Universidade Federal do Espírito Santo
Centro Tecnológico
Programa de Pós-Graduação em Informática

Leonardo Muniz de Lima

An alternative approach to parallel preconditioning for 2D finite element problems

Vitória - ES, Brasil

2018

An alternative approach to parallel preconditioning for 2D finite element problems

Leonardo Muniz de Lima

Tese submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Aprovada em 29 de junho de 2018 por:

Profª. Drª. Lucia Catabriga (Orientador)
UFES/ES

Prof. Dr. Alberto Ferreira de Souza (Examinador Interno)
UFES/ES

Prof. Dr. Isaac Pinheiro dos Santos (Examinador Interno)
UFES/ES

Prof. Dr. Renato Nascimento Elias(Examinador Externo)
UFRJ/RJ

Profª. Drª. Regina Célia Cerqueira de Almeida(Examinador Externo)
LNCC/RJ

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Vitória-ES, 29 de junho de 2018.

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Setorial Tecnológica,
Universidade Federal do Espírito Santo, ES, Brasil)

Lima, Leonardo Muniz de, 1978-
L732a An alternative approach to parallel preconditioning for 2D finite element problems / Leonardo Muniz de Lima. – 2018.
144 f. : il.

Orientador: Lucia Catabriga.
Tese (Doutorado em Ciência da Computação) – Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Sistemas lineares. 2. Método dos elementos finitos. 3. Método de decomposição. 4. Armazenamento de dados. 5. Precondicionadores. I. Catabriga, Lucia. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 004

Elaborada por Sandra Mara Borges Campos – CRB-6 ES-000593/O

"To every thing there is a season, and a time to every purpose under the heaven: a time to be born, and a time to die; a time to plant, and a time to pluck up that which is planted; a time to kill, and a time to heal; a time to break down, and a time to build up; a time to weep, and a time to laugh; a time to mourn, and a time to dance; a time to cast away stones, and a time to gather stones together; a time to embrace, and a time to refrain from embracing; a time to get, and a time to lose; a time to keep, and a time to cast away; a time to rend, and a time to sew; a time to keep silence, and a time to speak; a time to love, and a time to hate; a time of war, and a time of peace."

(King Solomon, Ecclesiastes 3:1-8)

To my beloved wife Suellen

Acknowledgements

I would like to thank, first God, for the life, health, and wisdom given to complete this work.

I would like to thank all my family and especially my wife Suellen for being by my side at all times.

I would like to thank my advisor Lucia Catabriga, one more time, for all her effort and dedication.

I would like to thank my colleague Sérgio Bento, a strong partnership.

I would like to thank the other colleagues on the Computational Mechanics group, Andrea, Isaac, Paulinho, Ramoni, and Riedson.

I would like to thank my colleague Brenno Lugon by the outstanding implementation of the SPIKE preconditioner.

I would like to thank all colleagues from the LCAD (High Performance Computing Laboratory), in particular, colleagues Rodrigo Berriel, Matheus Nogueira, and Marcos Spalenza.

At long last, I would like to thank you all that directly or indirectly contributed to make this work possible.

Abstract

We propose an alternative approach to parallel preconditioning for 2D finite element problems. This technique consists in a proper domain decomposition with reordering that produces narrow banded linear systems from finite element discretizations, allowing to apply, without significant efforts, traditional preconditioners as Incomplete LU Factorization (ILU) or even sophisticated parallel preconditioners as SPIKE. Another feature of that approach is the facility to recalculate finite element matrices whether for nonlinear corrections or for time integration schemes. That means parallel finite element application is performed indeed in parallel, not just to solve the linear system. We also employ preconditioners based on element-by-element storage with minimal adjustments. Robustness and scalability of these parallel preconditioning strategies are demonstrated for a set of benchmark experiments. We consider a group of two-dimensional fluid flow problems modeled by transport and Euler equations to evaluate ILU, SPIKE, and some element-by-element preconditioners. Moreover, our approach provides load balancing and improvement to MPI communications. The load balancing and MPI communications efficiency are verified through analyzer tools as TAU (Tuning Analysis Utilities).

Keywords: Parallel preconditioners, Narrow banded linear systems, Finite element method, Domain decomposition, Storage schemes.

Resumo

Neste trabalho é proposta uma abordagem de precondicionamento paralelo para problemas bidimensionais de elementos finitos. Essa técnica consiste em uma decomposição de domínio especial que produz sistemas lineares de banda oriundos de discretizações de elementos finitos, permitindo aplicar, sem maiores esforços, precondicionadores tradicionais como a fatoração LU incompleta (ILU) ou mesmo precondicionadores paralelos sofisticados como o SPIKE. Uma outra característica dessa abordagem é a facilidade para recalcular as matrizes de elementos finitos seja para correções não-lineares ou mesmo para esquemas de integração no tempo. Isso significa que uma aplicação paralela de elementos finitos é executada de fato em paralelo, não apenas os sistema lineares são resolvidos em paralelo. Além disso, precondicionadores baseados em armazenamento elemento-por-elemento podem ser aplicados com o mínimo de ajustes. A robustez e a escalabilidade dessa abordagem de precondicionamento paralelo é demonstrada através de uma série de experimentos. Um conjunto de problemas bidimensionais de fluxo de fluido modelados pelas equações do transporte e de Euler é considerado para avaliar os precondicionadores ILU, SPIKE, e alguns outros precondicionadores elemento-por-elemento. Mais que isso, essa abordagem fornece balanceamento de carga e melhorias nas comunicações MPI. As eficiências tanto do balanceamento de cargas e como das comunicações MPI são verificadas através de ferramentas de análise como o TAU (Tuning Analysis Utilities).

Palavras-chave: Precondicionadores paralelos, Sistemas lineares de banda, Método dos elementos finitos, Decomposição de domínio, Esquemas de armazenamento.

List of Figures

List of Tables

Contents

1 Introduction

Numerous practical computational fluid dynamics simulation problems require the solution of large and sparse linear systems. Saad (2003) comments that the complexity and size of the new generations of linear and nonlinear systems are some reasons that have made iterative methods more attractive than direct methods. Vorst (2003) emphasizes that, for such systems, it is required to exploit parallelism in combination with suitable solution techniques. There are many iterative methods for solving large and sparse linear systems of equations. Some of these methods are so-called Krylov projection-type methods, and they include popular methods as Conjugate Gradients (Hestenes and Stiefel, 1952), MINRES (Paige and Saunders, 1975) , SYMMLQ (Paige and Saunders, 1973), Bi-Conjugate Gradients (Fletcher, 1976), QMR (Freund and Nachtigal, 1991), Bi-CGSTAB (Sleijpen and Fokkema, 1993), CGS (Sonneveld, 1989), LSQR (Paige and Saunders, 1982), and GMRES (Saad and Schultz, 1986). Indeed, according to Wathen (2015), to solve such linear systems using an iterative method based on Krylov subspace may be a good option only if it converges with few iterations. That might be the case if a good preconditioner is applied. So, in the last decades, many efforts have been made to achieve that goal.

Benzi (2002) presents an essential survey about preconditioning of large linear systems. The author advocates in favor of algebraic methods, which use only information contained in the coefficient matrix. For him, the main classes of algebraic methods comprehend incomplete factorization techniques and sparse approximate inverses. He also states that, despite their popularity, incomplete factorization methods have their limitations, which include potential instabilities, difficult parallelization, and lack of algorithmic scalability. On the other hand, for Toselli and Widlund (2006), robust parallel solvers for many practical problems need proper preconditioning which cannot be constructed just by simple algebraic techniques, but the partial differential equations and respective discretizations must be taken into account.

Iterative methods based on Krylov subspaces, such as GMRES, have become an excellent option to solve large and sparse linear systems. This is because, the inner product and the matrix-vector product, the two main operations of Krylov subspaces methods, demand less memory and can be paralyzed more easily. In consonance with Saad (2003), the great question to ask is whether or not it is possible to find preconditioning techniques that have a high degree of parallelism, as well as good intrinsic qualities. As already stated, iterative solvers, such as GMRES, have slow convergence as the principal drawback. That determines the use of preconditioners. As stated by Wathen (2015):

"The convergence of GMRES has attracted a lot of attention, but the absence

> *of any reliable way to guarantee (or bound) the number of iterations needed to achieve convergence to any given tolerance in most situations means that there is currently no good a priori way to identify the desired qualities of a preconditioner. This is a major theoretical difficulty, but heuristic ideas abound.... In this sense, preconditioning will always be an art rather than a science".*

The advance of numerical methods for large algebraic systems is central in the development of efficient code for computational fluid dynamics, elasticity, and other core problems of continuum mechanics. There is a significant number of software packages (Eijkhout et al., 1998) which have incorporated iterative methods, as well as large algebraic system techniques, to solve linear systems for general applications. An example is the Portable, Extensible Toolkit for Scientific Computation (PETSc) (Balay et al., 2017) that is a suite of data structures and routines which provides the building blocks for the implementation of large-scale application code. Besides that, PETSc is a sophisticated set of software tools that supports many of the mechanisms needed within parallel application code, such as parallel matrix and vector assembly routines.

A natural idea to obtain parallelism is to split massive dimensional problems into subproblems, where each one is solved separately and an approximation of the overall problem solution is generated. As stated by Toselli and Widlund (2006), in practical applications, finite element or other discretizations reduces the problem to the solution of an often huge algebraic system of equations. Thus, solving one huge problem on a domain can not be a good option, it may be convenient to solve many smaller problems on smaller subdomains a certain number of times. Such process is named domain decomposition. There are many works in domain decomposition related to the selection of subproblems that ensure faster rates of convergence for the iterative method. Therefore, domain decomposition methods provide preconditioners that can accelerate Krylov subspace methods. Wathen (2015) cites domain decomposition, in the purest form, as a block diagonal preconditioner where each separate subproblem is represented by a single diagonal block which may or may not have an overlapping, that is, which may or may not share some variables with other subproblems. Wathen (2015) also points out that an important issue is how to treat the variables either in the overlap regions or on the introduced boundaries between subdomains.

Despite the popularization of Krylov's iterative methods for parallel computation, the robustness of direct methods has been unquestionable even for sparse matrices. In the 80s or even earlier, a series of parallel solvers based on narrow banded linear systems was developed (Arbenz and Gander, 1994). These solvers were established using direct methods which guaranteed robustness but questionable scalability. The solver's performances were only considered reasonable if the bandwidth was narrow. A interesting survey (Davis et

al., 2016) presented the recent efforts to produce solvers based on direct methods. It gives an in-depth presentation of the many algorithms and software available for solving sparse matrix problems, both sequential and parallel.

In 2006, the SPIKE (Polizzi and Sameh, 2006), a parallel preconditioner based on solvers of narrow banded linear systems, has been proposed. SPIKE was initially presented as a parallel solver for direct methods. However, it achieved better performance as an algebraic preconditioner based on the domain decomposition concept. According to Manguoglu et al. (2010), preconditioners based on narrow banded systems present robustness and scalability. However, in several applications, the resulting linear systems do not have narrow band. Because that, some reordering method, as reverse Cuthill-McKee (RCM) (Liu and Sherman, 1976), Spectral (Barnard et al., 1995), or Weighted Spectral (Manguoglu et al., 2010) should be applied to obtain a possible narrow banded linear system.

As reported by Polizzi and Sameh (2007), the SPIKE algorithm is a hybrid solver, once it can take advantage of the robustness of direct methods and the lower computational cost of iterative methods. A variant of SPIKE, named PSPIKE (Sathe et al., 2012), combines iterative methods with the direct solver PARDISO (Parallel Direct Sparse Solver Interface) (Schenk and Gärtner, 2006). Developers present the PARDISO software as a thread-safe, high-performance, robust, memory efficient and easy to use for solving large sparse symmetric and unsymmetric linear systems of equations on shared-memory and distributed-memory multiprocessors. Specifically, PARDISO solves the systems on each node based on a shared-memory parallelization, whereas the communication and coordination between the nodes are established by the distributed parallelization of the preconditioned iterative solver. In order to obtain a suitable preconditioning by SPIKE, a series of combinatorics should be employed to preprocess the coefficient matrix. A banded structure of the input matrix is required; then, several reordering techniques are applied to the original matrix to cover all entries of the matrix, ideally by the diagonal blocks. If all entries are confined within the blocks, an iterative solver based on Krylov subspace could converge in a few iterations. However, in practical applications this situation does not occur very often; thus, a realistic goal would be to find a preconditioner such that possibly all heavy-weighted entries are included in the block structures, and some small-weighted entries are not - that is, make some changes of rows and columns such that coefficients with greater value are closer to the main diagonal. In our work (Lima et al., 2017), we tested combinatorial techniques used to preprocess a linear system that would be solved using the SPIKE preconditioner. We also analyzed the computational efforts to improve the convergence of the original linear system.

Our initial goal was to propose the SPIKE as a parallel preconditioner for general purpose in finite element analysis. Unfortunately, finite element applications present

different issues when the subject is preconditioning. A nonlinear steady-state problem, for example, demands a significant amount of GMRES iterations per nonlinear correction while transient problems require less GMRES iterations per time step. So, the general combinatorics proposed in SPIKE preprocessing demonstrated a great overhead once working in a kind of "fork and join" is needed. That is, finite element matrices are calculated and combinatorial techniques should be applied, both in sequential form; after that the finite element matrix is split and the linear system using the parallel preconditioner SPIKE is solved. Because of that, we suggested an alternative arrangement to use SPIKE avoiding excessive overhead. In our work (Lima et al., 2016), we focused on combinatorial techniques that could be applied just once, precisely in the beginning of the process, which allows solving finite element applications indeed in parallel. Using such procedures, we aim to reduce the bottleneck created by the use of parallel computing for only a few parts of the problem numerical solution.

Therefore, we established a domain decomposition based on algebraic system solvers that provide scalability and better convergence properties. We highlight that our approach does not work in a kind of "fork and join". Once a proper preprocessing is established, the entire work is done in parallel, from reading finite element data, through the resolution of linear systems using preconditioners, to postprocessing. Thus, the hybrid SPIKE preconditioner can be applied to general 2D finite element applications. Furthermore, some difficulties related to the use of parallel versions of incomplete LU factorizations can be overcome. Also, our domain decomposition enables parallelization of element-by-element local preconditioners for finite element problems with one or more degrees of freedom per node.

1.1 Contributions of this thesis

In this work, we propose an alternative approach to parallel preconditioning for 2D finite element problems. This technique consists in a particular domain decomposition with reordering schemes that produces narrow banded linear systems from finite element discretization where we can apply, without significant efforts, traditional preconditioners as Incomplete LU Factorization (ILU) or even sophisticated parallel preconditioners as SPIKE. In particular, such adaptations applied to sequential incomplete LU factorizations lead to a kind of block Jacobi preconditioner with local ILU factorizations. Another feature of that approach is the facility to recalculate finite element matrices whether for nonlinear corrections or for time integration schemes. That means that the parallel finite element application is performed indeed in parallel, not just the solver of the linear system. We also employ preconditioners based in element-by-element storage with minimal adjustments. Moreover, our approach provides load balancing and improvements to MPI communications.

1.2 Thesis outline

This Ph.D. Thesis is organized as follows. Chapter 2 provides general ideas about preconditioning; parallel preconditioning; preconditioning for narrow banded linear systems such as SPIKE preconditioner and their relations with preconditioning for finite element analysis; and some considerations of local and global preconditioning. Chapter 3 presents a short overview of domain decomposition; introduces our domain decomposition approach; and demonstrates how our domain decomposition approach provides an *a priori* narrow banded linear system for finite element analysis. Chapter 4 describes in detail how to promote the parallel solution of a finite element application using a suitable preconditioning. In Chapter 5, numerical experiments are performed aiming to analyze the behavior of four benchmark problems when they use local and global preconditioners in combination with our domain decomposition approach. Chapter 6 lays out the Thesis's conclusions.

1.3 Publications related to this thesis

This section presents the publications related to thesis research.

- B. A. Lugon, **L. M. Lima**, M. T. P. Carrion, L. Catabriga, M. C. S. Boeres, and M. C. Rangel, *Combinatorial Optimization Techniques Applied to a Parallel Preconditioner Based on the SPIKE algorithm.* 1st. Pan-American Congress on Computational Mechanics (PANACM 2015), 27–29 April, 2015 - Buenos Aires, Argentina (Presentation);

- **L. M. Lima**, B. A. Lugon, and L. Catabriga, *An Alternative Approach of the SPIKE Preconditioner for Finite Element Analysis.* 23rd International Conference on High Performance Computing (HiPC 2016), 19–22 December, 2016 - Hyderabad, India;

- **L. M. Lima**, L. Catabriga, M. C. Rangel, and M. C. S. Boeres, *A Trade-off Analysis of the Parallel Hybrid SPIKE Preconditioner in a Unique Multi-core Computer.* 17th International Conference on Computational Science and Its Applications (ICCSA 2017), 3–6 July, 2017 - Trieste, Italy.

- L. K. Muller, **L. M. Lima**, and L. Catabriga, *A Comparative Study of Local and Global Preconditioners for Finite Element Analysis.* XXXVIII Iberian Latin-American Congress on Computational Methods in Engineering (CILAMCE 2017), 5–8 November, 2017 - Florianópolis, Brazil.

- **L. M. Lima**, S. S. Bento, R. Baptista, P. W. Barbosa, I. P. Santos, and L. Catabriga, *An Alternative Approach of Parallel Preconditioning for 2D Finite Element Problems.* 13th World Congress in Computational Mechanics (WCCM 2018), 22–27 July, 2018 - New York, USA; **(accepted)**

- **L. M. Lima**, S. S. Bento, R. Baptista, P. W. Barbosa, I. P. Santos, A. M. P. Valli, and L. Catabriga, *An Alternative Domain Decomposition to Local and Global Parallel Preconditioners for 2D Finite Element Problems*,..., 2018. **(to be submitted to a journal)**

2 Preconditioning

As pointed out by Olshanskii and Tyrtshnikov (2014), for very large systems arising in real-life applications, direct methods are often computationally nonfeasible. Although direct methods with lower complexity are known, they are more sensitive to roundoff errors and are more involved from the algorithmic viewpoint. In many cases, iterative methods for finding an approximate solution are more attractive. They represent a rich and lively research area with a representative number of published articles and books. Vorst (2003) emphasizes a particular class of iterative methods, that is, the so-called Krylov projection-type methods which include the popular methods such as Conjugate Gradients, MINRES, SYMMLQ, Bi-Conjugate Gradients, QMR, Bi-CGSTAB, CGS, LSQR, and GMRES. In addition, Wathen (2015) declares that preconditioning in conjunction with iterative methods is often the vital component in enabling the solution of large and sparse linear systems. Wathen also presents GMRES as the most common method to solve linear systems arising from large, sparse and unsymmetric matrices. However, it is not practical if many iterations are required. In short, GMRES combined with a good preconditioner becomes a suitable option for solving large and sparse linear systems arising from general finite element discretizations.

2.1 A Brief Note About Preconditioning

The concept of preconditioning is quite simple. Given a linear system $Ax = b$, the main idea behind a good preconditioner is to obtain an appropriate matrix M such that $M^{-1}A$ is well-conditioned (Saad, 2003). Theoretically, the best choice for M is A, but apparently to find A^{-1} is more laborious than to solve the system $Ax = b$. In this way, there is a trade-off problem, where it is necessary to improve the condition number of the matrix $M^{-1}A$ and at the same time to solve the preconditioned linear system $M^{-1}Ax = M^{-1}b$ more efficiently than the original linear system $Ax = b$. The effect of the new matrix $M^{-1}A$ appears in the matrix-vector operation of the Krylov projection-type methods algorithm. In practice, instead of determining M^{-1} explicitly, the preconditioning is achieved for each matrix-vector product operation $p = M^{-1}Av$, as follows: first, the matrix-vector $z = Av$ is computed and then the action of M^{-1} in $p = M^{-1}z$ is obtained through the resolution of the linear system $Mp = z$. From a practical point of view, since M is conveniently chosen, solving the preconditioned system $Mp = z$ is more efficient both in memory consumption and in the number of arithmetic operations involved.

Once the preconditioning matrix M is known, there are three ways to use the

preconditioner. It can be applied to the left:

$$M^{-1}Ax = M^{-1}b. \tag{2.1}$$

Alternatively, it can be applied to the right:

$$AM^{-1}u = b, \quad \text{with } x \equiv M^{-1}u. \tag{2.2}$$

Note the right preconditioning needs a change of variables $u = Mx$ and the solution of a linear system concerning the vector of unknowns u.

Finally, a typical situation is when the preconditioner is obtained in the factorized form:

$$M = M_L M_R \tag{2.3}$$

where, M_L e M_R are lower and upper triangular matrices, respectively. In that case, the preconditioner must be split as follows:

$$M_L^{-1}AM_R^{-1}u = M_L^{-1}b, \quad \text{with } x \equiv M_R^{-1}u. \tag{2.4}$$

For symmetric matrices is common to use the split preconditioning as presented in (2.4). However, there are other ways of preserving symmetry, or taking into account advantage of the symmetry, even if M is not available in the factored form (Chan et al., 1998).

One of the most genuine forms to define a preconditioner is to perform the incomplete factorization of the original matrix A. It requires a decomposition $A = LU - R$ where L and U have the same structure of nonzero elements as the lower and upper parts of A, respectively, and R is the residual or factorization error. Such incomplete factorization, named as $ILU(0)$, is easily computable. On the other hand, it can result in many iterations of the Krylov projection-type methods.

To remedy this problem, as can be seen in the work of Chow and Saad (1997), several incomplete factorization preconditioners have been developed considering a more substantial number of *fill-ins* in L and U. That is, a significant amount of nonzero coefficients is introduced during the elimination process at positions where they were initially null. In general, the most accurate ILU factorizations require a smaller number of iterations to converge, but the computational effort to provide the factors is high.

For finite element discretizations, as well known, assembling the entire global matrix is not always reasonable. In these cases, the preconditioners construction would be at the element level, named local or element-by-element preconditioner. The first works that proposed the construction of element-by-element preconditioners were (Hughes et al., 1983b) and (Ortiz et al., 1983). The work of Daydé et al. (1996) presents an efficient

alternative technique based on local preconditioning. The main idea of this technique suggests to factorize the element matrices into their corresponding LU factors and to replace the global backward and forward substitution operations associated with the LU factorization by a series of similar local operations into element level. Another analogous way of producing local preconditioners has been the edge-based form. Moreover, local preconditioners based on the edges, when compared to the element-based preconditioners, result in less processing time (Catabriga et al., 1998).

Sparse Approximate Inverses (SAI) (Grote and Huckle, 1997) preconditioners represent another alternative to improve linear systems convergence. As stated by Benzi (2002), approximate inverse techniques rely on the assumption that, for a given sparse matrix A, it is possible to find a sparse matrix M which is a good approximation, in some sense, of A^{-1}. However, this is not at all obvious, since the inverse of a sparse matrix is usually dense. According to Anzt et al. (2018), SAI preconditioners typically show good parallel performance but often fail to provide substantial convergence improvement. On the other hand, incomplete factorization preconditioners are often better preconditioners, but the sequential nature of exact triangular solves makes the preconditioner application particularly expensive on parallel architectures.

2.2 Parallel Preconditioning

Parallel computing is almost unanimous when dealing with massive computing tasks. Iterative methods, in turn, have become attractive to solve large and sparse linear systems, since they demand less storage and offer ease parallel implementation when compared with direct methods. In the mid-1990s, Yousef Saad (1994) stated that the degree of parallelism of the Krylov subspaces methods using standard preconditioners was limited and could lead to reduced performance when used on massively parallel computers. He identified this difficulty and considered two alternatives based on multi-coloring and polynomial preconditioning ideas. Besides that, methods which deal specifically with sparse matrices, such as those derived from finite element methods in unstructured meshes (Koric et al., 2014), have received particular attention.

Parallelizable preconditioners have proved to be indispensable since the parallel computing has became widespread. Saad (2003) states that methods based on incomplete factorization are more sequential in design, and although there are parallel variants, often speed of convergence is compromised by ordering and separating variables to obtain the necessary independence of calculations. The author emphasizes the importance of Multigrid and Algebraic Multigrid (AMG) methods, whose parallel forms retain much of the power of sequential implementation. Saad also enunciated that sparse approximate inverses were developed because they appeared to be a tractable way to compute a purely algebraic

('given a matrix') preconditioner in parallel. However, their effectiveness remains unclear. In consonance with Wathen (2015), very simple preconditioning, such as diagonal scaling, is obviously parallel. However, the efficiency of such approach is limited. Moreover, Wathen (2015) affirms that the paradigm of domain decomposition is ideal for parallel computation since it arose from such purpose.

There are two classes of algebraic preconditioners that come from domain decomposition approaches, namely, preconditioners based on overlapping and preconditioners based on nonoverlapping (Giraud and Tuminaro, 2006). Additive Schwarz preconditioners are the most common overlapping representatives. Parallel implementation of these preconditioners requires a factorization of a Dirichlet problem on each processor in the setup phase. Each invocation of the preconditioner requires two neighbor-neighbor communications. In general, more significant overlap usually leads to faster convergence up to a certain point at which increasing the overlap does not further improve the convergence rate. Unfortunately, more substantial overlap implies greater communication and computational requirements. The Restricted Additive Schwarz (RAS) (Cai and Sarkis, 1999) preconditioner is a variant of the classical additive Schwarz method which avoids one communication step when applying the preconditioning. Even within overlapping regions, each unknown is updated by only one sub-domain, while contributions from all overlapping sub-domains are summed in the classical additive Schwarz method. This variant does not have a natural counterpart in a mesh partitioning framework which by construction has overlapping sets of vertices. The concept of Schur complement establishes the main category of preconditioners subject to nonoverlapping domain decomposition. Moreover, the principal drawback of this approach is the fact that the interface between two adjacent sub-domain interiors has two layers. That means the corresponding Schur complement system is twice as large as necessary. Unfortunately, this complicates parallel implementation and load balancing decisions.

Giraud and Tuminaro (2006) also state that domain decomposition methods are more effective and require less tuning when they are employed as a preconditioner to accelerate the convergence of a Krylov method. In a parallel distributed framework, the construction and the application of these preconditioners should also be readily parallelizable. In the context of the solution of linear systems arising from partial differential equations (PDEs), domain decomposition preconditioners can be given from an algebraic perspective. For finite differences, the linear system is fully assembled, and the domain decomposition techniques correspond to matrix splittings. For finite elements, in turn, the prevalent practice is only partially assembling matrices on interfaces. That is, each processor is restricted so that it constructs matrix contributions coming only from finite elements belonged by the processor. In this case, domain decomposition techniques correspond to split the underlying mesh as opposed to split the matrix. From a parallel point of view, they require different data structures and possibly enable different algorithmic choices.

2.3 Preconditioning for Narrow Banded Linear Systems

Let A be a square sparse matrix of order n. The bandwidth bw of A is defined by

$$bw(A) = \max_{i,j:a_{ij}\neq 0} |i-j|. \tag{2.5}$$

The matrix A is a banded matrix if its bandwidth is reasonably small, that is, $bw << n$. A linear system composed by a banded matrix is called narrow banded linear system. Conforming to Gallopoulos et al. (2016), a narrow banded linear system $AX = F$ presents at least two advantages: (i) possibility to extract a suitable preconditioning matrix M from A; (ii) favorable conditions to implement on a parallel architecture with higher efficiency.

Let $AX = F$ be a large and sparse linear system with narrow bandwidth such that it can be divide into p partitions:

$$AX = \begin{bmatrix} A_1 & B_1 & & & \\ C_2 & A_2 & B_2 & & \\ & \ddots & \ddots & \ddots & \\ & & C_{p-1} & A_{p-1} & B_{p-1} \\ & & & C_p & A_p \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{p-1} \\ X_p \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_{p-1} \\ F_p \end{bmatrix} = F \tag{2.6}$$

where A_i, B_i, C_i are block matrices and X_i and F_i are block vectors ($i = 1, \ldots, p$).

Gallopoulos et al. (2016) also suggest large sparse systems, mainly those derived from the finite element method, can become narrow banded systems just using graph manipulation schemes. Reverse Cuthill-McKee (RCM) (Liu and Sherman, 1976), Spectral (Barnard et al., 1995), or Weighted Spectral Ordering (WSO) algorithms promote a suitable matrix rearrangement such that a chosen central band produces a good preconditioner for Krylov iterative methods. Commonly, such operations produce a positive effect on the preconditioning of the linear system since they lead to a robust narrow banded linear system. However, for problems whose assembly and reassembly of the finite element matrices are numerous, such modifications can become almost infeasible to the parallel processing due to computational demand.

Some efforts have been made to produce suitable preconditioning based on narrow banded linear systems arising from partial differential discretizations. Ortigosa et al. (2003) proposed an assessment of the behavior of the memory hierarchy and communication pattern in the parallel implementation of the conjugate gradient method for banded matrices which arise from finite difference or finite element methods on diverse computer architectures. The work (Chen and Taha, 2014) presents a parallelized iterative solver for large sparse linear systems (whose matrices are already in banded form) implemented on a GPGPU cluster.

A family of solvers namely SPIKE (Manguoglu et al., 2009)(Sathe et al., 2012) is another representative of narrow banded linear system solvers. According to the authors,

they are a family of parallel hybrid linear system solvers that are more robust than other available preconditioned iterative methods, and more scalable than parallel sparse direct solvers. Initially, SPIKE solvers worked just with narrow banded systems. Currently, the PSPIKE+ (Zhu and Sameh, 2017), as it is now named, also has become an excellent option for a medium-band linear system. That means the preconditioner consists of non-overlapping diagonal blocks plus small off-diagonal coupling blocks (Sathe et al., 2012).

2.4 The SPIKE Preconditioner

The SPIKE preconditioner (Polizzi and Sameh, 2006) provides a preconditioning matrix M which is a high-quality approximation of A (the original matrix of the linear system $AX = F$ in Eq. (2.6)) and whose corresponding linear system can be solved in parallel. SPIKE was conceived as a hybrid parallel solver for narrow banded linear systems. Once it takes advantage of the robustness of direct methods and the lower computational cost of iterative methods. Furthermore, if the linear systems are diagonally dominant, the truncated SPIKE version can be used as a preconditioner for external iterative scheme. Three phases define the SPIKE preconditioner: first, the preprocessing phase, in which the narrow banded matrix M is obtained; second, the factorization of the matrix M; and, finally, the postprocessing, where the preconditioning actually acts.

Combinatorial strategies – detailed in Sec.2.4.1 – are used in the first phase to transform the original sparse matrix A into a banded matrix as stated in Eq. (2.6). The banded matrix A is divided according to the appropriate number of processors. The preconditioning matrix M is formed from the banded matrix A, taking their diagonal blocks A_i $(i = 1, 2, ..., p)$ and extracting respective coupling block matrices B_i $(i = 1, 2, ..., p-1)$ and C_i $(i = 2, ..., p)$, where p is the number of partitions. A_i has dimension $n_i \times n_i$ in each partition i and the coupling blocks B_i and C_i have dimension $k \times k$, where k value is chosen conveniently (Manguoglu et al., 2010). To finalize the first phase, the preconditioning matrix M is obtained discarding the coefficients outside the coupling blocks B_i and C_i.

The second phase begins when the banded matrix A is factored into DS, where D is a diagonal block matrix and S is named the SPIKE matrix. Figure 2.1 illustrates a factorization divided in four partitions. D is equal A without coupling blocks. S has diagonal blocks I_i given by the corresponding identity matrix and dense matrices V_i $(i = 1, 2, ..., p-1)$ and W_i $(i = 2, ..., p)$ whose dimension are $n_i \times k$.

The SPIKE matrices V_i and W_i are defined as:

$$V_i = A_i^{-1} \begin{pmatrix} 0 \\ B_i \end{pmatrix} \text{ e } W_i = A_i^{-1} \begin{pmatrix} C_i \\ 0 \end{pmatrix}, \tag{2.7}$$

and we can calculate them solving the systems

$$A_i V_i = \begin{pmatrix} 0 \\ B_i \end{pmatrix} \text{ e } A_i W_i = \begin{pmatrix} C_i \\ 0 \end{pmatrix}, \tag{2.8}$$

These systems are solved using LU factorizations, with $A_i = \mathbf{L}_i \mathbf{U}_i$, and can be represented in a reduced form as

$$\mathbf{L}_i \mathbf{U}_i [V_i, W_i] = \left[\left(\begin{smallmatrix} 0 \\ B_i \end{smallmatrix} \right), \left(\begin{smallmatrix} C_i \\ 0 \end{smallmatrix} \right) \right]. \tag{2.9}$$

The bottom of the SPIKE V_i (denominated $V_i^{(b)}$) can be computed using only the bottom $k \times k$ blocks of L and U. Similarly, the top of the SPIKE W_i (denominated $W_i^{(t)}$) may be obtained if it performs the UL-factorization. Thus, after some communications, a good approximation of S, named $\tilde{S}$, is built and the preconditioning matrix $M = D\tilde{S}$ is defined. Then a parallel LU-factorization is applied to each partition of $\tilde{S}$.

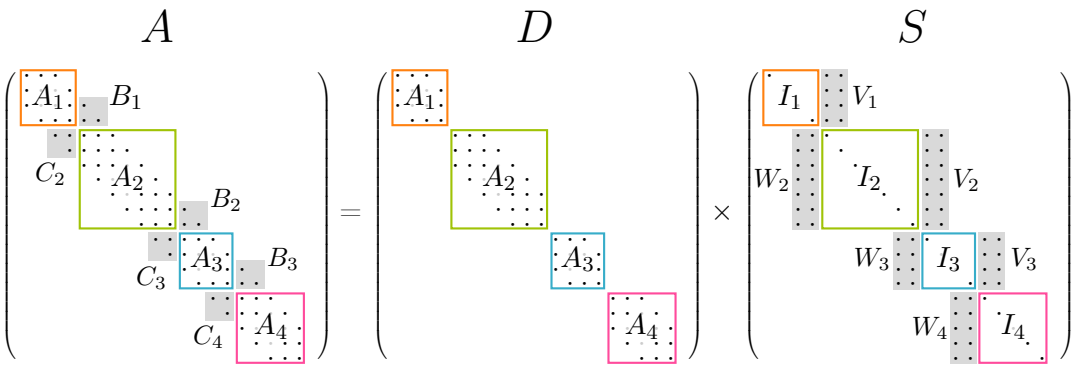


Figure 2.1 – Matrix A factorization into DS: example with four partitions.

In the last phase, called postprocessing, the preconditioning is performed, whenever the iterative method calls for a matrix-vector product like $\tilde{A}u = z$ and $\tilde{A} = M^{-1}A$. Obviously, the M^{-1} matrix is not computed, but a parallel solution of the linear system $M\tilde{x} = z$ is obtained in two steps:

$$\tilde{D}_i \tilde{g}_i = z_i \tag{2.10}$$

$$\tilde{S}_i \tilde{x}_i = \begin{bmatrix} I & V_i^{(b)} \\ W_{i+1}^{(t)} & I \end{bmatrix} \begin{bmatrix} x_i^{(b)} \\ x_{i+1}^{(t)} \end{bmatrix} = \begin{bmatrix} g_i^{(b)} \\ g_{i+1}^{(t)} \end{bmatrix} = \tilde{g}_i \tag{2.11}$$

The application of SPIKE preconditioner to solve the system $AX = F$ is presented in Algorithm 1.

2.4.1 Combinatorial techniques applied to SPIKE preprocessing

The main step of the SPIKE algorithm is concerned with the computation of a narrow banded linear system, where reordering procedures permute the original matrix

Algorithm 1 Solve a linear system $AX = F$ using SPIKE preconditioner for p processors

Require: Given matrix A stored in compact format.

1: Apply reordering techniques to reduce the matrix bandwidth of A.
{Improvements in preconditioning can be reached if other combinatorial techniques are utilized in matrix A (see (Sathe et al., 2012))}

2: Employ on the right side vector F the same techniques applied to matrix A in step 1.

3: Perform the partitioning of A and obtain preconditioning matrix M.

4: Send A_i, B_i and C_i blocks to processor i.
{processor 1 has only B_1 and processor p only C_p}

5: Make LU and UL factorization of A_i blocks using PARDISO (Schenk and Gärtner, 2014) software in parallel.

6: Solve with PARDISO: $L_iU_i[V_i^{(b)}, W_i^{(t)}] = \left[\left(\begin{smallmatrix} 0 \\ B_i \end{smallmatrix}\right), \left(\begin{smallmatrix} C_i \\ 0 \end{smallmatrix}\right)\right]$.
{Processor 1 calculates only $V_1^{(b)}$ and processor p only $W_p^{(t)}$}.

7: Send matrix $W_i^{(t)}$ of size k to processor $i-1$ and assembles truncated reduced matrices $\tilde{S}_i$ from Eq. (2.10) of size $2k$.
{Processor 1 only receives data and processor p, only sends it}.

8: Apply LU factorization in truncated reduced matrices $\tilde{S}_i$ of Eq.(2.11).

9: Use a parallel iterative solver as GMRES (Saad and Schultz, 1986). In each iteration, after an efficient matrix-vector product $Au = z$, solve the system $M\tilde{x} = z$.
{In the beginning of this step, send to processor i the corresponding partition of matrix A and vector b}
{Truncated system $\tilde{S}_i\tilde{x}_i = g_i$ of Eq.(2.10) is also performed by PARDISO*}*

10: Gather all partition x_i of vector solution and undo the reordering applied in step 1.

A. That aims to have all its entries covered by the diagonal blocks A_i or by the coupling blocks B_i and C_i (Sathe et al., 2012). A matrix organized with diagonal and coupling blocks can be denoted as a matrix with a SPIKE structure (see Fig. 2.1). As in real applications, this ideal matrix format is hard to achieve; a good approximation should be a banded matrix with its heavy-weighted entries confined into the block matrix structures.

After applying the reordering procedures, a suitable preconditioning matrix can be obtained if three steps are managed. First, find the diagonal blocks; second, guarantee their non-singularity (i.e., they must admit an inverse); and, third, organize the original matrix into diagonal and coupling blocks, moving the most significant entries into the coupling blocks. These three goals can be respectively formalized as graph partitioning, graph matching, and quadratic knapsack problem (Sathe et al., 2012). Graph algorithms seem to be handy tools to transform the original matrix if they are designed to perform search procedures and solve these known combinatorial problems.

The application of the combinatorial strategies to the original matrix A defines the SPIKE preconditioner matrix M. For this purpose, the matrix A is multiplied by scaling factors and permutation matrices (reordering procedures) on the left and right. Therefore, row permutations are applied to move large entries onto the diagonal (graph matching problem), and techniques for solving the quadratic knapsack problem (used to maximize

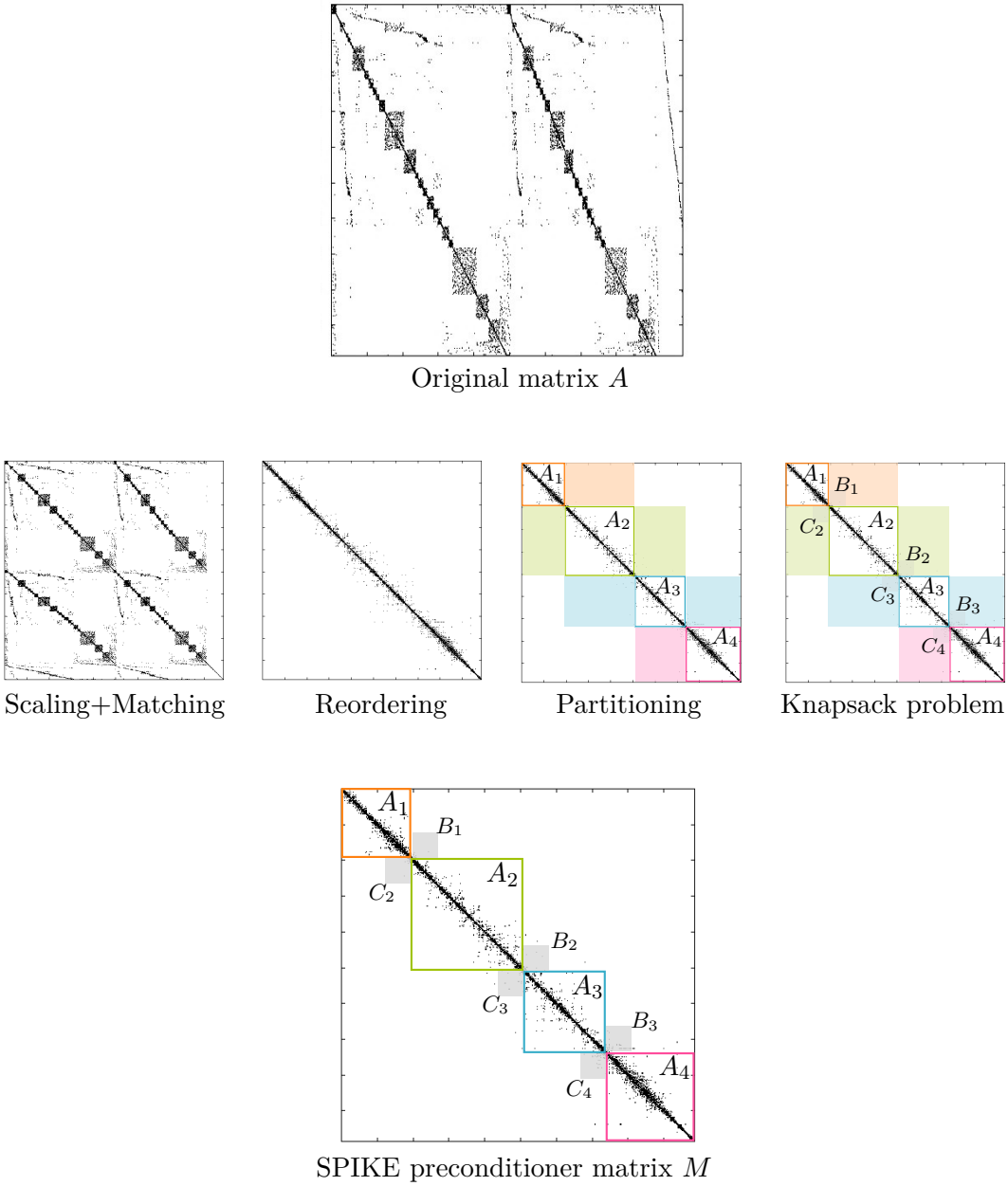


Figure 2.2 – Combinatorial strategies to obtain the SPIKE preconditioner matrix M from an original matrix A - An example of four partitions.

a quadratic objective function subject to a linear capacity constraint) are employed to maximize the entries weights in the coupling blocks. Finally, the matrix partitioning is performed for parallelization purposes. Figure 2.2 illustrates a sequence of combinatorial techniques applied to an original matrix A to obtain the respective SPIKE preconditioner matrix M divide into four partitions.

In the factorization phase of the SPIKE algorithm (Algorithm 1, line 5) the diagonal blocks A_i must be solved by a direct method using LU/UL factorizations, which can be applied only if these blocks are non-singular. Every preconditioning matrix M can be associated to a graph G employing the adjacency information of every matrix row. Thus,

the non-singularity of M can be achieved if a perfect matching (Sathe et al., 2012) is found in G, in a way that it can move the most significant elements to the main diagonal. A perfect matching in G is a set of independent edges that are incident to all vertices of G. For more details on graph concepts, see (Diestel, 2017).

An example of the algorithm to obtain a perfect matching in G is called *weighted bipartite matching* and is available in HSL_MC64[1] (Duff and Koster, 2001). This algorithm returns row permutations that maximize the smallest element on the diagonal, maximize the sum of the diagonal entries and also finds scaling factors that may be used to scale the original matrix. Therefore, the nonzero diagonal entries of the permuted and scaled matrix are equal to one in absolute value, and all the off-diagonal entries are less than or equal to one, also in absolute values.

Three different algorithms can perform the matrix reordering and almost always are able to assure a narrow banded linear system: reverse Cuthill-Mckee (RCM) (Liu and Sherman, 1976), spectral algorithm (Barnard et al., 1995), and weighted spectral ordering (WSO) (Manguoglu et al., 2010). All of them try to put almost all entries onto the SPIKE structure and leave only a few elements uncovered. The WSO algorithm also attempts to move the most significant coefficients as close as possible to the main diagonal, ensuring a more efficient SPIKE preconditioner.

Any matrix A can be seen as the adjacency matrix of the associated graph G. The RCM algorithm visits all vertices of G, according to some criterion. The order of the vertices visited corresponds to a permutation of rows and columns of the matrix A. The algorithm uses the inverse order of the performed visits to reduce the bandwidth and the envelope of the matrix A (Liu and Sherman, 1976). The Spectral and WSO algorithms consider the concept of the algebraic connectivity, also known by Fiedler vector of the unweighted or weighted Laplacian matrix of the original matrix A. These algorithms compute a symmetric permutation that reduces the profile and wavefront of A by using a multilevel algorithm (Manguoglu et al., 2010). Both algorithms codes are available in HSL_MC73[2].

For an efficient parallelization of the SPIKE algorithm, the partitioning step needs to balance the processor loads during each phase of Algorithm 1, considering strategies to reduce the communication cost. An outstanding option is to consider the well known *chains-on-chains* partitioning problem (Pınar and Aykanat, 2004) to model this step. The *MinMax* algorithm, used in this work, finds the exact solution of the *chains-on-chains* partitioning problem in polynomial time (Manne and Sorevik, 1995).

Considering the blocks $A_1, \ldots, A_p$ of the banded matrix (see, as an example, the last part of Fig. 2.2), it is important to have $k \times k$ coupling matrices B_i and C_{i+1}, for

[1] http://www.hsl.rl.ac.uk/catalogue/mc64.html
[2] http://www.hsl.rl.ac.uk/catalogue/hsl_mc73.html

$i = 1, \ldots, p-1$, with many of their entries as close as possible to the diagonal blocks. One possibility to achieve that matrix structure pattern is to consider the quadratic knapsack problem and to solve it by the *DeMin* heuristic – this heuristic starts with the given N as an initially infeasible solution and discards the row or column with the least contribution to the remaining matrix until a $k \times k$ matrix is obtained – as described in (Sathe et al., 2012).

Although the SPIKE preconditioner presents robustness and scalability as it can be noticed in (Sathe et al., 2012), our work (Lima et al., 2017) showed that the combinatorial strategies require a considerable preprocessing time what for general finite element applications could cause excessive computational runtime for the preconditioning step. In another work (Lima et al., 2016), we suggested a version of the SPIKE algorithm that could be applied to the finite element context. Such version intents to improve SPIKE's performance for problems where the finite element matrices are recalculated at each time step (or nonlinear iteration). Thus, combinatorial strategies as Scaling, Matching, and Knapsack Problem are neglected. That is why all matrix rearrangements need to be made in the preprocessing phase just once. That is, the matrix sparsity pattern is assigned before the finite element matrix coefficients are obtained. For the same reason, reordering algorithms as Weighted Spectral Ordering (WSO) are also disregarded.

2.4.2 The SPIKE preconditioner for finite element context

The traditional SPIKE preconditioner was proposed supposing the matrix of the linear system is already known. However, the finite element matrices are assembled while each element contribution is calculated across the whole domain. The SPIKE approach produces a linear system with suitable properties for preconditioning, but the matrix coefficients cannot indeed be calculated in parallel. That causes a momentary loss of balance in memory usage and increases execution time.

In the SPIKE parallel processing, only one CPU (as a *Master*) reads and calculates all data arising from a unique mesh. After preprocessing, the *Master* distributes matrix coefficients throughout the network. Such data, depending on the application, considering mesh size, and degrees of freedom, can be enormous. Beyond that, other difficulties can include transient and nonlinear problems because the resulting matrices may need to be recalculated in each time step or each nonlinear iteration.

These difficulties become more apparent if a cluster of workstations is used, for example. Solving the problem in parallel requires jobs submission in a queue, which in turn gives priority according to the number of processors and memory usage. In this case, dividing a problem of size ω among p processors would not mean allocating memory proportional to $\frac{\omega}{p}$ for each processor. After all, one of the processors must compute the whole matrix data sequentially. Then a preprocessing step would divide data across all

processors by using Message Passing Interface (MPI)(Dongarra et al., 2013) or saving the information in a disk. Thus the SPIKE application could cause an unnecessary overload in the cluster.

In the meantime, we proposed a SPIKE preconditioner approach (Lima et al., 2016) able to unify reasonable preconditioning properties and an efficient parallel recalculation of the finite element matrices. This alternative SPIKE does not store the linear system matrix A initially, as considered in Algorithm 1. It is performed literally in parallel. Thus, step 4 of Algorithm 1 is eliminated, avoiding the amount of prior communication reported in step 9. In the next chapter, a well-elaborated preprocessing scheme, which assure the success of our SPIKE proposal is stated.

2.5 Our Preconditioning Approach

We propose a domain decomposition technique that allows building an a priori narrow banded linear system, as described in Eq. (2.6), arising from 2D finite element discretizations. This narrow banded linear system format is adapted to the context of parallel computing in combination with global preconditioners based on Compressed Storage Row (CSR) (Saad, 2003) format and local preconditioners based on element-by-element structures. The SPIKE preconditioner and an adaptation of incomplete LU factorization are chosen as the global preconditioners representatives. Some simple local preconditioners based on element-by-element structures according to our experiments proposed in (Muller et al., 2017) are also evaluated.

2.5.1 Global preconditioners

Our studies on the SPIKE (Lima et al., 2016) preconditioner, as well as the application of combinatorial optimization techniques (Lima et al., 2017), have led to the development of an alternative approach to 2D finite element problems. Since SPIKE is a representative of domain decomposition preconditioners, it is almost immediate, to reflect on the fact that other forms of parallel preconditioning could be created. As reported by Benzi (2002), domain decomposition techniques can be used to introduce parallelism in incomplete factorization methods. Likewise, when the subdomains are solved approximately using an incomplete factorization, the resulting preconditioner can be thought as a "parallel ILU" strategy. Nevertheless, as emphasized by Benzi, the convergence can become very slow for solvers based on Krylov subspaces. Thus, our experience with the SPIKE enabled extending our parallel approach to ILU preconditioners.

2.5.2 Incomplete LU-factorization

ILU are one of the most used preconditioners in the recent years. For this reason, Wathen (2015) states that GMRES with ILU preconditioner is a 'go-to' technique for many practical problems. On the other hand, ILU preconditioners were widely believed to be ill-suited for implementation on parallel computers with more than a few processors (Benzi, 2002). The reason is due to the fact that ILU computations – in general Gaussian elimination operations – offer limited scope for parallelization. Benzi (2002) also highlights a series of excellent works about parallel incomplete LU factorizations. In spite of those papers' evidence, the Benzi's last sentence about parallel ILU preconditioners can be emphasized:

> *"It has now become clear that high parallelism can be achieved with ILU preconditioners, albeit considerable sophistication and ingenuity are needed if one is to develop efficient implementations".*

Such words stimulate our research because using our domain decomposition approach, the sequential ILU preconditioners can be readily employed in parallel computation. In our work, an application of the most common sequential incomplete LU factorization is chosen, that is, the well-known ILUm (Meijerink and Vorst, 1977), where m represents the number of *fill-in* levels admitted.

In this work, we consider the ILU implementations as block Jacobi preconditioners, that is, block Jacobi with local ILU factorizations. From a theoretical point of view, a parallel incomplete LU factorization is employed just by taking the preconditioning matrix M as an approximation of the SPIKE preconditioner matrix (see Fig. 2.2). That is, for each partition i, the coupling blocks B_i's and C_i's are neglected and only the sequential ILU of the blocks A_i's are considered (SPIKE performs complete LU/UL-factorizations using the PARDISO (Schenk and Gärtner, 2014) library).

2.5.3 Local preconditioners

This section presents the local parallel preconditioners developed in this thesis. (Muller et al., 2017) proposed versions of well known sequential element-by-element preconditioners, as: a diagonal preconditioner (DIAGe), a block diagonal preconditioner (BlockDIAGe), Gauss-Seidel preconditioners (SGSe and BlockSGSe), and LU factorization preconditioners (LUe and BlockLUe). Prefix "Block" means that the preconditioner was designed for problems with more than one degree of freedom per node. Suffix "e", in turn, denotes that the preconditioner is stored in element-by-element format. Considering our domain decomposition approach that is going to discuss in detail in Chapter 3, these preconditioners can be put in the parallel context with minimal adjustments. The parallel local preconditioners are described below.

Assume a domain decomposition with element overlaps such that the global banded matrix A, indicated in Eq. (2.6), partitioned into p divisions, can be represented as:

$$A = \bigcup_{i=1}^{p} A_i^{EBE} \text{ with } A_i^{EBE} = \mathop{\mathbf{A}}_{e=1}^{\mathtt{nel}_i} A^e \tag{2.12}$$

where, for a specific partition i, A_i^{EBE} is a substitute structure for blocks C_i, A_i, and B_i from Eq. (2.6) (for element-by-element schemes, the global matrix is never assembled); A^e is an elementary finite element matrix of size $\mathtt{ndof} \cdot \mathtt{nnoel} \times \mathtt{ndof} \cdot \mathtt{nnoel}$ ($\mathtt{ndof}$ is the number of degrees of freedom per node and $\mathtt{nnoel}$ is the number of nodes per element); and $\mathtt{nel}_i$ is the number of elements of the partition i. Operator $\mathbf{A}$ is an abstraction to represent the relations between the local matrices A^e within a partition i. In practice, this assembly is never performed when the element-by-element storage scheme is used.

Our local preconditioners are based on the diagonal (or block-diagonal for $\mathtt{ndof} > 1$) of local matrices A^e. A scaling is used as a pre-preconditioner to normalize the coefficients of A by its diagonal coefficients (or block diagonal). The original system defined in Eq. (2.6) may be rewritten as $\tilde{A}X = \tilde{F}$ in which

$$\tilde{A} = \bigcup_{i=1}^{p} \tilde{A}_i^{EBE} \text{ with } \tilde{A}_i^{EBE} = \mathop{\mathbf{A}}_{e=1}^{\mathtt{nel}_i} \tilde{A}^e \tag{2.13}$$

and

$$\tilde{F} = \bigcup_{i=1}^{p} \tilde{F}_i^{EBE} \text{ with } \tilde{F}_i^{EBE} = \mathop{\mathbf{A}}_{e=1}^{\mathtt{nel}_i} \tilde{F}^e, \tag{2.14}$$

where $\tilde{A}^e$ and $\tilde{F}^e$ are local scaling of A^e and F^e:

$$\tilde{A}^e = \begin{bmatrix} A_I^{-1}A_{11}^e & A_I^{-1}A_{12}^e & A_I^{-1}A_{13}^e \\ A_J^{-1}A_{21}^e & A_J^{-1}A_{22}^e & A_J^{-1}A_{23}^e \\ A_K^{-1}A_{31}^e & A_K^{-1}A_{32}^e & A_K^{-1}A_{33}^e \end{bmatrix} \tag{2.15}$$

and

$$\tilde{F}^e = \begin{bmatrix} A_I^{-1}F_1^e \\ A_J^{-1}F_2^e \\ A_K^{-1}F_3^e \end{bmatrix}, \tag{2.16}$$

with A_{11}^e, A_{12}^e, A_{13}^e, A_{21}^e, A_{22}^e, A_{31}^e, A_{32}^e, and A_{33}^e block matrices of dimension $\mathtt{ndof} \times \mathtt{ndof}$ for each element matrix A^e. The submatrices A_I, A_J, and A_K also have dimension $\mathtt{ndof} \times \mathtt{ndof}$ but represent the global block matrices according to the number of degrees of freedom of the global nodes I, J, and K, respectively. F_1^e, F_2^e, and F_3^e, in turn, are blocks of dimension $\mathtt{ndof} \times 1$ for each element vector F^e. The inverse of each global block matrix A_I, with $1 \leqslant I \leqslant \mathtt{nnodes}_i$ ($\mathtt{nnodes}_i$ defines the number of nodes in a partition i of the mesh), is calculated explicitly.

REMARK 1: The scaling described in this section is not the combinatoric operation enunciated for SPIKE preconditioner described in Sec. 2.4.1.

REMARK 2: The scaling (or block-scaling) is proposed with exactly 3 subgroups because only triangular linear elements are considered in our finite element formulations.

REMARK 3: To obtain a block A_I, all the contributions of the matrices of the elements that have node I as a common node are added. For example, if a node I is surrounded by 6 elements, the contributions of these 6 element matrices must be considered to compute the block A_I. For simplicity, consider a problem with one degree of freedom per node. In this case, A_I would be a real number represented by the sum of all coefficients of the element matrices that are related to the node I. In the general case, when $\texttt{ndof} > 1$, A_I will be a matrix block of dimension $\texttt{ndof} \times \texttt{ndof}$.

REMARK 4: After the calculation of the block A_I inverse matrix, is necessary to check if some row of A_I – or degree of freedom – is associated with Dirichlet boundary conditions. In the affirmative case, the corresponding row and column of A_I^{-1} should be set with the row and column of the equivalent identity matrix.

REMARK 5: We set zeros in the element vector $\tilde{F}^e$ positions in which the Dirichlet boundary condition is defined.

2.5.3.1 DIAGe and BlockDIAGe Preconditioners

The numerical results of the DIAGe and BlockDIAGe preconditioners are equivalent to the scaling processes, as described in Eqs. (2.13) and (2.14). That is, such preconditioners are basically a simulation of a scaling. There are two possibilities to perform this issue:

(i) DIAGe and BlockDIAGe are executed just once as a scaling described in Eqs. (2.13) and (2.14), exactly to perform the GMRES algorithm on the linear system $AX = F$;

(ii) DIAGe and BlockDIAGe are not executed as described in Eqs. (2.13) and (2.14). After each matrix-vector product, we perform some convenient scalar multiplications on each resulting vector. The effective preconditioning is a simple multiplication of the diagonal (or block diagonal) over the vector resulting from matrix-vector product.

Despite the preconditioners need to be applied for every iteration when a matrix-vector product is required, the runtime of the DIAGe or BlockDIAGe for the second possibility is smaller when compared to diagonal (or block diagonal) scalings presented by the first possibility. Thus, we establish that henceforth, the application of the DIAGe and BlockDIAGe preconditioners will always occur according to the second possibility described.

2.5.3.2 LUe and BlockLUe Preconditioners

Let $\bar{A}^e$ be an approximation of each element matrix $\tilde{A}^e$:

$$\bar{A}^e = \begin{bmatrix} I_{\texttt{ndof}} & A^e_{12} & A^e_{13} \\ A^e_{21} & I_{\texttt{ndof}} & A^e_{23} \\ A^e_{31} & A^e_{32} & I_{\texttt{ndof}} \end{bmatrix}, \tag{2.17}$$

where A^e_{mn}, with $1 \leqslant m, n \leqslant 3$, are block matrices of dimension $\texttt{ndof} \times \texttt{ndof}$ for each element matrix $\tilde{A}^e$ and $I_{\texttt{ndof}}$ is the identity matrix of order $\texttt{ndof}$.

Preconditioners LUe and BlockLUe are defined simply as LU decompositions of $\bar{A}^e$ in element level. The preconditioning matrix M_i can be defined as

$$M_i = \mathop{\mathbf{A}}_{e=1}^{\texttt{nel}_i} \bar{A}^e = \mathop{\mathbf{A}}_{e=1}^{\texttt{nel}_i} L^e U^e. \tag{2.18}$$

For each matrix-vector $p_i = M_i^{-1} \tilde{A}_i^{EBE} v_i$, with $\tilde{A}_i^{EBE}$ defined as in the Eq. 2.13, preconditioners LUe and BlockLUe are applied following two main steps:

Step 1: Perform the matrix-vector product $z_i = \tilde{A}_i^{EBE} v_i$;

Step 2: Calculate the lower and upper triangular systems from $p_i = M_i^{-1} z_i$,

$$\begin{aligned} & p_i = M_i^{-1} z_i \Longrightarrow M_i p_i = z_i \Longrightarrow \mathop{\mathbf{A}}_{e=1}^{\texttt{nel}_i} (L^e U^e p^e = z^e) \\ & (\texttt{lower triangular}) \Longrightarrow L^e q^e = z^e \\ & \text{and} \\ & (\texttt{upper triangular}) \Longrightarrow U^e p^e = q^e. \end{aligned}$$

REMARK 6: The triangular systems $L^e q^e = z^e$ and $U^e p^e = q^e$ are solved for each element. That is, a single loop with $e = \{1, ..., \texttt{nel}\}$ is executed whenever the preconditioner is applied.

2.5.3.3 SGSe and BlockSGSe Preconditioners

SGSe and BlockSGSe are split preconditioners given by the preconditioning matrices M_{Li} and M_{Ri}. These matrices are obtained by the trivial Gauss-Seidel decomposition of matrix $\bar{A}^e$ (defined in Eq. 2.17) at the element level as

$$M_{Li} = \mathop{\mathbf{A}}_{e=1}^{\texttt{nel}_i} L^e \text{ with } L^e = \begin{bmatrix} I_{\texttt{ndof}} & O & O \\ A^e_{21} & I_{\texttt{ndof}} & O \\ A^e_{31} & A^e_{32} & I_{\texttt{ndof}} \end{bmatrix} \tag{2.19}$$

and

$$M_{Ri} = \underset{\mathtt{e=nel}_i}{\overset{1}{\mathbf{A}}} U^e \text{ with } U_e = \begin{bmatrix} I_{\mathtt{ndof}} & A^e_{12} & A^e_{13} \\ O & I_{\mathtt{ndof}} & A^e_{23} \\ O & O & I_{\mathtt{ndof}} \end{bmatrix}. \tag{2.20}$$

For each matrix-vector $p_i = M_{L_i}^{-1} \tilde{A}_i^{EBE} M_{R_i}^{-1} v_i$, preconditioners SGS$e$ and BlockSGSe are applied following three main steps:

Step 1: Calculate the upper triangular system from $w_i = M_{R_i}^{-1} v_i$,

$$w_i = M_{R_i}^{-1} v_i \Longrightarrow M_{Ri} w_i = v_i \Longrightarrow \underset{\mathtt{e=nel}_i}{\overset{1}{\mathbf{A}}} \left(U^e w^e = v^e \right);$$

Step 2: Perform the matrix-vector product $z_i = \tilde{A}_i^{EBE} w_i$;

Step 3: Calculate the lower triangular system from $p_i = M_{L_i}^{-1} z_i$,

$$p_i = M_{L_i}^{-1} z_i \Longrightarrow M_{Li} p_i = z_i \Longrightarrow \overset{\mathtt{nel}_i}{\underset{e=1}{\mathbf{A}}} \left(L^e p^e = z^e \right).$$

3 Domain Decomposition

As stated by Korneev and Langer (2018), domain decomposition methods nowadays provide powerful tools for constructing efficient parallel solvers for large-scale systems of algebraic equations arising from the discretization of partial differential equations. The classical alternating Schwarz method and the classical substructuring technique have led to advanced overlapping and nonoverlapping domain decomposition solvers. In this context, preconditioners can be analyzed from a unified point of view now called Schwarz theory. This chapter is organized in two sections: an overview of the traditional domain composition approaches and a detailed description of our alternative domain decomposition approach.

3.1 Traditional Domain Decomposition Approaches

Toselli and Widlund (2006) refer to domain decomposition as a splitting or approximation of a partial differential equation into coupled problems on smaller subdomains forming a partition of the original domain. The authors state that this approach may be employed in three different ways: it may enter at the continuous level, where different physical models may be used in different regions; or at the discretization level, where it may be convenient to employ different approximation methods in different regions; or in the solution of the algebraic systems arising from the approximation of the partial differential equation. They also emphasize these three aspects are very often interconnected in practice.

More recently, Dolean et al. (2015) presented an excellent overview of the most popular domain decomposition methods for partial differential equations. The authors introduce the main classes of domain decomposition algorithms: Schwarz, Optimized Schwarz, Neumann-Neumann, and FETI (Finite Element Tearing and Interconnecting). Schwarz's method appeared initially in 1870 (Schwarz, 1870) but since then it has suffered many modifications. The most popular one is the Restricted Additive Schwarz (RSA) (Cai and Sarkis, 1999) that is the default parallel solver in PETSc (Balay et al., 2017). The significant advantages of RSA algorithms occur specifically when the overlap is minimal. In this case, RSA is equivalent to a Block-Jacobi Method (Saad, 2003). Optimized Schwarz methods, in turn, are based on classical domain decomposition, but they use more efficient transmission conditions than the conventional Dirichlet conditions at the interfaces between subdomains. Neumann-Neumann methods (Mandel, 1993) require pseudo inverses for local Neumann solves which can be ill-posed either in the formulation of the domain decomposition problem or in the formulation of the domain decomposition preconditioner.

FETI methods (Farhat and Roux, 1991) are based on the introduction of Lagrange multipliers on the interfaces to ensure a weak continuity of the solution.

Moreover, a partial differential equation when discretized over a certain grid or mesh can be solved by numerous methods such as the finite differences method or the finite elements method. The discretization also defines a system of linear equations that can be represented by a sparse matrix. According to (Buluç et al., 2016), while it is always possible to use that sparse matrix to do the actual computation over the mesh or grid, sometimes this can be wasteful when the matrix need not be formed explicitly. In this situation is recommended to provide the mesh partitioning using graph partitioning solvers. In addition, (Buluç et al., 2016) presents a useful survey concern traditional graph partitioning algorithms, recent advances, and a long list of the most used software, such as,JOSTLE (Walshaw and Cross, 2002), the well-known METIS family (Karypis and Kumar, 1998a)(Karypis and Kumar, 1998b), and SCOTCH (Pellegrini, 2012).

Performing domain decomposition for the numerical solution of partial differential equations that assures robustness and scalability is most often a very complicated task. Smith (1997) emphasizes that ensuring data locality, load balance, and low communication is essential. In general, the division of subdomains can be seen as a graph partitioning problem that naturally arises from the geometry of the discretized physical domain. Moreover, the partitioning can be obtained by using data from the matrix rather than the geometric information (e.g. in (Cai and Saad, 1996)(Balay et al., 2017)). Our domain decomposition approach, firstly suggested in (Lima et al., 2016), is also based on the matrix but we propose an efficient preprocessing of the mesh to set out the computation of the finite element application totally in parallel. In the next subsection, this process is described in details.

3.2 Alternative Domain Decomposition Approach

Our intent is to promote a domain decomposition approach that enables generating finite element linear systems with a suitable sparsity pattern. That means to produce a narrow banded linear system that provides more flexibility to apply preconditioning and to reduce the communication overhead. Our new domain decomposition approach assumes a series of adjustments. For the sake of simplicity, but not for the loss of generality, we consider an example to clarify the explanation in the beginning of this section. Following, we present the general ideas about the new domain decomposition approach.

3.2.1 A simple example to illustrate the new approach

In general, we follow some simple steps to apply our domain decomposition approach. We read the finite element mesh and obtain the sparsity pattern of the associated linear

system. This system, whose coefficients have not yet been calculated, is then reordered and partitioned in a balanced way. Finally, we return to the mesh and apply the partitioning of this conceptual linear system. This procedure is performed in a previous phase, even before any computation of the finite element matrices has been made.

In order to improve our domain decomposition approach understanding, we consider an example problem. Consider the unstructured finite element mesh of 22 nodes (`nnodes`) and 28 elements (`nel`) of Fig. 3.1. This example represents a problem with 3 degrees of freedom per node (`ndof`). The 7 highlighted nodes in gray are associated with 14 unknowns highlighted in blue. The symbol "$-$" indicates prescriptions and the rest of the unrelated nodes represents Dirichlet boundary conditions. We choose the highlighted gray nodes to form the preliminary sparsity which will be denoted as sparsity pattern from the mesh (see Fig. 3.2). Using a reordering algorithm as RCM (Liu and Sherman, 1976), we obtain a vector of permutation $P_{mesh} = \{7, 5, 6, 1, 4, 2, 3\}$ in which the conceptual linear system has rows and columns changed. P_{mesh} produces a better sparsity from the mesh. The original structure has bandwidth equal to 5 (Fig. 3.2a) , while the reordered structured has bandwidth equal to 2 (Fig. 3.2b).

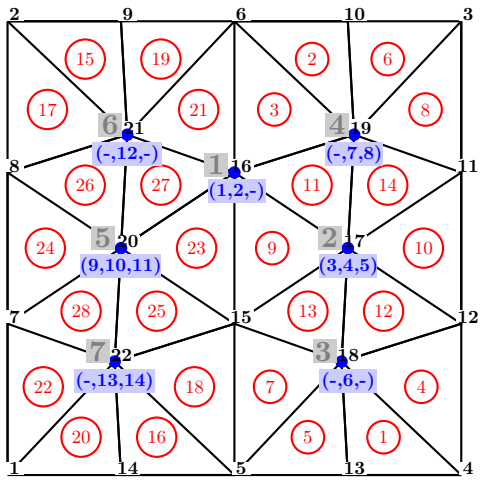


Figure 3.1 – A simple example mesh with 22 nodes and 28 elements. We indicate the 7 nodes (highlighted in gray) associated with 14 unknowns (highlighted in blue).

	1	2	3	4	5	6	7
1	•	•		•	•	•	
2	•	•	•	•			
3		•	•				
4	•	•		•			
5	•				•	•	•
6	•				•	•	
7					•		•

(a) Original

	1	2	3	4	5	6	7
1	•	•					
2	•	•	•	•			
3		•	•	•			
4		•	•	•	•	•	
5				•	•	•	
6				•	•	•	•
7						•	•

(b) Reordered

Figure 3.2 – Illustration of the sparsity pattern from the original mesh. (a) Original sparsity pattern from the mesh nodes that are related with some unknown. (b) Reordered sparsity pattern from these mesh nodes after using RCM.

The permutation P_{mesh} is used to produce the sparsity pattern of the finite element matrix according to unknowns and degrees of freedom, generating the permutation P_{matrix}. The unknowns (highlighted in blue) associated with finite element discretization produce a structure with much larger dimension (see Fig. 3.4a), because each node is associated with about 3 unknowns. The permutation P_{matrix} is generated considering the permutation P_{mesh} and its inverse $invP_{mesh}$ arranging the groups of 3 unknowns associated with each node. Fig. 3.3 illustrates the relation between the finite element matrix sparsity permutation P_{matrix} with the permutation of the mesh P_{mesh} and the inverse permutation $invP_{mesh}$ for this example problem. The matrix permutation $P_{matrix} = \{13, 14, 9, 19, 11, 12, 1, 2, 7, 8, 3, 4, 5, 6\}$ is applied to the sparsity pattern of the unknowns to produce a structure with a lower bandwidth. Original finite element matrix sparsity presents bandwidth equal to 11 (Fig. 3.4a), whereas the reordered structured has bandwidth equal to 6 (Fig. 3.4b).

P_{mesh}	$invP_{mesh}$	Reordering of unknows.	P_{matrix}
7	1	**(-,13,14)**	13,14
5	2	**(9,10,11)**	9,10,11
6	3	**(-,12,-)**	12
1	4	**(1,2,-)**	1,2
4	5	**(-,7,8)**	7,8
2	6	**(3,4,5)**	3,4,5
3	7	**(-,6,-)**	6

Figure 3.3 – Relation between permutation vectors P_{mesh}, $invP_{mesh}$, and P_{matrix} for the example problem.

After the finite element matrix sparsity pattern reordering, the algorithm chains-on-chains partitioning (CCP) (Pınar and Aykanat, 2004) generates an appropriate division of partitions. The CCP algorithm returns a vector of indices **d** with size according to the number of partitions. Each partition is balanced taking into account the number of the nonzero coefficients of the global finite element matrix. For this example, the partitioner

vector $\mathbf{d} = \{1, 6, 9, 15\}$ is obtained. Figure 3.5 shows the global finite element matrix sparsity pattern divided into 3 partitions. Partition 1 represents data belongs exclusively to rank 1 – analogously, Partitions 2 and 3 are an exclusive part of rank 2 and rank 3. Intersection of 1 and 2 represents the region of overlaps between Partition 1 and 2 – analogously, Intersection of 2 and 3 denotes the region of overlaps between Partitions 2 and 3. The use of the CCP algorithm over the reordered finite element matrix sparsity pattern (Fig. 3.4b) results the partitioning represented in Fig. 3.5a.

Finally, the partitioned model system (see Fig. 3.5a) is used to partition the finite element mesh (see Fig. 3.5b). This sequential preprocessing is not complicated. As an example, for each unknown in Partition 1 — degree of freedom numbers 13, 14, 9, 10, and 11 identify the related elements. For example, the degree of freedom 13 is related with elements number 16, 18, 20, 22, 25, and 28. Analogously, this procedure is repeated for each unknown in Partitions 2 and 3. The original finite element matrix sparsity pattern with bandwidth 11 (Fig. 3.4a) is defined as a banded matrix with bandwidth 6 (Fig. 3.4b) In practice, that partitioning gives a balanced load and fewer ranks sharing the same partition. In short, our new domain decomposition approach allows intersections with just two partitions. Elements associated with two partitions belong to an Intersection. In Fig. 3.5a, they are colored light blue (Intersection of 1 and 2) and purple (intersection of 2 and 3).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$		$a_{1,7}$	$a_{1,8}$	$a_{1,9}$	$a_{1,10}$	$a_{1,11}$	$a_{1,12}$		
2	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$		$a_{2,7}$	$a_{2,8}$	$a_{2,9}$	$a_{2,10}$	$a_{2,11}$	$a_{2,12}$		
3	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$	$a_{3,8}$						
4	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$	$a_{4,8}$						
5	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$	$a_{5,8}$						
6			$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$								
7	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$		$a_{7,7}$	$a_{7,8}$						
8	$a_{8,1}$	$a_{8,2}$	$a_{8,3}$	$a_{8,4}$	$a_{8,5}$		$a_{8,7}$	$a_{8,8}$						
9	$a_{9,1}$	$a_{9,2}$							$a_{9,9}$	$a_{9,10}$	$a_{9,11}$	$a_{9,12}$	$a_{9,13}$	$a_{9,14}$
10	$a_{10,1}$	$a_{10,2}$							$a_{10,9}$	$a_{10,10}$	$a_{10,11}$	$a_{10,12}$	$a_{10,13}$	$a_{10,14}$
11	$a_{11,1}$	$a_{11,2}$							$a_{11,9}$	$a_{11,10}$	$a_{11,11}$	$a_{11,12}$	$a_{11,13}$	$a_{11,14}$
12	$a_{12,1}$	$a_{12,2}$							$a_{12,9}$	$a_{12,10}$	$a_{12,11}$	$a_{12,12}$		
13									$a_{13,9}$	$a_{13,10}$	$a_{13,11}$		$a_{13,13}$	$a_{13,14}$
14									$a_{14,9}$	$a_{14,10}$	$a_{14,11}$		$a_{14,13}$	$a_{14,14}$

(a) Original

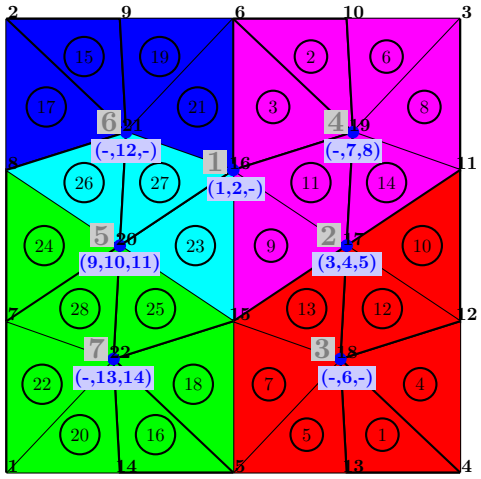
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	$a_{13,13}$	$a_{13,14}$	$a_{13,9}$	$a_{13,10}$	$a_{13,11}$									
2	$a_{14,13}$	$a_{14,14}$	$a_{14,9}$	$a_{14,10}$	$a_{14,11}$									
3	$a_{9,13}$	$a_{9,14}$	$a_{9,9}$	$a_{9,10}$	$a_{9,11}$	$a_{9,12}$	$a_{9,1}$	$a_{9,2}$						
4	$a_{10,13}$	$a_{10,14}$	$a_{10,9}$	$a_{10,10}$	$a_{10,11}$	$a_{10,12}$	$a_{10,1}$	$a_{10,2}$						
5	$a_{11,13}$	$a_{11,14}$	$a_{11,9}$	$a_{11,10}$	$a_{11,11}$	$a_{11,12}$	$a_{11,1}$	$a_{11,2}$						
6			$a_{12,9}$	$a_{12,10}$	$a_{12,11}$	$a_{12,12}$	$a_{12,1}$	$a_{12,2}$						
7			$a_{1,9}$	$a_{1,10}$	$a_{1,11}$	$a_{1,12}$	$a_{1,1}$	$a_{1,2}$	$a_{1,7}$	$a_{1,8}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	
8			$a_{2,9}$	$a_{2,10}$	$a_{2,11}$	$a_{2,12}$	$a_{2,1}$	$a_{2,2}$	$a_{2,7}$	$a_{2,8}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	
9							$a_{7,1}$	$a_{7,2}$	$a_{7,7}$	$a_{7,8}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	
10							$a_{8,1}$	$a_{8,2}$	$a_{8,7}$	$a_{8,8}$	$a_{8,3}$	$a_{8,4}$	$a_{8,5}$	
11							$a_{3,1}$	$a_{3,2}$	$a_{3,7}$	$a_{3,8}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$
12							$a_{4,1}$	$a_{4,2}$	$a_{4,7}$	$a_{4,8}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$
13							$a_{5,1}$	$a_{5,2}$	$a_{5,7}$	$a_{5,8}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$
14											$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$

(b) Reordered using the permutation P_{matrix}

Figure 3.4 – Illustration of the finite element matrix sparsity pattern reduction using the vector permutation P_{matrix}. (a) Original finite element matrix sparsity pattern. (b) Reordered finite element matrix sparsity pattern after using P_{matrix}.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	$a_{13,13}$	$a_{13,14}$	$a_{13,9}$	$a_{13,10}$	$a_{13,11}$									
2	$a_{14,13}$	$a_{14,14}$	$a_{14,9}$	$a_{14,10}$	$a_{14,11}$									
3	$a_{9,13}$	$a_{9,14}$	$a_{9,9}$	$a_{9,10}$	$a_{9,11}$	$a_{9,12}$	$a_{9,1}$	$a_{9,2}$						
4	$a_{10,13}$	$a_{10,14}$	$a_{10,9}$	$a_{10,10}$	$a_{10,11}$	$a_{10,12}$	$a_{10,1}$	$a_{10,2}$						
5	$a_{11,13}$	$a_{11,14}$	$a_{11,9}$	$a_{11,10}$	$a_{11,11}$	$a_{11,12}$	$a_{11,1}$	$a_{11,2}$						
6			$a_{12,9}$	$a_{12,10}$	$a_{12,11}$	$a_{12,12}$	$a_{12,1}$	$a_{12,2}$						
7			$a_{1,9}$	$a_{1,10}$	$a_{1,11}$	$a_{1,12}$	$a_{1,1}$	$a_{1,2}$	$a_{1,7}$	$a_{1,8}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	
8			$a_{2,9}$	$a_{2,10}$	$a_{2,11}$	$a_{2,12}$	$a_{2,1}$	$a_{2,2}$	$a_{2,7}$	$a_{2,8}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	
9							$a_{7,1}$	$a_{7,2}$	$a_{7,7}$	$a_{7,8}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	
10							$a_{8,1}$	$a_{8,2}$	$a_{8,7}$	$a_{8,8}$	$a_{8,3}$	$a_{8,4}$	$a_{8,5}$	
11							$a_{3,1}$	$a_{3,2}$	$a_{3,7}$	$a_{3,8}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$
12							$a_{4,1}$	$a_{4,2}$	$a_{4,7}$	$a_{4,8}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$
13							$a_{5,1}$	$a_{5,2}$	$a_{5,7}$	$a_{5,8}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$
14											$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$

(a) Finite element matrix sparsity partitioned by the CCP algorithm



(b) Partitioned finite element mesh

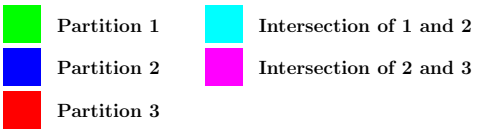


Figure 3.5 – Finite element information divided into 3 partitions.

An obvious question would be: why do we not obtain P_{matrix} directly from the finite element matrix sparsity? Our experiments demonstrated that, for problems with a degree of freedom per node greater than 1, the use of P_{mesh} in an intermediate way greatly reduces the number of overlaps.

3.2.2 General ideas about the new domain decomposition approach

In this subsection, there is a detailed description of how to receive a sequential mesh of finite elements and divide it into p partitions according to our domain decomposition approach. Just 12 main steps are enough to perform this task. Figure 3.6 presents the outline of these steps. A finite element mesh with `nnodes` nodes and `nel` elements is considered. We also propose a series of algorithms to perform each step of our finite

element domain decomposition. These algorithms are presented in Appendix A.

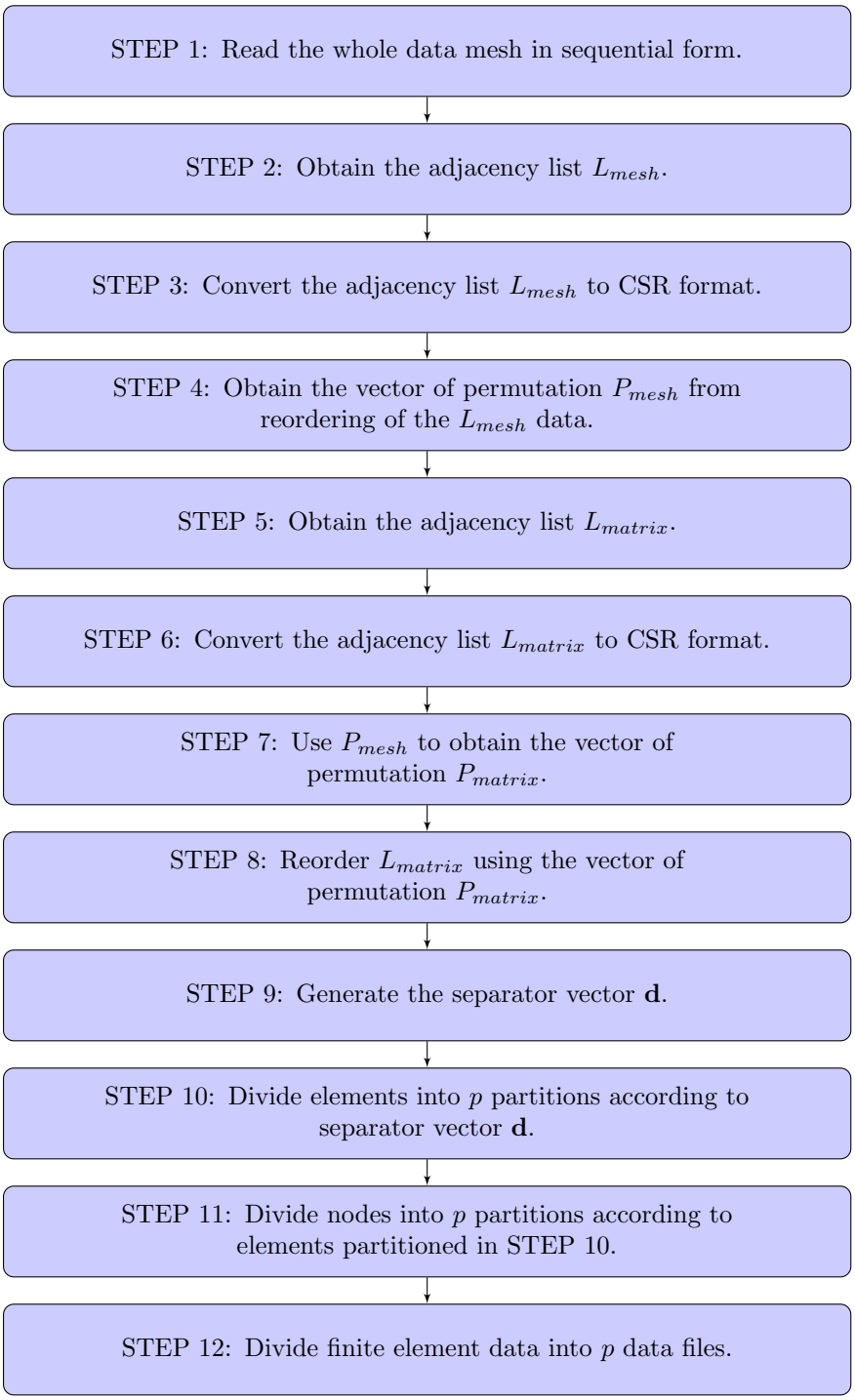


Figure 3.6 – The overall domain decomposition framework summarized by performing 12 steps.

3.2.2.1 STEP 1: Read finite element mesh

This step guides the sequential reading of the entire finite element mesh. The mesh data is arranged in a file with a specific format which is illustrated in Fig. 3.7. The first

line of the file contains the number of nodes `nnodes` that form the finite element mesh. The next `nnodes` lines correspond to the information for all nodes in the mesh. Values $\mathtt{x_a}$ and $\mathtt{y_a}$, with $1 \leqslant \mathtt{a} \leqslant \mathtt{nnodes}$ are the real coordinates of the node `a`. Terms $\mathtt{type_{ab}}$, with $1 \leqslant \mathtt{b} \leqslant \mathtt{ndof}$, indicate if the `b`th degree of freedom of the node `a` is an unknown ($\mathtt{type_{ab}} = 1$) or a prescribed node ($\mathtt{type_{ab}} = 0$). For example in Fig. 3.1, node 16 has 3 degrees of freedom, with 2 unknowns and 1 prescribed node, so that, $\mathtt{type_{16,1}} = 1$, $\mathtt{type_{16,2}} = 1$, and $\mathtt{type_{16,3}} = 0$. The following line displays the number `nel` of elements of the mesh. The next `nel` lines contain the connectivity of each element of the mesh. In our approach only triangular elements are used, so that each element is designated exactly by three nodes.

$$
\begin{array}{l}
< \mathtt{nnodes} > \\
< \mathtt{x_1}, \mathtt{y_1} > \quad < \mathtt{type_{1,1}} > \quad < \mathtt{type_{1,2}} > \quad \cdots \quad < \mathtt{type_{1,ndof}} > \\
< \mathtt{x_2}, \mathtt{y_2} > \quad < \mathtt{type_{2,1}} > \quad < \mathtt{type_{2,2}} > \quad \cdots \quad < \mathtt{type_{2,ndof}} > \\
\\
\qquad\qquad\qquad\qquad \vdots \qquad \vdots \qquad \vdots \\
\\
< \mathtt{x_{nnodes}}, \mathtt{y_{nnodes}} > \quad < \mathtt{type_{nnodes,1}} > \quad < \mathtt{type_{nnodes,2}} > \quad \cdots \quad < \mathtt{type_{nnodes,ndof}} > \\
< \mathtt{nel} > \\
< \mathtt{node_{1,1}} > \quad < \mathtt{node_{2,1}} > \quad < \mathtt{node_{3,1}} > \\
< \mathtt{node_{1,2}} > \quad < \mathtt{node_{2,2}} > \quad < \mathtt{node_{3,2}} > \\
\\
\qquad \vdots \\
\\
< \mathtt{node_{1,nel}} > \quad < \mathtt{node_{2,nel}} > \quad < \mathtt{node_{3,nel}} >
\end{array}
$$

Figure 3.7 – Sequential input file format.

3.2.2.2 STEP 2: Build the adjacency list L_{mesh}

After reading the mesh data, due to large sparsity arising from the finite element mesh, an adjacency list named L_{mesh} is created. L_{mesh} stores only the sparsity pattern designed by nodes which have at least one unknown associated with itself. For more details on how to get L_{mesh}, check Section A.1 in Appendix A.

3.2.2.3 STEP 3: Convert the adjacent list L_{mesh} to CSR format

Aiming for high performance, the adjacent L_{mesh} is converted to the Compressed Sparse Row (CSR) (Saad, 2003) format. As well known, CSR is a strategy to store a sparse matrix. With this objective, three arrays are taken into account:

- A real array AA contains the real values a_{ij} stored row by row, from row 1 to `n` (where `n` is the matrix order). The length of AA is `nnz` (the number of nonzero coefficients a_{ij});

- An integer array JA contains the column indices of the elements a_{ij} as stored in the array AA. The length of JA is also `nnz`;
- An integer array IA contains the pointers to the beginning of each row in the arrays AA and JA. Thus, the counter of $IA(i)$ is the position in arrays AA and JA where the i-th row starts. The length of IA is $\mathtt{n}+1$ with $IA(\mathtt{n}+1)$ containing the number $IA(1)+\mathtt{nnz}$, i.e., the address in AA and JA of the beginning of a fictitious row number $\mathtt{n}+1$.

As an example, consider the matrix

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

Matrix A may be stored using CSR format as follows:

AA	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
JA	1	4	1	2	4	1	3	4	5	3	4	5
IA	1	3	6	10	12	13						

Since the sparsity pattern provided by L_{mesh} needs to be stored, only two arrays, named, JA_{mesh} and IA_{mesh} are used (there is no real value a_{ij} to be stored in AA_{mesh}).

3.2.2.4 STEP 4: Reorder sparsity pattern provided by L_{mesh}

This step includes the reordering of L_{mesh} data. At this moment, all information of the adjacency list L_{mesh} is stored in two arrays; namely, JA_{mesh} and IA_{mesh}. During the development of our research, we studied three algorithms to reduce matrix bandwidth and profile:

- Reverse Cuthill-McKee (RCM) (Liu and Sherman, 1976);
- Spectral (Barnard et al., 1995);
- Weighted spectral ordering (WSO) (Manguoglu et al., 2010).

The Weighted Spectral Ordering (WSO) could not be used in this step because this reordering requires the content of AA_{mesh}. Thus, arrays JA_{mesh} and IA_{mesh} are passed as

arguments to the algorithms RCM or Spectral to reorder the sparsity pattern of L_{mesh} and the vector of permutation P_{mesh} is obtained.

3.2.2.5 STEP 5: Build the adjacency list L_{matrix}

In this step an adjacency list named L_{matrix} is obtained. Different from the adjacency L_{mesh}, L_{matrix} stores the sparsity pattern of the finite element problem. It coincides with the unknowns that will be solved in the linear system. For more details on how to build L_{matrix}, check Section A.2 in Appendix A.

3.2.2.6 STEP 6: Convert the adjacency list L_{matrix} to CSR format

Similarly to STEP 3, this step suggests converting L_{matrix} to CSR format. Thus, L_{matrix} is represented by two arrays; namely, JA_{matrix} and IA_{matrix}. Note, one more time, there is no real value a_{ij} to be stored in AA_{matrix}.

3.2.2.7 STEP 7: Generate the permutation vector P_{matrix}

In this step a vector of permutation named P_{matrix} is generated. This array is a rearrangement of the vector of permutation P_{mesh}. P_{matrix} larger than P_{mesh}, since it consider all unknowns of the finite element linear system. P_{matrix} also keeps the order proposed by P_{mesh} but reorganizes it by blocks of degrees of freedom per node. Therefore, P_{mesh} is equivalent to P_{matrix} when the number of degrees of freedom per node is 1. Section A.3 of the Appendix A shows how to obtain the permutation vector P_{matrix} from the vector P_{mesh}.

3.2.2.8 STEP 8: Reorder L_{matrix} data

As presented in STEP 6, L_{matrix} data is stored in CSR format, more precisely in the arrays JA_{matrix} and IA_{matrix}. As can be emphasized, this reordering step is a little bit different from STEP 4. The RCM or the Spectral algorithms are not directly used to reorder the CSR structures JA_{matrix} and IA_{matrix}. The permutation vector P_{matrix} is used to achieve this goal.

3.2.2.9 STEP 9: Generate separator vector $\mathbf{d}$

This step defines the core part of the partitioning, that is, the generation of the separator vector $\mathbf{d}$. The primary purpose of the partitioning is to ensure load balance and reduce communication overhead. Thus, each partition needs to be, approximately, of the same size. In our case, it means having a very similar amount of nonzero coefficients. As the other operations of our domain decomposition approach, partitioning was first thought to meet the demands of the Spike preconditioner. Thus, the problem is to find the diagonal block matrices A_i and the other block matrices C_i and B_i (see Eq. 2.6). Several heuristics

were proposed to solve the general problem of partitioning in graphs, and the most known are those proposed in (Kernighan and Lin, 1970) and (Fiduccia and Mattheyses, 1988).

In this thesis, the partitioning is modeled as a specific problem, named, Chains-on-chains partitioning (CCP) (Pınar and Aykanat, 2004). The algorithm uses contiguous regions of the matrix to perform the partitioning, in which, each row is visited one after other. The objective is to find a sequence of $p-1$ separator indices to divide a task chain with the computational weight associated to p consecutive parts, promoting load balancing. In our applications, the tasks represent the `n` rows of the matrix; and its weights denote the number of nonzero coefficients in the rows. The partitioning effect is to distribute the nonzero coefficients in a balanced form among the processors. The CCP algorithm can be solved in polynomial time when the algorithm MinMax (Manne and Sorevik, 1995) is used to obtain the exact solution of the problem.

3.2.2.10 STEP 10: Split elements

In this step the elements are split into p partitions considering the separator vector $\mathbf{d}$. For each element e, the three nodes which compose the mesh connectivity are recovered. Each degree of freedom of a given node is investigated using the separator vector $\mathbf{d}$. If a degree of freedom belongs to the partition i, the element e is also associated to the partition i. Thus, every element is assigned to its corresponding partition. An element e can belong to more than one partition. Because of this, our approach can be classified as a domain decomposition with overlapping. Section A.4 of Appendix A brings more details in how to execute this step.

3.2.2.11 STEP 11: Split nodes

After splitting the elements into p partitions, the nodes should be split according to elements division. For each element that belongs to a specific partition, we verify what nodes form this element, and assign these nodes to that specific partition. Section A.5 of Appendix A shows how to perform this division of nodes.

3.2.2.12 STEP 12: Split data files

In the last step, the finite element data is divided according to the number of partitions required. Let p be the number of partitions. We write the corresponding data about nodes and elements in p files which will be employed in the parallel processing. The mesh file input format of a given partition i (file $Mesh_i$) is presented in Fig. 3.8. The first line of the file $Mesh_i$ contains the number of nodes $nnodes_i$ that compose the partition i. The next $\texttt{nnodes}_i$ lines correspond to the information for all nodes in the mesh of the partition i. Values $\texttt{x}_\texttt{a}$ and $\texttt{y}_\texttt{a}$, with $1 \leqslant \texttt{a} \leqslant \texttt{nnodes}_i$ are the real coordinates of the node `a`.

Every tuple $< \mathtt{t}_{\mathtt{a,b}}, \mathtt{s}_{\mathtt{a,b}}, \mathtt{Id}_{\mathtt{a,b}} >$, with $1 \leqslant \mathtt{a} \leqslant \mathtt{nnodes}_i$ and $1 \leqslant \mathtt{b} \leqslant \mathtt{ndof}$, represents the following information about the degrees of freedom of each node:

- $< \mathtt{t}_{\mathtt{a,b}} >$: is an integer label to specify boundary conditions or parallel processing type.
- $< \mathtt{s}_{\mathtt{a,b}} >$: is an integer label that informs whether it should be sent or not to the neighbor partition;
- $< \mathtt{Id}_{\mathtt{a,b}} >$: is a mapping that assures the global and local interconnectivity that allows the parallel computation;

The following line of the file $Mesh_i$ displays $\mathtt{nel}_i$, the number of elements of the partition i. The next $\mathtt{nel}_i$ lines contain the connectivity of the mesh represented by the $\mathtt{nel}_i$ elements.

$$
\begin{array}{l}
< \mathtt{nnodes}_i > \\
< \mathtt{x}_1, \mathtt{y}_1 > < \mathtt{t}_{1,1}, \mathtt{s}_{1,1}, \mathtt{Id}_{1,1} > < \mathtt{t}_{1,2}, \mathtt{s}_{1,2}, \mathtt{Id}_{1,2} > \cdots < \mathtt{t}_{1,\mathtt{ndof}}, \mathtt{s}_{1,\mathtt{ndof}}, \mathtt{Id}_{1,\mathtt{ndof}} > \\
< \mathtt{x}_2, \mathtt{y}_2 > < \mathtt{t}_{2,1}, \mathtt{s}_{2,1}, \mathtt{Id}_{2,1} > < \mathtt{t}_{2,2}, \mathtt{s}_{2,2}, \mathtt{Id}_{2,2} > \cdots < \mathtt{t}_{2,\mathtt{ndof}}, \mathtt{s}_{2,\mathtt{ndof}}, \mathtt{Id}_{2,\mathtt{ndof}} > \\
\\
\qquad\qquad\qquad\qquad \vdots \qquad \vdots \qquad \vdots \\
\\
< \mathtt{x}_{\mathtt{nnodes}_i}, \mathtt{y}_{\mathtt{nnodes}_i} > < \mathtt{t}_{\mathtt{nnodes}_i,1}, \mathtt{s}_{\mathtt{nnodes}_i,1}, \mathtt{Id}_{\mathtt{nnodes}_i,1} > < \mathtt{t}_{\mathtt{nnodes}_i,2}, \mathtt{s}_{\mathtt{nnodes}_i,2}, \mathtt{Id}_{\mathtt{nnodes}_i,2} > \cdots \\
\qquad . \qquad\qquad\qquad\qquad\qquad\qquad \cdots < \mathtt{t}_{\mathtt{nnodes}_i,\mathtt{ndof}}, \mathtt{s}_{\mathtt{nnodes}_i,\mathtt{ndof}}, \mathtt{Id}_{\mathtt{nnodes}_i,\mathtt{ndof}} > \\
\\
< \mathtt{nel}_i > \\
< \mathtt{node}_{1,1} > \quad < \mathtt{node}_{2,1} > \quad < \mathtt{node}_{3,1} > \\
< \mathtt{node}_{1,2} > \quad < \mathtt{node}_{2,2} > \quad < \mathtt{node}_{3,2} > \\
\\
\qquad\qquad \vdots \\
\\
< \mathtt{node}_{1,\mathtt{nel}_i} > \quad < \mathtt{node}_{2,\mathtt{nel}_i} > \quad < \mathtt{node}_{3,\mathtt{nel}_i} >
\end{array}
$$

Figure 3.8 – $Mesh_i$: input file format.

Section A.6 in Appendix A describes, in detail, how to perform the mesh partitioning that provides the corresponding local-global mapping and types of the degrees of freedom in each partition i.

4 Parallel Finite Element Application

Chapter 3 discussed how to take a sequential finite element mesh and divide it into p partitions using our new domain decomposition approach. In this chapter the partitioning is used to implement the finite element application in parallel, in which five main steps are required

1. To read the parallel finite element mesh data;
2. To set the parallel structures to enable MPI updates;
3. To chose a storage scheme to accumulate the elementary matrices produced by finite element method;
4. To execute the finite element application in parallel;
5. To analyze the postprocessing data.

Steps (1), (2), and (3) are presented in Section 4.1. Subsection 4.1.1 explains how to read the parallel finite element mesh. Subsection 4.1.2 describes how to prepare the parallel structures which allow executions in distributed memory with required updates. Besides that, it reports how to perform the updates of finite element vectors. Subsections 4.1.3, 4.1.4, and 4.1.5 approach the storage schemes which are used in our implementations, namely, Compressed Sparse Row (CSR) (Saad, 2003) and Element-by-Element (EBE) (Hughes, 2012). The step (4) is discussed in Section 4.2, that is, the parallel processing of the finite element method, which occurs inside a predictor-multi-corrector scheme or even a loop of nonlinear iterations. In addition, two principal operations for parallel iterative solvers based on Krylov subspace are reported, the parallel matrix-vector product (which depends on storage scheme) and inner product. Finally, step (5) is summarized in Section 4.3, where the postprocessing generates output files to be visualized by the software ParaView (Ayachit, 2015).

4.1 Parallel finite element preprocessing

In this section, the preprocessing that ensures an efficient parallel execution throughout the finite element processing is discussed. Figure 4.1 shows an overview of the preprocessing composed of two primary actions: the reading of the finite element mesh in parallel and the creation of parallel structures to provide MPI updates. Besides, the second action is subdivided conform to CSR or EBE storage formats.

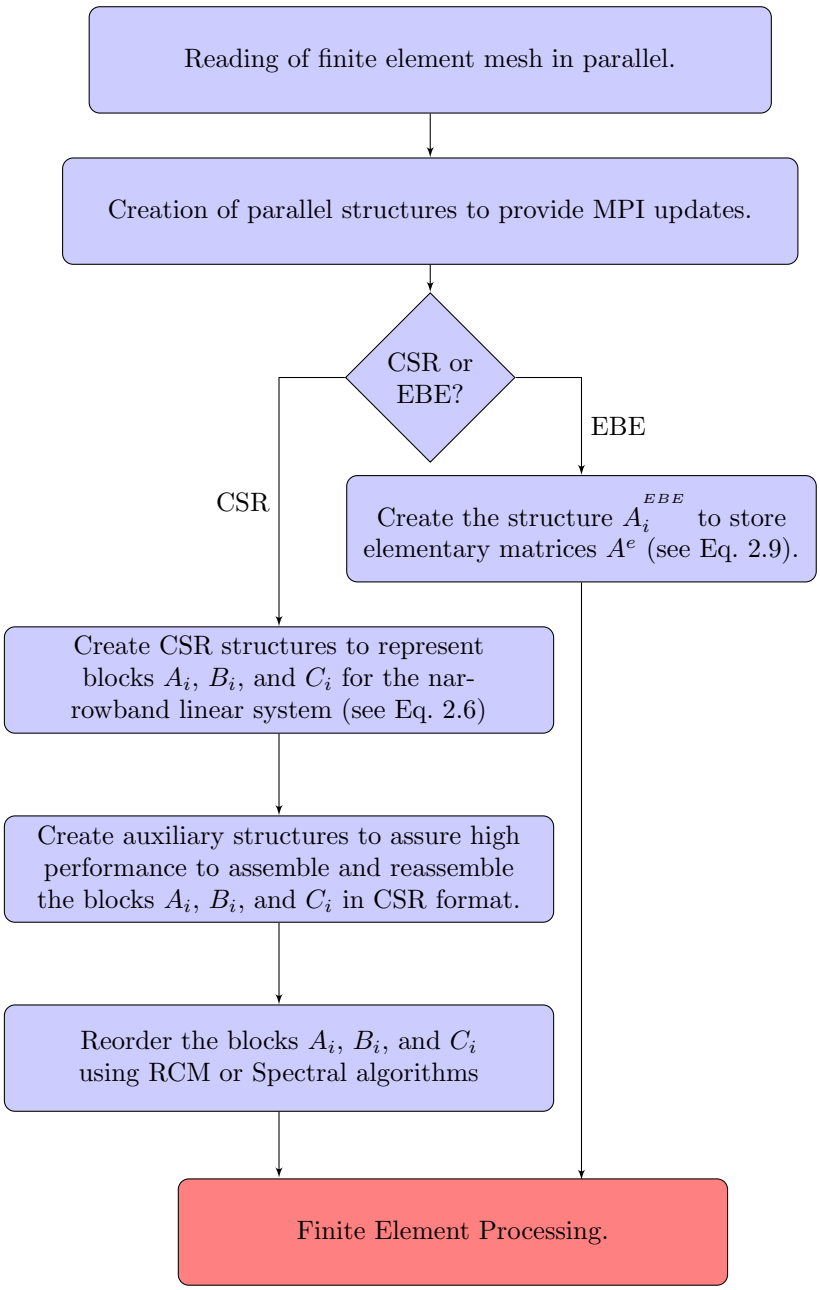


Figure 4.1 – Overview of the parallel finite element preprocessing.

4.1.1 Reading of finite element mesh in parallel

Suppose initial finite element mesh data is disposed in p files $Mesh_i$ (see Fig.3.8). Analogously to the sequential data reading, information of nodes and elements is read in structures *Node* and *Element* (see Subsection A.1). Admit *Node* with size $\texttt{nnodes}_i$ (number of nodes of the partition i) and fields *coord* (to store x,y coordinates), $Type[J]$, with $1 \leqslant J \leqslant \texttt{ndof}$ (to identify a degree of freedom of a node according to Table 4.1) and $Id[J]$, with $1 \leqslant J \leqslant \texttt{ndof}$, (to map the global and local connectivity). Also, data structure *Node* has another field named *Type_Send*, with size `ndof`, which identifies the degrees of freedom of a given node according to Table 4.1. The structure *Element*, in turn, has size $\texttt{nel}_i$ (number of elements of the partition i) and a field *Vertex* to store `nnoel` (number of nodes per element) nodes of $\texttt{nel}_i$ elements.

Table 4.1 – Description of identifiers type to promote parallel processing.

Variable	Value	Meaning
Type	0	Dirichlet boundary conditions in partition i
	1	Unknown in partition i related with partition i
	2	Unknown in partition $i-1$ related with partition i
	3	Unknown in partition $i+1$ related with partition i
Type_Send	1	No need to send
	2	Must be sent to partition $i-1$
	3	Must be sent to partition $i+1$
	23	Must be sent to partitions $i-1$ and $i+1$

4.1.2 Parallel structures to provide MPI update

As mentioned in the previous chapter, our new domain decomposition approach allows intersections between just two partitions. That means, the partition i is related only with the previous partition $i-1$ and the posterior partition $i+1$ (partition 1 is only related with partition 2 and partition p is only related with partition $p-1$). More precisely, if i is a partition composed of n_i unknowns it would be associated with n_{i-1} unknowns of partition $i-1$ and n_{i+1} unknowns of partition $i+1$. However, due to the sparsity pattern provided by finite element calculation, the associations are much smaller. In fact, the partition i needs to receive just $nrecv_{\texttt{bef}}$ indices from the partition $i-1$ and $nrecv_{\texttt{aft}}$ indices from the partition $i+1$. Despise that, these partitions have sizes n_{i-1} (much larger than $nrecv_{\texttt{bef}}$) and n_{i+1} (much larger than $nrecv_{\texttt{aft}}$). Analogously, partition i needs to send to partitions $i-1$ and $i+1$ only $nsend_{\texttt{bef}}$ and $nsend_{\texttt{aft}}$ indices.

Normally, the division of subdomains can be seen as a graph partition problem that naturally follows the geometry of the discretized physical domain. That means the partitioning of the finite element physical domain takes into account the connectivity of the elements themselves. Therefore, it is a natural division related to information of

nodes and elements. Our approach also includes the partitioning of the physical domain, but that task is performed in function of the linear system sparsity pattern derived from the discretization methods (see Eq. 2.6). As that sparsity arises from a narrow banded system, it may seem trivial to propose a set of messages to send and to receive through the Message Passing Interface (MPI)(Dongarra et al., 2013) protocol. However, if some cautions are not taken into account, MPI updates would be less efficient than what is desired because message sizes can be much larger. In this way, four data structures ensure a minimum amount of information traveling through the network. These structures are the mapping vectors $IdSend\_bef$, $IdReceive\_bef$, $IdSend\_aft$, and $IdReceive\_aft$. Fig. 4.2 illustrates how these structures map the relation between a partition i and the neighboring partitions $i-1$ and $i+1$. Note that vector $IdSend\_bef$ is a buffer to catch $nsend_{\texttt{bef}}$ coefficients of partition i in order to send them to partition $i-1$. Analogously, vector $IdSend\_aft$ catches $nsend_{\texttt{aft}}$ coefficients of partition i and sends them to partition $i+1$. On the other hand, vector $IRecv\_bef$ is a buffer to map $nrecv_{\texttt{bef}}$ coefficients that are received in partition i from partition $i-1$. Similarly, vector $IRecv\_aft$ maps $nrecv_{\texttt{aft}}$ coefficients received in partition i from partition $i+1$.

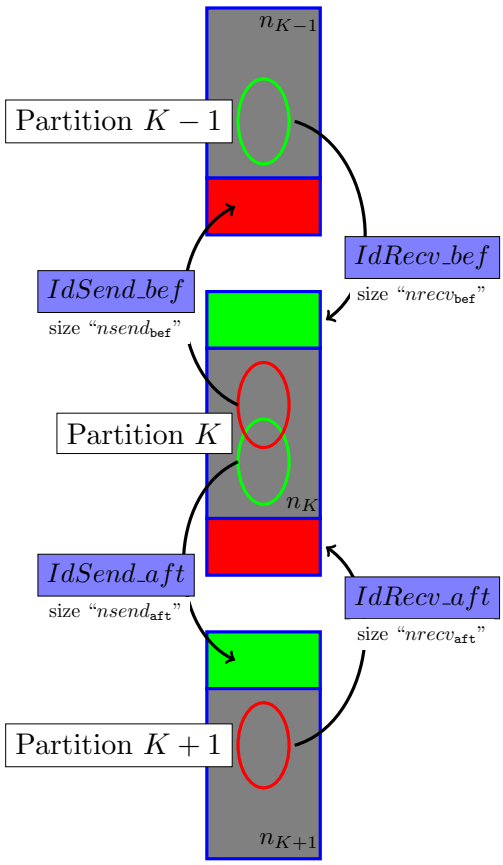


Figure 4.2 – Four mapping vectors: relations between partition i and its neighboring partitions $i-1$ and $i+1$.

Section B.1 in Appendix B describes how to obtain these four mapping vectors.

4.1.2.1 MPI update

In parallel computing, MPI updates are generally performed using vectors. Our approach maintains this trend. Thus, a generic vector U_i of size n_i from a partition i is assumed. The main region of the vector U_i with n_i coefficients will be named effective calculation area region. In order to have the communication between the neighboring partitions, two regions are considered:

- Previous communication region of the vector U_i with size $nrecv_{\texttt{bef}}$;
- Posterior communication region of the vector U_i with size $nrecv_{\texttt{aft}}$.

As can be emphasized, our domain decomposition enables MPI communications with just two other partitions. That is, partition i communicates with partitions $i-1$ and $i+1$. Of course, partition 1 communicates only with partition 2 and partition p only communicates with partition $p-1$. Thus, when the three U_i vector areas are considered, a generic vector U_i has a final size $n_i + nrecv_{\texttt{bef}} + nrecv_{\texttt{aft}}$. Figure 4.3 illustrates a complete generic vector U_i.

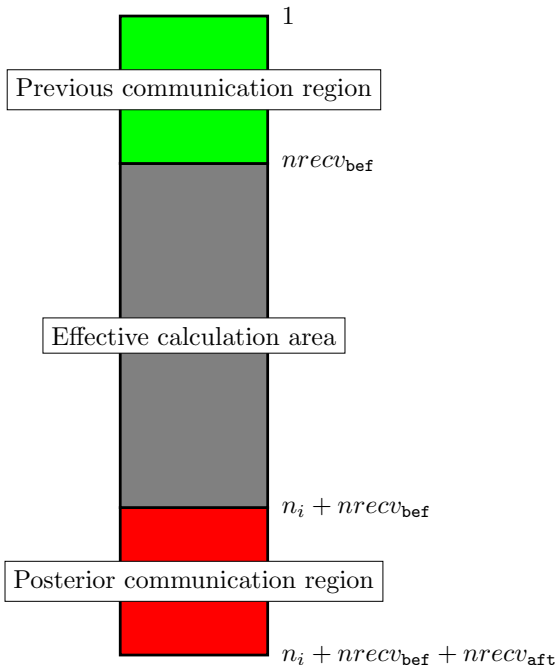


Figure 4.3 – A generic vector U_i.

In practice, a function to perform MPI updates must be developed. The Algorithm 2 describes how to perform MPI updates for any finite element generic vector U_i. That is, whenever synchronizations over a generic vector U_i are necessary, that function is executed. Specific data across effective calculation area of U_i are collected and sent to partitions $i-1$ and $i+1$. In particular, that information is loaded in buffers $SBuff\_bef$ and $SBuff\_aft$

and sent through MPI functions to corresponding partitions where it is unloaded properly in the regions of communications (see Figs. 4.2 and 4.3).

Algorithm 2 MPI Update of a generic vector U_i.

```
for I = 1, nsend_bef do
 |  SBuff_bef[I] ← U_i[IdSend_bef[I]]
end
for I = 1, nsend_aft do
 |  SBuff_aft[I] ← U_i[IdSend_aft[I]]
end
if i = 1 then
 |  MPI_Isend(SBuff_aft, nsend_aft, MPI_DOUBLE, 2)
 |  MPI_Recv(RBuff_aft, nrecv_aft, MPI_DOUBLE, 2)
end
else if i = p then
 |  MPI_Isend(SBuff_bef, nsend_bef, MPI_DOUBLE, p − 1)
 |  MPI_Recv(RBuff_bef, nrecv_bef, MPI_DOUBLE, p − 1)
end
else
 |  MPI_Isend(SBuff_bef, nsend_bef, MPI_DOUBLE, i − 1)
 |  MPI_Isend(SBuff_aft, nsend_aft, MPI_DOUBLE, i + 1)
 |  MPI_Recv(RBuff_bef, nrecv_bef, MPI_DOUBLE, i − 1)
 |  MPI_Recv(RBuff_aft, nrecv_aft, MPI_DOUBLE, i + 1)
end
for I = 1, nrecv_bef do
 |  U_i[IdRecv_bef[I]] ← RBuff_bef[I]
end
for I = 1, nrecv_aft do
 |  U_i[IdRecv_aft[I]] ← RBuff_aft[I]
end
```

4.1.3 Finite element storage

As well known, finite element method produces elementary matrices that need to be stored. Compressed Sparse Row (CSR) (Saad, 2003), Element-by-Element (EBE) (Hughes, 2012), and Edge-by-Edge (EDE) (Coutinho et al., 2001) are examples of ways to provide this issue. Performance to resolve problems discretized by finite elements depends on storage schemes. That happens because storage schemes are associated with the techniques used to calculate matrix-vector products and the linear system preconditioning. This subsection focuses on CSR format since it enables using preconditioners for narrow banded systems. Concluding the subsection, the EBE scheme is adapted to our domain decomposition context.

Figure 4.4 represents a generic resulting finite element narrow banded linear system $AX = F$ – see also Eq. (2.6) – generated by our domain decomposition approach. We assume that the system is divided into p partitions. In general, blocks A_i, B_i, and C_i, with $i = 1, \ldots, p$, store finite element data produced inside each partition. Table 4.2 explains the meaning and attributes of each theoretical parallel data structure used in all partitions. We say "theoretical" because the real form of A_i, B_i, and C_i depends on the chosen storage scheme.

$$AX = \begin{bmatrix} & & & & \\ \ddots & \ddots & \ddots & & \\ & A_i & B_i & C_i & \\ & & \ddots & \ddots & \ddots \\ & & & & \end{bmatrix} \begin{bmatrix} \vdots \\ X_{i-1} \\ X_i \\ X_{i+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ F_{i-1} \\ F_i \\ F_{i+1} \\ \vdots \end{bmatrix} = F$$

Figure 4.4 – Structures of partition i.

Table 4.2 – Structures of partition i.

Structure	Size	Meaning and Attributes
A_i	$n_i \times n_i$	Store finite element data produced inside partition i and related just with partition i.
B_i	$n_i \times nrecv_{\texttt{aft}}$	Store finite element data produced inside partition i but related with partition $i+1$. (There is no block B when $i = p$).
C_i	$n_i \times nrecv_{\texttt{bef}}$	Store finite element data produced inside partition i but related with partition $i-1$. (There is no block C when $i = 1$).
X_i	$nrecv_{\texttt{bef}} + n_i + nrecv_{\texttt{aft}}$	Vector of unknowns of partition i. It has regions of communication with partitions $i-1$ and $i+1$.
F_i	$nrecv_{\texttt{bef}} + n_i + nrecv_{\texttt{aft}}$	Vector of independent terms of partition i. It has regions of communication with partitions $i-1$ and $i+1$.

4.1.4 Parallel CSR storage

Compressed Sparse Row (CSR) (Saad, 2003), described in Subsection 3.2.2.3, stores a finite element matrix A using three global vectors: AA, JA and IA. With regard to parallel computation, CSR concept is applied to blocks A_i, B_i, and C_i (see Table 4.2) where they are replaced respectively by AA_i, JA_i, IA_i, AB_i, JB_i, IB_i, AC_i, JC_i, and IC_i.

A finite element narrow banded linear system divided into 3 partitions is used to illustrate how each parallel CSR structure works. Figure 4.5 shows this example describing the structural vectors JA_i, IA_i, JB_i, IB_i, JC_i, and IC_i. The subscript i is omitted once it only distinguishes between what would be the sequential case and the parallel case. In practice, it can be suppressed since each variable only exists within its partition.

$IBaux$, $IAidex\_aft$, $ICaux$, and $IAidex\_bef$ are auxiliary structures used to allow the parallel CSR storage. $IBaux$ and $ICaux$ maps which rows of blocks B_i and C_i have at least a nonzero coefficient. $IAidex\_aft$ and $IAidex\_bef$, in turn, manage the CSR multiplication of blocks B_i and C_i since these blocks have rows formed exclusively by zeros, a fact which is not predicted in a traditional CSR scheme. In essence, there is no need to visit n_i rows of the block B_i once it has only nB_i rows with nonzeros (nB_i is much smaller than n_i); nor to visit n_i rows of the block C_i once it has only nC_i rows with

nonzeros (analogously, nC_i is much smaller than n_i – see Fig. 4.6). Thus, for a suitable parallel matrix-vector product, structures $IAidex\_aft$ and $IAidex\_bef$ map the effective rows of blocks B_i and C_i in CSR format.

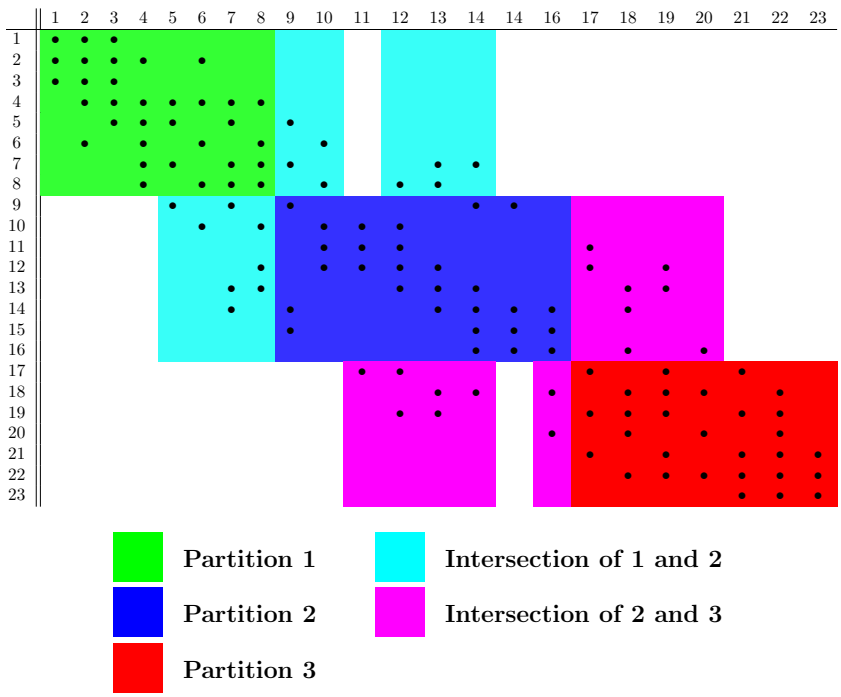
Structures $IBaux$ and $ICaux$ assign the corresponding rows to localize the multiplications performed by the blocks B_i and C_i. Once the values of finite element matrix coefficients are not known yet in the preprocessing phase, the structures AA_i, AB_i, and AC_i are not shown. For now, it is enough to know that AA_i, AB_i, and AC_i have the same size as the respective vectors JA_i, JB_i, and JC_i.

Section B.2 of Appendix B presents a series of algorithms to construct the CSR structural vectors JA_i, IA_i, JB_i, IB_i, JC_i, and IC_i. That algorithms also produce the auxiliary structures $IBaux$, $ICaux$, $IAidex\_aft$ and $IAidex\_bef$.

4.1.4.1 Other structures for the parallel CSR storage

The elementary matrices produced by the finite element method must be stored using a specific storage scheme. When CSR format is used, each element matrix A^e, with size $\texttt{ndof} \cdot \texttt{nnoel} \times \texttt{ndof} \cdot \texttt{nnoel}$, needs to be assembled on the global structures of CSR scheme. However, for general applications of finite element method, especially those involving algorithms of evolution in time or even calculations of nonlinearities, the time to reassemble the matrices is a primordial factor. That is, if an effective strategy is not adopted to regroup the data derived from the finite element method, storage using the CSR format becomes impracticable. Thus, three auxiliary structures are proposed, namely, CSR^e_A, CSR^e_B, and CSR^e_C. Such structures assure the assembly of each matrix A^e in a time complexity of $O(1)$.

Figure 4.7 illustrates how a given finite element matrix A^e from a specific partition i is assembled in parallel CSR structures AA_i, AB_i, and AC_i. In order to increase performance, each matrix A^e is rearranged in a vector format. Thus, all coefficients a^e_j of A^e, with $1 \leqslant j \leqslant (\texttt{ndof} \cdot \texttt{nnoel})^2$ are conveniently distributed between the structures AA_i, AB_i, and AC_i. To avoid an `If-else` conditional expression, each structure has a "scape" area, that is, AA_i, AB_i, and AC_i have their size increased in 1 unit to reach more performance. Suppose coefficients $a^e_{j_1}$, $a^e_{j_2}$, and $a^e_{j_3}$ of a local matrix A^e, as in Fig. 4.7. For example, $a^e_{j_2}$ should be stored in a theoretical block A_i (see Fig. 4.4 and Table 4.2) of the partition i is indeed stored in a CSR structure AA_i (of course, JA_i and IA_i are also used to complete the CSR storage). Thus, aiming to assure high performance, in the finite element preprocessing phase, $a^e_{j_2}$ is mapped in CSR^e_A with a index I_2 (that index points out to the convenient position of structure AA_i, where $a^e_{j_2}$ is added). $a^e_{j_2}$ is also mapped in CSR^e_B and CSR^e_C with indices $nnz_{\texttt{aft}} + 1$ and $nnz_{\texttt{bef}} + 1$ (that indices indicate $a^e_{j_2}$ did not belongs to the theoretical blocks B_i and C_i, according to Fig. 4.4 – indeed theoretical blocks B_i and C_i are replaced by CSR structures AB_i and AC_i). In addition, the coefficient $a^e_{j_2}$ is just



Partition 1																																
JA	1	2	3	1	2	3	4	6	1	2	3	2	3	4	5	6	7	8	3	4	5	7	2	4	6	8	4	5	7	8	4	6	7	8
IA	1	4	9	12	19	23	27	31	35																									
JB	1	2	1	4	5	2	3	4																										
IB	1	2	3	6	9																													
$IBaux$	5	6	7	8																														
$IAidex\_aft$	5	5	5	5	1	2	3	4	5																									
Partition 2																																		
JC	1	3	2	4	4	3	4	3																										
IC	1	3	5	6	8	9																												
$ICaux$	1	2	4	5	6																													
$IAidex\_bef$	1	2	6	3	4	5	6	6	6																									
JA	1	6	7	2	3	4	2	3	4	2	3	4	5	4	5	6	1	5	6	7	8	1	6	7	8	6	7	8						
IA	1	4	7	10	14	17	22	26	29																									
JB	1	1	3	2	3	2	2	4																										
IB	1	2	4	6	7	9																												
$IBaux$	3	4	5	6	8																													
$IAidex\_aft$	6	6	1	2	3	4	6	5	6																									
Partition 3																																		
JC	1	2	3	4	5	2	3	5																										
IC	1	3	6	8	9																													
$ICaux$	1	2	3	4																														
$IAidex\_bef$	1	2	3	4	5	5	5	5																										
JA	1	3	5	2	3	4	6	1	2	3	5	6	2	4	6	8	1	3	5	6	7	2	3	4	5	6	7	5	6	7				
IA	1	4	8	13	17	22	28	31																										

Figure 4.5 – Parallel CSR structures applied to a finite element narrow banded system divided into 3 partitions.

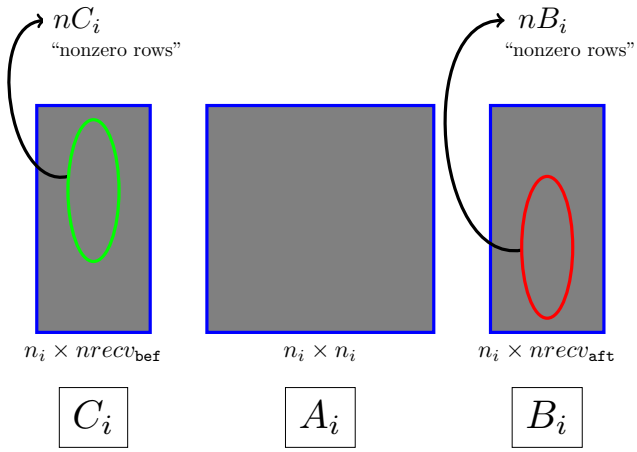


Figure 4.6 – Nonzero rows of blocks B_i and C_i.

summed to "scape" area of AB_i and AC_i. The procedure is analogous if $a^e_{j_1}$ belongs to theoretical block C_i and $a^e_{j_3}$ belongs to theoretical block B_i of the partition i. That is, $a^e_{j_1}$ is summed to AC_i and "scape" area of AA_i and AB_i; $a^e_{j_3}$, in turn, is summed to AB_i and "scape" area of AA_i and AC_i.

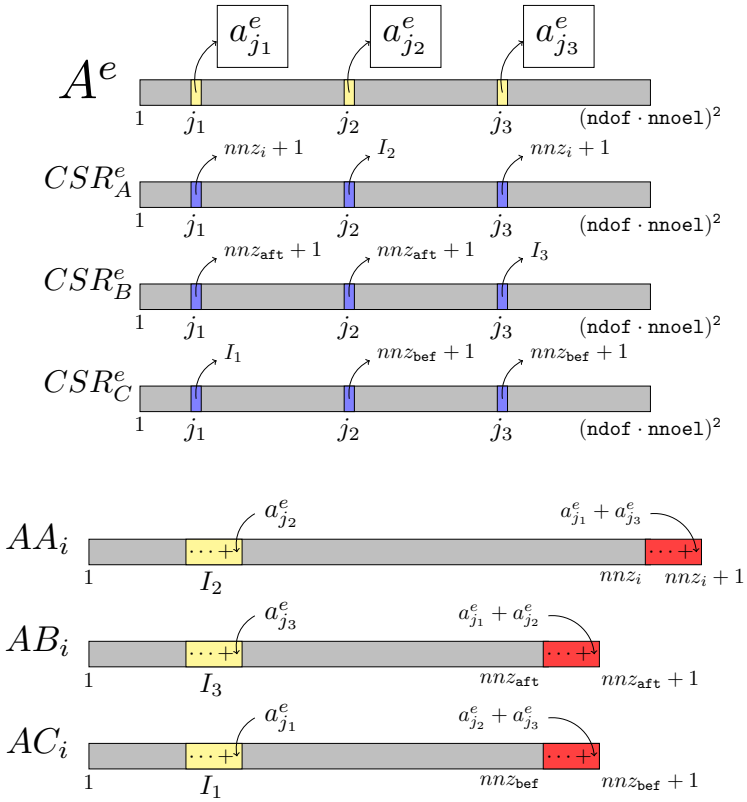


Figure 4.7 – Schematic illustration of the parallel storage of A^e using parallel CSR structures AA_i, AB_i, and AC_i in time complexity $O(1)$.

Section B.3 of the Appendix B describes how to build structures CSR^e_A, CSR^e_B, and CSR^e_C.

4.1.4.2 Reordering of the blocks A_i, B_i, and C_i in CSR format

CSR data structures reordering is the last step of the finite element preprocessing. Our objective here is to improve the preconditioning for blocks what leads to a better global preconditioning. Thus, the blocks A_i, B_i, and C_i from Table 4.2 are reordered as PA_iP^T, PB_i, and PC_i, where P is a permutation matrix. However, for implementations purpose, P is considered as a vector. So, from this point P is assumed as a permutation vector instead of a permutation matrix. Indeed, structures AJ_i, IA_i, and CSR_A^e are reordered with effect over rows and columns of the block A_i and structures JB_i, IB_i, CSR_B^e, AC_i, JC_i, IC_i, and CSR_C^e are reordered with effect just over rows of blocks B_i and C_i.

In Appendix B, Section B.4, there is a set of algorithms to perform a unique a priori reordering that is used to improve the preconditioning during the whole finite element processing phase.

4.1.5 Parallel EBE storage

This section is concluded talking about the EBE storage scheme, introduced in the 1980s by Hughes et al. (1983b). Ever since, many efforts have been employed to simplify the storage process of finite element matrices. This scheme is an alternative to reduce memory consumption. Also, it allows an arbitrary ordering of the elements so that no limitation is imposed by the mesh topology. According to Hughes et al. (1983a), the EBE procedures are independent of bandwidth and thus achieve significant operation count advantages. In consonance with the authors, the advantage increase in nonlinear applications to which frequently refactorizations are necessary.

The traditional EBE format have been adapted to our domain decomposition approach in a straightforward way. Just one structure is considered to store each element matrix A^e of a partition i, that is, A_i^{EBE} (other details about A_i^{EBE}, see Eq. (2.12)). Thus, A_i^{EBE} has size $\texttt{nel}_i \times (\texttt{ndof} \cdot \texttt{nnoel})^2$. As can be noted, each row of A_i^{EBE} is composed by an elementary vector A^e (see Fig. 4.8).

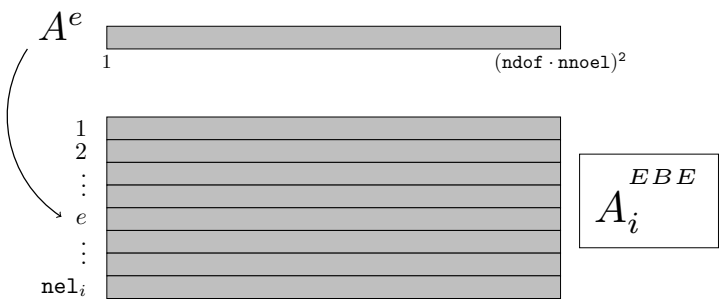


Figure 4.8 – Structure A_i^{EBE}: finite element data storage in EBE format.

REMARK 7: All matrices A^e are stored in an array of vectors instead of an array of matrices. This improves the performance because only 2 indices are used to retrieve the data. That is, accessing a^e_j, with $1 \leqslant e \leqslant \texttt{nel}_i$ and $1 \leqslant j \leqslant (\texttt{ndof} \cdot \texttt{nnoel})^2$ is much faster than accessing $a^e_{j,k}$ with $1 \leqslant j, k \leqslant \texttt{ndof} \cdot \texttt{nnoel}$.

4.2 Parallel finite element processing

In the previous section, there is a description of how to read the mesh data divided into p partitions and to prepare the structures that enable the parallel processing of the finite element application. This section gives a general idea of the finite element data processing (see Fig. 4.9). Two categories of 2D finite element applications are considered: steady-state and transient problems - both considering a nonlinear loop. For steady-state problems, the stiffness matrix K and the load vector F are evaluated at nonlinear iteration. Considering the predictor-multi-corrector (Hughes, 2012) algorithm for transient problems are defined: the mass matrix M, the stiffness matrix K, and the residue vector R.

For each nonlinear iteration of the steady-state or transient problem a parallel preconditioner solver based on the well known Generalized Minimal Residual (GMRES) (Saad and Schultz, 1986) method is considered. For steady-state problems, GMRES solves $KU = F$, in which a point fixed method is used once K and F are U dependent. In the context of the predictor-multi-corrector transient algorithm, the GMRES is used to solve $M^*\Delta dU = R$, where $M^* = M + \alpha\Delta t K$ is an effective matrix and ΔdU is the increment to update U (vector of unknowns) and dU (vector of unknown derivatives) for each time step – Δt is the time-step and α is a time advancing parameter, considered in all experiment as 0.5.

The following two subsections describe how to accommodate the predictor-multi-corrector scheme as well the loop of nonlinear iterations in our parallel process. After, the remaining algorithms are detailed, namely, the parallel preconditioned GMRES, the parallel matrix-vector products CSR and EBE, and, finally, the parallel inner product.

4.2.1 Parallel loop of nonlinear iterations

Algorithm 3 is used to perform the nonlinear loop of steady-state finite element applications. We consider U_i and F_i as local representatives of the global solution vector U and the load vector. Thus, K_i is one abstraction of the finite element matrix K in a partition i. The real format of K_i depends on the storage scheme (CSR or EBE). The superscripts j is the nonlinear iterations counter. The lines highlighted in red indicate points where MPI syncronizations are required. The finite element matrix calculations and assemblies as well as the GMRES solutions are performed according to the number of nonlinear iterations. For each nonlinear iteration, finite element matrices are calculated (see Fig. 4.7 for CSR and Fig. 4.8 for EBE storage format), preconditioners are set up (see

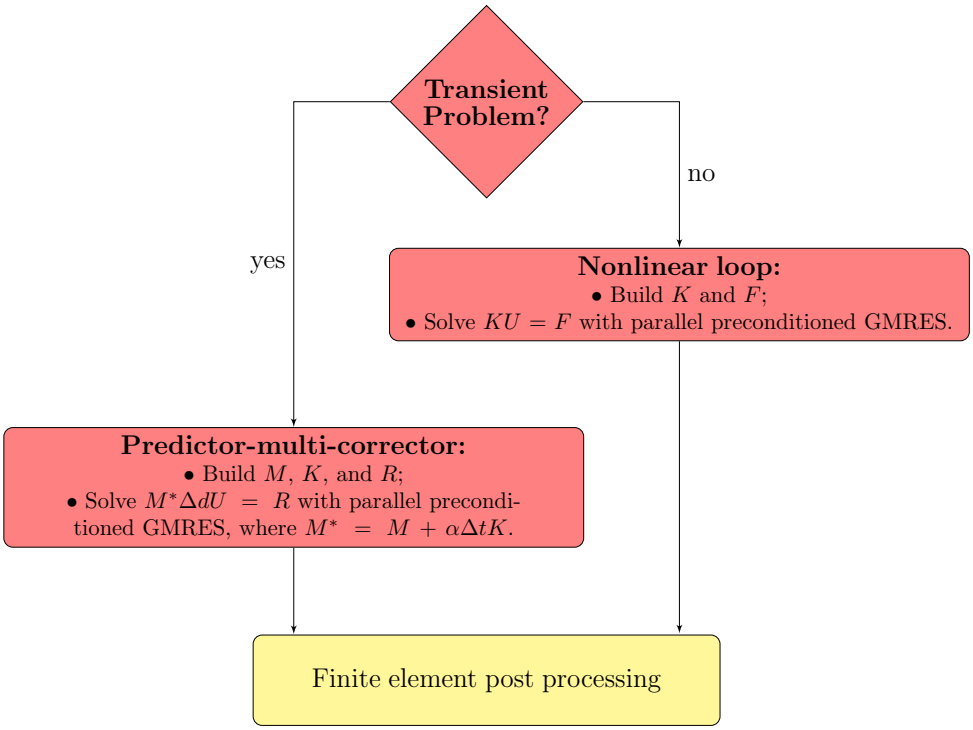


Figure 4.9 – Workflow of the Finite element data processing: transient or steady-state problems.

Section 2.5), and a GMRES solution is required. Also, scaling operation is performed just for LU and Gauss-Seidel local preconditioners (see Subsections 2.5.3.2 and 2.5.3.3).

Algorithm 3 Parallel loop of nonlinear iterations.

$j \leftarrow 0$
do
  MPI Update of U_i^j.
  Calculation and assembly of matrix K_i and vector F_i^j from each elementary A^e.
  Scaling of matrix K_i^* and vector F_i^j.
  Setup of the preconditioner.
  Solve $K_i U_i^{j+1} = F_i^j$ using the **parallel preconditioned GMRES**.
  Apply the **parallel inner product** to obtain the norm $|U_i^j|$.
  Apply the **parallel inner product** to obtain the norm $|U_i^{j+1}|$.
  $j \leftarrow j + 1$
while $(j < j_{MAX}\ .AND.\ \frac{|U_i^j|}{|U_i^{j+1}|} > tol)$

4.2.2 Parallel predictor-multi-corrector

The Algorithm 4 is a version of the implicit predictor-multi-corrector (Hughes, 2012) algorithm adapted to our parallel approach. U_i and dU_i are generic finite element representative vectors (see Fig. 4.3) of the global solution vector U and the global vector of derivatives dU in a partition i and R_i is the residue vector for the partition i. Matrix M^* has its representative in a partition i, that is, M_i^* (this local matrix can be stored in CSR or EBE format). The superscripts $n+1$ and n mean, respectively, the time solution on indices

$n+1$ and n; the superscript j, in turn, is the multi-corrector counter. Lines of function MPI Update (see Subsection 4.1.2.1), parallel inner products (see Subsection 4.2.6), and parallel preconditioned GMRES (see Subsection 4.2.3) are highlighted in red. These are points where the MPI syncronizations occur. There is a main loop for providing the time evolution and for each time step there is an inner loop for multi-corrections. For each multi-correction, finite element matrices are calculated and assembled (see Fig. 4.7 for CSR and Fig. 4.8 for EBE storage format), preconditioners are set up (see Section 2.5), and a GMRES solution is required. Besides, scaling operation is performed just for LU and Gauss-Seidel local preconditioners (see Subsections 2.5.3.2 and 2.5.3.3).

Algorithm 4 Parallel predictor-multi-corrector.

do
  Predictor phase:
  $j \leftarrow 0$
  $U_i^{n+1,0} \leftarrow U_i^n + (1-\alpha)\Delta t dU_i^n$
  $dU_i^{n+1,0} \leftarrow 0$
  Multi-corrector phase:
  do
    MPI Update of $U_i^{n,j}$.
    MPI Update of $dU_i^{n,j}$.
    Calculation and assembly of matrix M_i^* and vector $R_i^{n,j}$ from each elementary A^e.
    Scaling of matrix M_i^* and vector $R_i^{n,j}$
    Setup of the preconditioner.
    Solve $M_i^* \Delta dU_i = R_i^{n,j}$ using the **parallel preconditioned GMRES**.
    $U_i^{n,j+1} \leftarrow U_i^{n,j} + \alpha \Delta t \Delta dU_i$.
    $dU_i^{n,j+1} \leftarrow dU_i^{n,j} + \Delta dU_i$.
    Apply the **parallel inner product** to obtain the norm $|dU_i^{n,j}|$.
    Apply the **parallel inner product** to obtain the norm $|\Delta dU_i|$.
    $j \leftarrow j+1$
  while $(j < j_{MAX}$ $.AND.$ $\frac{|\Delta dU|}{|dU|} > tol)$
  $t_{n+1} \leftarrow t_n + \Delta t$
while $(t_n \leqslant t_{final})$

4.2.3 Parallel preconditioned GMRES

Algorithm 5 presents the parallel preconditioned GMRES which has been developed to solve a narrow banded linear system $AX = F$. Suppose all coefficients of the matrix A and of the vector F are stored in p partitions according to CSR or EBE schemes (see Figs.4.7 and 4.8). We also assume that subscript i means the structure belongs to partition i. MPI syncronizations are highlighted in red and the preconditioning is highlighted in blue considering right and to left computations. Specifically, only Guass-Seidel preconditioners consider right computations (see Subsection 2.5.3.3). The parallel matrix-vector product is executed by a specific algorithm, according to the storage scheme chosen. That is, Algorithm 6 performs CSR format and Algorithm 7 performs EBE format.

Algorithm 5 Parallel preconditioned GMRES.

Apply the **parallel inner product** to obtain the norm $|F_i|$.
do
  $j \leftarrow 1$.
  Right Precondition X_i according to the setup of Algorithm 4 or 3.
  Apply the **parallel matrix-vector product** $U_i^j \leftarrow A_i * X_i$.
  Left Precondition U_i^j according to the setup of Algorithm 4 or 3.
  $U_i^j \leftarrow F_i - U_i^j$.
  Apply the **parallel inner product** in U_i^j to obtain e_j.
  $U_i^j \leftarrow \frac{1}{e_j} * U_i^j$.
  $\rho \leftarrow \sqrt{e_j}$.
  do
    Right Precondition U_i^j according to the setup of Algorithm 4 or 3.
    Apply the **parallel matrix-vector product** $U_i^{j+1} \leftarrow A_i * U_i^j$.
    Left Precondition U_i^{j+1} according to the setup of Algorithm 4 or 3.
    Gram-Schmidt orthogonalization:
    for $k = 1, j+1$ **do**
      Apply the **parallel inner product** between U_i^k and U_i^{j+1} to obtain h_{kj}.
      $U_i^{j+1} \leftarrow U_i^{j+1} - h_{kj} * U_i^{j+1}$
    end
    Apply the **parallel inner product** to obtain $h_{j+1,j}$.
    $U_i^{j+1} \leftarrow \frac{1}{\sqrt{h_{j+1,j}}} * U_i^{j+1}$
    QR Algorithm:
    for $k = 1, j-1$ **do**
      $aux_1 \leftarrow c_k * h_{kj} + s_k * h_{k+1,j}$
      $aux_2 \leftarrow -s_k * h_{kj} + c_k * h_{k+1,j}$
      $h_{kj} \leftarrow aux_1$
      $h_{k+1,j} \leftarrow aux_2$
    end
    $r \leftarrow \sqrt{h_{jj}^2 + h_{j+1,i}^2}$
    $c_j \leftarrow h_{jj}/r$
    $s_j \leftarrow h_{j+1,j}/r$; $h_{jj} \leftarrow r$; $h_{j+1,j} \leftarrow 0$
    $j \leftarrow j+1$
  while $(\frac{\rho}{|F_i|} > tol\ .AND.\ k < k_{MAX})$
  $j \leftarrow j-1$
  Solve the local triangular linear system $hy = e$
  for $k = 1, j$ **do**
    $X_i \leftarrow X_i + U_i^k * y_k$
  end
  $l \leftarrow l+1$
while $(\frac{\rho}{|F_i|} > tol\ .AND.\ l < l_{MAX})$

4.2.4 Parallel CSR matrix-vector product

The Algorithm 6 describes how to perform the parallel CSR matrix-vector product. That is developed taking into account blocks A_i, B_i, and C_i, described in Table 4.2, which are stored in CSR structures AA_i, AB_i, AC_i (see Fig. 4.7) and mapped by the structures JA_i, IA_i, JB_i IB_i, $IBAux$, JC_i, IC_i and $ICaux$ (example in Fig. 4.5). Each call of CSR matrix-vector product demands a MPI update.

REMARK 8: Each vector of a partition i is admitted as a generic finite element vector (see Fig. 4.3). Since a generic vector has 3 regions: previous communication region (with size $nrecv_{\texttt{bef}}$), effective calculation area (with size n_i), and posterior communication region (with size $nrecv_{\texttt{bef}}$); theses vectors are allocated with total size $n_i + nrecv_{\texttt{bef}} + nrecv_{\texttt{aft}}$. However,

Algorithm 6 CSR matrix-vector product.

```
MPI update of U_i.
Z_i ← 0
for I = 1, nB_i do
    for J = IB_i[I], IB_i[I + 1] do
        Z_i[IBaux[I] + nrecv_bef] ← Z_i[IBaux[i] + nrecv_bef] + AB_i[j] * U_i[JB_i[J]]
    end
end
for I = 1, nC_i do
    for J = IC_i[I], IC_i[I + 1] do
        Z_i[ICaux[I] + nrecv_bef] ← Z_i[ICaux[I] + nrecv_bef] + AC_i[J] * U_i[JC_i[J] + n_i + nrecv_bef]
    end
end
for I = 1, n_i do
    for J = IA_i[I], IA_i[I + 1] do
        Z_i[I + nrecv_bef] ← Z_i[I + nrecv_bef] + AA_i[J] * U_i[JA_i[J] + nrecv_bef]
    end
end
```

the ranges $[1, nrecv_{\texttt{bef}}]$ and $[n_i + nrecv_{\texttt{bef}} + 1, n_i + nrecv_{\texttt{bef}} + nrecv_{\texttt{aft}}]$ are used only for MPI updates. The effective calculations occur inside the range $[nrecv_{\texttt{bef}} + 1, n_i + nrecv_{\texttt{bef}}]$ whose size is n_i.

4.2.5 Parallel EBE matrix-vector product

The EBE matrix-vector product, describe in Algorithm 7, is quite simple to implement. Thanks to the particular construction of the LM structure (see Algorithm 18), parallel EBE version is very similar to the sequential algorithm. As reported in Subsection 4.1, finite element matrix data is represented trivially in EBE format when the structure A_i^{EBE} (see Fig. 4.8) is considered. Each call of EBE matrix-vector product also demands a MPI update.

Algorithm 7 EBE matrix-vector product.

```
MPI update of U_i.
U_i[n_i + nrecv_bef + nrecv_aft + 1] ← 0
for I = 1, nel_i do
    count ← 1
    for J = 1, nnoel * ndof do
        L_J ← LM[I][J]
        Z_i[L_J] ← 0
        for K = 1, nnoel * ndof do
            L_K ← LM[I][K]
            Z_i[L_J] ← Z_i[L_J] + A_i^EBE [I][count] * U_i[L_K]
            count ← count + 1
        end
    end
end
```

REMARK 9: The EBE matrix-vector product can be implemented taking into account that, generally `nnoel` and `ndof`, are constant values. Thus, an EBE version considering one loop instead three is much faster. Whenever possible, it is better to unroll the loop.

4.2.6 Parallel inner product

The parallel inner product of two generic finite element vectors X_i and Y_i (see Fig. 4.3) is easily implemented in Algorithm 8, once the effective calculation areas of X_i and Y_i can be accessed directly. Thus, sequential and parallel versions are almost identical. The parallel inner product needs to finalize with a $MPI\_Allreduce$ operation, that demands a considerable time to be performed. In our implementation it is required just in this operations.

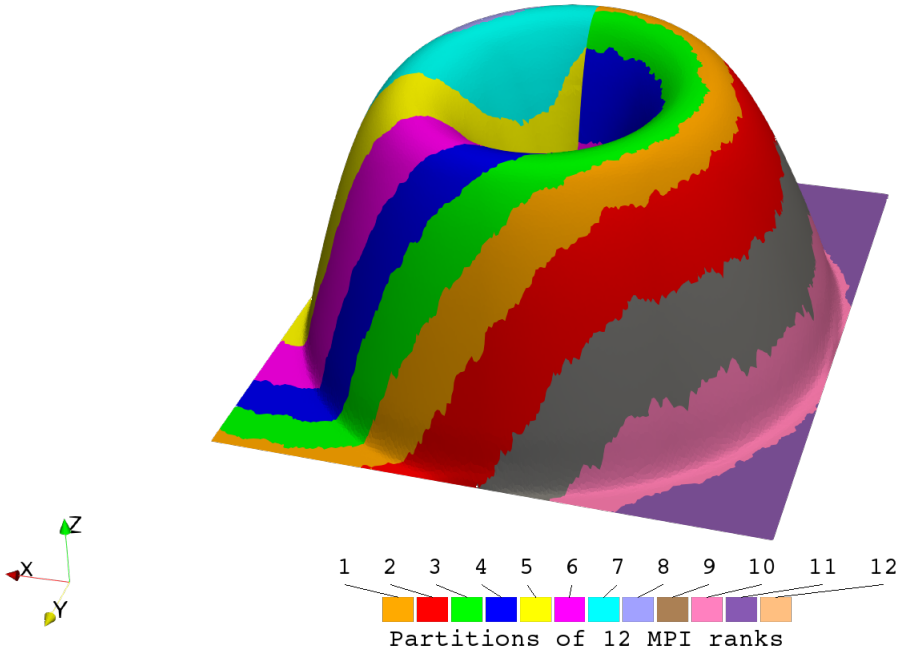
Algorithm 8 Parallel inner product of X_i and Y_i.

$sum \leftarrow 0$
for $I = nrecv_{\texttt{bef}} + 1, nrecv_{\texttt{bef}} + n_i$ **do**
  $sum \leftarrow sum + X_i[I] * Y_i[I]$
end
$MPI\_Allreduce(sum, GSUM, 1, MPI\_DOUBLE, MPI\_SUM)$
return $GSUM$

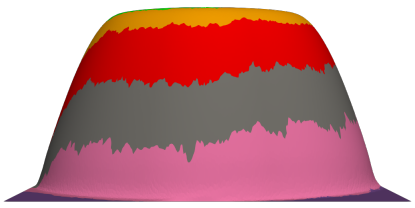
4.3 Finite element postprocessing

As discussed in Section 3 and Subsection 4.1, all efforts have been made to produce an *a priori* narrow banded linear system arising from finite element discretization. Such system keeps the same sparsity pattern since its creation until the postprocessing. That is, the unknowns receive labels that do not change. So, the postprocessing is a phase just to print the final results using Paraview[1] or a similar tool. Figure 4.10 shows an illustration of the parallel solution of a finite element analysis application divided into 12 MPI ranks (see Problem 1 in Chapter 5) using the Paraview plataform. The partition seems odd, but it is important to remember that the decomposition was defined by the linear system and not by the mesh.

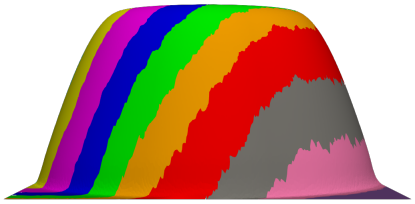
[1] <http://www.paraview.org> – The parallel multi-platform data analysis and visualization application Paraview is used to visualize the final results.



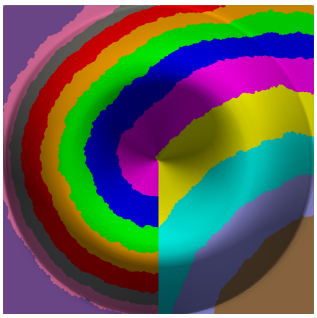
(a) View of the parallel solution



(b) yz-plane view perspective



(c) xz-plane view perspective



(d) xy-plane view perspective

Figure 4.10 – Paraview visualization: Illustration of a parallel solution with 12 MPI ranks.

5 Numerical Experiments

In this chapter, the robustness and stability of our approach considering the parallel preconditioning are demonstrated. Four finite element problems modeled by transport and Euler equations are examined: a steady-state problem with 1 degree of freedom per node (`ndof` = 1); a transient problem with 1 degree of freedom per node (`ndof` = `1`); and two other transient problems with 4 degrees of freedom per node (`ndof` = `4`). The well-known numerical stabilization Streamline Upwind Petrov-Galerkin (SUPG) (Brooks and Hughes, 1982), coupled with the $YZ\beta$ shock-capturing operator (Tezduyar and Senga, 2006) for compressible Euler equations and adapted to transport equation according to Bazilevs et al. (2007) is adopted to obtain accurate solutions.

Our approach is validated using local and global preconditioners. Local preconditioning is performed taking into account Element-by-Element (EBE) storage. Six local preconditioners are evaluated:

- DIAGe: a diagonal preconditioner used for problems with 1 degree of freedom per node (see Subsection 2.5.3.1);
- LUe: a local LU factorization preconditioner used for problems with 1 degree of freedom per node (see Subsection 2.5.3.2);
- SGSe: a local Gauss-Seidel preconditioner used for problems with 1 degree of freedom per node (see Subsection 2.5.3.3);
- BlockDIAGe: a block diagonal preconditioner used for problems with more than one degree of freedom per node (see Subsection 2.5.3.1);
- BlockLUe: a local LU factorization preconditioner used for problems with more than one degree of freedom per node (see Subsection 2.5.3.2);
- BlockSGSe: a local Gauss-Seidel preconditioner used for problems with more than one degree of freedom per node (see Subsection 2.5.3.3);

Global preconditioners, in turn, are based on Compressed Sparse Row (CSR) storage. Theoretically, they can be applied to problems with any number of degrees of freedom per node. Two global preconditioners are evaluated:

- ILUm: Incomplete LU factorization, where p represents the *fill-in* level (see Subsection 2.5.2);

- SPIKEk_t: a truncated SPIKE version which works together with PARDISO, where k means the size of the coupling blocks and t means the amount of OpenMP threads (see Section 2.4);

The parallel preconditioned GMRES, as described in Algorithm 5, is used to solve the arising linear systems from finite element discretization. The GMRES solver tolerance $\varepsilon_{GMRES} = 10^{-8}$ is considered for nonlinear problems and $\varepsilon_{GMRES} = 10^{-6}$ for transient problems. In general, the number of nonlinear iterations is small for nonlinear and transient applications for the experiments considered in this work. The GMRES solver tolerance for the nonlinear problems is chosen smaller because the GMRES is resolved few times - about 2 times. For the transient problems, the GMRES is resolved hundreds of times, once it is required at nonlinear iterations of each time step. In all cases, Krylov's base is set with 30 vectors. The nonlinear tolerance is $\varepsilon_{nonlinear} = 10^{-3}$ for steady-state problems, whereas for transient problems are considered 3 fixed nonlinear iterations for each time step. Next two sections discuss the Software and Hardware used, and metrics to measure the experiments' performance. The following four sections present the evaluation of the preconditioners' behavior for the 2D finite element benchmark problems in the context of our domain decomposition approach. The last three sections organize considerations about the preconditioners tested, memory usage, load balance, MPI communications, and our own domain decomposition approach.

5.1 Software and Hardware

5.1.1 Software used in the project development

Our codes have been developed in C language and compiled with Intel compiler version 2017.5.239. The parallel environment takes into account the same Intel version of the Message Passing Interface (MPI) protocol. Two optimization flags are also used, named, -Ofast and -march=native. According to icc manual, -Ofast sets compiler options -O3, -no-prec-div, and -fp-model fast=2. Flag -march=native, in turn, causes the compiler to auto-detect the architecture of the build computer. Unfortunately, according to gcc manual, this feature is only supported on GNU/Linux, and possibly was ignored by Intel compiler.

5.1.2 Other software and libraries used

Few adaptations were proposed in the routine ILUK from ITSOL[1] library (developed by Yousef Saad's team) to perform ILUm preconditioners. The algebraic factorization is an ITSOL library function that predicts *fill-in* positions. It is executed just once,

[1] <https://www-users.cs.umn.edu/~saad/software/ITSOL/>

because all transformations in the matrix sparsity pattern occur in the finite element preprocessing phase (see Subsection 4.1). The most common vector routines `ddot`, `daxpy`, `dscal`, `dcopy`, `dzero` were implemented based on codes of BLAS[2] level1 version 3.8.0. All SPIKE preconditioner steps involving complete LU factorizations and using direct methods were solved through the library pardiso500-INTEL1301-X86-64 of PARDISO[3] software.

Aiming to demonstrate the effectiveness and generality of our domain decomposition approach, only unstructured meshes with nontrivial sparsity patterns are used. Every problem is discretized by unstructured triangular meshes, using Delaunay triangulation through the software Gmsh (Geuzaine and Remacle, 2009). Thus, algorithms for reducing the bandwidth generated by such sparsity pattern are needed. For this task, two algorithms are chosen: the reverse Cuthill- McKee(RCM) (Liu and Sherman, 1976), for which a C language version of the function `symrcm` from the Octave 4.0 package was adopted; Spectral (Barnard et al., 1995), which incorporates the Fortran routines of the hsl[4] library.

5.1.3 Cluster Loboc

All tests were performed on the Cluster Lobo Carneiro (Loboc)[5] from Núcleo Avançado de Computação de Alto Desempenho (NACAD), Federal University of Rio de Janeiro (UFRJ), Brazil. Loboc has 504 CPUs Intel Xeon E5-2670v3 (Haswell), totalizing 6048 cores. It has 252 processing nodes, and each node has 64GB of RAM and 24 cores (48 with Hyper-Threading). The cluster's network is an Infiniband FDR - 56 Gs (Hypercube) and the operating systems is the Suse Linux Enterprise (SLE).

5.2 Performance Metrics

A series of measurements are made for each experiment. Initially, tables with the number of GMRES iterations according to each preconditioner and CPU time to solve every situation are presented. The total runtime includes: the finite element preprocessing (Subsection 4.1), the finite element processing (Subsection 4.2) and, the postprocessing (Subsection 4.3). CPU time, speedup, efficiency, total memory usage, and memory usage per processing node, are also shown. Finally, a tool named TAU[6] (Tuning Analysis Utilities) is used to investigate the proportional CPU time spent by each function for different preconditioners. TAU also provides analysis of the Message Passing Interface (MPI) operations.

[2] <http://www.netlib.org/blas/#_level_1>
[3] <http://www.pardiso-project.org/>
[4] <http://www.hsl.rl.ac.uk>
[5] <http://portal.nacad.ufrj.br/recurso-icex.html>
[6] <https://www.cs.uoregon.edu/research/tau/home.php>

5.2.1 Runtime: CPU time, speedup, and efficiency

The CPU time of each experiment is analyzed using the following criteria: each experiment is performed five times. The lowest and the longest CPU times are eliminated, and the average of the remaining three intermediate CPU times is taken as the reference value. Two other metrics are also employed: speedup and efficiency. A processing node always uses its full capacity, then partitions are divided into multiples of 24, as enunciated in Subsection 5.1.3. Thus, both metrics are redefined proportionally to 24 MPI ranks:

- $Speedup(n) = \dfrac{\text{CPU time of 24 MPI ranks}}{\text{CPU time of } n \text{ MPI ranks}}$
- $Efficiency(n) = 100\dfrac{24 \cdot Speedup(n)}{n}$

REMARK 10: Because that proportionality to 24 MPI ranks, the ideal speedup will be represented by a line passing through the points (24,1), (48,2), (72,3), (96,4), (192,8) and (384,16).

REMARK 11: When we apply our domain decomposition approach, all sequential portions of the experiments are executed a priori – such parts are analyzed in the beginning of each section, in the tables entitled "Domain Decomposition Approach - Bandwidth, Preprocessing CPU time, and Memory Usage". The other parts of the experiments are strictly parallel and are investigated in details for each numerical example. Thus, our speedup and efficiency metrics do not consider the limitations imposed by the Amdahl (Amdahl, 1967) and Gustafson's (Gustafson, 1988) laws. As result, the experiments can present superlinear speedups (Gustafson, 1990) and efficiencies above 100%.

REMARK 12: In the context of high performance computing, the efficiency metric can be evaluated using two common notions of scalability: strong and weak scaling (Khare et al., 2012). Strong scaling spread the same size problem across more nodes. Weak scaling keeps the problem-per-node size constant while increasing the number of nodes. Our efficiency metric considers only the strong scaling.

5.2.2 Memory usage

Memory usage of each preconditioner is evaluated. We use the Valgrind[7] package in combination with the tool named Massif[8] with that purpose. Each experiment is executed by appending to each MPI call a statement similar to `valgrind - -tool = massif`. As a

[7] <http://valgrind.org/>

[8] Massif is a heap profiler. It performs detailed heap profiling by taking regular snapshots of a program's heap. It produces a graph showing heap usage over time, including information about which parts of the program are responsible for the most memory allocations.<http://valgrind.org/info/tools.html#massif>

result, files `massif.out` are generated according to the number of MPI ranks. The variables `mem_heap_B` and `mem_heap_extra_B` arising from files `massif.out` store the amount of memory used at each instant. Thus, it is enough to take in each MPI rank the maximum value of `mem_heap_B`, say `max_mem_heap_B`, and add it to the corresponding value of `mem_heap_extra_B`, say `used_mem_heap_extra_B`, to get the peak memory usage in that rank. These values allow to define the two following metrics:

- **Total memory usage:** memory usage in each MPI rank is calculated as

  $$\texttt{max\_mem\_heap\_B} + \texttt{used\_mem\_heap\_extra\_B}.$$

  All the memory usage of the ranks are added and the total memory usage is obtained.

- **Average memory usage per processing node:** in order to obtain the average memory usage in the processing nodes, it is sufficient for each case to divide the total memory usage by 24, 48, 72, 96, 192, or 384 to find the average memory usage in 1, 2, 3, 4, 8, or 16 processing nodes, respectively.

Load balancing can also be measured by checking the variation of the amount of memory used in each MPI rank. If the difference between the maximum and minimum value of memory usage in each rank is relatively small, this indicates domain decomposition has been successful. Thus, for each experiment, we measure the load balancing measured of a local preconditioner and a global preconditioner.

5.2.3 Functions runtime analysis

TAU is a tool that allows evaluating the runtime of each function of a serial or parallel program. In applications involving MPI, each rank produces a log file containing information about all functions executed. This tool also allows analyzing the average runtime of the functions. From the implementation point of view, there are several functions whose runtime could be measured. We evaluate the runtime of the same ten functions or groups of functions in order to systematically analyze the code. Runtimes of the other functions are assigned to a group named Others. The following is the list of functions whose runtime is analyzed:

- **Build_Matrices**: responsible of building the matrix of the coefficients as well as the vector of independent terms of the linear systems derived from the finite element method. For the steady-state problem, it builds the matrix K and the vector F. For transient problems, it builds matrices M, K and the load vector R. Besides, the effective matrix M^* is also calculated. For details, see Fig. 4.9;

- **Matrix_Vector_Product**: process each matrix-vector product according to the chosen storage method. For details, see Subsections 4.2.4 and 4.2.5;

- **Preconditioner**: responsible for applying the preconditioning. Therefore it is applied soon after each matrix-vector product. For details, see Algorithm 5;
- **Preconditioner_Setup**: prepare the structures to be used in the preconditioning step. As an example, complete or incomplete factorizations of finite element matrices can be cited. For details, see Algorithms 4 and 3;
- **Scaling**: this operation is only performed in cases involving the local preconditioners LUe, SGSe, BlockLUe and BlockSGSe. For details, see Subsection 2.5.3.
- **Vectors**: this operation is formed by the set of all most common vector operations, namely ddot, daxpy, dscal, dcopy, dzero, as implemented in BLAS level 1.
- **MPI_Allreduce**: native MPI function to get the sum of all MPI ranks' partial internal products. For details see the Algorithm 8;
- **MPI_Barrier**: native MPI function to promote the synchronization of MPI ranks. It was applied to the input of data and more effectively in the preconditioners LUe, SGSe, BlockLUe and BlockSGSe.
- **MPI_Isend**: native MPI function to send the requested data to a designated MPI rank. The prefix "I" in Isend indicates the sending is asynchronous. This function was used within the context of the update function presented in the Algorithm 2.
- **MPI_Recv**: native MPI function to receive the requested data from a designated MPI rank. This function was used within the context of the update function presented in the Algorithm 2.
- **Others**: represents the remaining functions.

5.3 Sine hill in a rotating fluid flow field - Problem 1 : Steady-State case with `ndof=1`

Our first experiment is a steady-state problem with 1 degree of freedom per node. The exact solution consists in a pure advection of a sine hill in a rotating fluid flow field as presented in (Brooks and Hughes, 1982). Figure 5.1 shows the problem statement, modeled by the steady-state advection-diffusion equation. The computational domain is a square domain, $\Omega = [-10, 10] \times [-10, 10]$, the constant diffusivity is $\epsilon = 10^{-8}$, the source term is $f = 0$, and the velocity field is $\beta = (-y, x)^T$. Such values characterize an advection-dominated problem so that the condition on OA is almost purely advected along circular streamlines. Because of the shock capture term $YZ\beta$ (Bazilevs et al., 2007), this problem becomes a nonlinear example. Thus, there is a parallel loop of nonlinear iterations which is solved using the Algorithm 3.

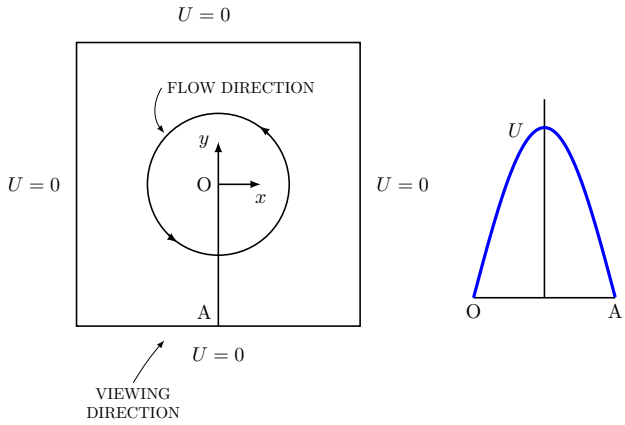


Figure 5.1 – Advection in a rotating fluid flow field: Problem statement.

Three sizes of unstructured mesh are considered, namely $SmallMesh$ with 2,356,673 nodes and 4,708,008 elements; $MediumMesh$ with 5,290,192 nodes and 10,572,382 elements; and $LargeMesh$ with 10,794,301 nodes and 21,577,168 elements. Table 5.1 shows the CPU time, bandwidth sizes, and memory usage related to the domain decomposition approach applied to all meshes reordered by RCM and Spectral algorithms. The CPU time to obtain the domain decomposition using Spectral algorithm was estimated because the bandwidth size was not enough to divide the linear system into 192 partitions. The estimated domain decomposition time using the Spectral algorithm was considered to be the sum of the reordering time using the Spectral algorithm and the time of the domain decomposition obtained by the RCM algorithm minus the reordering runtime, once the difference between both decompositions is just the reordering algorithm. As expected, as the mesh grows, CPU time increases considerably and the CPU time is directly related to the bandwidth.

Table 5.1 – Problem 1 : Domain Decomposition Approach - Bandwidth, Preprocessing CPU time, and Memory Usage.

	Bandwidth before reordering	Bandwidth after reordering	Time for Reordering (in seconds)	Total time to preprocess the domain decomposition into 192 partitions (in seconds)	Memory Usage (in MB)
$SmallMesh$ (RCM)	2,350,225	2,606	6.929	63.49	1461
$MediumMesh$ (RCM)	5,278,451	3,869	17.17	139.13	3217
$LargeMesh$ (RCM)	10,779,780	5,527	39.06	292.40	6843
$LargeMesh$ (Spectral)	10,779,780	120,127	89.29	342.63*	6967

*estimated CPU time: Spectral failed to split into 192 partitions.

Figure 5.2 presents the Problem 1 solution for $SmallMesh$ obtained with the preconditioner ILU2 and 24 MPI ranks. The graphic is plotted using Paraview software after the postprocessing phase (see Section 4.3).

In this experiment, nine preconditioning cases are evaluated. Three of them are local

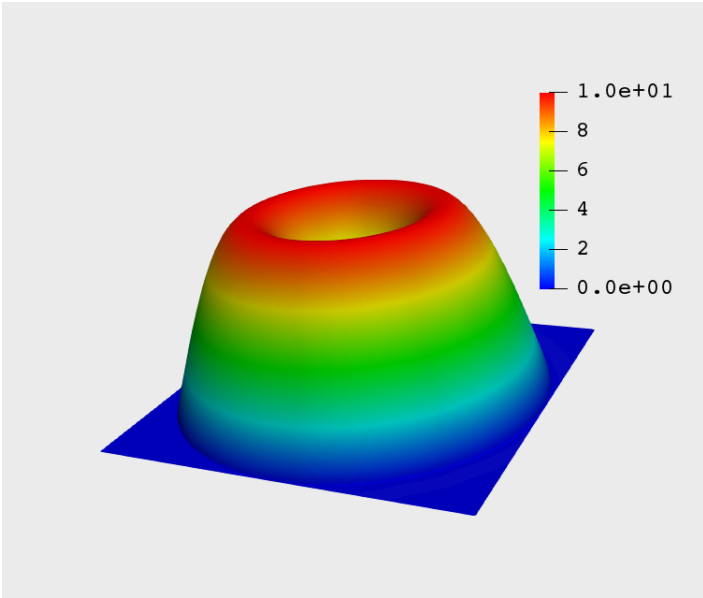


Figure 5.2 – Problem 1 : Solution for $SmallMesh$ with the preconditioner ILU2 and 24 MPI ranks.

preconditioners based on EBE storage, namely DIAGe, LUe, and SGSe. The remaining cases are global preconditioners based on CSR storage taken from incomplete LU factorizations (ILUm) and SPIKE preconditioners. ILUm preconditioners assumed *fill-in* level m as 2, 3, and 10, that is, preconditioners ILU2, ILU3, and ILU10. (For Problem 1, ILU0 and ILU1 did not converge – as reported by Benzi (2002), for many problems arising from Computational Fluid Dynamic, such as cases of highly nonsymmetric and indefinite matrices, there is the necessity of a level of *fill-in* greater than zero). SPIKE preconditioners have two parameters: the coupling blocks size k and the number of threads per MPI rank t. Empirically, we fix $k = 50$ in this work. It is a reasonable size for the coupling blocks, taking into account the number of floating point operations, as reported by Lima et al. (2017). An amount of 2, 4, and 8 threads per MPI rank is used for each SPIKE preconditioner. That cases are namely SPIKE50$_2$, SPIKE50$_4$, and SPIKE50$_8$. Problem 1 is a nonlinear steady-state problem, however, all solutions needed just two nonlinear steps to achieve the tolerance.

In this section, the preconditioners' performance is also analyzed when the sparsity of the mesh is reordered by RCM and Spectral algorithms. As mentioned earlier, partitions are divided into multiples of 24 precisely, so that a processing node always uses its full capacity. Thus, nine sets of numerical experiments are performed considering 24, 48, 72, 96, and 192 partitions. In the particular case of SPIKE, the grouping of MPI ranks per node was somewhat different. Because the architecture allows 48 hyper-threads per processing node, SPIKE50$_2$ is grouped in 24 MPI ranks per node, SPIKE50$_4$ is grouped with 12 MPI ranks per node, and finally SPIKE50$_8$ is grouped with 6 MPI ranks per node. Thus, compared with SPIKE50$_2$, SPIKE50$_4$ uses the double number of nodes and SPIKE50$_8$

uses the quadruple number of nodes for each simulation.

Tables 5.2, 5.3, and 5.4 present, the total CPU time (in seconds) and the number of GMRES iterations (iter) for all preconditioners, respectively to $SmallMesh$, $MediumMesh$, and $LargeMesh$. For all cases we consider 24, 48, 72, 96, 192 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Table 5.5, in turn, presents the same information for $LargeMesh$, but Spectral algorithm reorders the sparsity of the mesh. In this case, particularly, experiments with 192 MPI ranks are not performed, because the reduction of the bandwidth size was not enough to divide matrix order into 192 partitions (see Table 5.1). Note that, once the mesh size is fixed, the preconditioner DIAGe yields a nearly constant number of iterations. Such behavior is expected since this preconditioner is just a scaling based on the main diagonal of the matrix (see Subsection 2.5.3.1). Local preconditioners LUe and SGSe present a number of iterations much smaller than that reached with the preconditioner DIAGe. However, these preconditioners suffer a relative increase in the number of iterations when the number of processors increases. The preconditioner LUe becomes less attractive than SGSe as the mesh size increases. This behavior is more significant when the mesh reordering is obtained by Spectral algorithm.

All global preconditioners present a considerable increase in the number of iterations as the number of MPI ranks increases. That behavior is also expected because these preconditioners are applied to smaller and smaller blocks A_i (Fig. 4.4 defines blocks A_i with i=1,2,..., the number of MPI ranks). For higher indices of *fill-ins*, greater is the increase of the number of iterations as the number of MPI ranks grows. Initially, ILU3 presented the best results. However, this configuration changes when the mesh and the number of MPI ranks increase. For 192 MPI ranks, ILU2 becomes the best option for all meshes.

The number of threads does not change the number of iterations for the SPIKE preconditioners, that is, SPIKE50$_2$, SPIKE50$_4$, and SPIKE50$_8$ have an identical number of iterations for the same number of MPI ranks. However, runtime decreases, since matrix factorizations and triangular linear system calculations are performed by more threads in each MPI rank. Note also the preconditioner ILU10 has a closer behavior to SPIKE50$_8$ (number of iterations and CPU time). However, the preconditioner SPIKE50$_8$ uses the quadruple of processing nodes than that used for the preconditioner ILU10. The Spectral reordering did not provide significant improvement taking into account the number of GMRES iterations.

Figures 5.3, 5.4, and 5.5 show, respectively to $SmallMesh$, $MediumMesh$, and $LargeMesh$, the CPU time, speedup, and efficiency for the parallel preconditioners DIAGe, LUe, SGSe, ILU2, ILU3, ILU10, SPIKE50$_2$, SPIKE50$_4$, and SPIKE50$_8$ considering 24, 48, 72, 96, 192 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Graphics are organized in groups of local (on the left) and global (on the right)

Table 5.2 – Problem 1 - *SmallMesh* - RCM reordering: CPU time and the number of GMRES iterations from 24 to 192 partitions.

Preconditioner	Number of MPI ranks									
	24		48		72		96		192	
	time	iter	time	iter	time	iter	time	iter	time	iter
DIAG*e*	260.76	14188	111.33	14198	76.45	14191	58.16	14172	30.45	14169
LU*e*	117.51	4940	53.31	4953	36.40	5001	28.37	5005	15.35	5050
SGS*e*	134.17	4609	61.30	4614	41.24	4627	31.28	4634	15.51	4665
ILU2	13.17	578	7.28	663	5.29	752	4.46	862	3.02	1291
ILU3	12.39	498	7.00	578	5.43	692	4.55	780	3.31	1242
ILU10	16.19	369	11.29	546	8.76	659	7.47	768	5.50	1215
SPIKE50$_2$	54.03	372	35.91	571	26.59	675	21.63	776	15.90	1257
SPIKE50$_4$	27.07	372	18.35	571	13.76	675	11.46	776	7.84	1257
SPIKE50$_8$	19.04	372	12.88	571	9.75	675	5.30	776	3.64	1257

Table 5.3 – Problem 1 - *MediumMesh* - RCM reordering: CPU time and the number of GMRES iterations from 24 to 192 partitions.

Preconditioner	Number of MPI ranks									
	24		48		72		96		192	
	time	iter	time	iter	time	iter	time	iter	time	iter
DIAG*e*	1443.83	18782	466.51	18772	249.00	18782	174.55	18774	92.30	18777
LU*e*	598.04	6159	195.12	6209	109.49	6173	78.61	6182	42.36	6231
SGS*e*	713.64	6320	231.38	6324	136.99	6329	98.25	6334	53.25	6355
ILU2	43.62	647	20.20	789	13.67	820	10.67	863	6.50	1095
ILU3	41.10	573	19.65	705	13.97	767	11.38	816	7.39	1064
ILU10	50.22	443	29.55	625	21.30	691	17.39	763	11.10	1002
SPIKE50$_2$	160.34	451	99.30	650	65.32	682	52.80	765	35.13	1072
SPIKE50$_4$	75.09	451	49.04	650	33.15	682	26.98	765	17.64	1072
SPIKE50$_8$	51.49	451	33.98	650	23.24	682	18.81	765	12.38	1072

Table 5.4 – Problem 1 - *LargeMesh* - RCM reordering: CPU time and the number of GMRES iterations from 24 to 192 partitions.

Preconditioner	Number of MPI ranks									
	24		48		72		96		192	
	time	iter	time	iter	time	iter	time	iter	time	iter
DIAG*e*	5764.58	26497	2306.58	26495	1103.99	26496	698.01	26495	270.70	26497
LU*e*	4199.05	15502	1660.85	15310	872.14	16111	563.58	16375	206.52	15082
SGS*e*	2941.33	9052	1157.74	9058	577.77	9066	378.73	9070	155.89	9089
ILU2	125.52	842	64.30	937	38.42	1062	30.61	1193	18.67	1513
ILU3	101.53	638	57.10	786	35.84	918	27.99	998	18.87	1362
ILU10	128.45	525	75.85	672	46.24	700	38.64	812	27.53	1208
SPIKE50$_2$	361.44	465	226.19	660	139.03	662	121.39	807	83.86	1219
SPIKE50$_4$	171.41	465	106.54	660	68.96	662	60.55	807	42.21	1219
SPIKE50$_8$	112.54	465	72.02	660	46.94	662	41.36	807	32.13	1219

Table 5.5 – Problem 1 - *LargeMesh* - Spectral reordering: CPU time and the number of GMRES iterations from 24 to 96 partitions.

Preconditioner	Number of MPI ranks							
	24		48		72		96	
	time	iter	time	iter	time	iter	time	iter
DIAG*e*	7088.83	26498	2589.99	26497	1220.89	26497	748.22	26496
LU*e*	4564.25	13658	2564.67	21064	1392.53	24031	725.58	20123
SGS*e*	3683.03	9055	1290.89	9060	629.04	9065	390.45	9072
ILU2	152.81	996	78.89	1109	50.58	1371	41.10	1560
ILU3	119.15	730	64.48	843	42.29	1048	33.33	1144
ILU10	118.78	461	77.64	652	55.98	841	51.79	1083
SPIKE50$_2$	327.40	380	221.61	626	160.70	796	148.15	1023
SPIKE50$_4$	157.86	380	104.29	626	81.09	796	76.19	1023
SPIKE50$_8$	100.46	380	69.56	626	55.95	796	52.43	1023

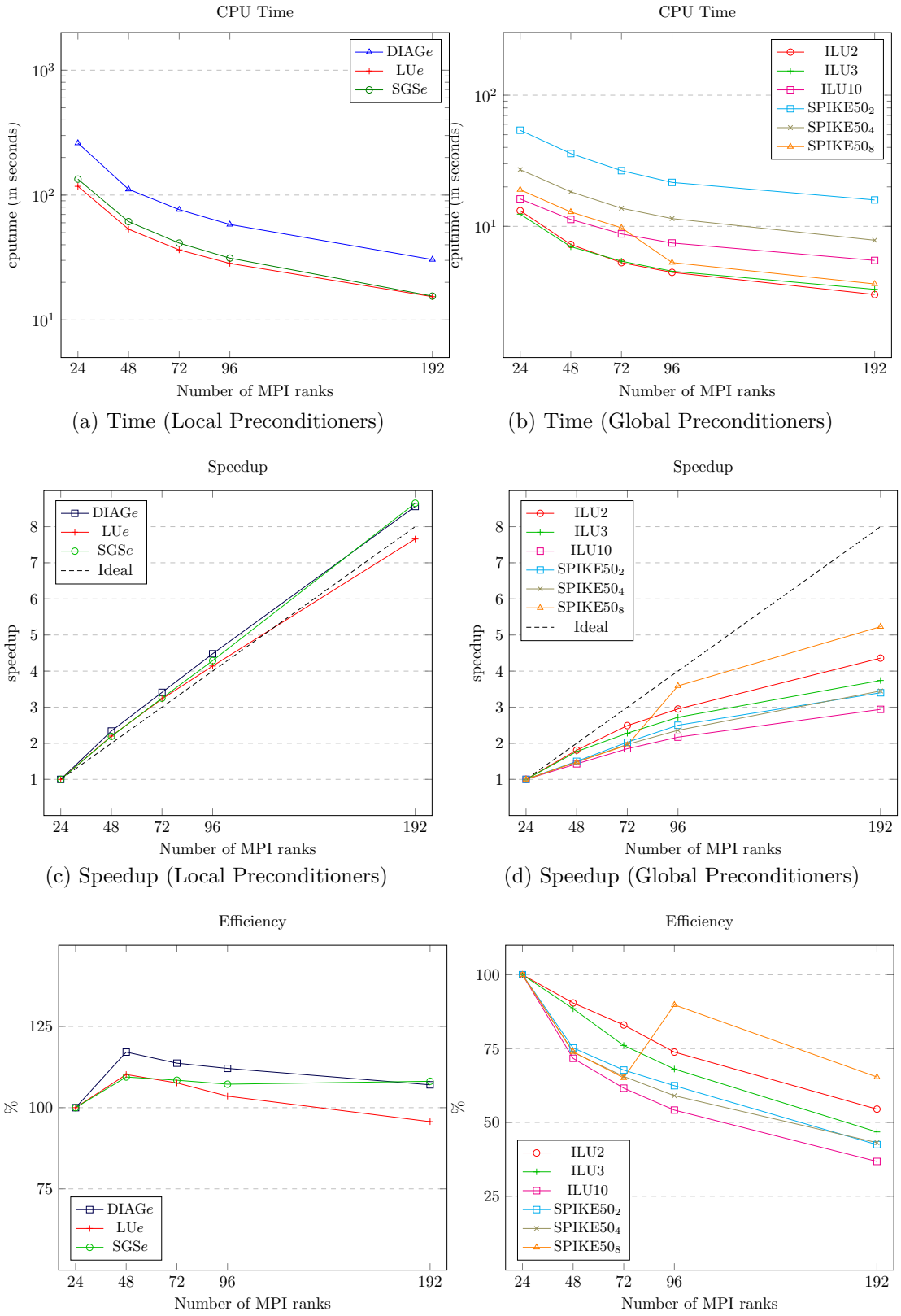
preconditioners. Figure 5.6 shows the same information for $LargeMesh$, but Spectral algorithm reorders the sparsity of the mesh.

Local preconditioners DIAGe, LUe, and SGSe present superlinear speedups. That behavior becomes more notable as the mesh size increases. We can not see any particular reason that could explain this behavior. Possibly, for larger partitions, the cache memory is misused. Note also among the local preconditioners, LUe presents the best CPU time for the meshes $SmallMesh$ and $MediumMesh$. However, for the mesh $LargeMesh$, LUe loses its position for the preconditioner SGSe. LUe is also more sensitive when the mesh is reordered by the Spectral algorithm.

As the size of the meshes increases, global preconditioners become about an order of magnitude faster than local preconditioners. Speedup and efficiency graphs also show significant improvement with the increase of the mesh. From the point of view of efficiency, the ILU2 preconditioner presents the best results, although in some locations the SPIKE50$_8$ preconditioner reveals greater efficiency ($SmallMesh$ mesh with 96 and 192 MPI ranks). In short, global preconditioners with lower *fill-in* number achieve better speedup and efficiency.

Figures 5.7a, 5.8a, and 5.9a show, respectively to $SmallMesh$, $MediumMesh$, and $LargeMesh$, the total memory usage for the parallel preconditioners DIAGe, LUe, SGSe, ILU2, ILU3, ILU10, SPIKE50 (case with only 1 thread) considering 24, 48, 72, 96, 192 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Figure 5.10a, in turn, shows the same information for $LargeMesh$, but Spectral algorithm reorders the sparsity of the mesh. Local preconditioners require less memory consumption than global preconditioners. The preconditioner SPIKE50, in particular, consumes an expressive amount of memory since it needs to store the complete LU factorization of the coefficient matrix. Such storage causes a significant number of *fill-ins*. For this same reason, ILU preconditioners use more memory as the *fill-in* level is increased. For smaller meshes, the variation of total memory usage as the number of partitions grows is more relevant. However, as the mesh increases or when it is reordered by the Spectral algorithm, the memory consumption becomes almost constant independent of the number of MPI ranks.

Figures 5.7b, 5.8b, and 5.9b show, respectively to $SmallMesh$, $MediumMesh$, and $LargeMesh$, the bar min and max graphs of average memory usage per node for the local preconditioner DIAGe and the global preconditioner ILU2 considering 1, 2, 3, 4, and 8 processing nodes and the sparsity of the mesh reordered by the RCM algorithm. The main bar value represents the average memory usage for a specific number of processing nodes. The min and max bar values represent the minimum and maximum memory usage by some MPI rank among all MPI ranks. Figure 5.10b, in turn, shows the same information for $LargeMesh$, but Spectral algorithm reorders the sparsity of the mesh. As expected, the average memory usage per processing node is reduced almost proportionally as more



(a) Time (Local Preconditioners)

(b) Time (Global Preconditioners)

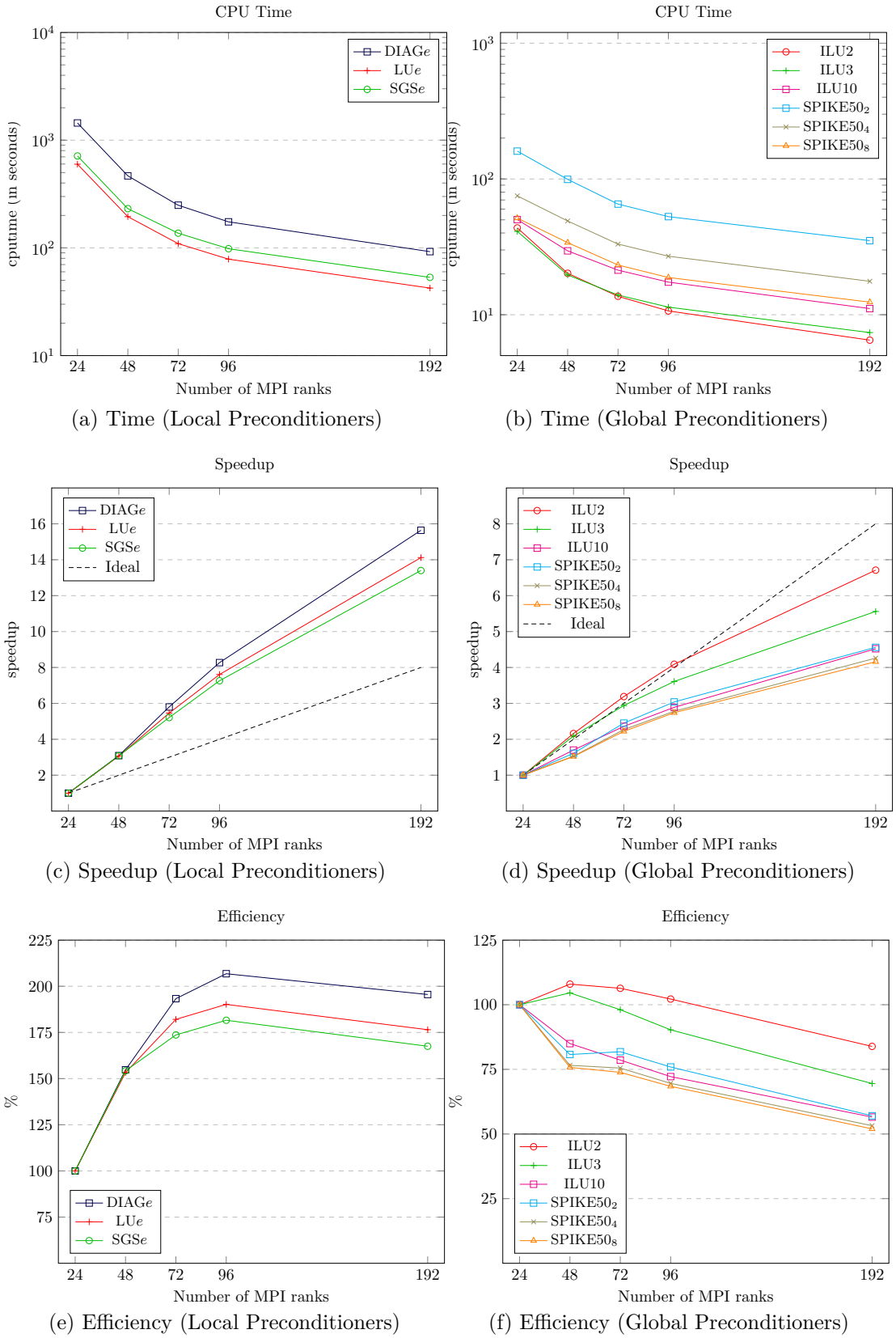
(c) Speedup (Local Preconditioners)

(d) Speedup (Global Preconditioners)

(e) Efficiency (Local Preconditioners)

(f) Efficiency (Global Preconditioners)

Figure 5.3 – Problem 1 - *SmallMesh* - Reordering RCM: CPU time, speedup, and efficiency.



(a) Time (Local Preconditioners)

(b) Time (Global Preconditioners)

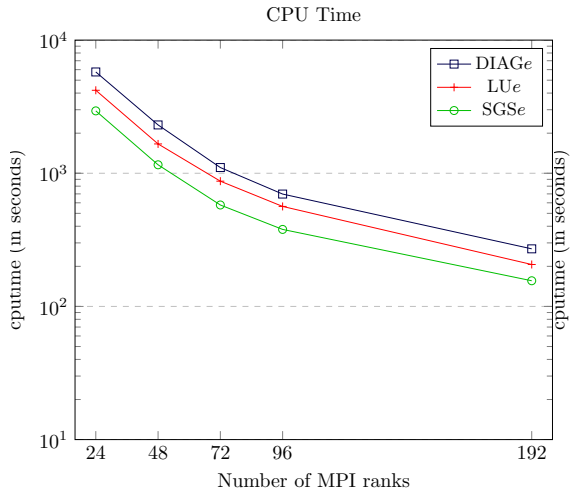
(c) Speedup (Local Preconditioners)

(d) Speedup (Global Preconditioners)

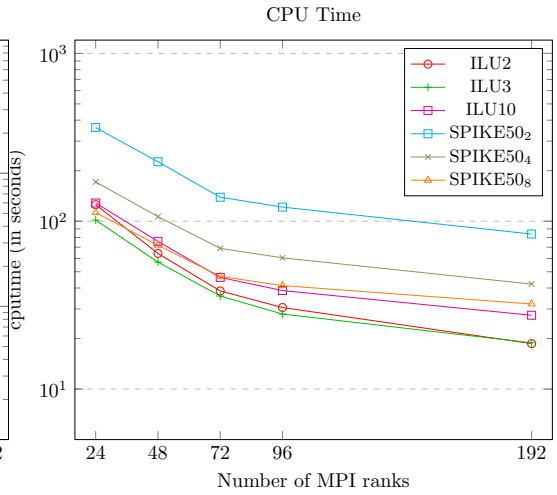
(e) Efficiency (Local Preconditioners)

(f) Efficiency (Global Preconditioners)

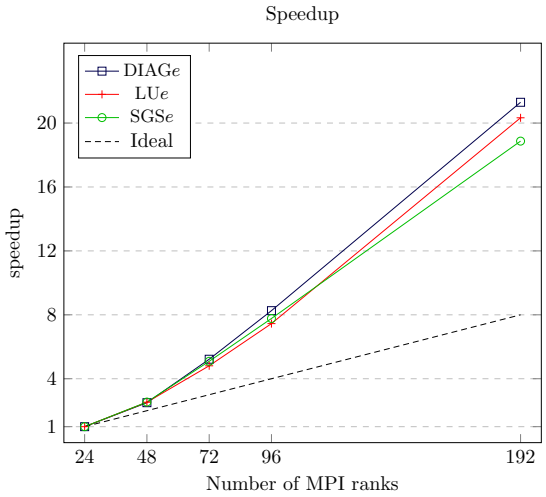
Figure 5.4 – Problem 1 - *MediumMesh* - Reordering RCM: CPU time, speedup, and efficiency.



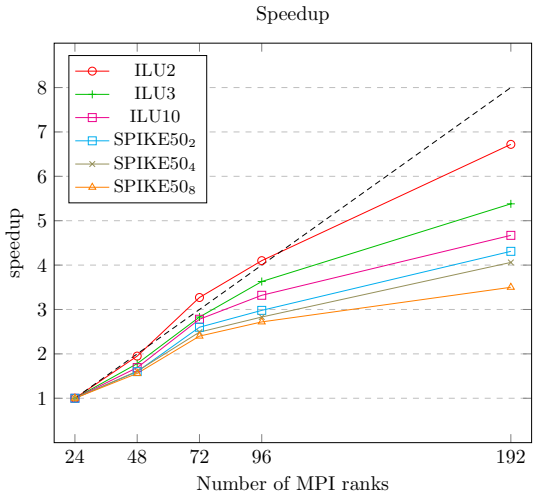
(a) Time (Local Preconditioners)



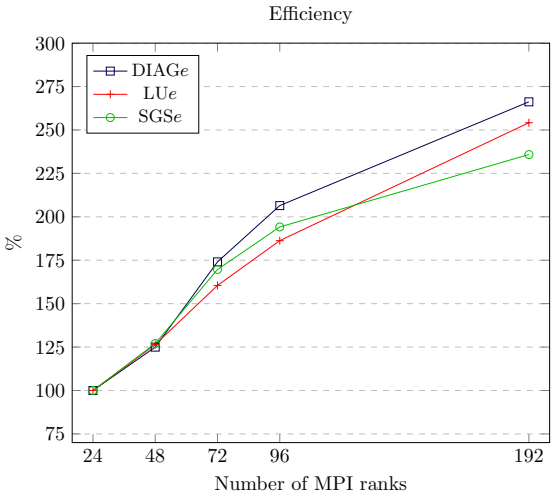
(b) Time (Global Preconditioners)



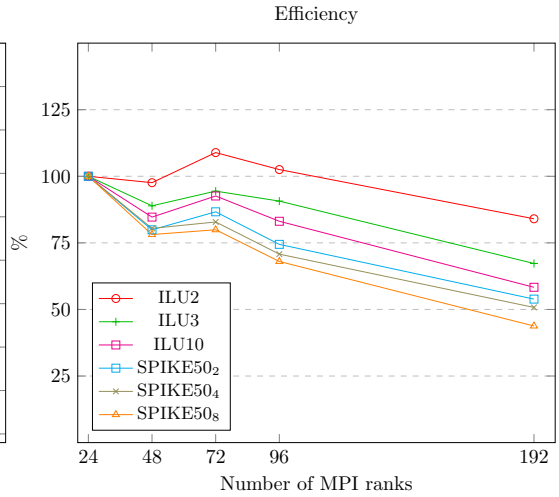
(c) Speedup (Local Preconditioners)



(d) Speedup (Global Preconditioners)

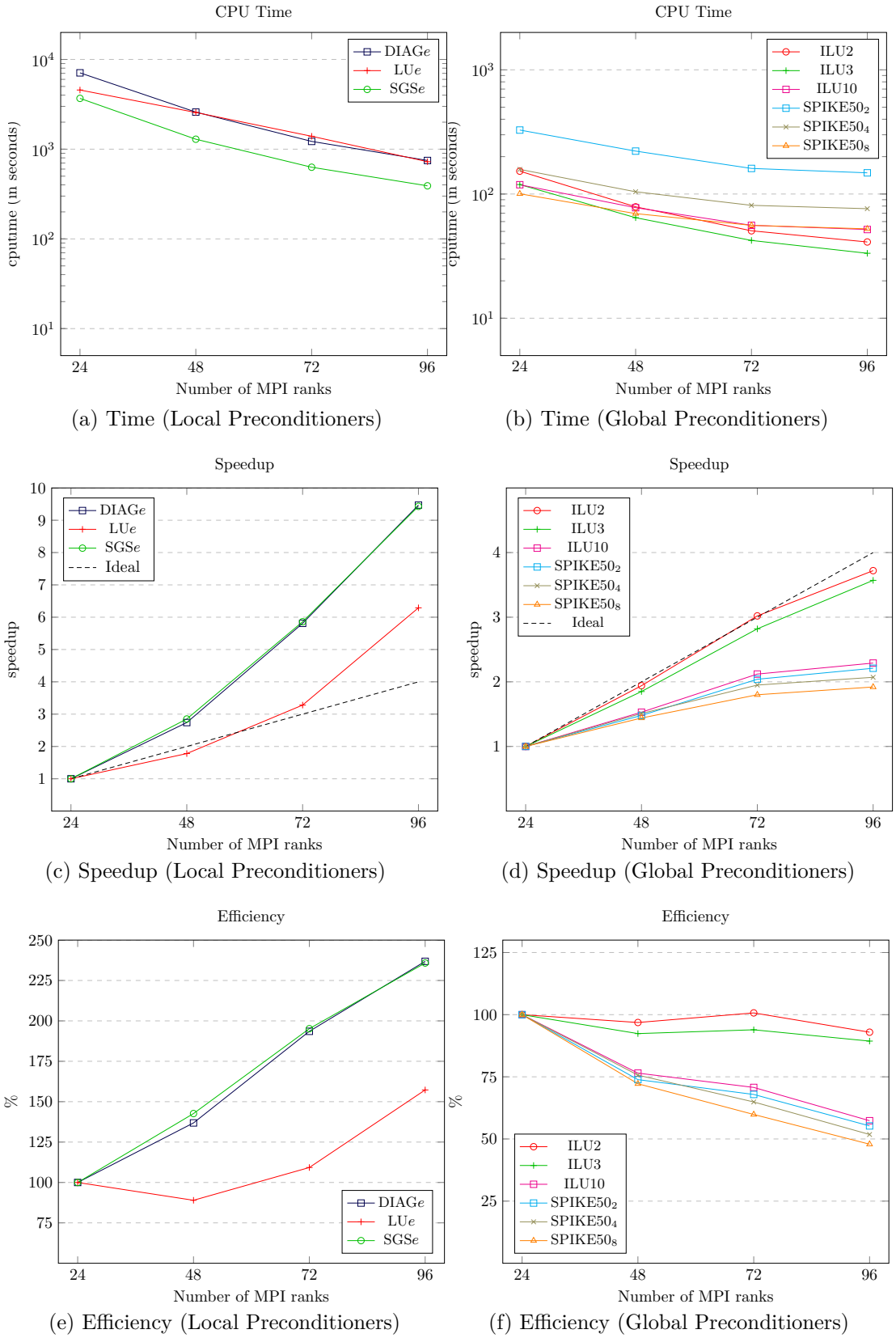


(e) Efficiency (Local Preconditioners)



(f) Efficiency (Global Preconditioners)

Figure 5.5 – Problem 1 - *LargeMesh* - Reordering RCM : CPU time, speedup, and efficiency.



(a) Time (Local Preconditioners)

(b) Time (Global Preconditioners)

(c) Speedup (Local Preconditioners)

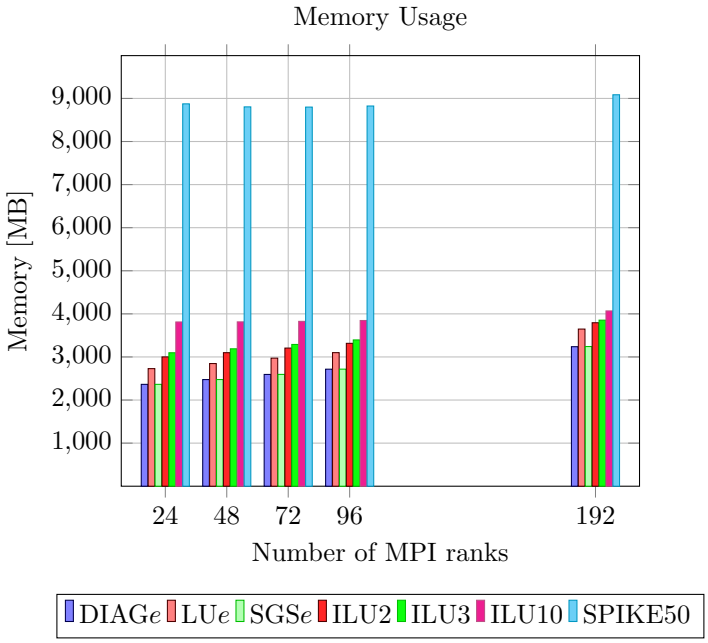
(d) Speedup (Global Preconditioners)

(e) Efficiency (Local Preconditioners)

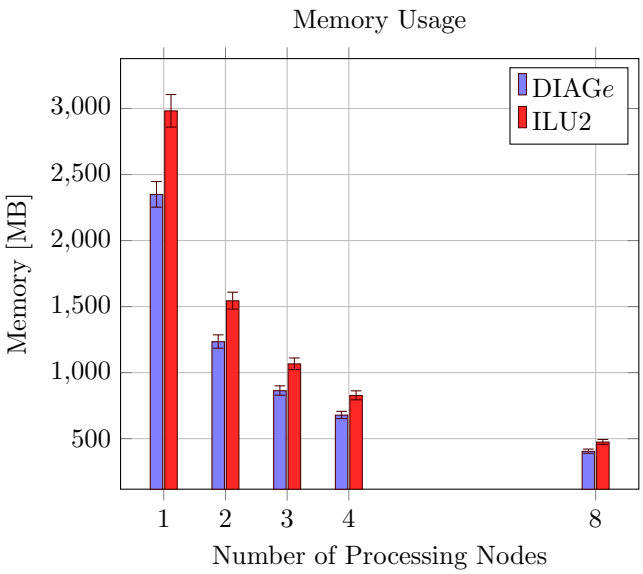
(f) Efficiency (Global Preconditioners)

Figure 5.6 – Problem 1 - *LargeMesh* - Reordering Spectral : CPU time, speedup, and efficiency.

nodes are used. There is not a very large discrepancy between the minimum and maximum memory values used in each MPI rank. This fact suggests that our domain decomposition approach, even initially proposed for global preconditioners with CSR storage, has also been successful in load balancing for local preconditioners with EBE storage.



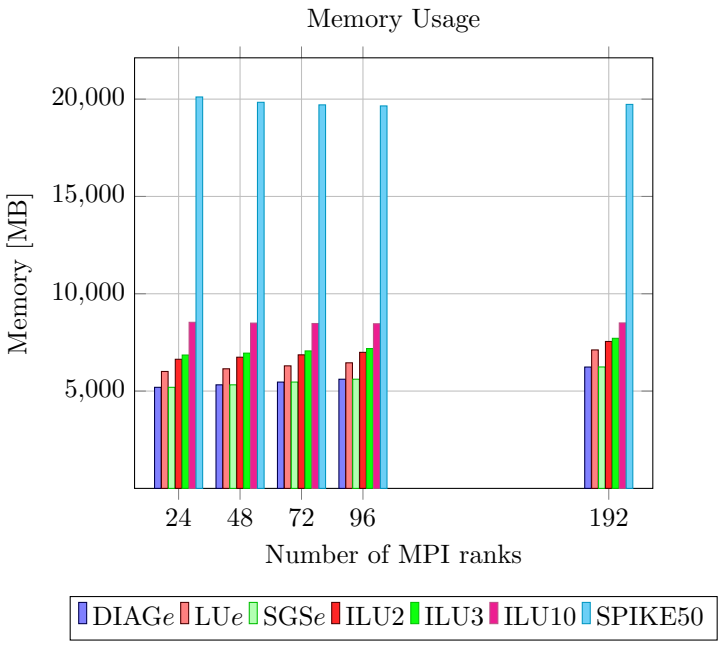
(a) Total Memory Usage (All Preconditioners)



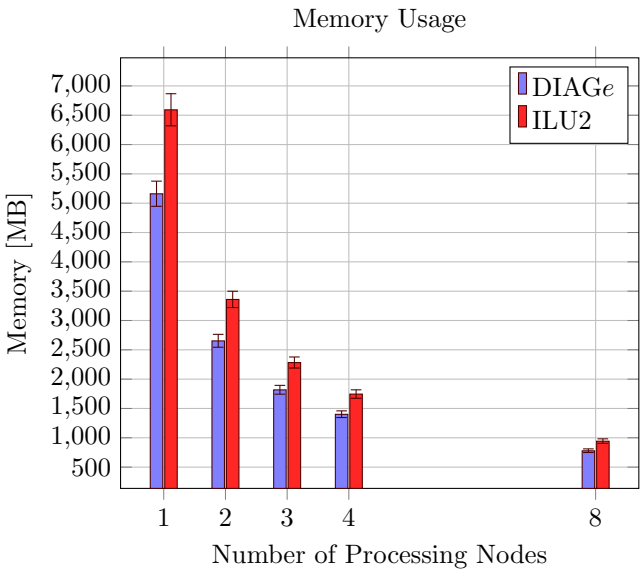
(b) Average Memory Usage per Node (DIAG*e* and ILU2)

Figure 5.7 – Problem 1 - *SmallMesh* - Reordering RCM : Memory Usage.

Figures 5.11, 5.12, and 5.13 show, respectively to *SmallMesh*, *MediumMesh*, and *LargeMesh*, the functions runtime analysis for the parallel preconditioners DIAG*e*, LU*e*,

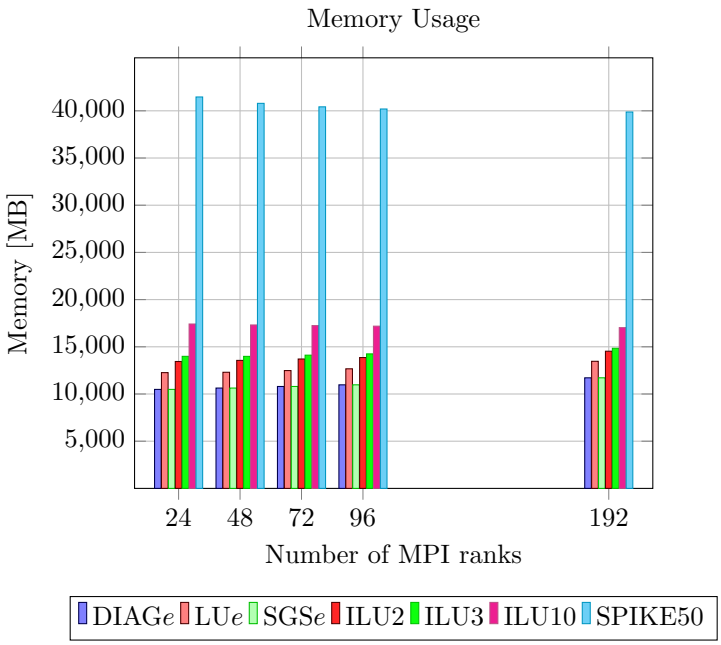


(a) Total Memory Usage (All Preconditioners)

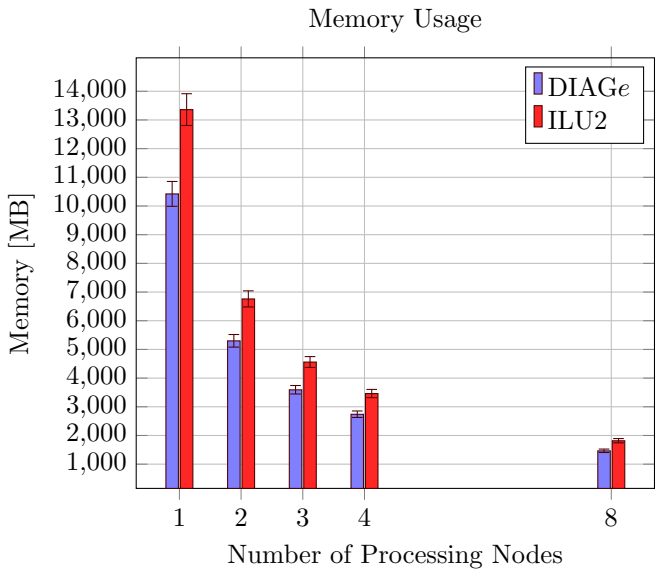


(b) Average Memory Usage per Node (DIAG*e* and ILU2)

Figure 5.8 – Problem 1 - *MediumMesh* - Reordering RCM : Memory Usage.

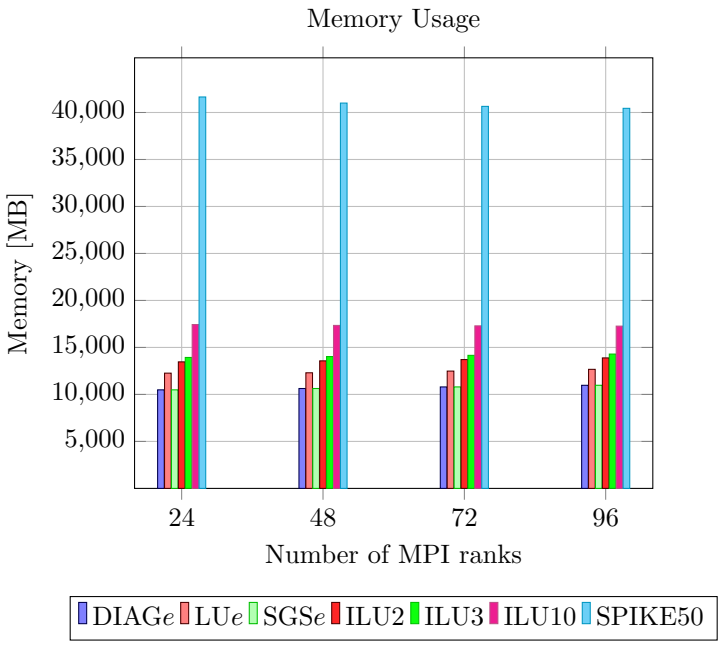


(a) Total Memory Usage (All Preconditioners)

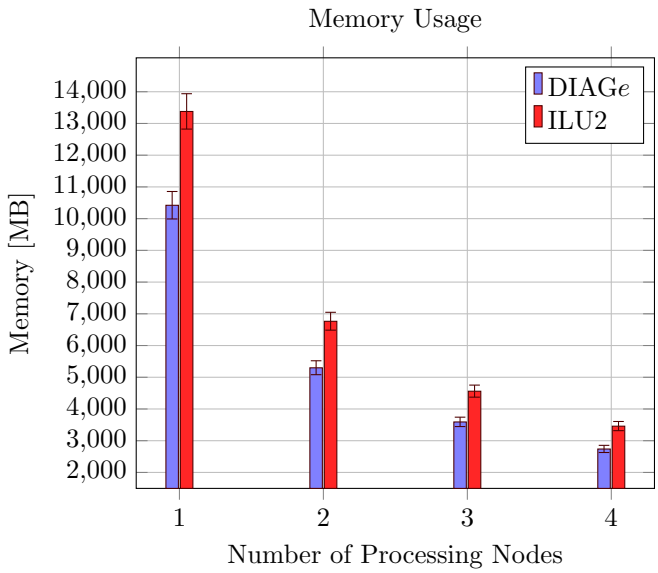


(b) Average Memory Usage per Node (DIAG*e* and ILU2)

Figure 5.9 – Problem 1 - *LargeMesh* - Reordering RCM : Memory Usage.



(a) Total Memory Usage (All Preconditioners)



(b) Average Memory Usage per Node (DIAG*e* and ILU2)

Figure 5.10 – Problem 1 - *LargeMesh* - Reordering Spectral : Memory Usage.

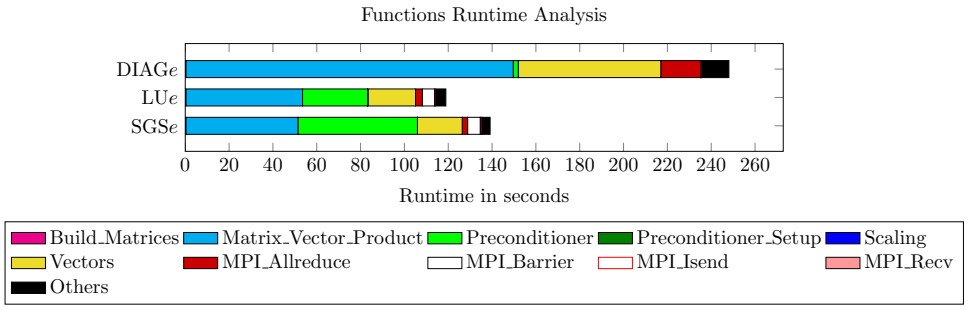
SGSe, ILU2, ILU3, ILU10, SPIKE50$_2$, SPIKE50$_4$, and SPIKE50$_8$ considering 24 and 192 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Graphics are organized into groups of local and global preconditioners. Figure 5.14, in turn, shows the same information, but Spectral algorithm reorders the sparsity of the mesh. Particularly, in this case, experiments with 96 MPI ranks are presented instead of 192 because Spectral reordering failed to split the domain into 192 partitions.

Build_Matrices runtime is almost imperceptible because in this experiment it was run only twice since this experiment is a steady-state case. GMRES iterations performed by local preconditioners are significantly higher than those performed by global preconditioners. The local preconditioners have a higher demand for matrix-vector products (function Matrix_Vector_Product) and another vector operations (function Vectors). Preconditioners LUe and SGSe are the largest demanders of MPI_Barrier because they require additional synchronizations. Once SGSe is a split preconditioner, it needs twice synchronization — the preconditioning is applied in two steps, that is, to the right and to the left. LUe and SGSe are the only preconditioners that use the function Scaling but they demand a negligible runtime from that function.

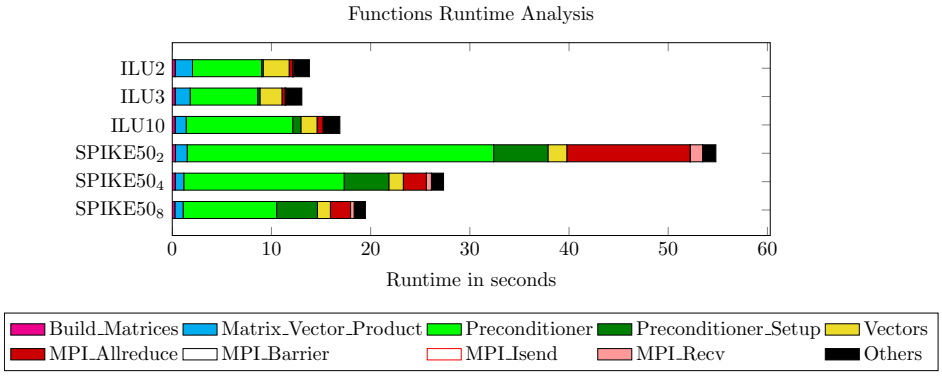
In the context of ILU preconditioners, a higher *fill-in* level allows the reduction of the number of iterations. However, ILU10 incomplete factorizations (see function Preconditioner_Setup) demand a longer runtime than ILU2 incomplete factorizations, for example. Note in addition, the solutions of the triangular systems (see function Preconditioner) by ILU preconditioners with higher higher *fill-in* levels are also more costly, because the generated triangular systems become less sparse as the *fill-in* level increases.

In the case of SPIKE preconditioners, the number of threads has influenced the runtime of several functions. The solutions of the triangular systems (see function Preconditioner), for example, need half CPU time when four threads are used instead of two threads. That suggests the runtime of the solutions of the triangular systems for ILU preconditioners could also be reduced if PARDISO was used. Another point of SPIKE that calls attention is the use of the MPI_Recv function. Compared to other preconditioners, MPI_Recv demands a much longer runtime. This fact is justified by the exchange of messages required by the coupling blocks (each coupling block has size $k \times k$, see Section 2.4).

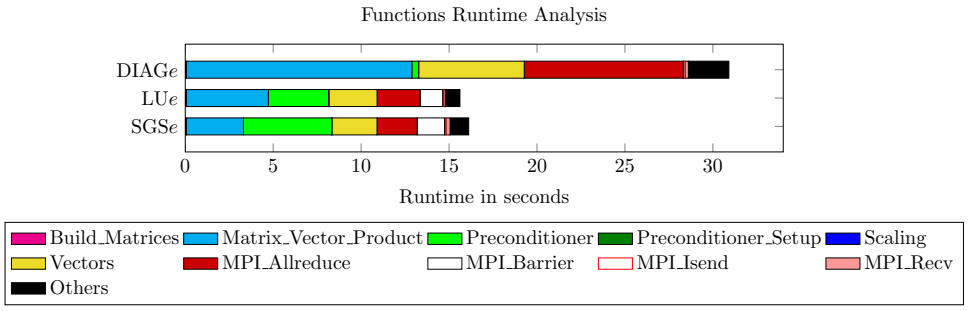
As expected, MPI_Allreduce runtime is proportionally larger for smaller meshes combined with a high number of MPI ranks. MPI_Allreduce runtime is also influenced by the number of GMRES iterations, because MPI_Allreduce is generally used to finalize an inner product operation. Runtimes of MPI_Isend and MPI_Recv can also be negligible (exceptions just to the SPIKE cases). Finally, RCM and Spectral reordering methods do not present significant differences in the functions runtime analysis.



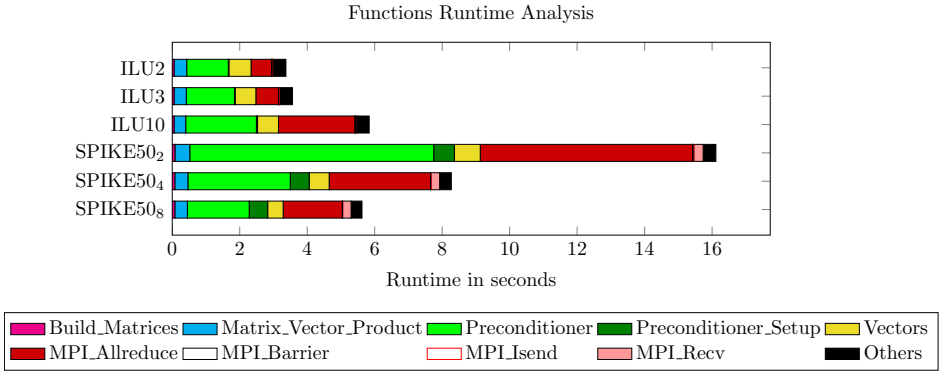
(a) Local Preconditioners : Functions analysis for 24 MPI ranks



(b) Functions Analysis (Global Preconditioners) for 24 MPI ranks

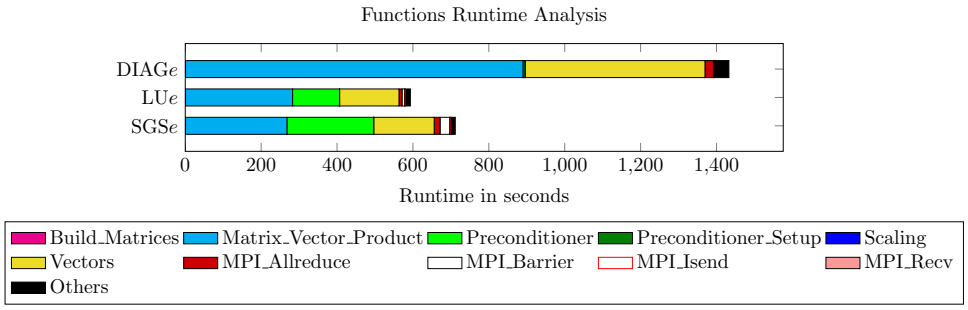


(c) Local Preconditioners: Functions analysis for 192 MPI ranks

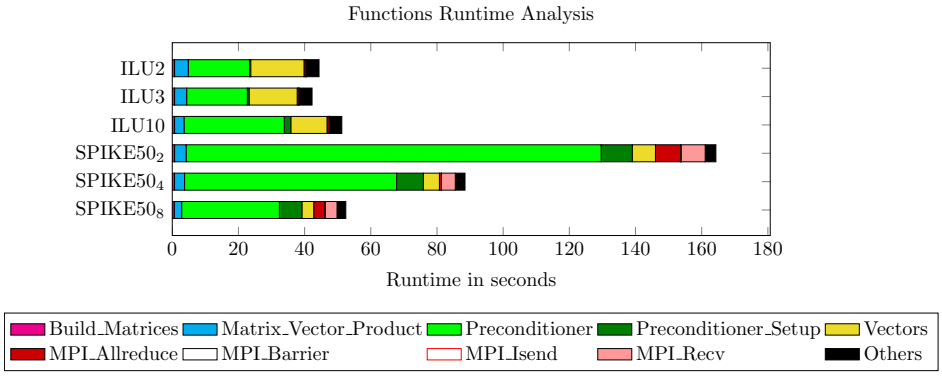


(d) Global Preconditioners : Functions analysis for 192 MPI ranks

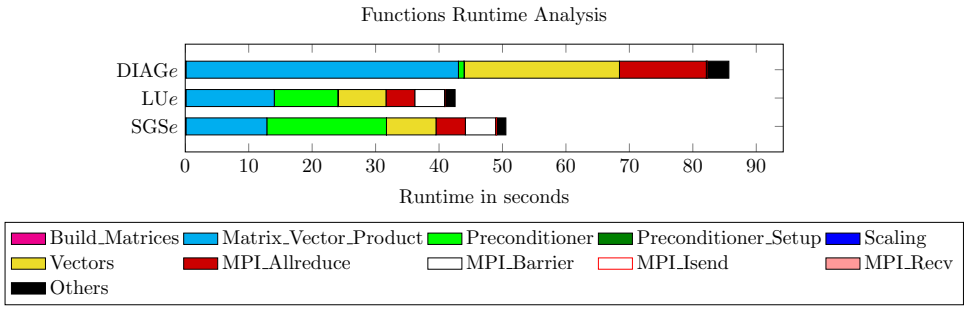
Figure 5.11 – Problem 1 - *SmallMesh* - Reordering RCM: Functions runtime analysis.



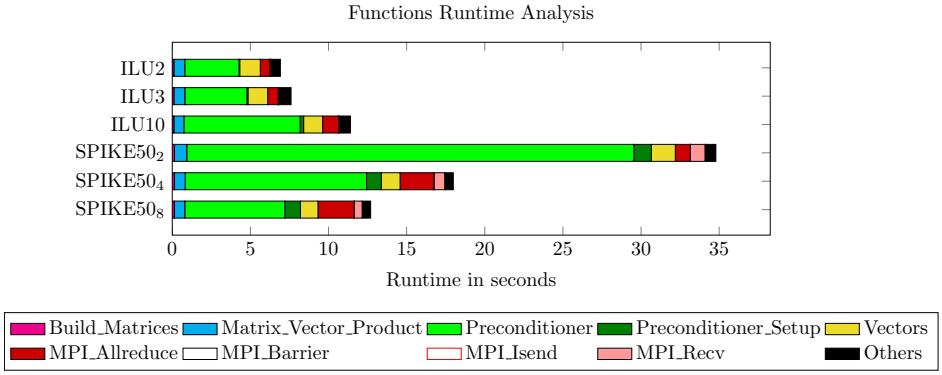
(a) Local Preconditioners : Functions analysis for 24 MPI ranks



(b) Global Preconditioners : Functions analysis for 24 MPI ranks

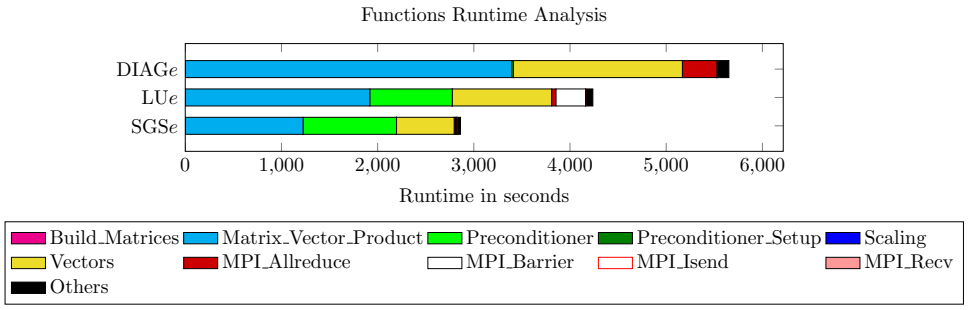


(c) Local Preconditioners : Functions analysis for 192 MPI ranks

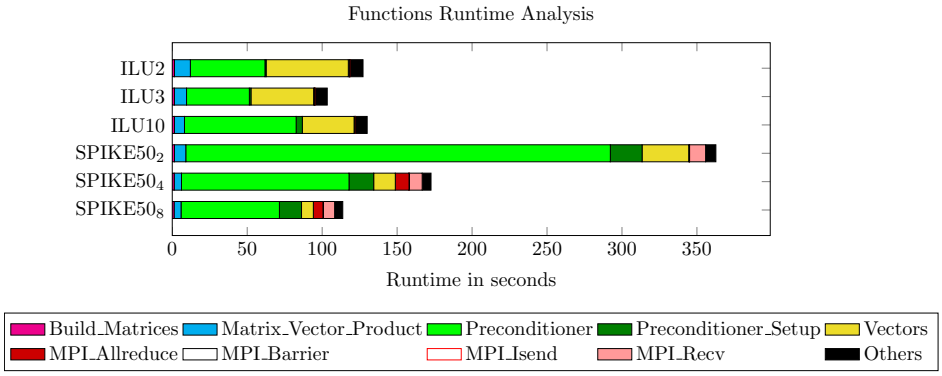


(d) Global Preconditioners : Functions analysis for 192 MPI ranks

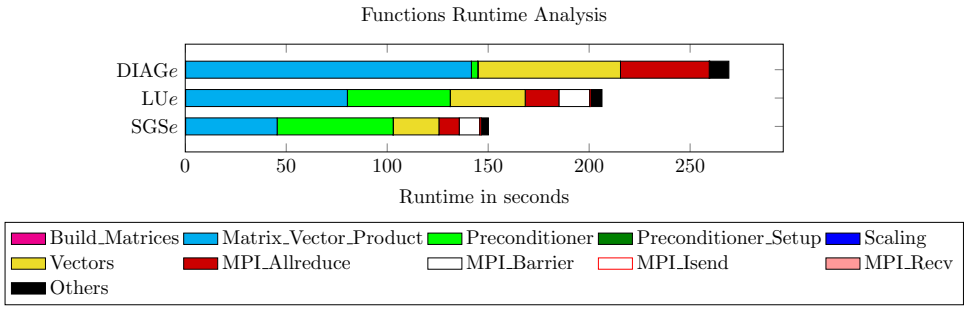
Figure 5.12 – Problem 1 - *MediumMesh* - Reordering RCM : Functions runtime analysis.



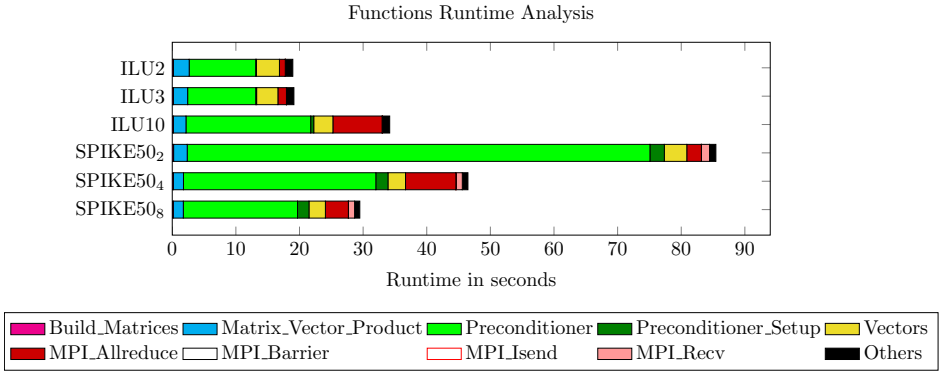
(a) Local Preconditioners : Functions analysis for 24 MPI ranks



(b) Global Preconditioners : Functions analysis for 24 MPI ranks

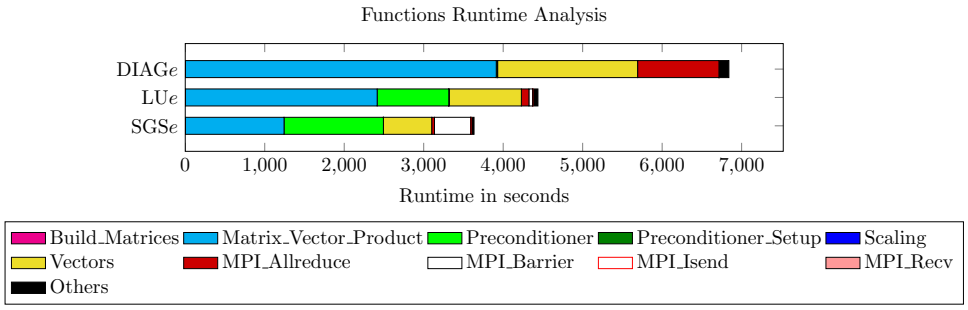


(c) Local Preconditioners : Functions analysis for 192 MPI ranks

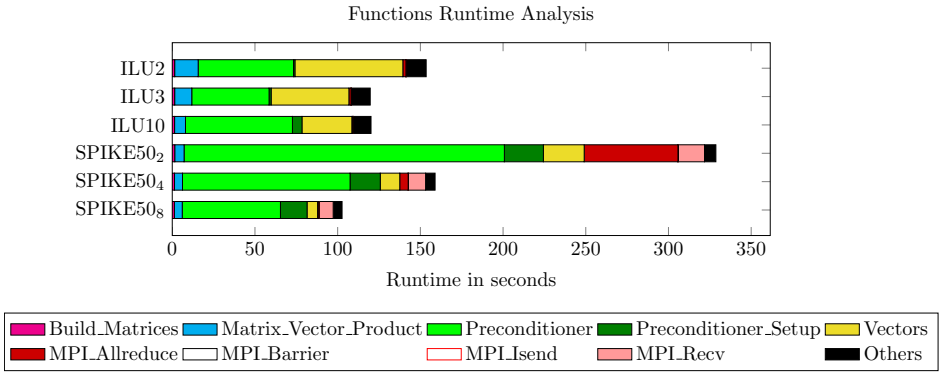


(d) Global Preconditioners : Functions analysis for 192 MPI ranks

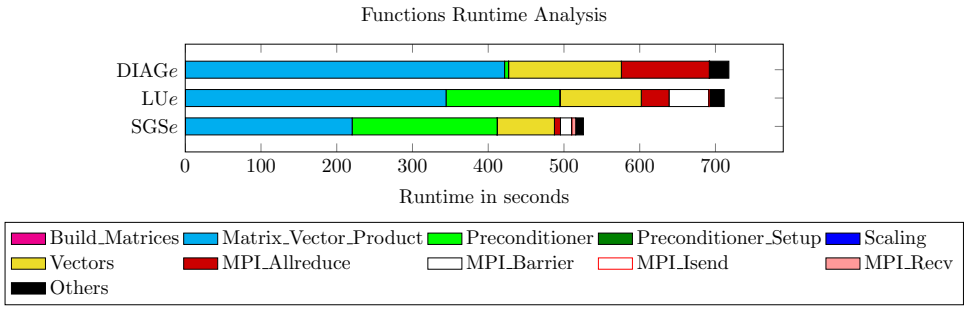
Figure 5.13 – Problem 1 - *LargeMesh* - Reordering RCM : Functions runtime analysis.



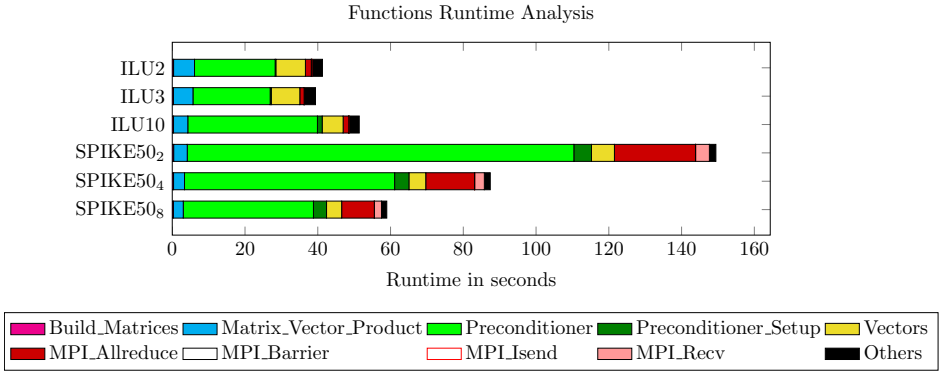
(a) Local Preconditioners : Functions analysis for 24 MPI ranks



(b) Global Preconditioners : Functions analysis for 24 MPI ranks



(c) Local Preconditioners : Functions analysis for 96 MPI ranks



(d) Global Preconditioners : Functions analysis for 96 MPI ranks

Figure 5.14 – Problem 1 - *LargeMesh* - Reordering Spectral : Functions runtime analysis.

5.4 Rotating cone - Problem 2 : Transient case with `ndof=1`

The second problem is a transient problem with 1 degree of freedom per node. The exact solution, modeled by the transport transient advection-diffusion equation as described by Brooks and Hughes (1982), consists of a rigid rotation of the cone about the center of the mesh. Figure 5.15 shows the problem statement, where the computational domain is a square domain, $\Omega = [0, 10] \times [0, 10]$, diffusivity constant is $\epsilon = 10^{-8}$, source term is $f = 0$, the velocity field is $\beta = (-y, x)^T$. The time advancing is solved using the parallel predictor-multi-corrector algorithm as described in Algorithm 4. The final time $t_{final} = 6.28$ and the time step $\Delta t = 10^{-2}$ are chosen corresponding to a full 360° rotation of the cone. For each time step is set 3 fixed multi-corrections. In short, the parallel preconditioned GMRES (see Algorithm 5) runs exactly 1884 times to solve this experiment.

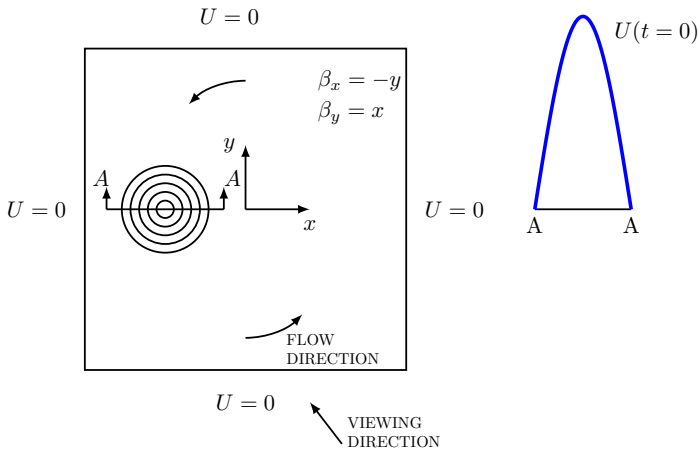


Figure 5.15 – Rotating cone: Problem statement.

An unstructured mesh with 1,321,646 nodes and 2,639,290 elements is used. Table 5.6 shows the CPU time, bandwidth sizes, and memory usage concern our domain decomposition approach corresponding to divide the sequential mesh into 192 partitions using the RCM or Spectral algorithm. Besides, the reduction of bandwidth size is presented when both reordering algorithms are used.

Table 5.6 – Problem 2 : Domain Decomposition Approach - Preprocessing Bandwidth, CPU Time, and Memory Usage.

	Bandwidth before reordering	Bandwidth after reordering	Time for Reordering (in seconds)	Total time to preprocess the domain decomposition into 192 partitions (in seconds)	Memory Usage (in MB)
Mesh (RCM)	1,317,181	2,077	3.74	36.744	843
Mesh (Spectral)	1,317,181	6,129	10.01	43.014	832

Figure 5.16 presents the Problem 2 solution at time $t = 6.28$ seconds, obtained with

the ILU0 preconditioner and 24 MPI ranks. The graphic is plotted using Paraview software after postprocessing phase (see Section 4.3).

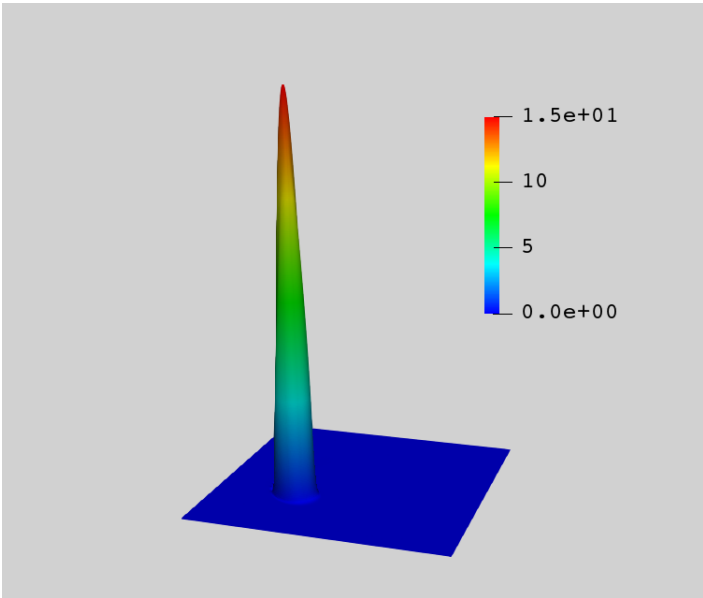


Figure 5.16 – Problem 2 : Solution at $t = 6.28$ seconds with the ILU0 preconditioner and 24 MPI ranks.

This experiment also evaluates local and global preconditioners. DIAGe, LUe, and SGSe are taken as local preconditioners representatives. We analyze Incomplete LU factorizations and SPIKE preconditioners in the global preconditioners context. The incomplete LU factorizations (ILUm) are used with *fill-in* levels 0, 1 and 2 which generate the preconditioners ILU0, ILU1, and ILU2. Only the SPIKE preconditioner SPIKE50$_4$ is analyzed, which means we use coupling block size $k = 50$ and 4 threads per MPI rank. Like in Problem 1, SPIKE50$_4$ uses the double number of processing nodes to run. In this section, the preconditioners' performance is also compared when RCM and Spectral algorithms reorder the sparsity of the mesh. Thus, there are two sets of numerical experiments considering 24, 48, 72, 96, and 192 MPI ranks.

Tables 5.7 and 5.8 present the total CPU time and the average number of GMRES iterations per GMRES execution (iter$_{av}$) – the total number of GMRES iterations divided by the number of GMRES executions to solve the problem (1884). Those tables show, respectively, the parallel preconditioners DIAGe, LUe, SGSe, ILU0, ILU1, ILU2, and SPIKE50$_4$ considering 24, 48, 72, 96, 192 MPI ranks and the sparsity of the mesh reordered by RCM and Spectral algorithms. Local preconditioners DIAGe, LUe, and SGSe keep their iter$_{av}$ almost constant when the number of MPI ranks increases. LUe presents the best results both to iter$_{av}$ and to the CPU time. The reordering procedure does not influence the behavior of the ILU preconditioners. ILU0 is the best option when global preconditioners are used. SPIKE50$_4$ does not present competitive CPU times though iter$_{av}$ be very similar to ILU preconditioners for RCM and Spectral algorithms.

Table 5.7 – Problem 2 - RCM reordering : CPU time and the average number of GMRES iterations from 24 to 192 partitions.

Preconditioner	Number of MPI ranks					
	24		48		72	
	time	$iter_{av}$	time	$iter_{av}$	time	$iter_{av}$
DIAGe	1157.21	65.8	559.76	64.3	388.07	64.2
LUe	355.12	11.5	178.52	11.6	123.09	11.6
SGSe	771.58	24.5	380.89	24.7	252.06	24.6
ILU0	372.57	12.8	186.10	14.8	119.64	16.0
ILU1	375.43	12.4	190.54	14.3	124.21	15.5
ILU2	420.88	12.4	212.63	14.3	142.58	15.5
SPIKE50$_4$	3342.97	12.7	1662.48	14.2	1192.77	15.6

Preconditioner	Number of MPI ranks			
	96		192	
	time	$iter_{av}$	time	$iter_{av}$
DIAGe	285.89	65.0	133.60	65.5
LUe	89.77	11.6	43.29	12.0
SGSe	179.38	24.5	72.14	24.4
ILU0	84.72	17.0	43.97	20.5
ILU1	91.79	16.8	45.16	20.1
ILU2	105.01	16.8	50.73	20.3
SPIKE50$_4$	984.38	16.8	751.13	20.3

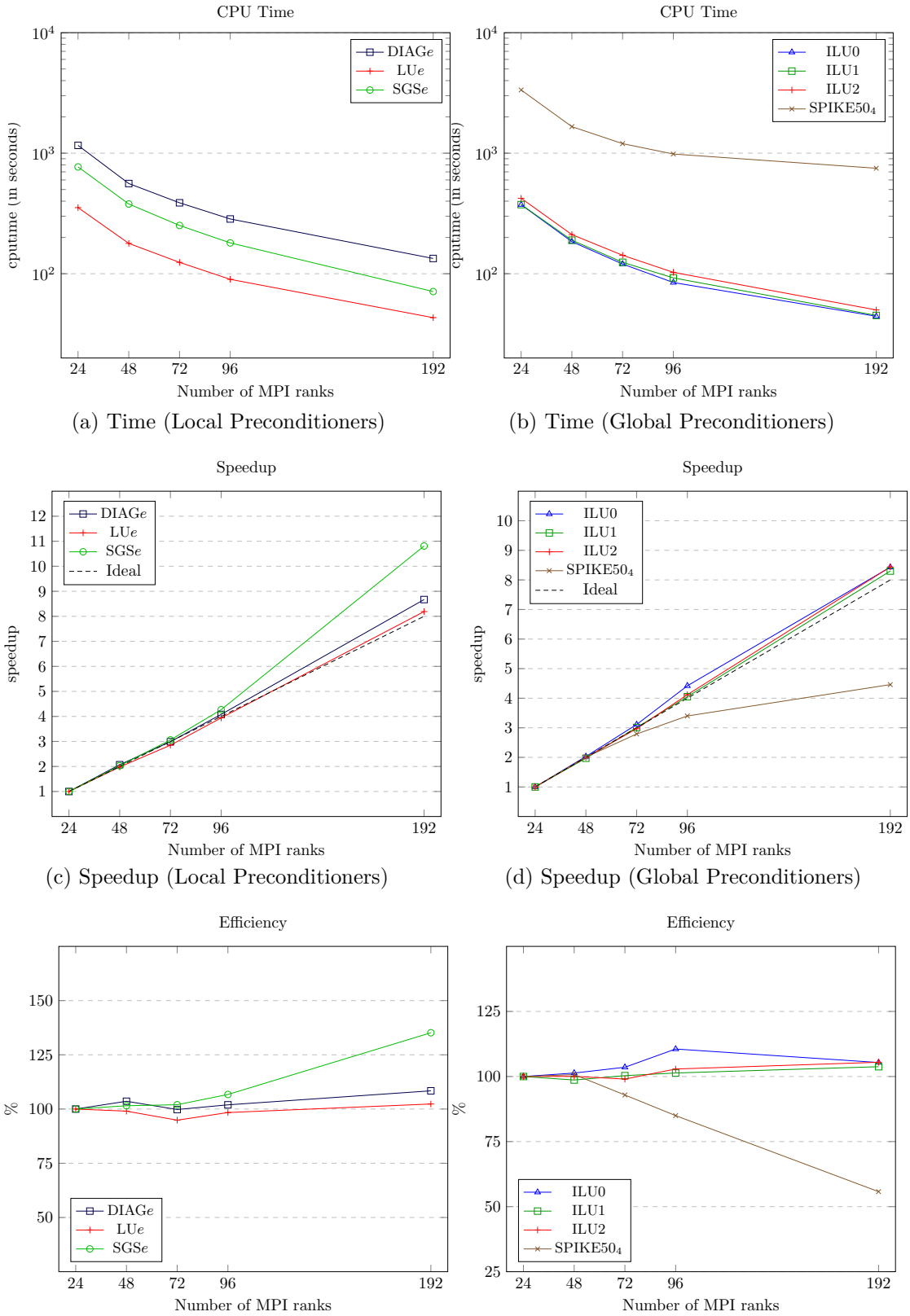
Table 5.8 – Problem 2 - Spectral reordering : CPU time and the average number of GMRES iterations from 24 to 192 partitions.

Preconditioner	Number of MPI ranks					
	24		48		72	
	time	$iter_{av}$	time	$iter_{av}$	time	$iter_{av}$
DIAGe	1143.83	64.3	568.31	63.5	382.85	62.5
LUe	361.24	11.5	179.82	11.6	120.65	11.6
SGSe	777.93	24.7	381.19	24.4	244.92	24.6
ILU0	375.45	12.9	182.54	14.6	117.09	15.9
ILU1	377.18	12.5	187.87	14.3	124.60	15.6
ILU2	428.71	12.4	215.12	14.1	145.16	15.6
SPIKE50$_4$	3516.93	13.6	1793.40	15.6	1210.62	16.5

Preconditioner	Number of MPI ranks			
	96		192	
	time	$iter_{av}$	time	$iter_{av}$
DIAGe	276.98	63.5	113.60	63.9
LUe	87.67	11.7	38.18	12.0
SGSe	176.75	25.0	64.69	24.6
ILU0	82.74	16.9	42.39	20.3
ILU1	89.51	16.7	44.73	20.1
ILU2	106.50	16.8	50.10	19.9
SPIKE50$_4$	998.26	16.6	755.26	20.0

Figures 5.17 and 5.18 show CPU time, speedup, and efficiency for the parallel preconditioners DIAGe, LUe, SGSe, ILU0, ILU1, ILU2, and SPIKE50$_4$ considering 24, 48, 72, 96, 192 MPI ranks and the sparsity of the mesh reordered by RCM or Spectral algorithms. Graphics are organized into groups of local and global preconditioners. All local preconditioners present superlinear speedup, but LUe is about three times faster than DIAGe and twice as fast as SGSe. ILU preconditioners present scalability for both reordering algorithms. Although scalability of SPIKE50$_4$ does not decrease dramatically, the CPU time for all experiments is one order of magnitude higher compared to LUe and ILU preconditioners.

Figures 5.19a and 5.20a show the total memory usage for the parallel preconditioners



(a) Time (Local Preconditioners)

(b) Time (Global Preconditioners)

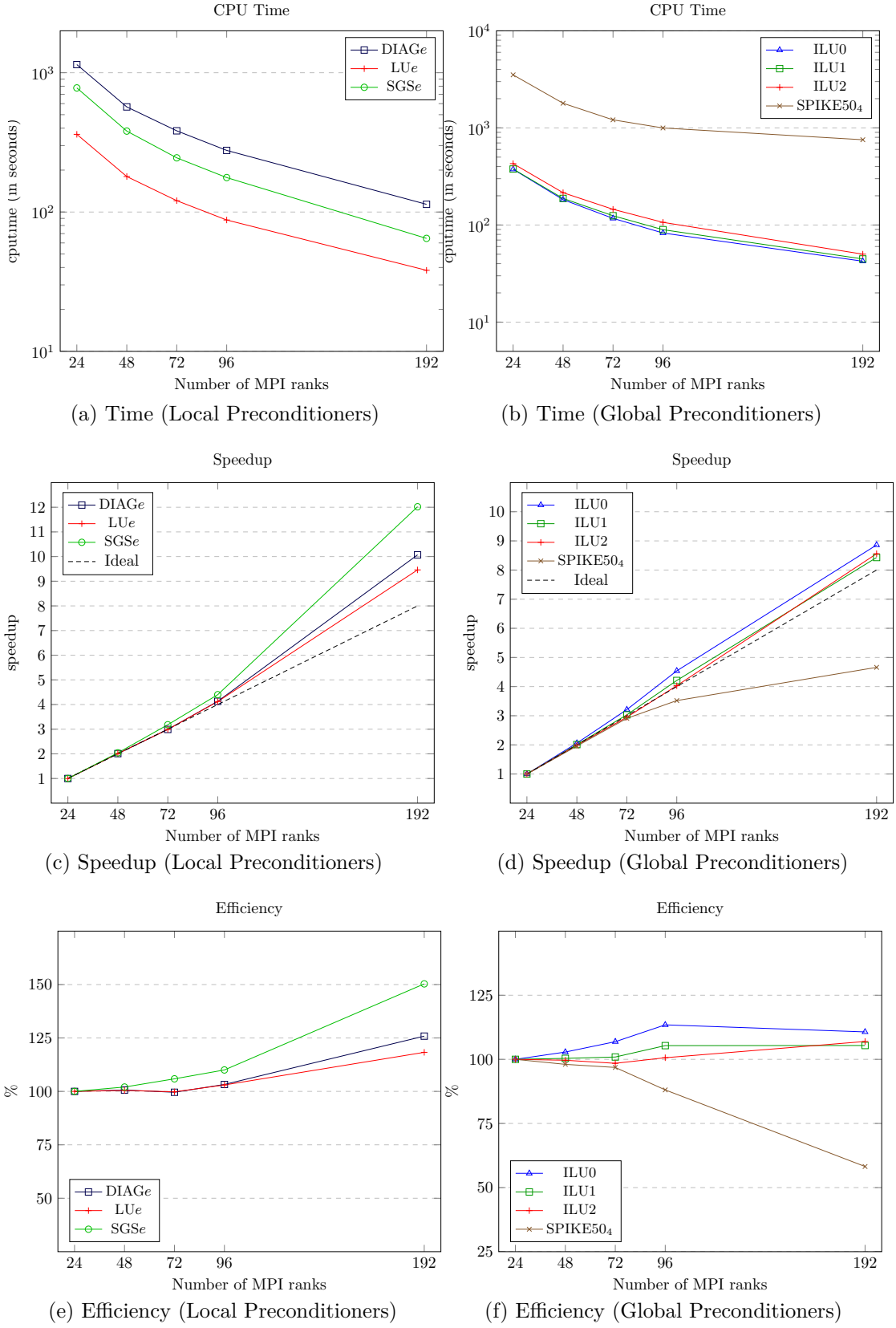
(c) Speedup (Local Preconditioners)

(d) Speedup (Global Preconditioners)

(e) Efficiency (Local Preconditioners)

(f) Efficiency (Global Preconditioners)

Figure 5.17 – Problem 2 - Reordering RCM : Graphics of CPU time, speedup, and efficiency.



(a) Time (Local Preconditioners)

(b) Time (Global Preconditioners)

(c) Speedup (Local Preconditioners)

(d) Speedup (Global Preconditioners)

(e) Efficiency (Local Preconditioners)

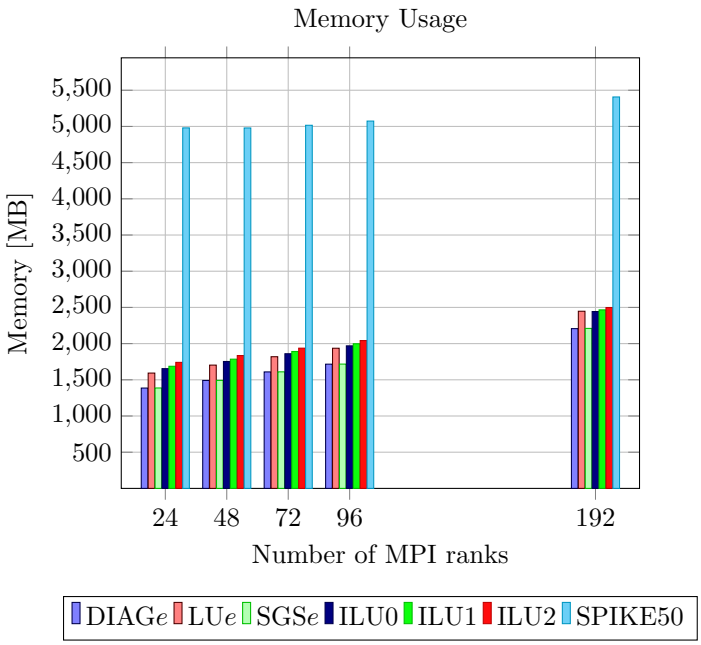
(f) Efficiency (Global Preconditioners)

Figure 5.18 – Problem 2 - Reordering Spectral : CPU time, speedup, and efficiency.

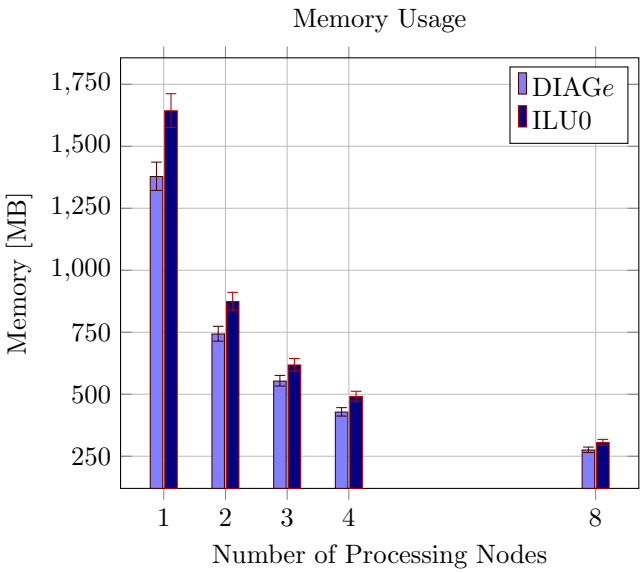
DIAGe, LUe, SGSe, ILU0, ILU1, ILU2, SPIKE50 (case with only 1 thread) considering 24, 48, 72, 96, 192 MPI ranks and the sparsity of the mesh reordered by the RCM or Spectral algorithms. As in Problem 1, local preconditioners require less memory usage than global preconditioners. The preconditioner SPIKE50 also consumes a significant amount of memory because of the complete LU factorizations. ILU preconditioners use more memory as the *fill-in* level is increased. The increase in total memory usage as the number of MPI ranks grows is noticed more sharply when compared to the analysis presented in Problem 1. This happens because in the transient experiment studied in Problem 2 the mesh is coarser and the matrix of coefficients has a much smaller order and the number of partitions is maintained. Thus, overlap regions are proportionally larger.

Figures 5.19b and 5.20b show the bar min and max graphs of average memory usage per node for the local preconditioner DIAGe and the global preconditioner ILU0 considering 1, 2, 3, 4, and 8 processing nodes and the sparsity of the mesh reordered by the RCM or Spectral algorithms. As a consequence of the overlap regions are proportionally large, the average memory usage in 8 processing nodes is about 5 times lower than the average memory usage in a single processing node – in Problem 1 was about 7 times. The discrepancy between the minimum and maximum memory values used in each MPI rank in this experiment is similar to that presented in Problem 1.

Figures 5.21 and 5.22 show the functions runtime analysis for the parallel preconditioners DIAGe, LUe, SGSe, ILU0, ILU1, ILU2, and SPIKE50$_4$ considering 24 and 192 MPI ranks and the sparsity of the mesh reordered by the RCM or Spectral algorithms. In general form, all global preconditioners demand similar runtimes to perform functions Build_Matrices, Matrix_Vector_Product, and Vector. That is because they presented similar average number of GMRES iterations (see Tables 5.7 and 5.8). Local preconditioners Build_Matrices runtime is slightly smaller when compared with corresponding global preconditioners functions. That occurs due to complexity of CSR matrix assembly (see Subsection 4.1.4). On the other hand, local preconditioners Matrix_Vector_Product runtimes are the largest. Even so, the preconditioner LUe is the fastest because the runtime of the functions Preconditioner and Precontitioner_Setup are much smaller compared to the corresponding functions of ILU preconditioners (fact assured by the smaller number of GMRES iterations obtained by the preconditioner LUe – see Tables 5.7 and 5.8). Complete LU factorizations provided by SPIKE preconditioners make function Preconditioner_Setup runtime extremely long. On the other hand, ILU preconditioners achieve similar number of GMRES iterations when compared with SPIKE50$_4$, but ILU runtimes are notoriously smaller. As reported in steady-state case (Problem 1), SPIKE preconditioners demand much more MPI_Allreduce and MPI_Recv runtimes. Again, RCM and Spectral reordering algorithms do not present significant differences in the functions runtime analysis.

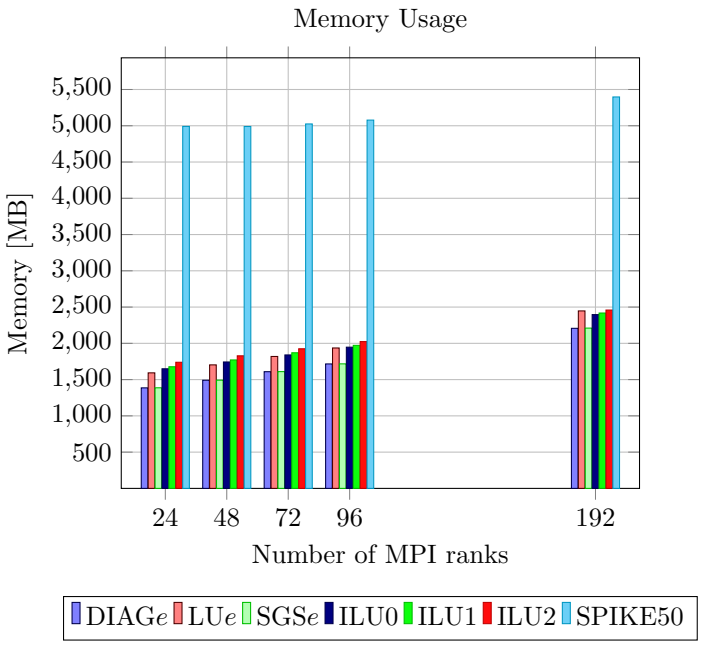


(a) Total Memory Usage (All Preconditioners)

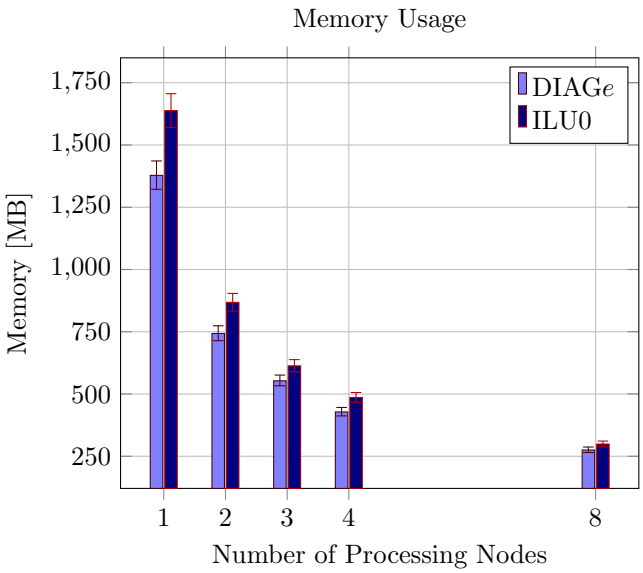


(b) Average Memory Usage per Node (DIAG*e* and ILU0)

Figure 5.19 – Problem 2 - Reordering RCM : Memory Usage.

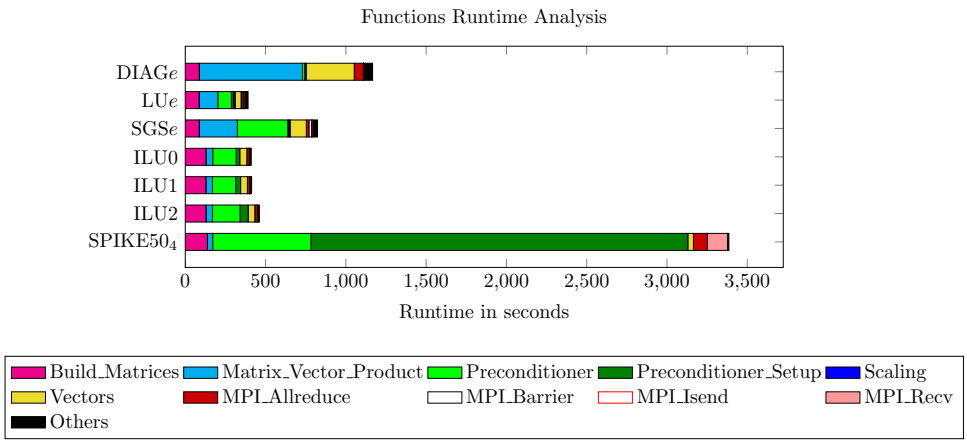


(a) Total Memory Usage (All Preconditioners)

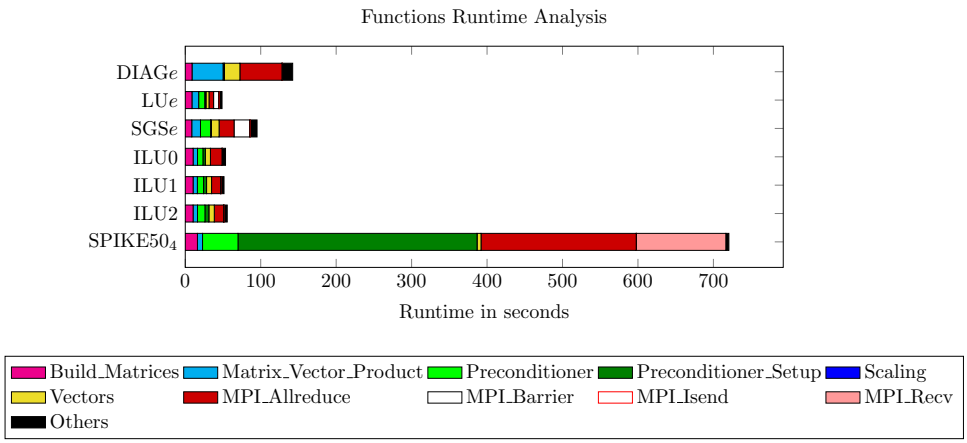


(b) Average Memory Usage per Node (DIAG*e* and ILU0)

Figure 5.20 – Problem 2 - Reordering Spectral : Memory Usage.



(a) Functions analysis for 24 MPI ranks



(b) Functions analysis for 192 MPI ranks

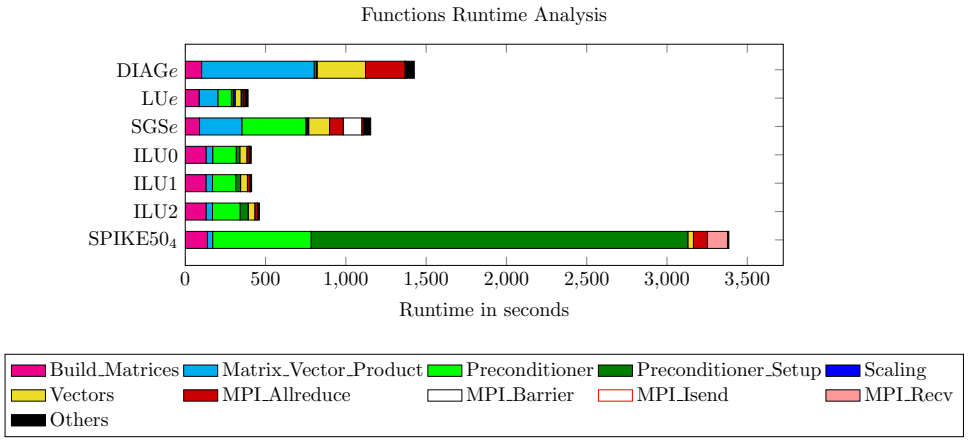
Figure 5.21 – Problem 2 - Reordering RCM : Functions runtime analysis.

5.5 Explosion - Problem 3 : Transient case with `ndof=4`

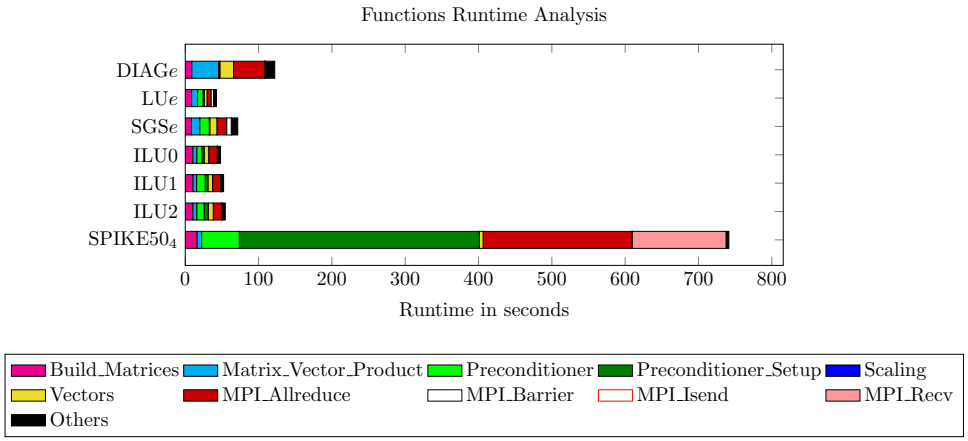
The third problem considered is a transient problem with 4 degrees of freedom per node. That problem, known as explosion problem, is described in (Toro, 2013) and studied by Bento et al. (2016). The 2D Euler equations are solved in a 2.0×2.0 square domain in the $xy-$plane. The initial condition consists of the region inside of a circle with radius $R = 0.4$ centered at $(1, 1)$ and the region outside the circle, see Fig. 5.23. The flow variables are constant in each of these regions and are separated by a circular discontinuity at time $t = 0$. The two constant states are chosen as

$$\text{ins} \quad \begin{cases} \rho &= 1.0 \\ u &= 0.0 \\ v &= 0.0 \\ p &= 1.0 \end{cases} \qquad \text{out} \quad \begin{cases} \rho &= 0.125 \\ u &= 0.0 \\ v &= 0.0 \\ p &= 0.1 \end{cases} \tag{5.1}$$

The time advancing is solved using the parallel predictor-multi-corrector algorithm as described in Algorithm 4. The final time $t_{final} = 0.25$ and the time step $\Delta t = 10^{-3}$



(a) Functions analysis for 24 MPI ranks



(b) Functions analysis for 192 MPI ranks

Figure 5.22 – Problem 2 - Reordering Spectral : Functions runtime analysis.

are chosen to represent the numerical solution. For each time step is set 3 fixed multi-corrections. In short, the parallel preconditioned GMRES (see Algorithm 5) runs exactly 750 times to solve each experiment.

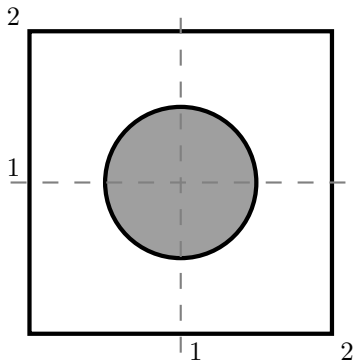


Figure 5.23 – Explosion: Problem statement.

An unstructured mesh with 531,166 nodes, 1,059,798 elements is used. Table 5.9 shows

the CPU time, bandwidth sizes, and memory usage concern our domain decomposition approach applied to divide the sequential mesh into 384 partitions using the RCM or Spectral algorithms. Besides, the reduction of bandwidth size can be noted when both reordering algorithms are used. As can be seen, the number of unknowns is about four times greater than the number of nodes because it is a problem with 4 degrees of freedom per node. The CPU time to obtain the domain decomposition using Spectral was also estimated – as in Problem 1, because the bandwidth size was not enough to divide the linear system order into 384 partitions.

Table 5.9 – Problem 3 : Domain Decomposition Approach - Bandwidth, Preprocessing CPU time, and Memory Usage.

	Bandwidth before reordering	Bandwidth after reordering	Time for Reordering (in seconds)	Total time to preprocess the domain decomposition into 384 partitions (in seconds)	Memory Usage (in MB)
Mesh (RCM)	2,114,275	5,255	6.10	54.58	2663
Mesh (Spectral)	2,114,275	10,727	7.73	56.21*	2094

*estimated CPU time: Spectral failed to split into 384 partitions.

Figure 5.24 presents the Problem 3 density solution at time $t = 0.25$ second, obtained with the preconditioner ILU0 and 24 MPI ranks. The graphic is plotted using Paraview software after postprocessing phase (see Section 4.3).

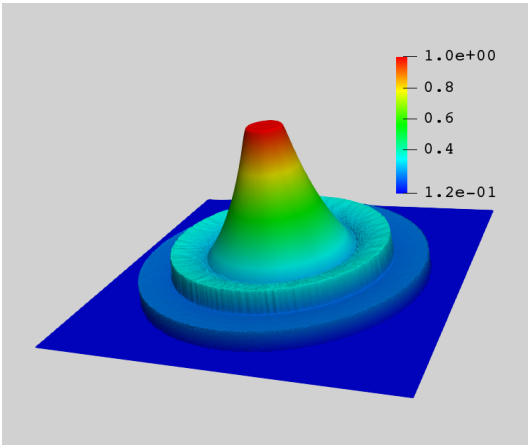


Figure 5.24 – Problem 3 : Density solution at $t = 0.25$ second with the preconditioner ILU0 and 24 MPI ranks.

Table 5.10 presents the total CPU time and average number of GMRES iterations per GMRES execution (iter_{av}) – the total number of GMRES iterations divided by the number of GMRES executions to solve the problem (750). This table shows, respectively, the parallel preconditioners BlockDiag$e$, BlockLUe, BlockSGSe, ILU0, and ILU1 considering 24, 48, 96, 192, 384 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Table 5.11, in turn, presents the same information, but the Spectral algorithm reorders the sparsity of the mesh until 192 partitions. No version of SPIKE preconditioner

is considered because preliminary tests demonstrated very high CPU time to perform this application. This behavior can be explained once, the number of point float operations in each partition is larger for matrices derived from problem with more than 1 degree of freedom per node, resulting in greater CPU time in the Preconditioner_Setup function.

As can be noted, the number of GMRES iterations performed by the BlockDIAG*e* preconditioner is basically constant. Such behavior is expected since this preconditioner is just a block scaling based on the main block diagonal of the matrix (see Subsection 2.5.3.1) and it has the same effect of preconditioning independent of the number of MPI ranks. Local preconditioners BlockLU*e* and BlockSGS*e* present a number of GMRES iterations ($\text{iter}_a v$) much smaller than that reached by the preconditioner BlockDIAG*e*, however, the CPU times are larger. According to our work (Muller et al., 2017), that fact occurs probably because of block scaling operations (see Subsection 2.5.3) demand a substantial runtime.

For this experiment, the number of GMRES iterations performed by ILU preconditioners increased only about 30% – small, if it is compared with ILU preconditioners for Problem 1, where number of GMRES iterations increased about 150%. CPU times of the Spectral algorithm cases are slightly smaller, but the RCM algorithm has the advantage to get a more significant number of partitions. BlockDiag*e* presented the best CPU times until 96 partitions, however, for 192 or more partitions preconditioner ILU0 became more advantageous.

Table 5.10 – Problem 3 - RCM reordering : CPU time and the average number of GMRES iterations from 24 to 384 partitions.

Preconditioner	Number of MPI ranks					
	24		48		72	
	time	iter_{av}	time	iter_{av}	time	iter_{av}
BlockDIAG*e*	334.54	13.7	161.75	13.8	109.28	13.7
BlockLU*e*	428.41	8.2	206.32	8.2	141.48	8.3
BlockSGS*e*	442.32	7.5	216.81	7.5	146.31	7.6
ILU0	382.23	7.4	195.75	8.0	130.36	8.3
ILU1	453.44	7.4	232.10	8.0	154.34	8.2
Preconditioner	Number of MPI ranks					
	96		192		384	
	time	iter_{av}	time	iter_{av}	time	iter_{av}
BlockDIAG*e*	83.81	13.7	49.83	13.7	35.26	13.7
BlockLU*e*	110.81	8.4	65.87	8.4	46.47	8.7
BlockSGS*e*	112.75	7.5	67.18	7.5	45.41	7.6
ILU0	98.50	8.6	47.57	9.3	23.35	10.5
ILU1	115.92	8.6	56.45	9.3	26.98	10.5

Figure 5.25 shows the CPU time, speedup, and efficiency for the Problem 3 considering 24, 48, 72, 96, 192, and 384 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Graphics are organized in groups of local and global preconditioners. Figure 5.26, in turn, shows the same information, but the Spectral algorithm reorders the sparsity of the mesh until 192 partitions. Local preconditioners BlockDIAG*e*, BlockLU*e*, and BlockSGS*e* present scalability until 192 partitions. However, for 384 partitions, the

Table 5.11 – Problem 3 - Spectral reordering : CPU time and the average number of GMRES iterations from 24 to 192 partitions.

Preconditioner	Number of MPI ranks					
	24		48		72	
	time	$iter_{av}$	time	$iter_{av}$	time	$iter_{av}$
BlockDIAG*e*	337.95	13.7	159.16	13.7	105.28	13.8
BlockLU*e*	427.61	7.9	201.62	8.0	134.20	8.1
BlockSGS*e*	444.15	7.5	210.86	7.5	139.46	8.5
ILU0	374.67	7.4	192.60	8.0	128.59	8.3
ILU1	449.01	7.4	228.99	8.0	152.79	8.3

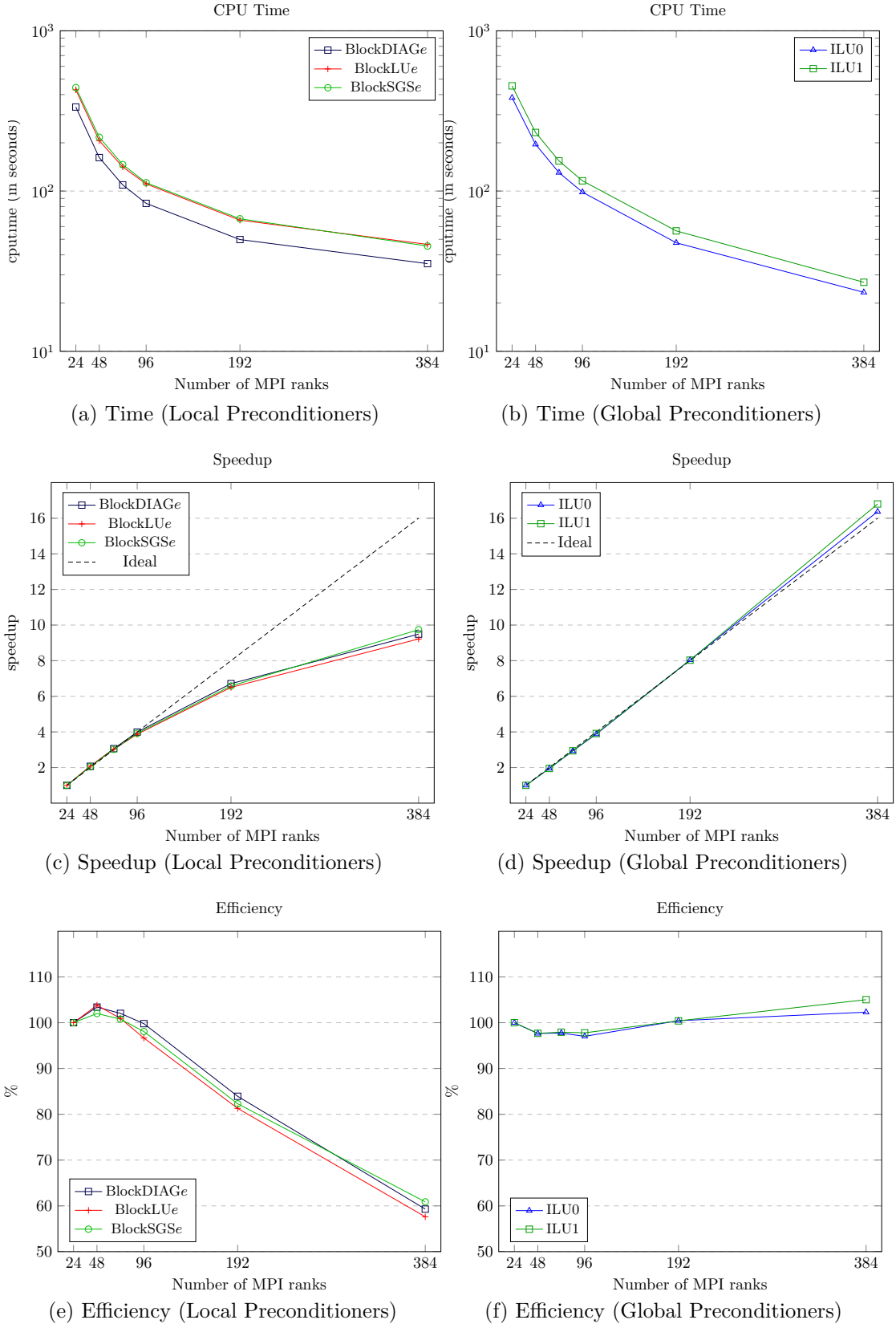
Preconditioner	Number of MPI ranks			
	96		192	
	time	$iter_{av}$	time	$iter_{av}$
BlockDIAG*e*	79.97	13.7	45.50	13.7
BlockLU*e*	102.75	8.1	59.89	8.2
BlockSGS*e*	106.16	7.5	61.36	7.5
ILU0	96.50	8.6	46.90	9.2
ILU1	114.32	8.5	55.44	9.2

speedup and the efficiency dropped significantly. Such behavior is related to the fact that EBE matrix-vector products for problems with `ndof` > 1 tend to suffer more with the overlapping generated by the greater number of partitions. As can be emphasized, BlockLU*e* and BlockSGS*e* reduced the number of iterations, but the performance is not substantial when compared with the preconditioner BlockDIAG*e*. Speedup and efficiency of ILU preconditioners point out good scalability.

Figure 5.27a shows the total memory usage for the parallel preconditioners BlockDIAG*e*, BlockLU*e*, BlockSGS*e*, ILU0, and ILU1 considering 24, 48, 72, 96, 192, and 384 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Figure 5.28a, in turn, shows the same information, but the Spectral algorithm reorders the sparsity of the mesh until 192 partitions. The difference between the total memory usage of global preconditioners and the total memory usage of local preconditioners becomes more evident. Global preconditioners use more memory during processing, however, this difference tends to decrease as the number of MPI ranks increases.

Figure 5.27b shows the bar min and max graph of average memory usage per node for the local preconditioner BlockDIAG*e* and the global preconditioner ILU0 considering 1, 2, 3, 4, and 8 processing nodes and the sparsity of the mesh reordered by the RCM algorithm. Figure 5.28b, in turn, shows the same information, but Spectral algorithm reorders the sparsity of the mesh. In this case, particularly, experiments with 192 MPI ranks are presented instead of 384. Again, different mesh reordering algorithms do not imply substantial changes in memory consumption. Also in this experiment, there is no large discrepancy between the minimum and maximum memory values used in each MPI rank.

Figure 5.29 shows the functions runtime analysis for the parallel preconditioners BlockDIAG*e*, BlockLU*e*, BlockSGS*e*, ILU0, and ILU1 considering 24 and 384 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Figure 5.30, in turn, presents



(a) Time (Local Preconditioners)

(b) Time (Global Preconditioners)

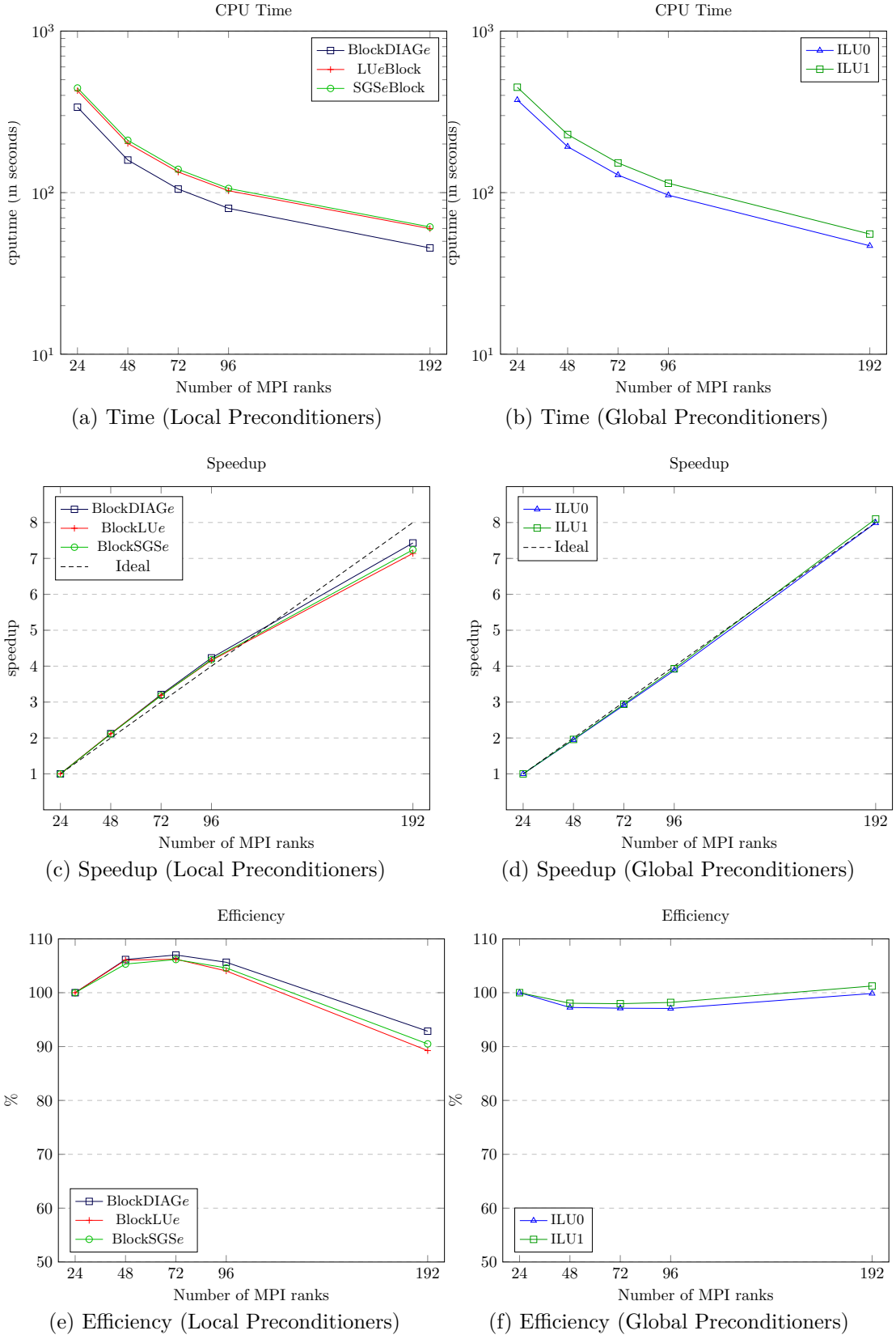
(c) Speedup (Local Preconditioners)

(d) Speedup (Global Preconditioners)

(e) Efficiency (Local Preconditioners)

(f) Efficiency (Global Preconditioners)

Figure 5.25 – Problem 3 - Reordering RCM : CPU time, speedup, and efficiency.



(a) Time (Local Preconditioners)

(b) Time (Global Preconditioners)

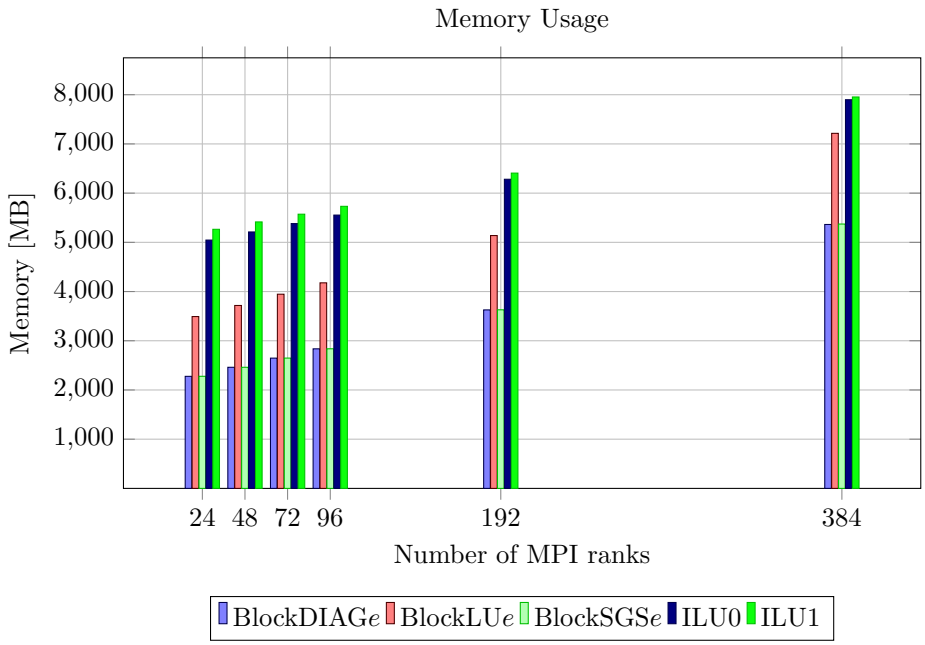
(c) Speedup (Local Preconditioners)

(d) Speedup (Global Preconditioners)

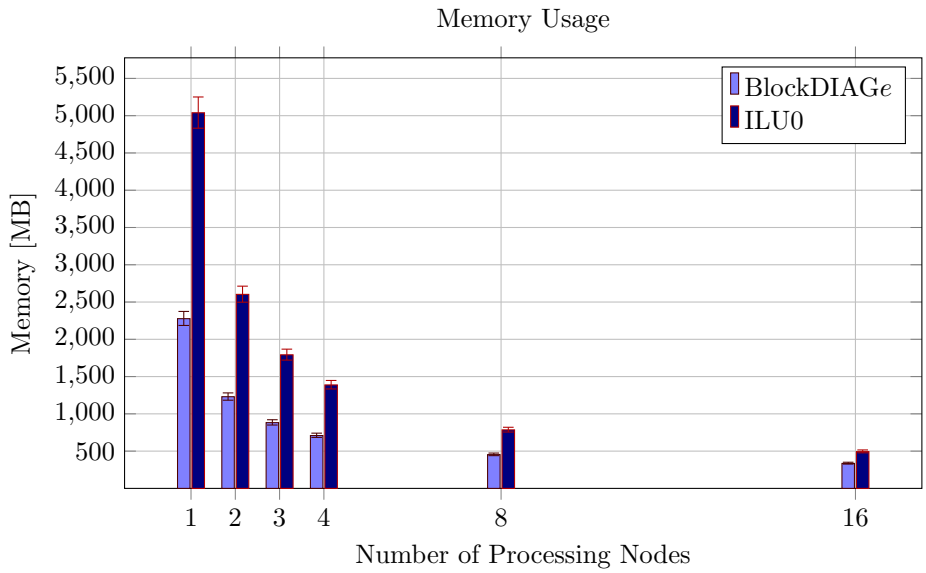
(e) Efficiency (Local Preconditioners)

(f) Efficiency (Global Preconditioners)

Figure 5.26 – Problem 3 - Reordering Spectral : CPU time, speedup, and efficiency.

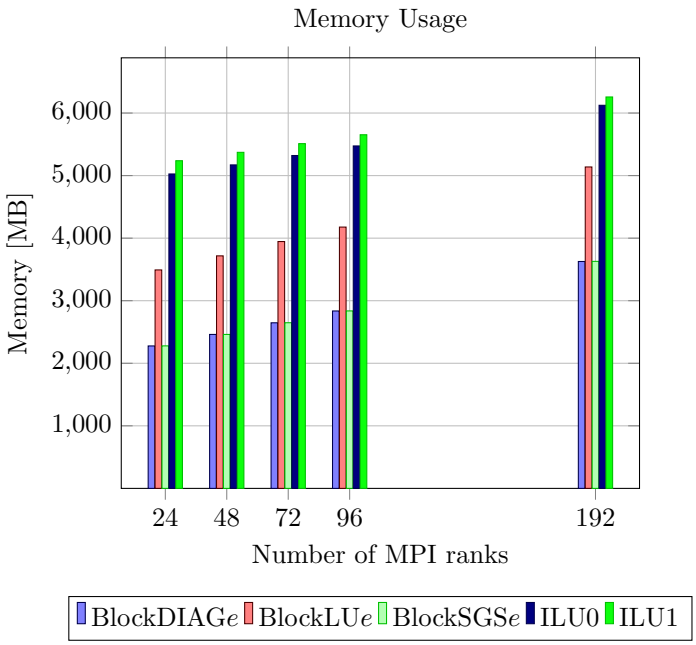


(a) Total Memory Usage (All Preconditioners)

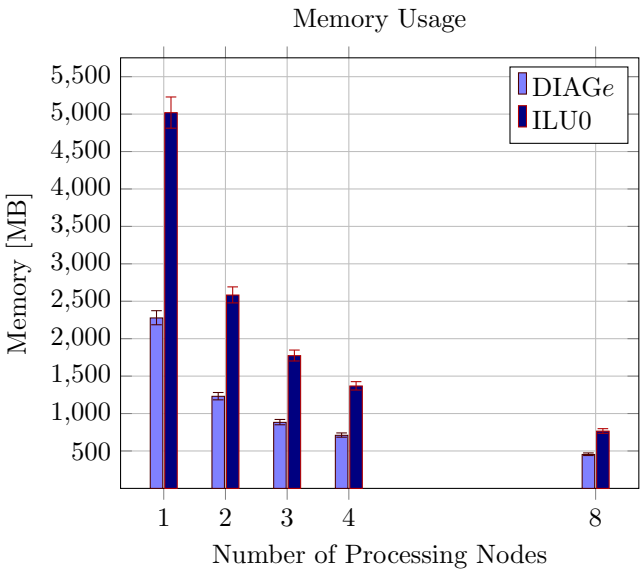


(b) Average Memory Usage per Node (BlockDIAG*e* and ILU0)

Figure 5.27 – Problem 3 - Reordering RCM : Memory Usage.



(a) Total Memory Usage (All Preconditioners)

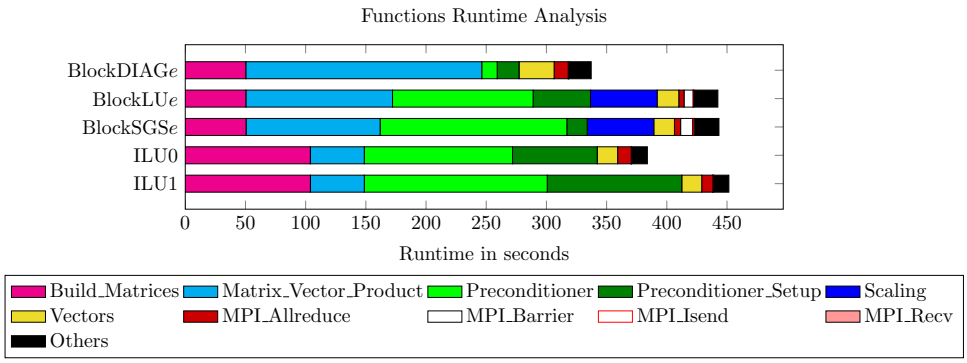


(b) Average Memory Usage per Node (BlockDIAG*e* and ILU0)

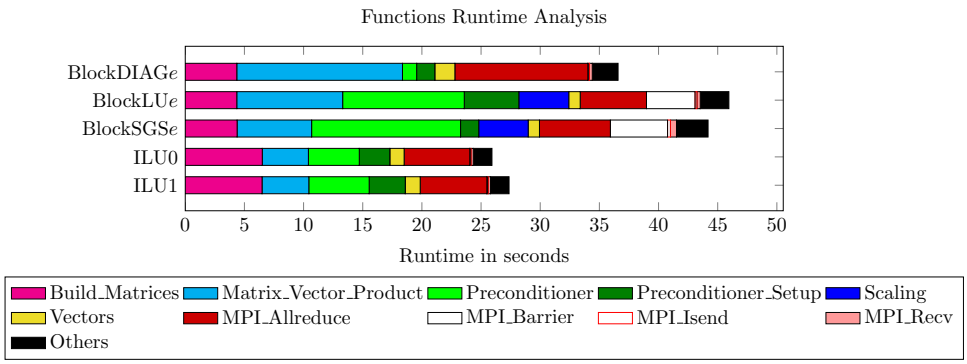
Figure 5.28 – Problem 3 - Reordering Spectral : Memory Usage.

the same information, but the Spectral algorithm reorders the sparsity of the mesh until 192 partitions. When 24 MPI ranks are considered, BlockDIAG*e* present the best overall runtime even its Matrix_Vector_Product demands the greatest runtime. As the number of MPI ranks increases, ILU preconditioners become the best options, because the runtimes of the Preconditioner and Preconditioner_Setup ILU functions decrease significatively. Function Build_Matrices keeps similar behavior as in Problem 2 – Build_Matrices is slower for global preconditioners because CSR assembly is more complex (see Subsection 4.1.4). Note the Matrix_Vector_Product function demands a greater runtime for local preconditioners, i.e., in our implementation, CSR matrix-vector products are more efficient than EBE ones.

The local preconditioners BlockLU*e* and BlockSGS*e* reduce the number of GMRES iterations (see Tables 5.10 and 5.11) but, do not reduce the execution time. The runtime of the functions Scaling, MPI_Barrier and even of the functions Preconditioner and Preconditioner_Setup make these local preconditioners a bad choice to solve this experiment. As in the previous experiments, MPI_Allreduce runtime is proportionally larger for a high number of MPI ranks. Runtimes of MPI_Isend and MPI_Recv can also be considered proportionally tiny. RCM and Spectral reordering methods also do not present significant differences in the functions runtime analysis.

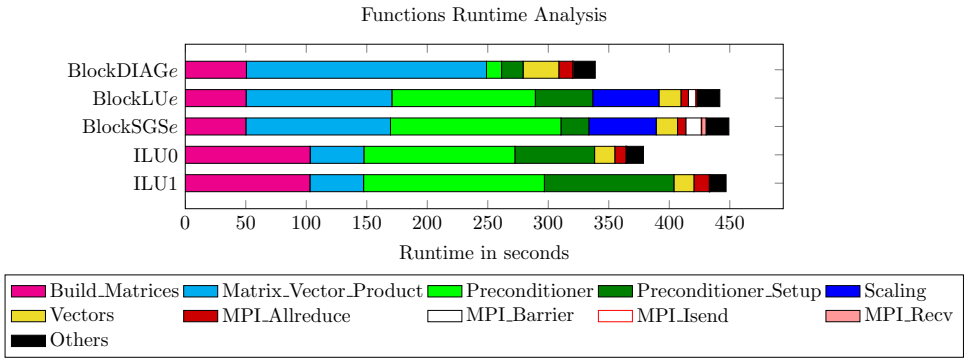


(a) Functions analysis for 24 MPI ranks

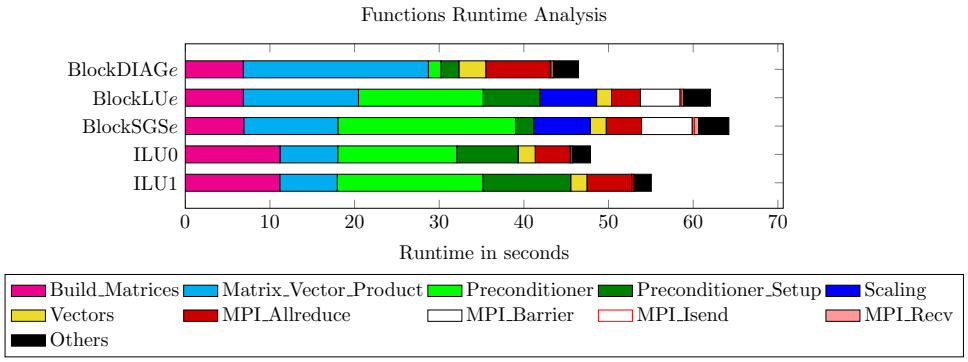


(b) Functions analysis fort 384 MPI ranks

Figure 5.29 – Problem 3 - Reordering RCM : Functions runtime analysis.



(a) Functions analysis for 24 MPI ranks



(b) Functions analysis for 192 MPI ranks

Figure 5.30 – Problem 3 - Reordering Spectral : Functions runtime analysis.

5.6 Wind tunnel - Problem 4 : Transient case with `ndof=4`

The last problem considered is another transient problem with 4 degrees of freedom per node and also modeled by the Euler Equations. This two-dimensional test problem was initially introduced by Emery (1968) and since then this problem has proven to be a useful test for a large number of methods in fluid dynamics. The computational domain is shown in Fig. 5.31 and the inflow data on the left boundary is set up by

$$\text{inflow} \begin{cases} M &= 3.0 \\ \rho &= 1.4 \\ u &= 3.0 \\ v &= 0.0 \\ p &= 1.0 \end{cases} . \tag{5.2}$$

Along the walls of the tunnel reflecting boundary conditions are applied, and no boundary condition is imposed at the outflow boundary. The time advancing is solved using the parallel predictor-multi-corrector algorithm as described in Algorithm 4. The final time $t_{final} = 0.5$ and the time step size $\Delta t = 10^{-4}$ are chosen to represent the numerical solution. For each time step is set 5 fixed multi-corrections. In short, the parallel preconditioned GMRES (see Algorithm 5) runs exactly 25000 times to solve each experiment.

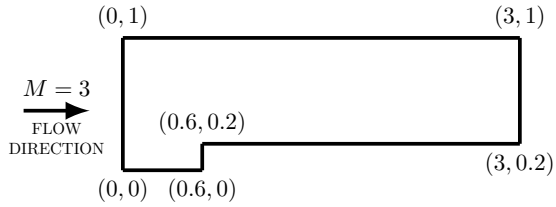


Figure 5.31 – Wind tunnel: statement problem.

An unstructured mesh with 534,143 nodes and 1,065,084 elements is used. Table 5.6 shows the CPU time, bandwidth sizes, and memory usage concern our domain decomposition approach applied to divide the sequential mesh into 384 partitions using the RCM algorithm to reorder the sparsity of the mesh. When `ndof` > 1 the sparsity of the linear system does not coincide with the sparsity of the mesh (see example in Figs. 3.1, 3.2, 3.3, and 3.4). Despite that, our domain decomposition approach allows reducing matrix bandwidth significantly.

Table 5.12 – Problem 4 : Domain Decomposition Approach - Bandwidth, Preprocessing CPU time, and Memory Usage.

	Bandwidth before reordering	Bandwidth after reordering	Time for Reordering (in seconds)	Total time to preprocess the domain decomposition into 384 partitions (in seconds)	Memory Usage (in MB)
Mesh (RCM)	2,128,947	2,890	6.16	50.77	2698

Figure 5.32 presents the Problem 4 density solution at time $t = 0.5$ second, obtained with the preconditioner ILU0 and 24 MPI ranks. The graphic is plotted using Paraview software after postprocessing phase (see Section 4.3).

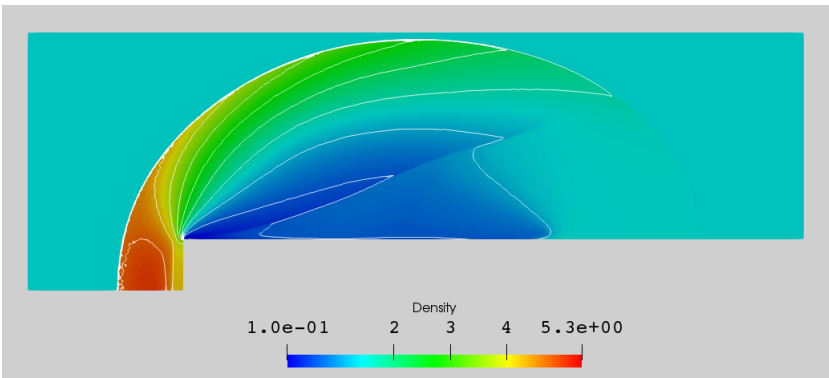


Figure 5.32 – Problem 4 : Density solution at $t = 0.5$ second with the preconditioner ILU0 and 24 MPI ranks.

Table 5.13 presents the total CPU time and the average number of GMRES iterations per execution (iter_{av}) – the total number of GMRES iterations divided by the number of GMRES executions to solve the problem (25000). This table, shows, respectively, the parallel preconditioners BlockDIAGe, ILU0, and ILU1 considering 24, 48, 72, 96, 192, 384

MPI ranks and the sparsity of the mesh reordered by RCM algorithm. iter_{av} is practically constant for BlockDIAG*e* preconditioner and presents a small increment for ILU0 and ILU1 cases when the number of MPI ranks increases. The CPU time for ILU0 preconditioner was smaller when compared with BlockDIAG*e*. For 384 MPI ranks, ILU0 was almost twice faster. Increasing *fill-in* level from 0 to 1 was not enough to achieve better CPU time.

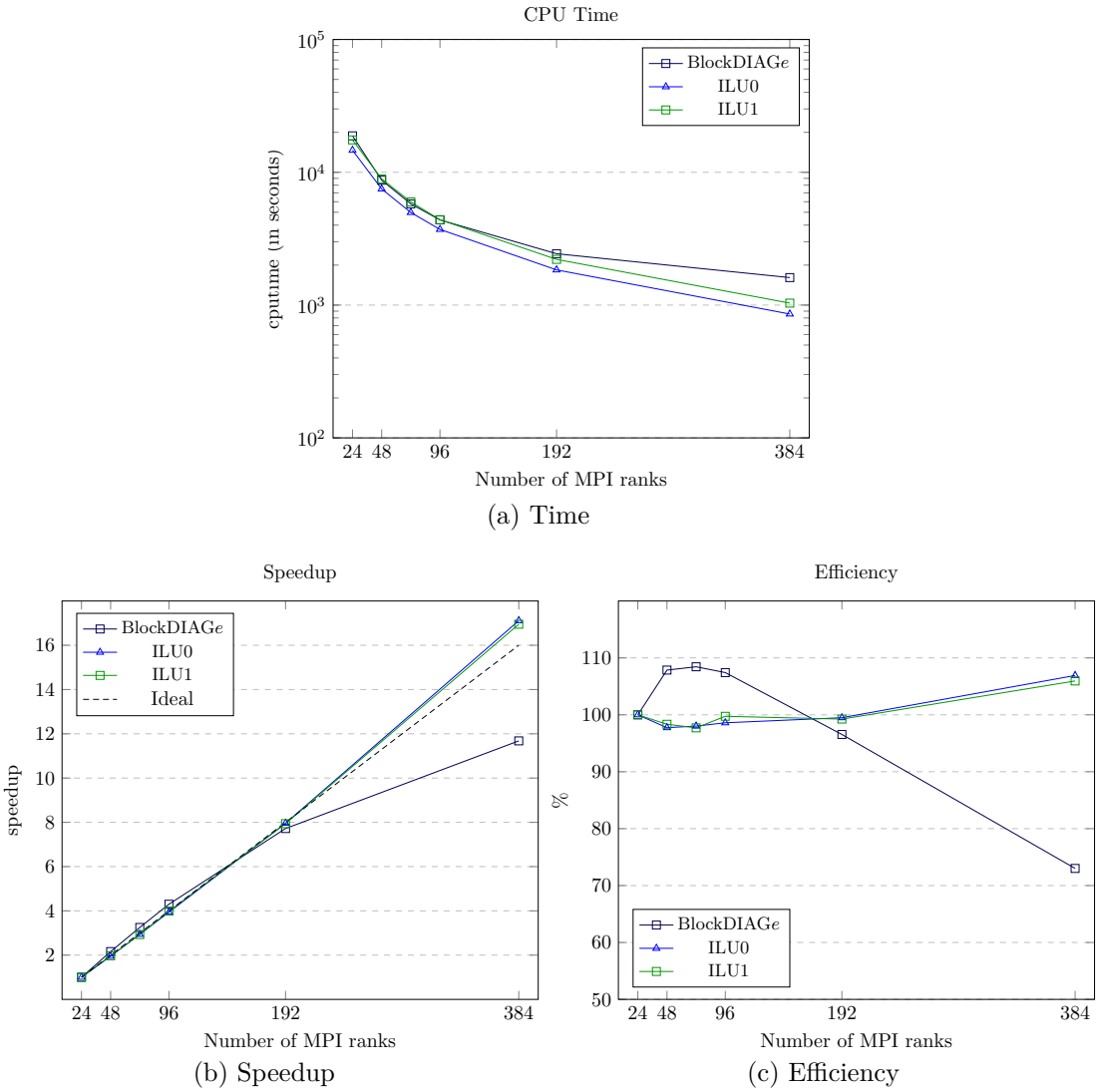
Table 5.13 – Problem 4 - RCM reordering: CPU time and the average number of GMRES iterations from 24 to 384 partitions.

Preconditioner	Number of MPI ranks					
	24		48		72	
	time	iter_{av}	time	iter_{av}	time	iter_{av}
BlockDIAG*e*	18813.94	26.4	8721.81	26.5	5783.12	26.5
ILU0	14651.05	10.3	7494.81	10.9	4982.09	11.1
ILU1	17535.63	10.2	8915.44	10.8	5981.97	11.0
Preconditioner	Number of MPI ranks					
	96		192		384	
	time	iter_{av}	time	iter_{av}	time	iter_{av}
BlockDIAG*e*	4378.62	26.5	2435.90	26.5	1610.61	26.5
ILU0	3714.72	11.2	1840.70	12.4	856.47	13.0
ILU1	4395.41	11.1	2208.41	12.3	1034.39	12.9

Figure 5.33 shows the CPU time, speedup, and efficiency for the Problem 4 considering 24, 48, 72, 96, 192, and 384 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. As can be noted, preconditioner BlockDIAG*e*'s speedup and efficiency decrease as the number of MPI ranks grows. On the other hand, speedup and efficiency of ILU preconditioners improve as the number of MPI ranks increases. In short, ILU preconditioners point out to better scalability when compared to preconditioner BlockDIAG*e*.

Figure 5.34a shows the total memory usage for the parallel preconditioners BlockDIAG*e*, ILU0, and ILU1 considering 24, 48, 72, 96, 192, and 384 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. The total memory usage of global preconditioner is larger than of local preconditioner. However, this difference tends to decrease as the number of MPI ranks increases. Figure 5.34b shows the bar min and max graph of average memory usage per node for the local preconditioner BlockDIAG*e* and the global preconditioner ILU0 considering 1, 2, 3, 4, 8 and 16 processing nodes and the sparsity of the mesh reordered by the RCM algorithm. Again, there is no significant discrepancy between the minimum and maximum memory values used in each MPI rank what suggests a good load balancing independent of the storage scheme.

Figure 5.35 shows the functions runtime analysis for the parallel preconditioners BlockDIAG*e* and ILU0, considering 24 and 384 MPI ranks and the sparsity of the mesh reordered by the RCM algorithm. Runtimes of BlockDIAG*e* functions Preconditioner and Preconditione_Setup are substantially smaller when compared with corresponding ILU runtime functions. However, one more time, ILU Matrix_Vector_Product runtime is smaller smaller and offsets the BlockDIAG*e* overall runtime. Again, MPI_Allreduce



(a) Time

(b) Speedup

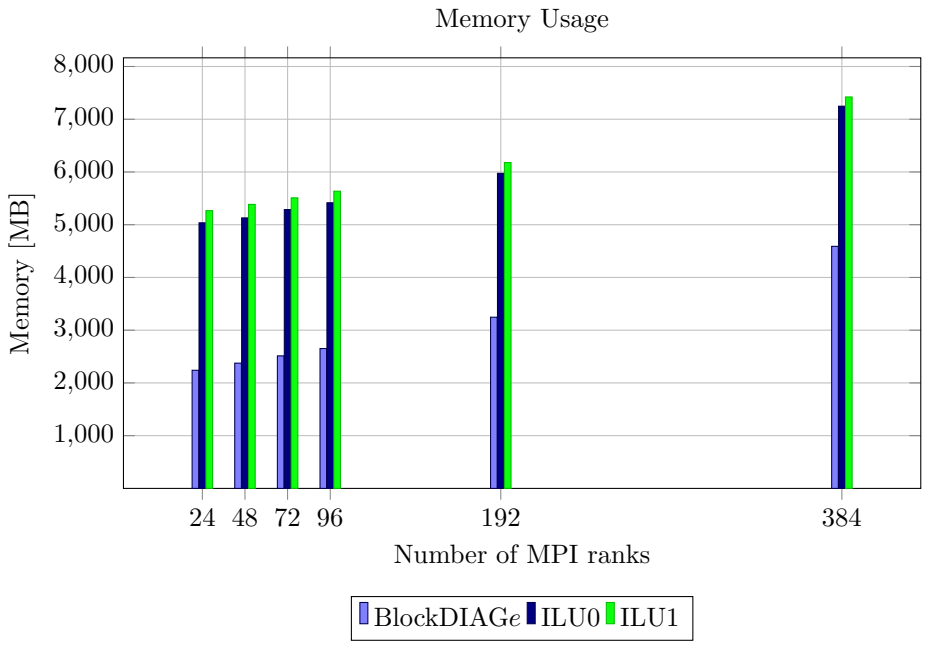
(c) Efficiency

Figure 5.33 – Problem 4 - Reordering RCM : CPU time, speedup, and efficiency.

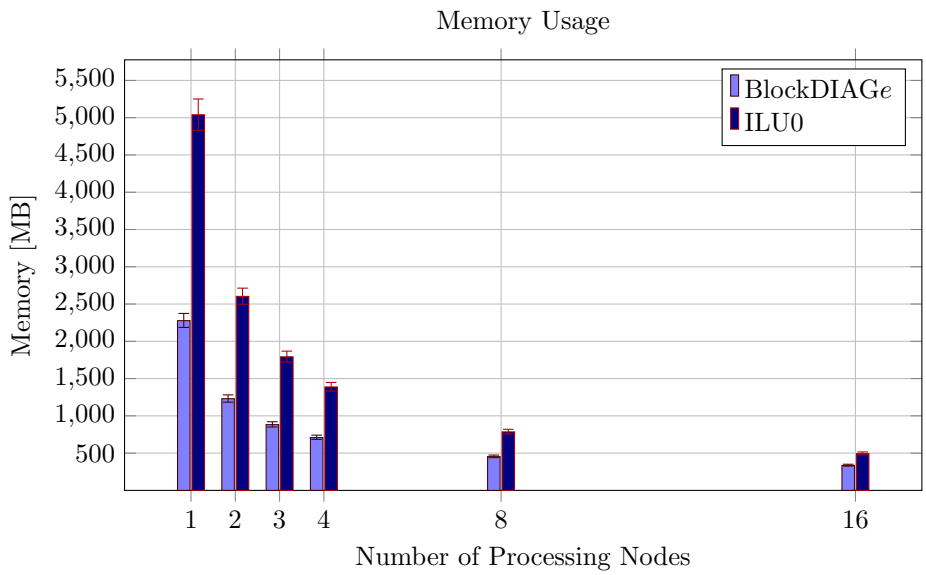
runtime grows as the number of MPI ranks increases. Runtimes of MPI_Isend and MPI_Recv, as in the previous experiments, are almost undetectable.

5.7 Considerations about the preconditioners tested

Although our domain decomposition approach was initially designed for the global preconditioner SPIKE (Lima et al., 2016), it was possible to extend this approach to the context of other global preconditioners such as incomplete LU factorization. More than that, this extension also included local preconditioners. In the following we give a summary of the behavior of each of the preconditioners applied to our domain decomposition approach.

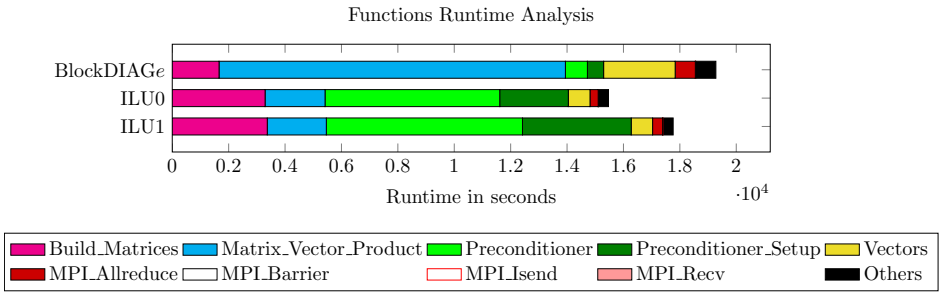


(a) Total Memory Usage (All Preconditioners)

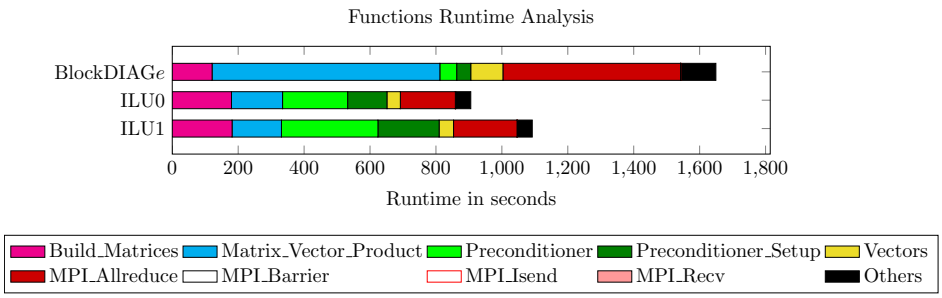


(b) Average Memory Usage per Node (BlockDIAG*e* and ILU0)

Figure 5.34 – Problem 4 - Reordering RCM : Memory Usage.



(a) Functions analysis for 24 MPI ranks



(b) Functions analysis for 384 MPI ranks

Figure 5.35 – Problem 4 - Reordering RCM : Functions runtime analysis.

5.7.1 Preconditioner DIAG*e*

The preconditioner DIAG*e* consists primarily of a matrix scaling, that is, a division of each matrix coefficient by their corresponding coefficient in the diagonal matrix. In consonance with Wathen (2015), preconditioners as DIAG*e*, which are a type of rescaling of the matrix, present good results when used in a well-conditioned matrix which is just poorly scaled. Unfortunately, in general, diagonal preconditioning is likely to achieve very little regarding reducing iterations or computation time. Besides, there are situations where the DIAG*e* does not reduce the number of iterations for an acceptable range. On the other hand, Rychkov (2017) reports the effect that a functional reorganization of the elements can provide in the execution of the product matrix-vector element by element, particularly in the scalability for parallel computation. This condition can be noted in Problem 1: the speedup reaches values much higher than expected as the size of the mesh increases. For larger partitions, the disorder of the elements possibly generated a bad use of the cache memory (See Figs. 5.3c, 5.4c, and 5.5c). For the Problem 2, DIAG*e* presented scalability, but the large number of GMRES iterations did not allowed better performance (see Tables 5.7 and 5.8).

5.7.2 Preconditioner LU*e*

LU*e* is a local preconditioner based on LU factorization of each element matrix and developed for problems with `ndof` = `1`. This preconditioner did not present the best performance when applied in a steady-state problem (Problem 1), for a more refined

mesh (see Tables 5.4 and 5.5). However, for smaller meshes, it was the best representative of local preconditioners (see Tables 5.2 and 5.3). For the transient problem modeled by transport equation (Problem 2) , preconditioner LUe provided the best performance of all (see Tables 5.7 and 5.8). The preconditioner LUe as well as the preconditioner DIAGe presented superlinear speedups (see Figs. 5.3c, 5.4c, 5.5c, 5.17c, and 5.18c).

5.7.3 Preconditioner SGSe

SGSe is also a local preconditioner developed for problems with `ndof` = `1`. It is based on Gauss-Seidel factorization of each element matrix. In general form, it assumed an intermediary position among local preconditioners since its performance was better than DIAGe but worst than LUe. However, for larger meshes, SGSe was the best local preconditioner representative.

5.7.4 Preconditioner BlockDIAGe

The preconditioner BlockDIAGe can be considered a natural expansion of DIAGe for `ndof` > `1`. The matrix diagonal is seen as a diagonal of blocks, where each block has size `ndof` × `ndof`. However, unlike what was observed for problems with `ndof` = `1`, BlockDIAGe achieved more expressive performance than the other local preconditioners per block, namely BlockLUe and BlockSGSe (See Figs. 5.25a and 5.26a). BlockDIAGe still showed scalability up to a certain number of partitions. However, above 192 MPI ranks, it showed a marked drop in speedup and efficiency (see Figs. 5.25, 5.25, and 5.33). BlockDIAGe could be stored in structures based on edges instead of elements, since the result of the preconditioning obtained by both is equivalent. Coutinho et al. (2001) suggested edge-by-edge storage combined with preconditioner BlockDIAGe which presented expressive results (Lins et al., 2009).

5.7.5 Preconditioners BlockLUe and BlockSGSe

BlockLUe and BlockSGSe are the representatives of LUe and SGSe for problems with `ndof` > `1`. As opposite of local preconditioners with $ndof = 1$, these preconditioners did not overcome. They reduced the number of GMRES iterations considerably, but not the runtime when compared with the BlockDIAGe preconditioner.

5.7.6 Incomplete LU Factorizations Preconditioners

ILU preconditioners became an excellent surprise. Many authors question the feasibility of building an efficient parallel ILU preconditioner. In the book Iterative Krylov Methods for Large Linear Systems by Vorst (2003), the following statement can be found:

> *"For large enough problem sizes the inner products, vector updates, and matrix-vector product are easily parallelized and vectorized. The more successful preconditionings, i.e., based upon incomplete LU decomposition, are not easily parallelizable. For that reason often the use of only diagonal scaling as a preconditioner on highly parallel computers."*

Similarly, in the book Parallelism in Matrix Computations by Gallopoulos et al. (2016), authors also state:

> *"Compared to the parallel banded LU-factorization schemes in the existing literature, the Spike algorithm reduces the required memory references and interprocessor communications at the cost of performing more arithmetic operations... A distinct feature of the Spike algorithm is that it avoids the scalable parallel scheme of obtaining the classical global LU factorization of the narrow-banded coefficient matrix".*

Despite these observations, our domain decomposition approach has provided the almost immediate adequation of the sequential ILU preconditioners to the context of finite element parallel applications. Such adaptations applied to sequential incomplete LU factorizations lead to a kind of block Jacobi with a local ILU factorizations. In addition, ILU preconditioners provided the best results for the experiments performed. In general, the best option would be to choose the ILU preconditioner with the lowest possible *fill-in* level associated with the mesh reordering obtained by the RCM algorithm. That is because RCM is more effective to reduce the bandwidth of the associated linear system, which for our domain decomposition approach means more partitions. In short, ILU preconditioner with lower *fill-in* level associated with the greatest number of MPI ranks would result in the shortest runtime for the 2D finite element applications. It is worth mentioning that such adaptations applied to sequential incomplete LU preconditioners leads to a kind of Jacobi block with local ILU factorizations and not a strictly parallel form of a ILU preconditioner.

5.7.7 SPIKE Preconditioners

SPIKE preconditioners as described in (Sathe et al., 2012) demonstrated robustness and scalability to solve linear systems from problems of the most miscellaneous application areas (Manguoglu et al., 2010). Thanks to a group of combinatorial techniques that can be applied in the matrices if the linear systems (see Subsection 2.4.1). However, the application of these combinatorial techniques required a considerable runtime (Lima et al., 2017), which would make almost infeasible to use the SPIKE preconditioner in the context where the matrices should be reconstructed (for example, nonlinear or transient finite element problems).

Unlike ILU preconditioners, the reordering of A_i blocks does not present gains for the SPIKE preconditioners, that is because the complete LU factorization proposed for this preconditioner is made using the software PARDISO as a black box, where the A_i blocks (see Fig. 4.4) are inserted in the CSR storage format, regardless of its sparsity pattern. SPIKE preconditioners presented regular performance only for Problem 1 (steady-state case). SPIKE performance was equivalent to ILU preconditioners, just when 8 threads per MPI process are considered. However, the number of processing nodes required by SPIKE preconditioners was four times higher to achieve such performance. Concerning the time-advancing problem (Problem 2), results are even less expressive, since the processing time to obtain the solution of such problems with SPIKE preconditioner is at least an order of magnitude higher than the other preconditioners.

5.8 Considerations about memory usage, load balance, and MPI communications

Local preconditioners consumed less memory than global preconditioners. Among the local preconditioners, LUe and BlockLUe consumed more memory, because they need to store the LU factorization of each element matrix. SPIKE preconditioners require a massive amount of memory since they store the complete factorizations of the global finite element matrices, for each rank, in addition to the coupling blocks (these blocks are dense with size $k \times k$, see Subsection 2.4). ILU preconditioners memory usage, in turn, depends on the *fill-in* level chosen.

Our domain decomposition approach is based on the division of the mesh data taking into account the sparsity pattern of the linear system associated with the two-dimensional finite element problems. More precisely, chains-on-chains partitioning (CCP) (Pınar and Aykanat, 2004) algorithm has been used. That algorithm aims to achieve load balancing according to the number of non-zero coefficients of the linear system matrix derived from the finite element discretization. As can be seen in the column charts entitled "Average Memory Usage per Node", the memory usage in each MPI rank is balanced for both the global preconditioners (CSR storage) and local preconditioners (EBE storage).

All graphics entitled "Functions Runtime Analysis" provide an evaluation of the runtime of main MPI routines, that is, MPI_Allreduce, MPI_Barrier, MPI_Isend, and MPI_Recv. As can be observed, MPI_Allreduce is the only routine whose runtime stands out (MPI_Barrier also stands out, but only in specific cases involving the local preconditioners LUe, SGSe, BlockLUe, and BlockSGSe because they require additional synchronizations). As well known, MPI_Allreduce is performed only after an inner product which depends on the number of MPI ranks and the number of iterations of the solver. On the other hand, the proportionality of the routines MPI_Isend and MPI_Recv is almost

imperceptible in the charts (MPI_Recv also stands out in SPIKE preconditioners cases because it depends on additional communication for coupling blocks, see Section 2.4). That means the domain decomposition approach is efficient enough to reduce the message exchanges which need to be performed by the update function (see Algorithm 2).

5.9 Considerations about the domain decomposition

The domain decomposition as presented in Section 3.2 provided good results when associated with convenient preconditioning. Moreover, that approach can be applied in $O(n)$ time and $O(n + \frac{n}{\texttt{ndof}})$ memory usage, where n is the order of the linear system associated with finite element discretization and `ndof` is the number of degrees of freedom per node. This fact can be explained once the algorithm demands one list to store a linear system pattern sparsity and another one to store the mesh pattern sparsity.

More specifically, the CPU time to preprocess the sequential mesh was relatively small compared to the overall runtime for processing the parallel application. Problem 4, for example, demanded just 50 seconds to apply our domain decomposition (see Table 5.12) while the parallel finite element processing using 384 MPI ranks required more than 800 seconds (see Table 5.13). The quantity of partitions is another issue. If p is admitted as the maximum number of partitions, p is limited by the quotient between the matrix order n and the linear system bandwidth bw. In Problem 4, for example, since the matrix order is $n = 2,128,947$ and bandwidth is $bw = 2,890$, it is possible to get up to $\frac{2,128,947}{2,890} = 736$ as the maximum number of partitions.

6 Conclusions

6.1 Review of results

An alternative domain decomposition approach for finite element analysis that provides robustness and scalability has been proposed. Due to this proper domain decomposition, finite element discretizations lead to narrow banded linear systems that can be preconditioned using either global preconditioners (CSR storage) or local preconditioners (Element-by-Element storage). Versions of sequential preconditioners as DIAGe, LUe, SGSe, BlockDIAGe, BlockLUe, BlockSGSe, and ILUm can be easily adjusted for parallel computing. We also consider the SPIKE preconditioner – a sophisticated parallel preconditioners based on narrow banded linear systems. Two reordering algorithms enabled our approach, namely, reverse Cuthill-McKee (RCM) (Liu and Sherman, 1976) and Spectral (Barnard et al., 1995).

The local preconditioner DIAGe did not present a considerable reduction in the number of GMRES iterations and CPU time, but presented scalability. LUe and SGSe preconditioners achieved better CPU times when applied to the transient problem. The BlockDIAGe preconditioner, applied to problems with `ndof` > 1, has been well adjusted to the parallel finite element processing since it presented favorable CPU time and scalability. The local preconditioners BlockLUe and BlockSGSe, developed for problems with `ndof` > 1, did not present good results: although they have reduced the number of GMRES iterations, their overall runtimes exceeded acceptable values.

Our approach enabled an almost straightforward adaptation of the sequential ILU preconditioners to the context of parallel computing. Moreover, ILU preconditioners have presented good scalability, for steady-state or transient problems, considering one or more degrees of freedom per node – as emphasized previously, our ILU preconditioner version is a kind of block Jacobi preconditioner with local ILU factorizations and not a strictly parallel form of the ILU preconditioner. On the other hand, SPIKE preconditioners demonstrated to be unsuitable for such finite element applications because it demanded excessive runtime and memory usage to perform the complete LU factorizations, particularly for transient problems.

In general, RCM and Spectral reordering schemes presented similar results regarding the preconditioning obtained for the set of experiments studied in this work. However, the RCM algorithm can be considered more efficient since it allowed to use a larger number of MPI ranks.

6.2 Future work

In this work, we proposed a domain decomposition approach for 2D parallel preconditioning finite element problems allowing all phases of the parallel computation indeed occurs in parallel. The domain decomposition permits: the reading of the finite element data mesh, the build of the finite element matrices and solutions of the preconditioned linear systems, and the postprocessing where the output data files are generated to be visualized by Paraview software. Our approach has well adapted to parallel preconditioners using CSR and EBE storage schemes. In addition, it guaranteed load balancing and a significant reduction in the size of MPI messages.

In order to contribute to continuous improvement in the development of our approach and even the implementation of the related algorithms, we suggest some future work:

- To extend the domain decomposition approach to computational fluid dynamics problems governed by Navier-Stokes Equations. To employ auxiliary preconditioning, as proposed in (Elman, 1999), for assuring GMRES effectivity when either the discretization mesh size or the viscosity approaches to zero;
- To use the PARDISO software for solving ILU triangular linear systems;
- To use Graphical Processing Unit (GPU) for accelerating matrix-vector products. Bertaccini and Durastante (2018) state that GPU combined with CSR storage is the fastest way to perform the matrix-vector products for preconditioned large and sparse linear systems;
- To use OpenMP for assembling finite element matrices. Guo et al. (2015) suggests a suitable form to assembly finite element matrices where an OpenMP parallel algorithm uses graph coloring to identify independent sets of elements that can be assembled concurrently with no race conditions;
- To extrapolate the domain decomposition approach to contexts where meshes are so large that they do not fit into a single processing node as stated in (Devine et al., 2009)(Sahni et al., 2009). In particular, to 3D finite element applications.

Bibliography

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: ACM. *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967. p. 483–485.

ANZT, H.; HUCKLE, T. K.; BRÄCKLE, J.; DONGARRA, J. Incomplete sparse approximate inverses for parallel preconditioning. *Parallel Computing*, Elsevier, v. 71, p. 1–22, 2018.

ARBENZ, P.; GANDER, W. A survey of direct parallel algorithms for banded linear systems. *ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Wissenschaftliches Rechnen*, ETH Zurich, v. 221, 1994.

AYACHIT, U. The paraview guide: a parallel visualization application. Kitware, Inc., 2015.

BALAY, S.; ABHYANKAR, S.; ADAMS, M.; BROWN, J.; BRUNE, P.; BUSCHELMAN, K.; DALCIN, L.; EIJKHOUT, V.; GROPP, W.; KAUSHIK, D. et al. *PETSc Users Manual Revision 3.8*, 2017.

BARNARD, S. T.; POTHEN, A.; SIMON, H. A spectral algorithm for envelope reduction of sparse matrices. *Numerical linear algebra with applications*, Wiley Online Library, v. 2, n. 4, p. 317–334, 1995.

BAZILEVS, Y.; CALO, V. M.; TEZDUYAR, T. E.; HUGHES, T. J. YZβ discontinuity capturing for advection-dominated processes with application to arterial drug delivery. *International Journal for Numerical Methods in Fluids*, Wiley Online Library, v. 54, n. 6-8, p. 593–608, 2007.

BENTO, S. S.; LIMA, L. M. de; SEDANO, R. Z.; CATABRIGA, L.; SANTOS, I. P. A nonlinear multiscale viscosity method to solve compressible flow problems. In: SPRINGER. *International Conference on Computational Science and Its Applications*, 2016. p. 3–17.

BENZI, M. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, Elsevier, v. 182, n. 2, p. 418–477, 2002.

BERTACCINI, D.; DURASTANTE, F. Iterative methods and preconditioning for large and sparse linear systems with applications. Chapman and Hall/CRC, 2018.

BROOKS, A. N.; HUGHES, T. J. Streamline upwind/Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations. *Computer methods in applied mechanics and engineering*, Elsevier, v. 32, n. 1-3, p. 199–259, 1982.

BULUÇ, A.; MEYERHENKE, H.; SAFRO, I.; SANDERS, P.; SCHULZ, C. Recent advances in graph partitioning. In: *Algorithm Engineering*: Springer, 2016. p. 117–158.

CAI, X.-C.; SAAD, Y. Overlapping domain decomposition algorithms for general sparse matrices. *Numerical linear algebra with applications*, Wiley Online Library, v. 3, n. 3, p. 221–237, 1996.

CAI, X.-C.; SARKIS, M. A restricted additive schwarz preconditioner for general sparse linear systems. *Siam journal on scientific computing*, SIAM, v. 21, n. 2, p. 792–797, 1999.

CATABRIGA, L.; MARTINS, M. A.; COUTINHO, A. L. G. A.; ALVES, J. L. Clustered edge-by-edge preconditioners for non-symmetric finite element equations. In: CITESEER. *4th World Congress on Computational Mechanics*, 1998.

CHAN, T. F.; CHOW, E.; SAAD, Y.; YEUNG, M.-C. Preserving symmetry in preconditioned Krylov subspace methods. *SIAM Journal on Scientific Computing*, SIAM, v. 20, n. 2, p. 568–581, 1998.

CHEN, C.; TAHA, T. M. A communication reduction approach to iteratively solve large sparse linear systems on a GPGPU cluster. *Cluster Computing*, Springer, v. 17, n. 2, p. 327–337, 2014.

CHOW, E.; SAAD, Y. ILUS: an incomplete lu preconditioner in sparse skyline format. *International Journal for Numerical Methods in Fluids*, Wiley, v. 25, n. 7, p. 739–748, 1997.

COUTINHO, A. L.; MARTINS, M. A.; ALVES, J. L.; LANDAU, L.; MORAES, A. Edge-based finite element techniques for non-linear solid mechanics problems. *International Journal for Numerical Methods in Engineering*, Wiley Online Library, v. 50, n. 9, p. 2053–2068, 2001.

DAVIS, T. A.; RAJAMANICKAM, S.; SID-LAKHDAR, W. M. A survey of direct methods for sparse linear systems. *Acta Numerica*, Cambridge University Press, v. 25, p. 383–566, 2016.

DAYDÉ, M. J.; L'EXCELLENT, J.-Y.; GOULD, N. I. Preprocessing of sparse unassembled linear systems for efficient solution using element-by-element preconditioners. In: SPRINGER. *European Conference on Parallel Processing*, 1996. p. 34–43.

DEVINE, K.; DIACHIN, L.; KRAFTCHECK, J.; JANSEN, K.; LEUNG, V.; LUO, X.; MILLER, M.; OLLIVIER-GOOCH, C.; OVCHARENKO, A.; SAHNI, O. et al. Interoperable mesh components for large-scale, distributed-memory simulations. In: IOP PUBLISHING. *Journal of Physics: Conference Series*, 2009. v. 180, n. 1, p. 012011.

DIESTEL, R. *Graph theory*: Springer, 2017.

DOLEAN, V.; JOLIVET, P.; NATAF, F. *An introduction to domain decomposition methods: algorithms, theory, and parallel implementation*: SIAM, 2015. v. 144.

DONGARRA, J. et al. Mpi: A message-passing interface standard version 3.0. *High Performance Computing Center Stuttgart (HLRS)*, 2013.

DUFF, I. S.; KOSTER, J. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, SIAM, v. 22, n. 4, p. 973–996, 2001.

EIJKHOUT, V. et al. Overview of iterative linear system solver packages. *NHSE review*, v. 3, n. 1, 1998.

ELMAN, H. C. Preconditioning for the steady-state Navier–Stokes equations with low viscosity. *SIAM Journal on Scientific Computing*, SIAM, v. 20, n. 4, p. 1299–1316, 1999.

EMERY, A. F. An evaluation of several differencing methods for inviscid fluid flow problems. *Journal of Computational Physics*, Elsevier, v. 2, n. 3, p. 306–331, 1968.

FARHAT, C.; ROUX, F.-X. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, Wiley Online Library, v. 32, n. 6, p. 1205–1227, 1991.

FIDUCCIA, C. M.; MATTHEYSES, R. M. A linear-time heuristic for improving network partitions. In: ACM. *Papers on Twenty-five years of electronic design automation*, 1988. p. 241–247.

FLETCHER, R. Conjugate gradient methods for indefinite systems. In: *Numerical analysis*: Springer, 1976. p. 73–89.

FREUND, R. W.; NACHTIGAL, N. M. QMR: a quasi-minimal residual method for non-hermitian linear systems. *Numerische mathematik*, Springer, v. 60, n. 1, p. 315–339, 1991.

GALLOPOULOS, E.; PHILIPPE, B.; SAMEH, A. H. *Parallelism in matrix computations*: Springer, 2016.

GEUZAINE, C.; REMACLE, J.-F. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, Wiley Online Library, v. 79, n. 11, p. 1309–1331, 2009.

GIRAUD, L.; TUMINARO, R. Algebraic domain decomposition preconditioners. *Mesh partitioning techniques and domain decomposition methods*, p. 187–216, 2006.

GROTE, M. J.; HUCKLE, T. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, SIAM, v. 18, n. 3, p. 838–853, 1997.

GUO, X.; LANGE, M.; GORMAN, G.; MITCHELL, L.; WEILAND, M. Developing a scalable hybrid MPI/OpenMP unstructured finite element model. *Computers & Fluids*, Elsevier, v. 110, p. 227–234, 2015.

GUSTAFSON, J. L. Reevaluating amdahl's law. *Communications of the ACM*, ACM, v. 31, n. 5, p. 532–533, 1988.

GUSTAFSON, J. L. Fixed time, tiered memory, and superlinear speedup. In: IEEE PRESS. *Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, 1990. p. 1255–1260.

HESTENES, M. R.; STIEFEL, E. *Methods of conjugate gradients for solving linear systems*: NBS Washington, DC, 1952. v. 49.

HUGHES, T. J. *The finite element method: linear static and dynamic finite element analysis*: Courier Corporation, 2012.

HUGHES, T. J.; LEVIT, I.; WINGET, J. Element-by-element implicit algorithms for heat conduction. *Journal of Engineering Mechanics*, American Society of Civil Engineers, v. 109, n. 2, p. 576–585, 1983.

HUGHES, T. J.; LEVIT, I.; WINGET, J. An element-by-element solution algorithm for problems of structural and solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, Elsevier, v. 36, n. 2, p. 241–254, 1983.

KARYPIS, G.; KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, SIAM, v. 20, n. 1, p. 359–392, 1998.

KARYPIS, G.; KUMAR, V. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, Elsevier, v. 48, n. 1, p. 96–129, 1998.

KERNIGHAN, B. W.; LIN, S. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, Nokia Bell Labs, v. 49, n. 2, p. 291–307, 1970.

KHARE, A.; HUANG, Y.; DOAN, H.; KANWAL, M. Scalabity. In: *A Fresh Graduate's Guide to Software Development Tools and Technologies*: Singapure, 2012. cap. 6.

KNUTH, D. E. *The art of computer programming*: Pearson Education, 1997. v. 3.

KORIC, S.; LU, Q.; GULERYUZ, E. Evaluation of massively parallel linear sparse solvers on unstructured finite element meshes. *Computers & Structures*, Elsevier, v. 141, p. 19–25, 2014.

KORNEEV, V. G.; LANGER, U. Domain decomposition methods and preconditioning. *Encyclopedia of Computational Mechanics Second Edition*, Wiley Online Library, p. 1–37, 2018.

LIMA, L. M. de; CATABRIGA, L.; RANGEL, M. C.; BOERES, M. C. S. A trade-off analysis of the parallel hybrid SPIKE preconditioner in a unique multi-core computer. In: SPRINGER. *International Conference on Computational Science and Its Applications*, 2017. p. 422–437.

LIMA, L. M. de; LUGON, B. A.; CATABRIGA, L. An alternative approach of the SPIKE preconditioner for finite element analysis. In: IEEE. *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, 2016. p. 331–340.

LINS, E. F.; ELIAS, R. N.; GUERRA, G. M.; ROCHINHA, F. A.; COUTINHO, A. L. G. A. Edge-based finite element implementation of the residual-based variational multiscale method. *International Journal for Numerical Methods in Fluids*, Wiley Online Library, v. 61, n. 1, p. 1–22, 2009.

LIU, W.-H.; SHERMAN, A. H. Comparative analysis of the Cuthill–McKee and the reverse Cuthill–McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, SIAM, v. 13, n. 2, p. 198–213, 1976.

MANDEL, J. Balancing domain decomposition. *International Journal for Numerical Methods in Biomedical Engineering*, Wiley Online Library, v. 9, n. 3, p. 233–241, 1993.

MANGUOGLU, M.; KOYUTÜRK, M.; SAMEH, A. H.; GRAMA, A. Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers. *SIAM journal on Scientific Computing*, SIAM, v. 32, n. 3, p. 1201–1216, 2010.

MANGUOGLU, M.; SAMEH, A. H.; SCHENK, O. PSPIKE: A parallel hybrid sparse linear system solver. In: SPRINGER. *European Conference on Parallel Processing*, 2009. p. 797–808.

MANNE, F.; SOREVIK, T. Optimal partitioning of sequences. *Journal of Algorithms*, Elsevier, v. 19, n. 2, p. 235–249, 1995.

MEIJERINK, J. A.; VORST, H. A. van der. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of computation*, v. 31, n. 137, p. 148–162, 1977.

MULLER, L. K.; LIMA, L. M. de; CATABRIGA, L. A comparative study of local and global preconditioners for finite element analysis. In: *Proceedings of the XXXVIII Iberian Latin-American Congress on Computational Methods in Engineering*, 2017.

OLSHANSKII, M. A.; TYRTSHNIKOV, E. E. *Iterative methods for linear systems: theory and applications*: SIAM, 2014. v. 138.

ORTIGOSA, E. M.; ROMERO, L. F.; RAMOS, J. Parallel scheduling of the PCG method for banded matrices rising from FDM/FEM. *Journal of Parallel and Distributed Computing*, Elsevier, v. 63, n. 12, p. 1243–1256, 2003.

ORTIZ, M.; PINSKY, P. M.; TAYLOR, R. L. Unconditionally stable element-by-element algorithms for dynamic problems. *Computer Methods in Applied Mechanics and Engineering*, Elsevier, v. 36, n. 2, p. 223–239, 1983.

PAIGE, C.; SAUNDERS, M. *Solution of sparse indefinite systems of equations and least squares problems*, 1973.

PAIGE, C. C.; SAUNDERS, M. A. Solution of sparse indefinite systems of linear equations. *SIAM journal on numerical analysis*, SIAM, v. 12, n. 4, p. 617–629, 1975.

PAIGE, C. C.; SAUNDERS, M. A. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM transactions on mathematical software*, v. 8, n. 1, p. 43–71, 1982.

PELLEGRINI, F. Scotch and PT-scotch graph partitioning software: an overview. *Combinatorial Scientific Computing*, Chapman and Hall/CRC, p. 373–406, 2012.

PINAR, A.; AYKANAT, C. Fast optimal load balancing algorithms for 1d partitioning. *Journal of Parallel and Distributed Computing*, Elsevier, v. 64, n. 8, p. 974–996, 2004.

POLIZZI, E.; SAMEH, A. SPIKE: A parallel environment for solving banded linear systems. *Computers & Fluids*, Elsevier, v. 36, n. 1, p. 113–120, 2007.

POLIZZI, E.; SAMEH, A. H. A parallel hybrid banded system solver: the spike algorithm. *Parallel computing*, Elsevier, v. 32, n. 2, p. 177–194, 2006.

RYCHKOV, V. Accelerating assembly operation in element-by-element fem on multicore platforms. *Supercomputing: Second Russian Supercomputing Days, RuSCDays 2016, Moscow, Russia, September 26–27, 2016, Revised Selected Papers*, Springer, v. 687, p. 3, 2017.

SAAD, Y. Highly parallel preconditioners for general sparse matrices. In: *Recent Advances in Iterative Methods*: Springer, 1994. p. 165–199.

SAAD, Y. *Iterative methods for sparse linear systems*: siam, 2003. v. 82.

SAAD, Y.; SCHULTZ, M. H. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, SIAM, v. 7, n. 3, p. 856–869, 1986.

SAHNI, O.; CAROTHERS, C. D.; SHEPHARD, M. S.; JANSEN, K. E. Strong scaling analysis of a parallel, unstructured, implicit solver and the influence of the operating system interference. *Scientific Programming*, Hindawi, v. 17, n. 3, p. 261–274, 2009.

SATHE, M.; SCHENK, O.; UÇAR, B.; SAMEH, A. A scalable hybrid linear solver based on combinatorial algorithms. *Combinatorial Scientific Computing*, CRC Press, p. 95–127, 2012.

SCHENK, O.; GÄRTNER, K. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, v. 23, n. 1, p. 158–179, 2006.

SCHENK, O.; GÄRTNER, K. *Parallel Sparse Direct And Multi-Recursive Iterative Linear Solvers. PARDISO. User Guide Version 5.0. 0.* 2014.

SCHWARZ, H. A. *Ueber einen Grenzübergang durch alternierendes Verfahren*: Zürcher u. Furrer, 1870.

SLEIJPEN, G. L.; FOKKEMA, D. R. BiCGstab (l) for linear equations involving unsymmetric matrices with complex spectrum. *Electronic Transactions on Numerical Analysis*, v. 1, n. 11, p. 2000, 1993.

SMITH, B. F. Domain decomposition methods for partial differential equations. In: *Parallel Numerical Algorithms*: Springer, 1997. p. 225–243.

SONNEVELD, P. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM journal on scientific and statistical computing*, SIAM, v. 10, n. 1, p. 36–52, 1989.

TEZDUYAR, T. E.; SENGA, M. Stabilization and shock-capturing parameters in SUPG formulation of compressible flows. *Computer methods in applied mechanics and engineering*, Elsevier, v. 195, n. 13-16, p. 1621–1632, 2006.

TORO, E. F. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*: Springer Science & Business Media, 2013.

TOSELLI, A.; WIDLUND, O. *Domain decomposition methods-algorithms and theory*: Springer Science & Business Media, 2006. v. 34.

VORST, H. A. van der. *Iterative Krylov methods for large linear systems*: Cambridge University Press, 2003. v. 13.

WALSHAW, C.; CROSS, M. Parallel mesh partitioning on distributed memory systems. *Computational mechanics using high performance computing*, Saxe-Coburg Publications Stirling, p. 59–78, 2002.

WATHEN, A. J. Preconditioning. *Acta Numerica*, Cambridge University Press, v. 24, p. 329–376, 2015.

ZHU, Y.; SAMEH, A. H. Pspike+: A family of parallel hybrid sparse linear system solvers. *Journal of Computational and Applied Mathematics*, Elsevier, v. 311, p. 682–703, 2017.

A Extra algorithms for the domain decomposition steps - Fig.3.6

A.1 The adjacency list L_{mesh} algorithm

The global matrix for a 2D finite element application is highly sparse and their rows are composed by an average of $7 \cdot \texttt{ndof}$ nonzero coefficients, where `ndof` means the number of degrees of freedom per node. CSR format is used to store such matrices, but in general situations, it is not possible to predict structures AA, JA, and IA (see Subsection 3.2.2.3). Thus, finite element mesh data can be read in an adjacency list what in practice means a vector of linked lists. In order to obtain our decomposition approach, finite element mesh data is initially stored in an adjacency list named L_{mesh}. Algorithm 9 illustrates how to obtain the adjacent list L_{mesh}, STEP 2 in Fig. 3.6. Variables `nnodes` and `nel` represent the number of nodes and the number of elements of the mesh. Variables `ndof` and `nnoel`, in turn, means the number of degrees of freedom per node and the number of nodes per element. Admit $node\_in\_L$ as a vector with size `nnodes` which maps the nodes that are associated with at least one unknown. Also, admit $Element$ as a structure with size `nel` and a field $Vertex$ to store `nnoel` nodes. $Node$ is other structure with size `nnodes` and fields $coord$ (to store x and y coordinates), $Type[J]$ (to identify which degree of freedom of a node is an unknown or not) and $Id[J]$ (to enumerate the unknowns), with $1 \leqslant J \leqslant \texttt{ndof}$. Precisely, L_{mesh} is a vector of n_{mesh} linked lists, where n_{mesh} is the number of nodes that are associated with at least one unknown. The function $insert$ represents a sorted insertion in a linked list without repetition. The vector Id_{mesh} also has size `nnodes` and enumerates the nodes whose status in $node\_in\_L$ is $TRUE$. In short, Algorithm 9 enables predicting finite element mesh sparsity pattern in a vector of linked lists named L_{mesh}. Note L_{mesh} has memory complexity of $O(n_{mesh})$, because it maps n_{mesh} rows and every row maps an average of 7 positions. As $\frac{n}{\texttt{ndof}}$, where n is the order of the linear system arose from 2D finite element problem, is a good approximation to n_{mesh}, memory complexity of L_{mesh} can be redefined as $O(\frac{n}{\texttt{ndof}})$.

A.2 The adjacency list L_{matrix} algorithm

As stated in Section 3.2, our domain decomposition approach demands two adjacency lists. One is used to store the mesh sparsity pattern (in this case L_{mesh}, which has been already presented) and the another to store the sparsity pattern of the linear system arose from finite element discretization, that is, L_{matrix}. Algorithm 10 shows how to obtain the adjacency list L_{matrix}, STEP 5 in Fig. 3.6. That algorithm is quite similar to Algorithm 9.

Algorithm 9 To obtain the adjacency list L_{mesh} from mesh sparsity pattern

```
for I = 1, nnodes do
    node_in_L[I] ← .FALSE.
    for J = 1, ndof do
        if Node[I].Type[J] = 1 then
            node_in_L[I] ← .TRUE. {break}
        end
    end
end
for K = 1, nel do
    for I = 1, nnoel do
        v_a ← Element[K].Vertex[I]
        if node_in_L[v_a] = .TRUE. then
            for i = 1, nnoel do
                v_b ← Element[K].Vertex[i]
                if node_in_L[v_b] = .TRUE. then
                    L_mesh[Id_mesh[v_a]] ← insert(Id_mesh[v_b])
                end
            end
        end
    end
end
```

The structures $Node$ and $Element$ have already been defined in Algorithm 9. As mentioned before, $Node[I].Type[J] = 1$ means that Jth degree of freedom of the Ith node is an unknown. Field $Id[J]$ of structure $Element$ enumerates the unknowns. Precisely, L_{matrix} is a vector of n linked lists, where n is the order of the linear system arose from 2D finite element problem. The function $insert$ represents a sorted insertion in a linked list without repetition. In short, Algorithm 10 enables predicting sparsity pattern of the finite element linear system in a linked list named L_{matrix}. Note L_{matrix} has memory complexity of $O(n)$, because it maps n rows and every row maps an average of $7 \cdot \texttt{ndof}$ positions.

Algorithm 10 To build the adjacency list L_{matrix} from finite element matrix sparsity

```
for K = 1, nel do
    for I = 1, nnoel do
        v_a ← Element[K].Vertex[I]
        for II = 1, ndof do
            if Node[v_a].Type[II] = 1 then
                for J = 1, nnoel do
                    v_b ← Element[K].Vertex[J]
                    for JJ = 1, ndof do
                        if Node[v_b].Type[JJ] ≠ 0 then
                            a ← Node[v_a].Id[II]
                            b ← Node[v_b].Id[JJ]
                            L_matrix[a] ← insert(b)
                        end
                    end
                end
            end
        end
    end
end
```

A.3 Permutation P_{matrix} from permutation P_{mesh} algorithm

As discussed in Section 3.6, in order to provide our domain decomposition approach, the sparsity pattern of the finite element linear system is not reordered directly from RCM or Spectral algorithms. There is a intermediate step in which RCM or Spectral algorithm generate a permutation vector P_{mesh} – used to obtain another permutation vector P_{matrix}. Thus, Algorithm 11 shows how to obtain P_{matrix} from the permutation P_{mesh}, STEP 7 in Fig. 3.6. Variable n_{mesh} is the number of nodes that are associated with at least one unknown. The vector $invP_{mesh}$, with size n_{mesh}, is the inverse permutation of P_{mesh}. P_{aux}, with size `nnodes` $\times$ `ndof`, is an auxiliary structure. $Node$ and Id_{mesh} are the same structures defined in Algorithm 9. In short, Algorithm 11 receives the vector P_{mesh} with size n_{mesh} and return the permutation vector P_{matrix} with size n, where n is the order of the linear system from 2D finite element problem.

Algorithm 11 To obtain permutation P_{matrix} from permutation P_{mesh}

```
for I = 1, n_mesh do
    invP_mesh[P_mesh[I]] ← I
    for J = 1, ndof do
        P_aux[I][J] ← −1
    end
end
for I = 1, nnodes do
    if node_in_L[I] = .TRUE. then
        a ← Id_mesh[I]
        for J = 1, ndof do
            P_aux[invP_mesh[a]][J] ← Node[I].Id[J]
        end
    end
end
n ← 0
for I = 1, n_mesh do
    for J = 1, ndof do
        if P_aux[I][J] ≠ −1 then
            n ← n + 1
            P_matrix[n] ← P_aux[I][J]
        end
    end
end
```

A.4 Algorithm of elements partitioning

The Algorithm 12 divides the elements into p partitions considering a separator vector $\mathbf{d}$, STEP 10 in Fig. 3.6. A vector of linked lists $ElemByPart$, with size `nel`, is used to promote the partitioning. Structures $Element$ and $Node$, variables n, `nel`, and `nnoel` are defined as in the Algorithm 9. The function $insert$ is a sorted insertion in a linked list without repetition and $Search$ is an search function in a linked list. Initially, the inverse permutation vector $invP_{matrix}$ from P_{matrix} is built. The separator vector $\mathbf{d}$, as described in STEP 9 of Subsection 3.2.2, gives $p-1$ indices that allow finding which partition a

specific element belongs to. As can be highlighted, some elements can be in more the one partition. That is, there is a element overlapping.

Algorithm 12 To divide elements according to separator vector $\boldsymbol{d}$

```
for I = 1, n do
    invP_matrix[P_matrix[I]] ← I
end
for I = 1, nel do
    for J = 1, nnoel do
        v ← Element[I].Vertex[J]
        for i = 1, p do
            for JJ = 1, ndof do
                if Node[v].Type[JJ] ≠ 0 then
                    if d[i] ⩽ invP_matrix[Node[v].Id[JJ]] < d[i + 1] then
                        ElemByPart[I] ← insert(i) {break}
                    end
                end
            end
        end
    end
end
```

A.5 Node partitioning algorithm according to element division

The Algorithm 13 divides the nodes into p partitions considering the element division, STEP 11 in Fig. 3.6. A vector of linked lists $NodeByPart$, with size `nnodes`, is used to promote the partitioning. Structures $Element$ and $Node$, variables n, `nel`, and `nnoel` are defined as in the Algorithm 9. As in Algorithm 12, the function $insert$ is a sorted insertion in a linked list without repetition, and $Search$ is a search function in a linked list. Function $Search$ verifies if an element I belongs to a specific partition i. In the affirmative case, all nodes that form the element I should be splitted to the partition i.

Algorithm 13 To divide nodes according to element division

```
for I = 1, nel do
    for i = 1, p do
        if Search(i, ElemByPart[I]) = .TRUE. then
            for J = 1, nnoel do
                v ← Element[I].Vertex[J]
                NodeByPart[v] ← insert(i)
            end
        end
    end
end
```

A.6 Finite element mesh data partitioning algorithm

Algorithm 14 is used to divide the finite element mesh data into p files, STEP 12 in Fig. 3.6. Admit $count\_nodes$ and $count\_elem$ as the number of nodes and the number of

elements in a partition i. Variables $Type$, $Type_{\texttt{bef}}$, $Type_{\texttt{aft}}$, and $Type\_Send$ are integers used as labels to identify some attributes of the degrees of freedom for the parallel processing according to Table A.1. Id is the local position of a specific unknown in the partition i. The vector map is an association between the global numbering of a specific node and the local numbering in the partition i. In addition, there are five functions in this algorithm context. The function *open* initializes pointer $FILE$ and has two parameters: $Mesh_i$, the ith name of file mesh and a tag w that indicates a file in the write mode. The function *write* prints all parameters in the corresponding file. The function *counter* counts the number of occurrences of i in $NodeByPart$ or $ElemByPart$. The function $catch\_Type$, defined in Algorithm 15, determines the type of degree of freedom of the nodes according to parallel processing or boundary conditions (see Table A.1). Variables `nnodes` and `ndof` are the number of nodes of the mesh and the number of degrees of freedom per node. Structures $Node$, $Element$ are the same defined in Algorithm 9. The vector $invP_{matrix}$ is the inverse permutation obtained from permutation vector P_{matrix} in STEP 7 of Subsection 3.2.2. The vector d is the separator $\mathbf{d}$ defined in STEP 9 of Subsection 3.2.2. The vectors of linked lists $ElementByPart$ and $NodeByPart$, in turn, are detailed in Algorithms 12 and 13. In short, Algorithm 14 receives information of nodes and elements splitted in p partitions and writes p files whose format is proposed in Fig. 3.8.

Algorithm 14 To divide the mesh into p files

```
for i = 1, p do
    FILE ← open(Mesh_i, "w")
    count_nodes ← counter(NodeByPart, nnodes, i)
    write(FILE, count_nodes)
    count_nodes ← 0
    for I = 1, nnodes do
        if Search(i, NodeByPart[I]) =.TRUE. then
            write(FILE, Node[I].coord)
            for JJ = 1, ndof do
                Type ← catch_Type(Node, I, JJ, i, d, invP_matrix)
                a ← Node[I].Id[JJ]
                Id ← invP_matrix[a] − d[i]
                if i > 1 and Search(i − 1, NodeByPart[I]) =.TRUE. then
                    Type_bef ← catch_Type(Node, I, JJ, i − 1, d, invP_matrix)
                end
                else
                    Type_bef ← 1
                end
                if i < p and Search(i + 1, NodeByPart[I]) =.TRUE. then
                    Type_aft ← catch_Type(Node, I, JJ, i + 1, d, invP_matrix)
                end
                else
                    Type_aft ← 1
                end
                if Type = 0 or Type = 2 or Type = 3 then
                    Type_Send = 1
                end
                else if Type_bef = 3 and Type_aft = 2 then
                    Type_Send = 23
                end
                else if Type_bef = 3 then
                    Type_Send = 2
                end
                else if Type_aft = 2 then
                    Type_Send = 3
                end
                else
                    Type_Send = 1
                end
                write(FILE, Type, Type_Send, Id)
                count_nodes ← count_nodes + 1
                map[I] ← cont_nodes
            end
        end
    end
    count_elem ← counter(ElemByPart, nel, i)
    write(FILE, count_elem)
    for I = 1, nel do
        if Search(i, ElemByPart[I]) =.TRUE. then
            for J = 1, nnoel do
                v ← Element[I].Vertex[J]
                write(FILE, map[v])
            end
        end
    end
    close(FILE)
end
```

Algorithm 15 To determine the type of a node

```
Function catch_Type(Node, I, JJ, i, d, invP_matrix)
    if Node[I].Type[JJ] = 0 then
        Type ← 0
    end
    else if invP_matrix[Node[I].Id[JJ]] < d[i] then
        Type ← 2
    end
    else if invP_matrix[Node[I].Id[JJ]] ⩾ d[i + 1] then
        Type ← 3
    end
    else
        Type ← 1
    end
    return Type
```

Table A.1 – Description of degree of freedom types according to Algorithm 14

Variable	Value	Meaning
$Type$	0	Dirichlet boundary condition in partition i
	1	Unknown in partition i related with partition i
	2	Unknown in partition $i-1$ related with partition i
	3	Unknown in partition $i+1$ related with partition i
$Type_{\texttt{bef}}$	0	Dirichlet boundary condition in partition $i-1$
	1	Unknown in partition $i-1$ related with partition $i-1$
	2	Unknown in partition $i-2$ related with partition $i-1$
	3	Unknown in partition i related with partition $i-1$
$Type_{\texttt{aft}}$	0	Dirichlet boundary condition in partition $i+1$
	1	Unknown in partition $i+1$ related with partition $i+1$
	2	Unknown in partition i related with partition $i+1$
	3	Unknown in partition $i+2$ related with partition $i+1$
$Type\_Send$	1	No need to send
	2	Must be sent to partition $i-1$
	3	Must be sent to partition $i+1$
	23	Must be sent to partitions $i-1$ and $i+1$

B Extra algorithms for the parallel finite element preprocessing

B.1 Algorithms of structures *IdSend_bef*, *IdRecv_bef*, *IdSend_aft* and *IdRecv_aft*

Suppose finite element mesh data is splitted into p partitions according to file format proposed in Fig 3.8. As mentioned in Section 4.1, improving MPI communications means minimizing the amount of information that should be transmitted over the network. Thus, four structures, namely, $IdSend\_bef$, $IdRecv\_bef$, $IdSend\_aft$ and $IdRecv\_aft$, are demanded to manage MPI communication between partitions. Algorithm 16 aims to construct the vectors $IdSend\_bef$ and $IdRecv\_bef$ for partitions from 2 to p. Algorithm 17, in turn, constructs the vectors $IdSend\_aft$ and $IdRecv\_aft$ for partitions from 1 to $p-1$. Terms $\texttt{nel}_i$ and $\texttt{ndof}$ represent the number of elements in a partition i and the number of degree of freedom per node. Structures $Node$ and $Element$ are quite similar to those defined in Section A.1. However, $Node$ has new information in its fields named $Type\_Send[J]$, $Type[J]$ and $Id[J]$, with $1 \leqslant J \leqslant \texttt{ndof}$. Field $Type\_Send[J]$ stores an integer label to identify if a specific degree of freedom information should be or not be sent to neighboring partitions (see Table A.1). Field $Type[J]$ stores another integer label to identify if a specific degree of freedom represents a Dirichlet boundary condition, an unknown in the partition i, or an unknown in neighboring partitions (see Table A.1). Field $Id[J]$, in turn, stores an integer that maps local-global relation between a specific degree of freedom and their corresponding numeration as an unknown of the linear system. In addition, there is a long list of other auxiliary variables, whose description can be seen in Table B.1.

Algorithms 16 and 17 have four main loop-blocks. The first loop-block has the purpose of verifying how many ($nsend_{\texttt{bef}}$, $nrecv_{\texttt{bef}}$, $nsend_{\texttt{aft}}$ and $nrecv_{\texttt{aft}}$) and which ($IdSend\_bef$, $IdSend\_aft$, $IdRecv\_bef$, and $IdRecv\_aft$) degrees of freedom of the partition i must communicate with the neighboring partitions. This communication always occurs in two directions: sent messages and received messages (see Fig. 4.2). The vectors $IdSend\_bef$ and $IdSend\_aft$ store which degrees of freedom they should send messages to. These vectors contain integers within the range $[nrecv_{\texttt{bef}}, n_i + nrecv_{\texttt{bef}}]$ and therefore they are ready to be used in communication (see Fig. 4.3). On the other hand, the vectors $IdRecv\_bef$ and $IdRecv\_aft$ store which degrees of freedom should receive messages. However, in this case, these vectors contain integers outside the expected ranges that would be $[1, nrecv_{\texttt{bef}}]$ and $[n_i + nrecv_{\texttt{bef}}, n_i + nrecv_{\texttt{bef}} + nrecv_{\texttt{aft}}]$ (see Fig. 4.3). Thus, the

vectors $IdRecv\_bef$ and $IdRecv\_aft$ need to have their contents adjusted to these ranges. In this way, the next three loop-blocks of the algorithms fulfill this task. As the stored indices are not sorted, the sort function $qsort\_array1$ is executed and then the stored indices have their values increased (or decreased) until they belong to the appropriate ranges. Besides, these algorithms make a readjustment in the field Id of the structure $Node$ that contemplates the global-local mappings.

Algorithm 16 To obtain structures $IdSend\_bef$, $IdRecv\_bef$ and to adjust Id of $Node$

```
nsend_bef ← 0
nrecv_bef ← 0
for I = 1, nnodes_i do
    for JJ = 1, ndof do
        if Node[I].Type_Send[JJ] = 2 or Node[I].Type_Send[JJ] = 23 then
            nsend_bef ← nsend_bef + 1
            IdSend_bef[nsend_bef] ← Node[I].Id[JJ]
        end
        if Node[I].Type[JJ] = 2 then
            nrecv_bef ← nrecv_bef + 1
            IdRecv_bef[nrecv_bef] ← Node[I].Id[JJ]
            IdRecvAux_bef[nrecv_bef] ← I
            IdDegreeRecvAux_bef[nrecv_bef] ← JJ
        end
    end
end
for I, nrecv_bef do
    SortIdRecv_bef[I].array1 ← IdRecv_bef[I]
    SortIdRecv_bef[I].array2 ← I
end
qsort_array1(SortIdRecv_bef)
for I, nrecv_bef do
    while SortIdRecv_bef[I].array1 + nrecv_bef < I do
        SortIdRecv_bef[I].array1 ← SortIdRecv_bef[I].array1 + 1
    end
end
for I, nrecv_bef do
    a ← SortIdRecv_bef[I].array1
    b ← SortIdRecv_bef[I].array2
    IdRecv_bef[b] ← a
    Node[IdRecvAux_bef[b]].Id[IdDegreeRecvAux_bef[b]] = a + nrecv_bef
end
```

B.2 Algorithms of the parallel CSR structures and their auxiliary structures

Before creating the parallel CSR structures themselves, another structure named LM is required. LM with size $\texttt{nel}_i \times \texttt{ndof}$ has as task maps, element-by-element, all the unknowns which cover a partition i, and the previous and posterior communication regions of each generic vector U_i (see Fig. 4.3). Terms $\texttt{nel}_i$ and $\texttt{ndof}$ represent the number of elements in a partition i and the number of degree of freedom per node. Algorithm 18 shows how to obtain LM from the structure $Node$ proposed in Subsection 4.1.

Since LM is obtained, linked lists L_A, L_B, and L_C to provide the CSR structural vectors JA_i, IA_i, JB_i, IB_i, JC_i and IC_i should be generated. Algorithm 19 indicates how

Algorithm 17 To obtain structures $IdSend\_aft$, $IdRecv\_aft$ and to adjust Id of $Node$

```
nsend_aft ← 0
nrecv_aft ← 0
for I = 1, nnodes_i do
    for JJ = 1, ndof do
        if Node[I].Type_Send[JJ] = 3 or Node[I].Type_Send[JJ] = 23 then
            nsend_aft ← nsend_aft + 1
            IdSend_aft[nsend_aft] ← Node[I].Id[JJ]
        end
        if Node[I].Type[JJ] = 3 then
            nrecv_aft ← nrecv_aft + 1
            IdRecv_aft[nrecv_aft] ← Node[I].Id[JJ]
            IdRecvAux_aft[nrecv_aft] ← I
            IdDegreeRecvAux_aft[nrecv_aft] ← JJ
        end
    end
end
for I, nrecv_aft do
    SortIdRecv_aft[I].array1 ← IdRecv_aft[I]
    SortIdRecv_aft[I].array2 ← I
end
qsort_array1(SortIdRecv_aft)
for I, nrecv_aft do
    while SortIdRecv_aft[I].array1 < I + n_i do
        SortIdRecv_aft[I].array1 ← SortIdRecv_aft[I].array1 − 1
    end
end
for I, nrecv_aft do
    a ← SortIdRecv_aft[I].array1
    b ← SortIdRecv_aft[I].array2
    IdRecv_aft[b] ← a
    Node[IdRecvAux_aft[b]].Id[IdDegreeRecvAux_aft[b]] = a + nrecv_bef
end
```

Algorithm 18 To obtain LM from $Node$

```
for I, nel_i do
    pos ← 1
    for J, nnoel do
        v ← Element[I].Vertex[J]
        for JJ, ndof do
            if Node[v].Type[JJ] = 0 then
                LM[I][pos] ← nrecv_bef + n_i + nrecv_aft + 1
            end
            else
                LM[I][pos] ← Node[v].Id[JJ]
            end
            pos ← pos + 1
        end
    end
end
```

Table B.1 – Description of variable of Algorithms 16 and 17

Variable	Meaning and Attributions
$nsend_{\texttt{bef}}$	Number of unknowns to be sent to partition $i-1$.
$nrecv_{\texttt{bef}}$	Number of unknowns to be received from partition $i-1$.
$nsend_{\texttt{aft}}$	Number of unknowns to be sent to partition $i+1$.
$nrecv_{\texttt{aft}}$	Number of unknowns to be received from partition $i+1$.
$IdSend\_bef$	Vector of indexes of the unknowns to be sent to partition $i-1$.
$IdSend\_aft$	Vector of indexes of the unknowns to be sent to partition $i+1$.
$IdRecv\_bef$	Vector of indexes of the unknowns to be received from partition $i-1$.
$IdRecv\_aft$	Vector of indexes of the unknowns to be received from partition $i+1$.
$IdRecvAux\_bef$	Auxiliary vector involved in organization of $IdRecv\_bef$.
$IdRecvAux\_aft$	Auxiliary vector involved in organization of $IdRecv\_aft$.
$IdDegreeRecvAux\_bef$	Auxiliary vector involved in organization of degrees of freedrom of $IdRecv\_bef$.
$IdDegreeRecvAux\_aft$	Auxiliary vector involved in organization of degrees of freedrom of $IdRecv\_aft$.
$SortIdRecv\_bef$	Temporary structure with 2 fields used to sort $IdRecvAux\_bef$.
$SortIdRecv\_aft$	Temporary structure with 2 fields used to sort $IdRecvAux\_aft$.

to obtain L_A, L_B, and L_C from LM. These linked lists have a field J – to store a column index – and a field $next$ – to store a pointer to the next list. Also, admit the function $insert$ as a sorted insertion in a linked list without repetition and n_i as the number of unknowns of the partition i. Terms $nrecv_{\texttt{bef}}$ and $nrecv_{\texttt{aft}}$ are defined according to Table B.1. Variables nnz_i, $nnz_{\texttt{aft}}$, and $nnz_{\texttt{aft}}$ are the number of nonzero coefficients represented respectively in the blocks A_i, B_i and C_i (see Table 4.2).

Finally, the CSR structural vectors JA_i, IA_i, JB_i, IB_i, JC_i and IC_i can be constructed. As can emphasized, the blocks B_i and C_i (see Fig. 4.6) can have rows with all coefficients equal to zero. Thus, it is necessary to map the nB_i rows containing the nonzero coefficients of the block B_i into a set of n_i possible rows. Similarly, the nC_i rows of C_i should be mapped in n_i possible rows (see Fig. 4.6). For that, auxiliary vectors $IBaux$ and $ICaux$ are used. Remember $IAidex\_aft$ and $IAidex\_idex$, structures that enable the parallel matrix-vector product, are also generated in this stage. Algorithm 20 demonstrates how to fill all parallel CSR structures. In detail, the procedure $create\_JA\_IA$ receives the linked lists produced by Algorithm 19 and builds CSR structural vectors JA_i, IA_i, JB_i, IB_i, JC_i and IC_i (see Algorithm 21).

Algorithm 19 To obtain the linked lists L_A, L_B, and L_C from LM

```
nnz_i ← 0
nnz_bef ← 0
nnz_aft ← 0
for I, nel_i do
    for K, ndof * nnoel do
        II ← LM[I][K]  if nrecv_bef < II ⩽ n_i + nrecv_bef then
            for J, ndof * nnoel do
                JJ ← LM[I][J]  if JJ ⩽ nrecv_bef then
                    nnz_bef ← insert(L_C, II − nrecv_bef + 1, JJ, nnz_bef)
                end
                else if nrecv_bef < JJ ⩽ n_i + nrecv_bef then
                    nnz_i ← insert(L_A, II − nrecv_bef + 1, JJ − nrecv_bef + 1, nnz_i)
                end
                else if JJ ⩽ nrecv_bef + n_i + nrecv_aft then
                    nnz_aft ← insert(L_B, II − nrecv_bef + 1, JJ − n_i − nrecv_bef + 1, nnz_aft)
                end
            end
        end
    end
end
```

Algorithm 20 To obtain JA, IA, JB, IB, $IBaux$, $IAidex\_aft$,JC, IC, $ICaux$, and $IAidex\_bef$

```
create_JA_IA(L_A, JA, IA, n_i)
if i > 1 then
    create_JA_IA(L_B, JB, IB, n_i)
    nB_i ← 0
    for I = 1, n_i do
        if L_B[i] ≠ .NULL. then
            nB_i ← nB_i + 1
            IAidex_aft[I] ← nB_i
            IBaux[nB_i] ← I
        end
    end
end
else if i < p then
    create_JA_IA(L_C, JC, IC, n_i)
    nC_i ← 0
    for I = 1, n_i do
        if L_C[i] ≠ .NULL. then
            nC_i ← nC_i + 1
            IAidex_bef[I] ← nC_i
            ICaux[nC_i] ← I
        end
    end
end
```

Algorithm 21 To create the CSR structural vectors JA and IA

```
Procedure create_JA_IA(List, JA, IA, n)
    for I = 1, n do
        current ⟵ List[I]
        while current ≠ .NULL. do
            count ⟵ count + 1
            JA ⟵ current->J
            IA[I + 1] ⟵ IA[I + 1] + 1
            current ⟵ current->next
        end
    end
    for I = 2, n do
        IA[I + 1] ← IA[I + 1] + IA[I]
    end
```

B.3 Extra algorithms to improve parallel CSR performance

Suppose finite element mesh data is splitted into p partitions. Terms $\texttt{nel}_i$, $\texttt{ndof}$, and $\texttt{nnoel}$ represent, respectively, the number of elements in a partition i, the number of degree of freedom per node, and number of nodes per element. Auxiliary structures CSR_A^e, CSR_B^e, and CSR_C^e are used to improve the CSR storage performance. These three structures have size $\texttt{nel}_i \times (\texttt{ndof} \cdot \texttt{nnoel})^2$, and they enable assembling and reassembling the CSR structures AA_i, AB_i, and AC_i (vectors responsible for storing the coefficients of the blocks A_i, B_i, and C_i, described in Table 4.2) without further efforts. Algorithm 22 shows the main steps to create CSR_A^e, CSR_B^e, and CSR_C^e using LM from Algorithm 18, linked lists from Algorithm 19, and vectors $IAidex\_bef$ and $IAidex\_aft$ from Algorithm 20. In addition, consider the function $Search\_Position$ figures out the position of a column index inside a given linked list (see Algorithm 23).

Algorithm 22 To obtain CSR_A^e, CSR_B^e, and CSR_C^e

```
for I = 1, nel_i do
    for i = 1, (ndof * nnoel)^2 do
        CSR_A^e[I][i] ← nnz_i + 1
        CSR_B^e[I][i] ← nnz_bef + 1
        CSR_C^e[I][i] ← nnz_aft + 1
    end
    for i = 1, (ndof * nnoel)^2 do
        II ← LM[I][i]
        if nrec_bef < II ⩽ n_i + nrec_bef then
            for j = 1, (ndof * nnoel)^2 do
                JJ ← LM[I][j]
                v ← (i − 1) * ndof * nnoel + j
                if JJ ⩽ nrec_bef then
                    pos ← Search_Position(L_B[II − nrecv_bef + 1], JJ)
                    CSR_B^e[I][v] ← IB[IAidex_bef[II − nrecv_bef + 1]] + pos
                end
                else if nrec_bef < JJ ⩽ nrec_bef + n_i then
                    pos ← Search_Position(L_A[II − nrecv_bef + 1], JJ − nrecv_bef + 1)
                    CSR_A^e[I][v] ← IA[II − nrecv_bef + 1] + pos
                end
                else if JJ ⩽ nrec_bef + n_i + nrec_aft then
                    pos ← Search_Position(L_C[II − nrecv_bef + 1], JJ − nrecv_bef − n_i + 1)
                    CSR_C^e[I][v] ← IB[IAidex_aft[II − nrecv_bef + 1]] + pos
                end
            end
        end
    end
end
```

Algorithm 23 To calculate the position of a column index from a adjacency list

```
Function Search_Position(List, value)
    Position ⟵ 0
    while List ≠ .NULL. do
        Position ⟵ Position + 1
        if List->J = value then
            break
        end
        List ⟵ List->next
    end
    return Position
```

B.4 Parallel CSR structures reordering algorithms

As discussed in Subsection 4.1.4.1, to achieve high performance when using parallel CSR structures, auxiliary structures CSR_A^e, CSR_B^e, and CSR_C^e are created. Thus, all information related to reorder the blocks A_i, B_i and C_i (see Fig. 4.4) is also involved with reordering the mentioned structures. As can be noted in the first lines of Algorithm 24, there are procedures, namely, $Matrix\_Row\_Permutation$ and $Matrix\_Column\_Permutation$. Such procedures described in Algorithms 25 and 26 aim to reorder the structures JA_i and IA_i referring to the block A_i, taking into account a permutation vector P. In addition, these two procedures generate a new permutation vector $PermCSR_A^e$ that is used in the reordering of structure CSR_A^e. Another structure worth mentioning is LM, a standard mapping of the $\texttt{nel}_i$ elements for the EBE matrix-vector product, but also used in the assembly of the residue vector. As can be observed, structure LM fits the same reordering applied to structures JA_i and IA_i, since the permutation vector $invP$, inverse of P,is used. The field Id of the structure $Node$ (details about structure $Node$ see Subsection 4.1) is also rearranged according to the reordering proposed by P. In the next two sets of instructions, the blocks B_i and C_i are reordered, more specifically their CSR format representatives. The structures $IBaux$, $IdSend\_aft$, $ICaux$ and $IdSend\_bef$ are rearranged as well as the structures CSR_B^e and CSR_C^e. For this purpose, other two permutation vectors with their respective inverse permutation are built. That is, $PermCSR_B^e$ with inverse $invPermCSR_B^e$ and $PermCSR_C^e$ with inverse $invPermCSR_C^e$. The procedure $Matrix\_Sparse\_Row\_Permutation$, described in Algorithm 27, has almost the same purpose as the procedure $Matrix\_Row\_Permutation$. The difference lies in the fact that $Matrix\_Sparse\_Row\_Permutation$ only works on reordering the rows of blocks B_i and C_i and that these blocks are also sparse per rows, what does not occur with block A_i. Another point to be highlighted is the use of a function named $qsort\_array3\_array2$, presented in Algorithms 25 and 27. This function is a call of the sort algorithm Quick-Sort (Knuth, 1997), and it aims to reorganize the structure $Temp$, taking as the first sort criteria the data from field $array3$. Followed by second sort criteria through field $array2$ data.

Algorithm 24 To reorder CSR structures according to permutation P

```
for I = 1, nnz_i do
  | PermCSR_A^e[I] ← I
end
Matrix_Row_Permutation(n_i, nnz_i, JA_i, IA_i, P, PermCSR_A^e)
Matrix_Column_Permutation(n_i, nnz_i, JA_i, IA_i, P, PermCSR_A^e)
for I = 1, < nnz_i do
  | invPermCSR_A^e[PermCSR_A^e[I]] ← I
end
invPermCSR[nnz_i + 1] ← nnz_i + 1
for I = 1, n_i do
  | invP[P[I]] ← I
end
invP[n_i + 1] ← n_i + 1
for I = 1, nel_i do
  | for J = 1, (ndof · nnoel)^2 do
  |   | CSR_A^e[I][J] ← invPermCSR_A^e[CSR_A^e[I][J]]
  | end
  | for J = 1, ndof · nnoel do
  |   | if nrecv_bef < LM[I][J] ⩽ nrecv_bef + n_i then
  |   |   | LM[I][J] ← invP[LM[I][J] − nrecv_bef + 1] + nrecv_bef
  |   | end
  | end
end
for I = 1, nnodes_i do
  | for J = 1, ndof do
  |   | if Node[I].Type[I] = 1 then
  |   |   | Node[I].Id[J] ← invP[Node[I].Id[J]]
  |   | end
  | end
end
if i > 1 then
  | for I = 1, nB_i do
  |   | IBaux[I] ← invP[IBaux[I]]
  | end
  | for I = 1, nsend_aft do
  |   | IdSend_aft[I] ← invP[IdSend_aft[I]]
  | end
  | Matrix_Sparse_Row_Permutation(nB_i, nnz_aft, JB_i, IB_i, IBAux, PermCSR_B^e)
  | for I = 1, nnz_aft do
  |   | PermCSR_B^e[I] ← I
  | end
  | invPermCSR[nnz_aft + 1] ← nnz_aft + 1
  | for I = 1, nel_i do
  |   | for J = 1, (ndof · nnoel)^2 do
  |   |   | CSR_B^e[I][J] ← invPermCSR_B^e[CSR_B^e[I][J]]
  |   | end
  | end
end
if i < p then
  | for I = 1, nC_i do
  |   | ICaux[I] ← invP[ICaux[I]]
  | end
  | for I = 1, nsend_bef do
  |   | IdSend_bef[I] ← invP[IdSend_bef[I]]
  | end
  | Matrix_Sparse_Row_Permutation(nC_i, nnz_bef, JC_i, IC_i, ICaux, PermCSR_C^e)
  | for I = 1, nnz_bef do
  |   | PermCSR_C^e[I] ← I
  | end
  | invPermCSR[nnz_bef + 1] ← nnz_bef + 1
  | for I = 1, nel_i do
  |   | for J = 1, (ndof · nnoel)^2 do
  |   |   | CSR_C^e[I][J] ← invPermCSR_C^e[CSR_C^e[I][J]]
  |   | end
  | end
end
```

Algorithm 25 To permutate CSR structures according to rows of Block A_i

```
Procedure Matrix_Row_Permutation(n, nnz, JA, IA, P, PermCSR^e)
    Temp_IA[1] ← 1
    k ← 1
    for I = 1, n do
        for J = IA[P[I]], IA[P[I] + 1] − 1 do
            Temp_PermCSR^e[k] ← PermCSR^e[J]
            Temp_JA[k] ← JA[J]
            k ← k + 1
        end
        Temp_IA[I + 1] ← k
    end
    for I = 1, nnz do
        PermCSR^e[I] ← Temp_PermCSR^e[I]
        JA[I] ← Temp_JA[I]
    end
    for I = 1, n + 1 do
        IA[I] ← Temp_IA[I]
    end
```

Algorithm 26 To permutate CSR structures according to columns of Block A_i

```
Procedure Matrix_Column_Permutation(n, nnz, JA, IA, P, PermCSR^e)
    for I = 1, n do
        invP[P[I]] = i
    end
    k ← 1
    for I = 1, n do
        for J = IA[I], IA[I + 1] − 1 do
            Temp[k].array1 ← PermCSR^e[J]
            Temp[k].array2 ← invP[JA[J]]
            Temp[k].array3 ← I
            k ← k + 1
        end
        qsort_array3_array2(Temp, nnz)
        for I = 1, nnz do
            PermCSR^e[I] ← Temp[I].array1
            JA[I] ← Temp[I].array2
        end
    end
```

Algorithm 27 To permutate CSR structures according to rows of Blocks B_i and C_i

```
Procedure Matrix_Sparse_Row_Permutation(n, nnz, JA, IA, IAaux, PermCSR^e)
    for I = 1, n do
        Temp[I].array2 ← I
        Temp[I].array3 ← IAaux[I]
    end
    qsort_array3_array2(Temp, n)
    Temp_IA[1] ← 1
    for I = 1, n do
        for J = IA[Temp[I].array2], IA[Temp[i].array2 + 1] − 1 do
            Temp_PermCSR^e[k] ← PermCSR^e[J]
            Temp_JA[k] ← JA[j]
            k ← k + 1
        end
        Temp_IA[I + 1] ← k
    end
    for I = 1, n do
        IAaux[I] ← Temp[I].array3
    end
    for I = 1, nnz do
        PermCSR^e[i] ← Temp_PermCSR^e[I]
        JA[I] ← Temp_JA[I]
    end
    for I = 1, n + 1 do
        IA[I] ← Temp_IA[I]
    end
```