(A')
[$\sim$,q] $\leftarrow$ etree(C(p,p))
p $\leftarrow$ p(q)
$A_p \leftarrow A$(p,p)
$[L, U] \leftarrow$ LU(A_p)
$fI \leftarrow$ nnz($L(:, 1 : nSV)$)+nnz($U(1 : nSV, :)$)

id	matrix name	id	matrix name	id	matrix name
1	wing_100	3	wing_500	4	wing_1000
9	GD98_a	10	n3c6-b1	11	GD96_b
12	football	13	GD06_theory	19	bcsstm01
22	rel5	23	relat5	24	GD01_c
31	GD95_a	33	GD95_b	35	GlossGT
37	ch6-6-b1	38	n3c5-b2	39	i_laplace_500
40	ch7-6-b1	41	i_laplace_1000	43	mk10-b1
44	GD99_b	50	ch7-7-b1	54	wheel_5_1
55	mk11-b1	56	ch4-4-b2	57	GL7d11
59	GD97_c	60	GD99_c	62	bcsstm04
64	Sandi_authors	65	bcsstm03	67	GD96_d
73	GD98_b	77	SmallW	78	D_5
79	can_144	87	n4c5-b10	89	GL6_D_10
94	abb313	96	D_11	101	GD01_a
102	GD01_A	103	GL6_D_6	104	lock_700
106	Harvard500	107	Maragal_2	109	GD00_a
116	lpi_ex72a	117	lp_brandy	122	lpi_box1
125	GL6_D_9	127	lp_bore3d	128	dwt_245
129	Sandi_sandi	135	bcspwr04	137	n2c6-b9
141	cat_ears_4_1	142	GD00_c	143	lp_tuff
144	IG5-9	145	lpi_mondou2	146	GL6_D_7
148	D_10	149	GL6_D_8	150	D_6
153	lp_scorpion	154	lp_standgub	155	lp_ship04s
156	flower_7_1	157	dwt_419	158	model2
182	Erdos971	199	Erdos981	212	Erdos991
216	mbeause	217	mbeacxc	218	mbeaflw
219	beacxc	220	beause	221	beaflw
224	dwt_512	225	flower_8_1	227	D_9
229	dwt_992	233	IG5-10	234	lp_shell
241	heat_1000	242	lp_bnl1	244	lp_modszk1
245	dwt_758	247	dwg961a	249	lp_25fv47
252	dwt_869	257	Roget		

Table B.2 – List of matrices where the maximum absolute element of L_{21} is greater than $sqrt(n)$ where n is the number of columns of the matrix or a NaN. Here id is the identifier of the matrix in ids shown on the MATLAB code.

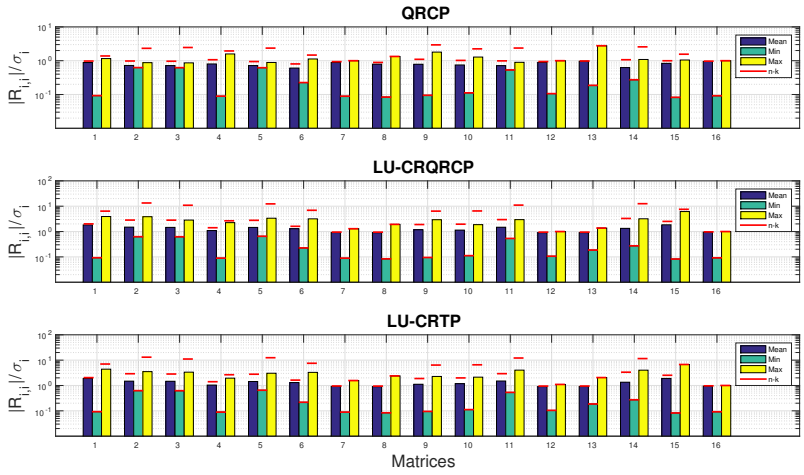


Figure B.3 – Accuracy of the approximate singular values computed by QRCP, LU-CRQRCP and LU-CRTP compared to the real singular values where the last k singular values have been removed. Matrix size is 256, $k = 16$ and $nSV = 256$

Minimizing communication for incomplete factorizations and low-rank approximations on large scale computers

Abstract

The impact of the communication on the performance of numerical algorithms increases with the number of cores. In the context of sparse linear systems of equations, solving $Ax = b$ on a very large computer with thousands of nodes requires the minimization of the communication to achieve very high efficiency as well as low energy cost. The high level of sequentiality in the Incomplete LU factorization (ILU) makes it difficult to parallelize. We first introduce in this manuscript a Communication-Avoiding ILU preconditioner, denoted CA-ILU(k), that factors A in parallel and then is applied at each iteration of a solver as GMRES, both steps without communication. Considering a row block of A, the key idea is to gather all the required dependencies of the block so that the factorization and the application can be done without communication. Experiments show that CA-ILU(k) preconditioner can be competitive with respect to Block Jacobi and Restricted Additive Schwarz preconditioners. We then present a low-rank algorithm named LU factorization with Column Row Tournament Pivoting (LU-CRTP). This algorithm uses a tournament pivoting strategy to select a subset of columns of A that are used to compute the block LU factorization of the permuted A as well as a good approximation of the singular values of A. Extensive parallel and sequential tests show that LU-CRTP approximates the singular values with an error close to that of the Rank Revealing QR factorization (RRQR), while the memory storage of the factors in LU-CRTP is up to 200 times lower than of the factors in RRQR. In this context, we propose an improvement of the tournament pivoting strategy that tends to reduce the number of Flops performed as well as the communication. A column of A is discarded when this column is a linear combination of other columns of A, with respect to a threshold τ. Extensive experiments show that this modification does not degrade by much the accuracy of LU-CRTP. Moreover, compared to the Communication-Avoiding variant of RRQR, our modification reduces the number of operations by a factor of up to 36.

Keywords: ilu factorization, low-rank approximation, preconditioner, reducing communication

---

Minimisation des commmunications lors de factorisations incomplètes et d'approximations de rang faible dans le contexte des grands supercalculateurs

Résumé

L'impact des communications sur les performances d'un code d'algèbre linéaire augmente avec le nombre de processeurs. Dans le contexte de la résolution de systèmes d'équations linéaires creux, la résolution de $Ax = b$, sur une machine composée de milliers de noeuds, nécessite la minimisation des communications dans le but d'atteindre une grande efficacité tant en terme de calcul qu'en terme d'énergie consommée. La factorisation LU, même incomplète, de la matrice A est connue pour être difficilement parallélisable. Ce manuscrit présente CA-ILU(k), un nouveau préconditionneur qui minimise les communications autant durant la phase de factorisation que durant son application à chaque itération d'un solveur tel que GMRES. L'idée est de considérer un sous-ensemble de lignes de A et de lui adjoindre des données de A tel que la factorisation du sous-ensemble, ainsi que l'application des facteurs obtenus, se fait sans communication. Les expériences réalisées montre que CA-ILU(k) rivalise avec les préconditionneurs Block Jacobi et Restricted Additive Schwarz en terme d'itérations. Nous présentons ensuite un algorithme de rang faible appelé la factorisation LU couplée à une permutation des lignes et des colonnes, LU-CRTP. Cet algorithme utilise une méthode par tournoi pour sélectionner un sous-ensemble de colonnes de A, permettant la factorisation par bloc de la matrice A permutée, ainsi qu'une approximation des valeurs singulières de A. Les test séquentiels puis parallèles ont permit de mettre en évidence que LU-CRTP retourne une approximation des valeurs singulières avec une erreur proche de celle obtenue par la factorisation QR révélant le rang de la matrice (RRQR). En outre, l'espace mémoire occupé par les facteurs de LU-CRTP est jusqu'à 200 fois plus faible que dans le cas de RRQR. Toujours dans le cadre d'une approximation de rang faible, nous proposons enfin une amélioration de la stratégie de pivotage par tournoi qui réduit le nombre d'opérations effectuées ainsi que les communications. Une colonne de A est retirée de la méthode si elle est une combinaison linéaire des autres colonnes de A, suivant un critère τ. Des tests sur un grand nombre de matrices montrent que cette modification ne dégrade pas significativement la précision de LU-CRTP. En outre, cette modification appliquée à la variante de RRQR minimisant les communications réduit par un facteur de 36 le nombre d'opérations.

Mots clés : factorisation ilu, approximation de rang faible, préconditionneur, réduction des communications

Laboratoire Jacques-Louis Lions

4 place Jussieu – 75005 Paris – France