Universidade do Estado do Rio de Janeiro

Centro de Ciência e Tecnologia

Faculdade de Engenharia

Pedro Juan Torres López

Parallel Implementation of Finite Element Code for Two-Dimensional Incompreensible Navier-Stokes Equations with Scalar Transport

Rio de Janeiro

2010

Pedro Juan Torres López

Parallel Implementation of Finite Element Code for Two-Dimensional Incompreensible Navier-Stokes Equations with Scalar Transport

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Mecânica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Fenômenos de Transporte

Orientador: Prof. Dr. Norberto Magiavacchi

Rio de Janeiro

2010

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

L864 López, Pedro Juan Torres.
Parallel implementation of finite element code for two-dimensional incompreensible navier-stokes equations with scalar transport / Pedro Juan Torres López. – 2010.
87f.

Orientador: Norberto Magiavacchi.
Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia Mecânica. 2. Método dos elementos finitos - Dissertações. 3. Condensação. I. Mangiavacchi, Norberto. II. Universidade do Estado do Rio de Janeiro. III. Título.

CDU 66.02/.04

Autorizo, apenas para fins acadêmicos e científicos, a reprodução total ou parcial desta tese, desde que citada a fonte.

Assinatura

Data

Pedro Juan Torres López

Parallel Implementation of Finite Element Code for Two-Dimensional Incompreensible Navier-Stokes Equations with Scalar Transport

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Mecânica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Fenômenos de Transporte

Aprovado em: de 2010.

Banca Examinadora:

_____

Prof. Dr. Norberto Mangiavacchi (Orientador)
Faculdade de Engenharia - UERJ

_____

Prof. Dr. Carlos Antônio de Moura
Faculdade de Matemática e Estatística - UERJ

_____

Prof. Dr. Luiz Mariano Paes de Carvalho Filho
Faculdade de Matemática e Estatística - UERJ

_____

Prof. Dr. Antonio Castelo Filho
Instituto de Ciências Matemáticas e de Computação - ICMC-USP

Rio de Janeiro

2010

DEDICATÓRIA

Dedicated to the people I love the most in this life:
my family.

AGRADECIMENTOS

I would like to thank my adviser Professor Norberto Mangiavacchi for all his help. I greatly appreciate his time, patient, efforts, and his advice, during the research and writing process. His knowledge and insight has been an invaluable source of guidance during this two years.

I thank to Professor Christian Schaerer, friend and mentor, for believe in me well-before I believed in myself and for show me the amazing path of the science.

I thank to all GESAR team Maxini, Sonia, George, Andre for all the help and my laboratory partners Esther, Raama, Michele, Felipe, Virginia, Katia, Gustavo for their friendship, support and for all the great times shared together.

Special thanks to my friend and "sushi's partner" Hyun, for share his wisdom and for the invaluable advice during the code development. To Hugo, Leon and Manolo for the friendship, partnership, support and for all the "slayer, team slayer and fake team slayer" shared together. I'll always be indebted with you.

Thanks to FAPERJ for the financial support.

Finally and most importantly, I would like to acknowledge my family for understanding the strain that this process put on me. I thank my parents Justina and Pedro, my sister Gracie, my lovely nieces Aleli and Lauri, and my nephew Alejandro, for bringing so much happiness into my life. I would like to say a special word of thanks to my Mother and Father, for all the support and for all the anxiety they had suffered on my behalf. Thanks you so much.

RESUMO

López, Pedro Juan Torres. Implementação paralela de um código de elementos finitos em 2D para as Equações de Navier-Stokes para fluidos incompressíveis com transporte de escalaresl. 2010. 87f. Dissertação (Mestrado em Engenharia Mecânica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2010.

O estudo do fluxo de água e do transporte escalar em reservatórios hidrelétricos é importante para a determinação da qualidade da água durante as fases iniciais do enchimento e durante a vida útil do reservatório. Neste contexto, um código de elementos finitos paralelo 2D foi implementado para resolver as equações de Navier-Stokes para fluido incompressível acopladas a transporte escalar, utilizando o modelo de programação de troca de mensagens, a fim de realizar simulações em um ambiente de cluster de computadores. A discretização espacial é baseada no elemento MINI, que satisfaz as condições de Babuska-Brezzi (BB), que permite uma formulação mista estável. Todas as estruturas de dados distribuídos necessárias nas diferentes fases do código, como pré-processamento, solução e pós-processamento, foram implementadas usando a biblioteca PETSc. Os sistemas lineares resultantes foram resolvidos usando o método da projeção discreto com fatoração LU por blocos. Para aumentar o desempenho paralelo na solução dos sistemas lineares, foi empregado o método de condensação estática para resolver a velocidade intermediária nos vértices e no centróide do elemento MINI separadamente. Os resultados de desempenho do método de condensação estática com a abordagem da solução do sistema completo foram comparados. Os testes mostraram que o método de condensação estática apresenta melhor desempenho para grandes problemas, às custas de maior uso de memória. O desempenho de outras partes do código também são apresentados.

Palavras-chave: Métodos de Elementos Finitos; Condensação Estática; Particionamento de Malhas; Sistema Linear de Grande; PETSc.

ABSTRACT

Lópes, Pedro Juan Torres. Parallel implementation of finite element code for twodimensional incompressible Navier-Stokes Equations with scalar transport. 2010. 87f. Dissertation (Master in Mechanical Engineering) – Faculty of Engineering, State University of Rio de Janeiro, Rio de Janeiro, 2010.

The study of the water flow and scalar transport in water reservoirs is important for the determination of the water quality during the initial stages of the reservoir filling and during the life of the reservoir. For this scope, a parallel 2D finite element code for solving the incompressible Navier-Stokes equations coupled with scalar transport was implemented using the message-passing programming model, in order to perform simulations of hidropower water reservoirs in a computer cluster environment. The spatial discretization is based on the MINI element that satisfies the Babuska-Brezzi (BB) condition, which provides sufficient conditions for a stable mixed formulation. All the distributed data structures needed in the different stages of the code, such as preprocessing, solving and post processing, were implemented using the PETSc library. The resulting linear systems were solved using the projection method implemented by an approximate block LU factorization. In order to increase the parallel performance in the solution of the linear systems, we employ the static condensation method for solving the intermediate velocity at the vertex and centroid nodes separately. We compare performance results of the static condensation method with the approach of solving the complete system. In our tests the static condensation method shows better performance for large problems, at the cost of an increased memory usage. Performance results for other intensive parts of the code in a computer cluster are also presented.

Keywords: Parallel Finite Element; Static Condensation; Mesh Partition; Large Linear System; PETSc.

LISTA DE FIGURAS

SUMÁRIO

u^{θ}

CHAPTER 1

INTRODUCTION

The biomass decomposition and water analysis are an important and necessary task during the initial stages of the reservoir filling and during the life of the reservoir. This is the main subject studied in the GESAR work group, involving different areas of science as mechanical engineering, computational science, geography, biology and chemistry.

As the development of the numerical model progresses, the complexity of the algorithms and the data sets needed for a reasonable quality of the computed results increase, as well as the computational cost. The concept of parallelism arises as solutions enabling data storage and computational power increases. Since computational cluster are made from commodity parts, this approach has become affordable and accessible.

Based on this facts this work aims to enhance the computational performance of the numerical models parallelizing an existing code, developed in the GESAR work group (ANJOS et al., 2007; SHIN, 2009). The parallelization was performed using functions and data structures from PETSc, which is based on the MPI standard. PETSc enables to handle matrix and vector objects at different levels of abstractions, which is a necessary property if we want to implement some problem specific approach for assemble and solve the linear system in an efficient way.

The numerical model solves the Navier-Stokes equations coupled with scalar transport using the Finite Element Method. In order to satisfies the Babuska-Brezzi (BB) condition, the spatial discretization is based on the MINI element, and the resulting linear system was resolved using an approximated block LU factorization.

Basically the workflow of the computational code has the following steps: decompose the domain using a parallel partition library, compute the matrix entries, assemble the matrix, solve the linear system and save the result. Since we must handle multiple degrees of freedom

(dof) per node, a mapping strategy must be adopted in order to attempt one important goal in parallel programming, which is to maintain the data locality (FOSTER, 1995). Also, when solving the intermediate velocity (chapter 2) two approaches were implemented. The first called *Method 1*, was solving the complete linear system, that is, dof on vertices plus on centroids resulting on a large size matrix. In the second, called *Method 2*, we used the idea of static condensation to create a smaller matrix or *condensed matrix* on each subdomain, eliminating the dof of the centroids. Next, we use each of this local matrices on each subdomain to mount the global system, solve this, and back on the local domain to solve the centroids, obtaining finally the whole solutions. Details of this approach are given in chapter 4.

Results are shown and discussed, comparing the static condensation approach presents superior in terms of parallel performance.

This dissertation is organized as follows:

- In chapter 2, will expose the motivations and the mathematics formulation of the problem. Some concepts about parallel computing and a brief review of the PETSc library are given.
- In chapter 3, will explain the main part of a parallel finite element code and the issues that involve the parallelization.
- In chapter 4, the code implementations will described.
- In chapter 5, the computational platform is described, and some benchmark results are given.
- In chapter 6, the results will be presented and discussed.

CHAPTER 2

MATHEMATIC MODEL OF THE PROBLEM

In this chapter we describe the mathematic solved model, introducing the spatial and time discretization. The Galerking semi-discrete method for spatial discretization and the semi-lagrangian for time. Finally the general procedure for solving the problem is presented and the resulting sparsity pattern is discussed.

2.1 Equations of the Model

The final aim is to perform simulations for water flow and scalar transport in water reservoirs like shown in figure 2.1, during the initial stages of the reservoir filling and during the life of the reservoir. The geometry of the domain is usually quite complex, involving many branches with dendritic structure. Therefore, it is desirable to employ a numerical method capable to deal with unstructured meshes to discretize the domain. Among the possible choices, we selected the Finite Element method to solve the model equations. The mathematical statement of this problem, is based on the continuity equations, the Navier-Stokes equations and the advection-diffusion equation for the scalar transport, subject to appropriate boundary and initial conditions.

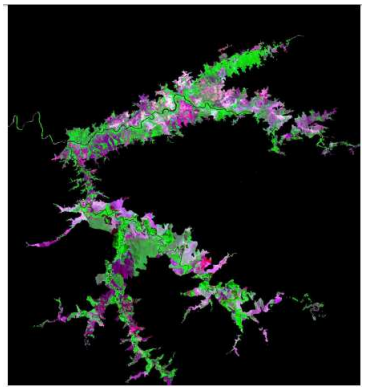


Figure 2.1: Reservoir.

2.1.1 The Model

The model solved, as we said before, is the Navier-Stokes equation for incompressible fluid flow and scalar transport equations (BATCHELOR, 2000; KUNDU; COHEN, 2002).

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot [\mu(\nabla \mathbf{u} \ + \nabla \mathbf{u}^T)] + g \tag{2.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2.2}$$

$$\frac{D\theta}{Dt} = \nabla \cdot (\alpha \nabla \theta) \tag{2.3}$$

The equations 2.1,2.2 and 2.3 are the Navier-Stokes, continuity and scalar transport equations respectively, where u is the velocity, p is the pressure, θ scalar quantity of magnitude of a property of the fluid, ρ is the density of the fluid, μ is the dynamic viscosity, and α is the molecular (heat or mass) diffusivity coefficient. The equation 2.1 can be expressed in terms of the substantive derivative operator $\frac{D}{Dt}$ as showed in the equation 2.4.

$$\frac{D\mathbf{u}}{Dt} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot [\mu(\nabla \mathbf{u} + \nabla \mathbf{u}^T)] + g \tag{2.4}$$

2.2 Variational approaches

Formally, a fluid flow solution is given by the function u, p and θ defined on $(\Omega, t) \subset \mathbb{R}^3 \times \mathbb{R}^+$, which satisfy the following differential equations system, in non-dimensional form

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho}\nabla p + \frac{1}{Re}\nabla \cdot [\mu(\nabla \mathbf{u} + \nabla \mathbf{u}^T)] + \frac{1}{Fr^2}g \tag{2.5a}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2.5b}$$

$$\frac{D\theta}{Dt} = \frac{1}{ReSc}\nabla \cdot (\alpha \nabla \theta) \tag{2.5c}$$

subject to the following boundary conditions,

$$u = u_{\Gamma_u} \quad \text{on} \quad \Gamma_u; \qquad \frac{\partial u}{\partial n} = u_{\Gamma_u^n} \quad \text{on} \quad \Gamma_u^n \tag{2.6a}$$

$$p = p_{\Gamma_p} \quad \text{on} \quad \Gamma_p; \qquad \frac{\partial p}{\partial n} = p_{\Gamma_p^n} \quad \text{on} \quad \Gamma_p^n \tag{2.6b}$$

$$\theta = \theta_{\Gamma_\theta} \quad \text{on} \quad \Gamma_\theta; \qquad \frac{\partial \theta}{\partial n} = \theta_{\Gamma_\theta^n} \quad \text{on} \quad \Gamma_\theta^n \tag{2.6c}$$

and initial conditions,

$$u_i = u_{t_0} \quad p = p_{t_0} \quad \theta = \theta_{t_0} \quad \text{in } \Omega \quad \text{at } t = t_0 \tag{2.7}$$

Equations 2.5 are known as *strong formulations* of the problem, that is, the solution u, p and θ, satisfy the system equation on every point of Ω. The problem can be equivalently formulated in the so-called variational form, also known as *weak formulation*, which is given by equations 2.8. Details about this procedure can be found in (ZIENKIEWICZ; TAYLOR, 2000a; BATCHELOR, 2000).

$$\int_\Omega \nabla w_p \cdot \mathbf{u} d\Omega = \int_{\Gamma_p^c} w_p \mathbf{u} \cdot n d\Gamma \tag{2.8a}$$

$$\int_\Omega w \frac{D\mathbf{u}}{Dt} d\Omega - \int_\Omega \nabla w p d\Omega + \int_\Omega \frac{1}{Re}[\nabla \mathbf{u} + \nabla \mathbf{u}^T] : w d\Omega = 0 \tag{2.8b}$$

$$\int_\Omega w_\theta \frac{D\theta}{Dt} d\Omega + \int_\Omega \frac{1}{Sc\, Re} \nabla w_\theta \cdot \nabla \theta d\Omega = 0 \tag{2.8c}$$

Here we suppose that the gravity term has no dynamic effect in our problem, so it can be neglected. The formulations could be read as, find functions $u \in \mathcal{H}^1_{u_{\Gamma_u}} \times \mathbb{R}^+$, $p \in \mathcal{L}^2 \times \mathbb{R}^+$ and $\theta \in \mathcal{H}^1_{\Gamma_\theta} \times \mathbb{R}^+$ that satisfy the weak formulation 2.8 for all w, w_p and $w_\theta \in \mathcal{H}^1_0$, where the spaces $\mathcal{H}^1_{u_{\Gamma_u}}$, $\mathcal{H}^1_{u_{\Gamma_u}}$ and $\mathcal{L}^2$ are defined as follows,

$$\mathcal{L}^2(\Omega) = \left\{ v : \Omega \to \mathbb{R}; \int_\Omega v^2 d\Omega < \infty \right\} \tag{2.9}$$

$$\mathcal{H}^1(\Omega) = \left\{ v : \Omega \to \mathbb{R}; v, \nabla v \in \mathcal{L}^2(\Omega) \right\} \tag{2.10}$$

$$\mathcal{H}^1_\xi(\Omega) = \left\{ v \in \mathcal{H}^1(\Omega); v = \xi \ \text{ in } \ \Gamma_\xi \right\}$$
$$\text{and where } \xi \text{ is any function defined in the boundary } \Gamma_\xi \tag{2.11}$$

Finally we can say that a solution of 2.5, also is a solution of 2.8. The oppositive is not true, in general.

From equantions 2.8 we can distinguish two kinds of functions. The functions belonging to the solutions space and those that belongs to the w functions space, first set are called *trial functions* and the ones belonging to the second are called *weight functions*.

2.3 Galerkin Finite element method and Matrix Form Equations

In order to compute an approximate solution to this problem, we need to introduce some restrictions in the search space, that is, *seek an approximate solutions in a finite-dimensional subspace of the space of solutions rather than in the whole space*. This is known as *Galerking's method*, (see (BECKER; CAREY; ODEN, 1981)). Now, since the choice of the basis function is arbitrary, we need a technique for constructing them. The *finite element method* provides a general and systematic technique for constructing basis functions (HUGHES, 2000; ZIENKIEWICZ; TAYLOR, 2000b; BECKER; CAREY; ODEN, 1981).

On the basis of these concepts we can perform the spatial discretization of equation 2.8, which result is the semi-discrete form of the equations. Details about this procedure for this problem can be found in (ANJOS et al., 2007; SHIN, 2009). The matrix form of the equations can be written as,

$$\mathbf{M}\dot{\tilde{\mathbf{u}}} - \mathbf{G}\tilde{\mathbf{p}} + \mathbf{K}\tilde{\mathbf{u}} = \boldsymbol{bcn}_2 \tag{2.12a}$$
$$\mathbf{D}\tilde{\mathbf{u}} = \boldsymbol{bcn}_1 \tag{2.12b}$$
$$\mathbf{M}_\theta\dot{\tilde{\theta}} + \mathbf{K}_\theta\tilde{\theta} = \boldsymbol{bcn}_3 \tag{2.12c}$$

where $\mathbf{M}$ is the mass matrix, $\mathbf{G}$ is the gradient matrix, $\mathbf{K}$ is the momentum diffusion matrix, $\mathbf{D}$ is the divergence matrix, $\mathbf{M}_\theta$ is the scalar mass matrix, $\mathbf{K}_\theta$ is the scalar diffusion matrix. In the 2D dimensinal case the matrices have the following structure,

$$\mathbf{D} = \begin{bmatrix} D_1 & D_2 \end{bmatrix}_{np \times 2nu} \tag{2.13a}$$
$$\mathbf{G} = \begin{bmatrix} G_1 \\ G_2 \end{bmatrix}_{2nu \times np} \tag{2.13b}$$
$$\mathbf{M}_\rho = \begin{bmatrix} M_\rho & 0 \\ 0 & M_\rho \end{bmatrix}_{2nu \times 2nu} \tag{2.13c}$$

$$\mathbf{K}_\rho = \left[\begin{array}{cc} 2K_{\rho 11} + K_{\rho 22} & K_{\rho 12} \\ K_{\rho 21} & K_{\rho 11} + 2K_{\rho 22} \end{array} \right]_{2nu \times 2nu} \tag{2.13d}$$

$$\mathbf{M}_\theta = \left[\begin{array}{c} M_\theta \end{array} \right]_{n\theta \times n\theta} \tag{2.13e}$$

$$\mathbf{K}_\theta = \left[\begin{array}{c} K_{\theta 1} + K_{\theta 2} \end{array} \right]_{n\theta \times n\theta} \tag{2.13f}$$

and the unknown variables vector are given by

$$\tilde{\mathbf{p}} = \left[\begin{array}{c} p \end{array} \right]_{np \times 1} \tag{2.14a}$$

$$\tilde{\mathbf{u}} = \left[\begin{array}{c} \tilde{u}_1 \\ \tilde{u}_2 \end{array} \right]_{2nu \times 1} \tag{2.14b}$$

$$\tilde{\theta} = \left[\begin{array}{c} \tilde{\theta} \end{array} \right]_{n\theta \times 1} \tag{2.14c}$$

Where nu is the number of nodes for velocity, np is the number of nodes for pressure and $n\theta$ is the number of nodes for the scalar concentration.

2.4 Time Discretizations - Semi-Lagrangian method

The resulting equations 2.12, is an ordinary differential equations system in time, therefore we need to perform a time discretization. The approach used in this work is the semi-lagrangian method, originally introduced in (CHARNEY; FJÖRTOFT; NEUMMAN, 1952; WIINNIELSEN, 1959; J.S., 1963; J., 1981). This technique allow us to use larger time stable step than a Eulerian approach. This technique try to take the best of the two known scheme *Eulerian* and *Lagrangian*. Basically this consist in use different set of particles at each time step, the set of particles being chosen such that they arrive exactly at the of the mesh at the of the time step. For a mathematical formulation we can start on representing the substantive derivative of the function ϕ at the point x_i discretized using a first order scheme as

$$\frac{\overline{D}\phi}{Dt} = \frac{\phi_m^{n+1} - \phi_d^n}{\Delta t} \tag{2.15}$$

where, $\phi_m^{n+1} = \phi(x_m, t^{n+1})$ is the image of ϕ at the point x_m and the time step $n+1$ and $\phi_d^n = \phi(x_d, t^n)$ is the image of ϕ at the point x_d and the time step n, obtained by interpolating the solution on the mesh nodes at time step n. The i-component of the position x_d is obtained using the expression

$$x_{id} = x_{im} - u_i \Delta t \quad (i = 1, 2, 3) \tag{2.16}$$

where $u_i = u_i(x_m, t^n)$ is the velocity vector at the point x_m and time step n.

Finally equations 2.12 discretized in time using a semilagrangian scheme looks like the

following,

$$\mathbf{M}\left(\frac{\tilde{\mathbf{u}}^{n+1}-\tilde{\mathbf{u}}_d^n}{\Delta t}\right)-\mathbf{G}\tilde{\mathbf{p}}^{n+1}+\mathbf{K}\left(\lambda\tilde{\mathbf{u}}^{n+1}+(1-\lambda)\tilde{\mathbf{u}}_d^n\right)=\boldsymbol{bcn}_2 \tag{2.17a}$$

$$\mathbf{D}\tilde{\mathbf{u}}^{n+1}=\boldsymbol{bcn}_1 \tag{2.17b}$$

$$\mathbf{M}_\theta\left(\frac{\tilde{\theta}^{n+1}-\tilde{\theta}_d^n}{\Delta t}\right)+\mathbf{K}_\theta\left(\lambda\tilde{\theta}^{n+1}+(1-\lambda)\tilde{\theta}_d^n\right)=\boldsymbol{bcn}_3 \tag{2.17c}$$

where, λ is a parameter to obtain different methods of discretization in time. For $\lambda = 0$ results a explicit discretization, for $\lambda = 1$ results a semi-implicit discretization and for $\lambda = \frac{1}{2}$ results the Crank-Nicolson method.

2.5 Ladyzhenskaya Babuska-Brezzi Conditions

As we said before the block matrix of the system 2.19 is **indefinite**, that is, it has positive and also negative eingenvalues. Then we would take a particular care to that system has a unique solution. It can be shown that, provided the kernel (null space) of matrix $\mathbf{G}$ is zero, the system 2.19 will have unique solution. To ensure this solution of the discrete problem, interpolation functions for velocity v and pressure p must be carefully matched. This requirements is known as *inf-sup* conditions, also known as *Ladyzhenskaya Babuska-Brezy* condition. To satisfy this stability conditions, we use in this work the MINI element, see (ZIENKIEWICZ; TAYLOR, 2000a; ZIENKIEWICZ; TAYLOR, 2000b; HUGHES, 2000).

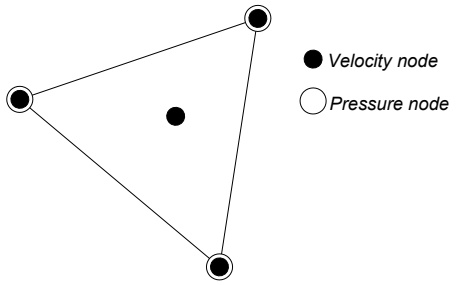


Figure 2.2: 2D Mini Element.

2.6 Solving The Linear Equations System - Projection method

Once we have made all the discretizations, we need to perform the resolution of the set of linear equations system resulted, so we can rewrite the equations 2.17 as follows,

$$\mathbf{D}\tilde{\mathbf{u}}^{n+1} = \boldsymbol{bcd}_1 + \boldsymbol{bcn}_1 \tag{2.18a}$$

$$\left(\frac{\mathbf{M}}{\Delta t} + \lambda\mathbf{K}\right)\tilde{\mathbf{u}}^{n+1} - \mathbf{G}\tilde{\mathbf{p}}^{n+1} = \left(\frac{\mathbf{M}}{\Delta t} - (1-\lambda)\mathbf{K}\right)\tilde{\mathbf{u}}_d^n + \boldsymbol{bcd}_2 + \boldsymbol{bcn}_2 \tag{2.18b}$$

$$\left(\frac{\mathbf{M}_\theta}{\Delta t} + \lambda\mathbf{K}_\theta\right)\tilde{\theta}^{n+1} = \left(\frac{\mathbf{M}_\theta}{\Delta t} - (1-\lambda)\mathbf{K}_\theta\right)\tilde{\theta}_d^n + \boldsymbol{bcd}_3 + \boldsymbol{bcn}_3 \tag{2.18c}$$

We can rewrite these equations in a simpler way, as follows

$$\begin{bmatrix} \mathbf{B} & -\mathbf{G} \\ \mathbf{D} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{u}}^{n+1} \\ \tilde{\mathbf{p}}^{n+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u^n \\ 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{bcd}_2 \\ \boldsymbol{bcd}_1 \end{bmatrix} + \begin{bmatrix} \boldsymbol{bcn}_2 \\ \boldsymbol{bcn}_1 \end{bmatrix} \tag{2.19}$$

$$\mathbf{B}_\theta\tilde{\theta}^{n+1} = \boldsymbol{r}_\theta^n + \boldsymbol{bcd}_3 + \boldsymbol{bcn}_3 \tag{2.20}$$

where, the matrix $\mathbf{B}$ and $\mathbf{B}_\theta$ are given by

$$\mathbf{B} = \frac{\mathbf{M}}{\Delta t} + \lambda\mathbf{K} \tag{2.21a}$$

$$\mathbf{B}_\theta = \frac{\mathbf{M}_\theta}{\Delta t} + \lambda\mathbf{K}_\theta \tag{2.21b}$$

and the vector $\boldsymbol{r}_u^n$ and $\boldsymbol{r}_\theta^n$ are given by

$$\boldsymbol{r}_u^n = \left(\frac{\mathbf{M}}{\Delta t} - (1-\lambda)\mathbf{K}\right)\tilde{\mathbf{u}}_d^n \tag{2.22a}$$

$$\boldsymbol{r}_\theta^n = \left(\frac{\mathbf{M}_\theta}{\Delta t} - (1-\lambda)\mathbf{K}_\theta\right)\tilde{\theta}_d^n \tag{2.22b}$$

The equation 2.20 is easy to solve, not so the system 2.19. This system is in general large, indeterminate and difficult to solve in an efficient way. The matrix of this system is known as saddle point and is one fundamental problem in scientific computing (BENZI et al., 2005). Most traditional solve methods like the projection or fractional step methods, were conceived to allow the system to be solved as a series of individuals equations, saving in this way computational efforts and produce simpler code implementations (PEROT, 1993; LEE; OH; KIM, 2001). The price of this approach is a certain loss of accuracy in the solution.

The projection method consists of decomposing the matrix of the equation 2.19 via a

block factorization. Applying the LU factorization, the following linear system is obtained

$$\begin{bmatrix} \mathbf{B} & \mathbf{0} \\ \mathbf{D} & \mathbf{D}\mathbf{B}^{-1}\mathbf{G} \end{bmatrix} \begin{bmatrix} \mathbf{I} & -\mathbf{B}^{-1}\mathbf{G} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{u}}^{n+1} \\ \tilde{\mathbf{p}}^{n+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u^n \\ 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{bc}_2 \\ \boldsymbol{bc}_1 \end{bmatrix} \tag{2.23}$$

where, $\boldsymbol{bc}_i = \boldsymbol{bcd}_i + \boldsymbol{bcn}_i$ $(i = 1, 2, 3)$.

In the first instance the intermediate solution is obtained by solving the following equation system

$$\begin{bmatrix} \mathbf{B} & \mathbf{0} \\ \mathbf{D} & \mathbf{D}\mathbf{B}^{-1}\mathbf{G} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}}^{n+1} \\ \hat{\mathbf{p}}^{n+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u^n \\ 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{bc}_2 \\ \boldsymbol{bc}_1 \end{bmatrix} \tag{2.24}$$

and then, the final solution is obtained by solving

$$\begin{bmatrix} \mathbf{I} & -\mathbf{B}^{-1}\mathbf{G} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{u}}^{n+1} \\ \tilde{\mathbf{p}}^{n+1} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{u}}^{n+1} \\ \hat{\mathbf{p}}^{n+1} \end{bmatrix} \tag{2.25}$$

Using the two equations (equation 2.24 and 2.25) the solution can be obtained by the following steps

$$\mathbf{B}\hat{\mathbf{u}}^{n+1} = \boldsymbol{r}_u^n + \boldsymbol{bc}_2 \tag{2.26a}$$
$$\mathbf{D}\mathbf{B}^{-1}\mathbf{G}\tilde{\mathbf{p}}^{n+1} = -\mathbf{D}\hat{\mathbf{u}}^{n+1} + \boldsymbol{bc}_1 \tag{2.26b}$$
$$\mathbf{B}_\theta\tilde{\theta}^{n+1} = \boldsymbol{r}_\theta^n + \boldsymbol{bc}_3 \tag{2.26c}$$
$$\tilde{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}^{n+1} + \mathbf{B}^{-1}\mathbf{G}\tilde{\mathbf{p}}^{n+1} \tag{2.26d}$$

This method relies on the Helmholtz-Hodge decomposition, which says that any vector can be decomposed into a component of a zero divergence and another with zero curl.

Solving the equations 2.26 is known as the Uzawa method. However, to solve 2.26b exactly is a very expensive step. Therefore, an approximation is performed in order to increase the computational efficiency, yielding the following approximate factorization

$$\mathbf{B}\hat{\mathbf{u}}^{n+1} = \boldsymbol{r}_u^n + \boldsymbol{bc}_2 \tag{2.27a}$$
$$\mathbf{D}\widetilde{\mathbf{B}}^{-1}\mathbf{G}\tilde{\mathbf{p}}^{n+1} = -\mathbf{D}\hat{\mathbf{u}}^{n+1} + \boldsymbol{bc}_1 \tag{2.27b}$$
$$\mathbf{B}_\theta\tilde{\theta}^{n+1} = \boldsymbol{r}_\theta^n + \boldsymbol{bc}_3 \tag{2.27c}$$
$$\tilde{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}^{n+1} + \widetilde{\mathbf{B}}^{-1}\mathbf{G}\tilde{\mathbf{p}}^{n+1} \tag{2.27d}$$

where $\widetilde{\mathbf{B}}$ is a diagonal (lumped) approximation of $\mathbf{B}$.

Finally we can describe the solutions algorithm with more details as following.

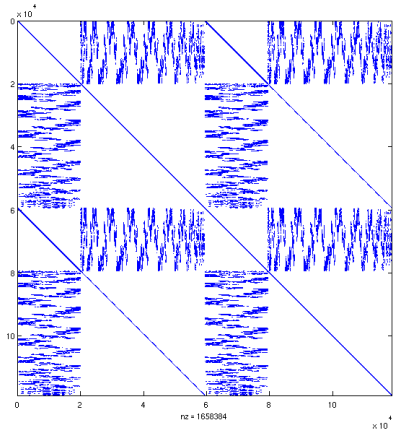
Algorithm 1: General Procedure

Assemble Matrix: $\mathbf{B}$,$\mathbf{D}$, $\mathbf{G}$, $\widetilde{\mathbf{B}}^{-1}$ and $\mathbf{B}_\theta$
for $i \leftarrow 1$ **to** $Nsteps$ **do**
  Apply Boundary Conditions.
  Resolve $\hat{\mathbf{u}}^{n+1}$ from:
  $\mathbf{B}\hat{\mathbf{u}}^{n+1} = \boldsymbol{r}_u^n + \boldsymbol{bc}_2$
  Resolve $\tilde{\mathbf{p}}^{n+1}$ from:
  $\mathbf{D}\widetilde{\mathbf{B}}^{-1}\mathbf{G}\tilde{\mathbf{p}}^{n+1} = -\mathbf{D}\hat{\mathbf{u}}^{n+1} + \boldsymbol{bc}_1$
  Find the final velocity: $\tilde{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}^{n+1} + \widetilde{\mathbf{B}}^{-1}\mathbf{G}\tilde{\mathbf{p}}^{n+1}$
  Calculate the Scalar Concentrations:
  $\mathbf{B}_\theta\tilde{\theta}^{n+1} = \boldsymbol{r}_\theta^n + \boldsymbol{bc}_3$
  Dump Solutions.
end

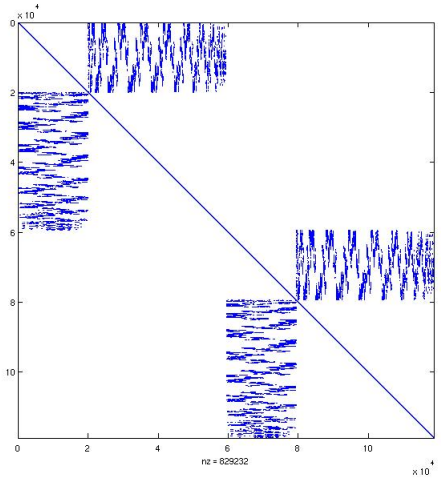
2.7 Sparsity Pattern

The resulting matrices, seen in the previous section, are large and sparse, as typically resulting from the PDE discretization. A matrix with very few nonzero elements is a sparse matrix. It is possible to take advantage of this fact, using special scheme to store them, like CSR, CSC, MRS and others see (SAAD, 1999) for details. Consequently algorithm that uses the matrix must be designed to work with this kind of storage (SAAD, 1999). The sparse matrices in algorithm 1 are unstructured, because their entries are non regularly located.

Figures 2.3(a) and 2.3(b) shown a typical sparsity of matrix K and M for this problem, with this particular type of element mesh and numerating all vertices first following by centroids, see (ANJOS et al., 2007; SHIN, 2009) for details.



(a)



(b)

Figure 2.3: Typical Sparsity Pattern for our problem: (a) Sparsity Pattern for Matrix K . (b) Sparsity Pattern for Matrix M.

The blocks matrix in the main diagonal correspond to each degree of freedom, X and Y in our case, note the influence of centroids in the final size of the matrix.

CHAPTER 3

FINITE ELEMENT PROGRAM AND THEIR PARALLELIZATION

In this chapter we introduce a general structure of a typical finite element code. The bottlenecks on each stage are explained. Next, the parallelization strategy on each stage of the code is discussed.

3.1 A General Structure of Finite Element Program

In general, a finite element program could be designed as a process with three independents stages, that is preprocessing, main processing and postprocessing. Each must be executed in right sequential order to produce the desired result. The preprocessing part includes: read input data, mesh generation, define data arrays, and element-related data. The postprocessing, which is the last stage, directly gives the solution in graphical form or may be linked to an external postprocessor via an interface. The main processing unit is responsible for the computational effort and often most of the computing (CPU) time during a calculation, consequently this part is the most time consuming part of whole work. Sometimes this part is called *kernel* and usually many finite element programs only consist of this unit, leaving the pre- and post processing to be performed by linking external programs. We can break up this unit into two parts: *assemble* and *solver*, see figure 3.1 . In the *assemble* part we mount the stiffness matrix and load vector and next in the *solver* we solve the entire equation system assembled in this way.

In the following sections we provide a short review of parallel computer architectures and programming models to introduce the parallel programming paradigm employed in this work.

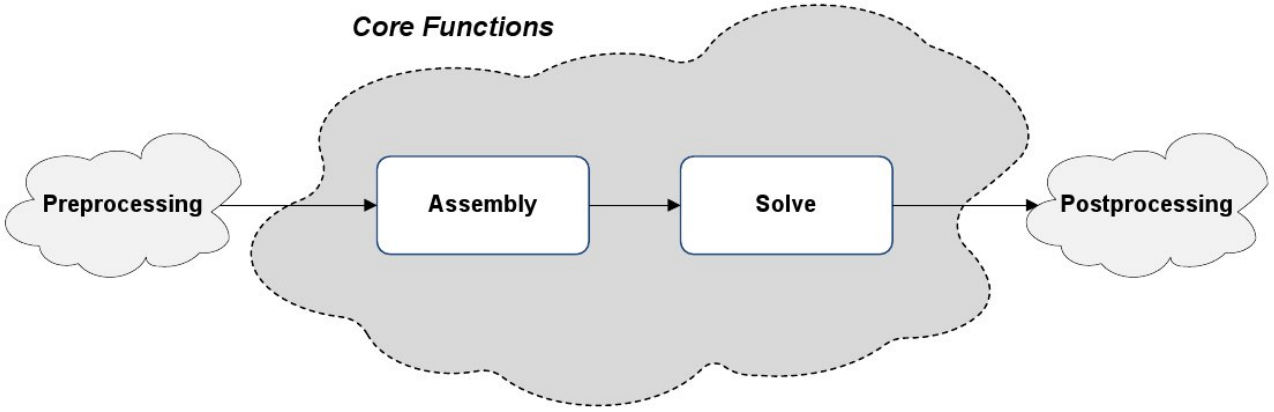


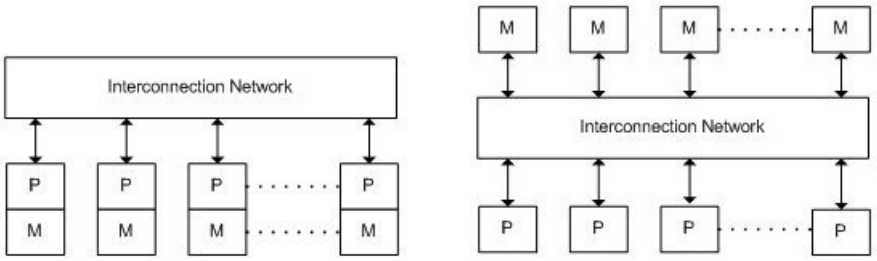
Figure 3.1: Typical FEM Program .

Algorithm 2: Finite Element Procedure

Preprocessing
while *i<Nsteps* **do**
    assemble matrix
    intermediate velocity calculation
    pressure calculation
    velocity correction calculation
    scalar calculation
end
Posprocessing

3.2 Basics Concepts of Parallel Computing

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem (FOSTER, 1995). Two types of parallel computers can be highlight, those with distributed memory (fig. 3.2(a)) and those with shared memory (fig. 3.2(b)). More extensive classifications and explanations can be founded in (FOSTER, 1995; KSHEMKALYANI; SINGHAL, 2008).



(a) (b)

Figure 3.2: Classifications of Parallel Computers by memory: (a) Distributed Memory. (b) Shared Memory.

In the distributed memory architecture each processors has its own memory and communications are performed through either specialized (InfiniBand, Quadrics, Myrinet, etc) or commodities (Gigabit Ethernet) networks. In shared memory architecture a single space address are shared between all the processors.

From the programming point of view parallel computers with shared memory are easy to program and sharing data is fast due to the proximity of memory to processors. In opposite to this there are not scalability between processors and memory, also they becomes expensive to design and produce with ever increasing numbers of processors. In other hand, distribute memory has the advantage that the memory is scalable with number of processors and cost effectiveness because can use commodity, i.e., off-the-self processors and networking. As a disadvantages they are hard to programming, because the programmers is responsible for many of the details associated with the data communications. The largest and fastest computers in the world today employ both shared and distributed memory architectures.

3.3 Programming Models

Several programming models are available for parallel computers, but each of them are associated to the underlying hardware. Some of them are shared memory, message passing, and hybrid. From this group we can stick out two, shared memory and message passing, for details of these approaches see (GRAMA; GUPTA; KUMAR, 2003).

Shared memory model programm share a common address space, that is why we need mechanisms for control the access to the memory space, however, with this approach the notion of memory ownership disappears, with the need of explicitly communications, therefore simplifies the program development (HUGHES; HUGHES, 2008; GRAMA et al., 2003). Some APIs that help the programming in this fashion are the POSIX Threads (BUTENHOF, 1997; BURNS; WELLINGS, 2009) and OpenMP (OPENMP, 2010).

In distributed memory model, each process owns a memory space and communications must be performed explicitly to exchange data between process.The message-passing program-

ming paradigm is one of the oldest and most widely used approaches for programming parallel computers (GRAMA et al., 2003). The specifications of APIs needed to perform communications in this model is Message Passing Interface (MPI) and there are several implementation, commercial also as non-commercial, for a complete list see (ANL/MSC, 2010a). The implementations used in this work is the MPICH2 (ANL/MSC, 2010b). An important and desirable features on this model is the *data locality*, since local data is less costly to access that the remote data.

3.4 The PETSc Library

The parallel code implemented in this work was development using the message passing programming model. The main library used to develop this work was PETSc, that in words of their authors "is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for parallelism" (BALAY et al., 2009), basically all the necessary distribute object and functions for implement the parallel code were taken from this library. PETSc provide basically four main object **Vec**, **Mat**, **KSP/PC** and **SNES** 3.3.

- **Vec**: for handle vectors sequentials/parallels and basic operations of linear algebra.
- **Mat**: for handle matrix sequential and parallel with different format.
- **KSP/PC**: implement a set of solvers algorithms based on Krylov Subspace and preconditioners such as ILU, ICC, algebraic multigrid, and others.
- **SNES**: implementations of Newton method, with global convergency (linear search, trusted region).

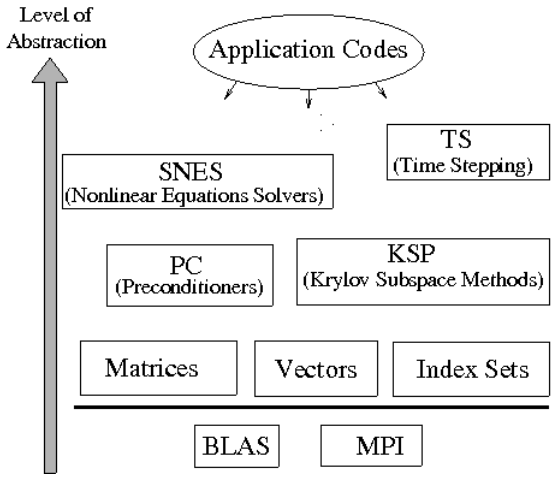


Figure 3.3: Diagram of PETSc Library (BALAY et al., 2009).

Others objects that was widely used in this work was the **IS** and **AO**. **IS**is an abstract object for general indexing and setup vector scatters and **AO** is an abstract object that manages mapping between different global numbering.

Implement a complete parallel unstructured grid finite element code with PETSc represent a good deal of work and requires a lot of application level coding, but it can manage all the linear solvers, which allow concentrate the effort on parallel unstructured grid manipulations.

Next, we will explain how is the treatment of distribute vectors and matrices in this library.

3.4.1 Matrix-Vector Parallel Layout

We will review now how is the parallel layout of vector and matrices in PETSc, with a simple but relevant example. The matrix are partitioned by contiguous chunks of rows across the processors, and all column are local, as showed in 3.4. Suppose we want to perform a matrix-vector $\mathcal{A}x = b$ product, we must take care choosing the parallel layout of each vector, that is, each vector layout must be consistent with matrix column or row parallel layout. Therefore we have in 3.4 that the left vector x must be compatible with column layout and the right b must be compatible with row layout. Fortunately PETSc have specialized function that given a matrix we get compatible vectors, avoiding inconsistencies in the code. In this manner on each local structure of distributed object matrix A we have two sequential submatrix named diagonal and off-diagonal matrix, details about algorithms of matrix-vector product in distributed environment can be see in (SAAD, 1999). The fact of the continuous distributions of rows across the process, become relevant at time of matrix assembly, resulting desirable proper numbering in each subdomains, this will be explained in further chapters.

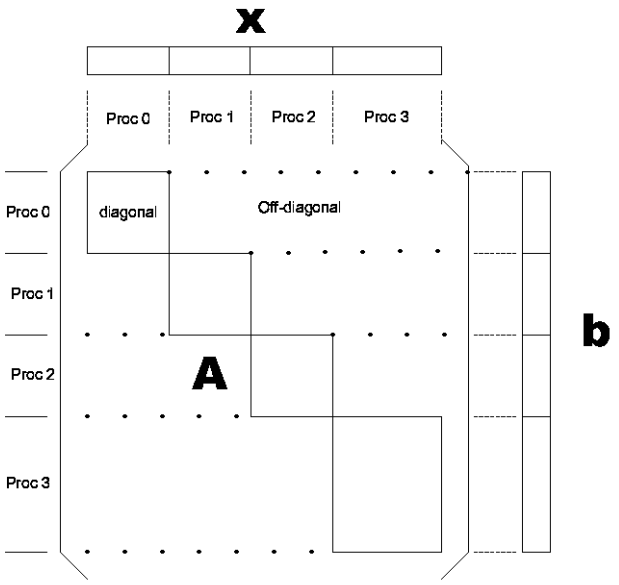


Figure 3.4: Matrix Parallel Layout used on PETSc.

3.5 Performance Bottlenecks

Most of the computational effort is made in the kernel of the program, as we said before. In order to perform more complex simulations, i.e., complex numerical models and geometry, we need to improve this unit, taking parallel programming technique as prime approach, based on successful results reported in the literature and the reduction of commodity hardware costs.

When performing large scale finite element computations, basically all stages of the program demand more effort. In fact, when the problem is large, the mesh is huge, demanding high cost of I/O work and memory usage. Inside the kernel this large mesh would affect the cost of assembly and solver stage consuming even more memory, this in turn would produce large amount of results that have to be saved.

As we explained, the most resources and time consuming stage is the kernel, because it need to be performed every time step. In principle we shall focus in the improvement of this unit, but we will need to accomplish some task in the preprocessing stage (like the mesh partition and decomposition) to achieve full parallelism in the kernel.

3.5.1 Parallel Programming Paradigm

In large application we could have serious bottlenecks in the preprocessing and post processing stages, since the NFS is not scalable for more than 64 nodes, see (STERLING; SALMON; SAVARESE., 1999) for details. To overcome this we should use some parallel file system for HPC like Lustre, PanFS,PVFS2,etc, see (LAYTON, 2007). Another option is to split input/ouput file avoiding the transfers and synchronization between nodes.

Inside the kernel, we have to improve two substage assemble and solve. Algorithm 3 shows a typical element-by-element style assembly from FEM theory, see (HUGHES, 2000; GOCKENBACH, 2006). In this algorithms, each element has its own local matrix and adds his contributions to the global matrix. In this way the program will iterate over each element in order to build the global matrix. Since each element can be iterated and processed in an independent order, this bring to us a natural approach for parallelism. Employing either the distribute memory or a shared memory paradigm. As mentioned before we selected the message-passing model because it maps the available hardware. This allows us to distribute work (in a distributed environment like a cluster) across the processes, enabling each process to work on it own data, consisting on a set of elements independently, saving a considerable amount of time in the computation of the global matrix. Once we have assembled the matrix (or matrices) and vector (or vectors), we need to proceed to solve the system equation. There are excellent works about this issue employing PETSc (BALAY et al., 1997; BALAY et al., 2010; BALAY et al., 2009), Trilinos (HEROUX, 2005), SPOOLES (ASHCRAFT; GRIMES, 1999) and others.

Algorithm 3: Element-oriented algorithm for assembly

Allocate memory and create the sparse matrix A
foreach *element* $e \in \mathcal{D}$ **do**
  construct elemental local matrix a of the element
  foreach $i_e \in e$ **do**
    foreach $j_e \in e$ **do**
      add contribution to the global matrix A
      $A_{\mathcal{M}(i_e)\mathcal{M}(j_e)} += a_{i_e j_e}$
    end
  end
end
where $\mathcal{M}$ is a function that associates a local numbering to a global numbering

Necessarily, we will have data exchanging between process, in order to complete assemble and solve stages. Since the communication bandwidth is a bottleneck in the message passing programming models, *minimizing communication* is the most important problem to solve with this programming models.

3.6 Mesh Partition and Decomposition

The general parallelization strategy is to decompose the problem domain into subdomains and distribute them onto the processors. This partition must be such that minimizes the communication and balances the computational load on each processors, i.e., the number of elements must be equal or near equal on each subdomain. In some problem this load balancing

would be inefficient because the computational effort varies from element to element. In such case another strategy must be applied, see for example (LU, 2008).

The partitioning problem for p processor can be expressed as findings p subsets $\omega_1 \dots \omega_p$ of elements such that:

$$\mathcal{D} = \bigcup_{i=1}^{p} \omega_i \quad and \quad \forall i,j \quad i \neq j \quad \omega_i \cap \omega_j = A_{ij}, \quad i,j = 1 \dots p \tag{3.1}$$

with the constraint that the workload $W(\omega_i) = constant \; \forall \; i = 1 \dots p$, and the subset A_{ij} is such that the connections between subdomains is minimum. In practice, the workload in each subdomain may not be equal, as long as the connections result in reduced inter-process communications.

This problem is known to be NP-complete problem in general (GAREY; JOHNSON; STOCKMEYER, 1974). However, since the problem arises in many applications areas, many approximation algorithms have been developed over the years.

Basically partitions methods are divided into those which use the spatial coordinates of the nodes called *geometric methods* and those which use connectivity of the mesh called *graph based methods*. The geometric methods are usually fast but do not necessarily have or use any knowledge of edges, resulting sometime in a disconnected mesh and in a not optimized data dependencies/communication (WALSHAW; CROSS, 1999). Behind the graph based methods we have the *spectral* and the *multilevel* methods. The spectral partitioning has proven to provide good quality of mesh partitions but demand very high computational effort (KARYPIS; KUMAR, 1995). Multilevel method algorithms have been shown to be very fast and produce very high quality partitioning. The idea is to minimize the problem size by contracting the graph into coarser approximations which are used to solve the actual problem. This methods is described in (KARYPIS; KUMAR, 1995).

Geometric methods as well as graph based methods have been implemented in a variety of package like PARMETIS (LAB, 2010), JOSTLE (WALSHAW, 2010), SCOTH (PELLEGRINI, 2010), CHACO (HENDRICKSON, 2010), etc. In this work we chose as initial tool PARMETIS, but thanks to the abstract interface given by PETSc, we can work transparently with any package. In chapter 4 we will explain details about this implementation.

3.7 Assemble and Solve - The Kernel

The assemble an the solve substage are the core operations of the system and they need to run very efficiently, since they execute many times, requiring consideration not only of the algorithm but also of the hardware where it will be executed. In this section we will explain the approach taken for the assemble and solve substages.

3.7.1 Assemble

As we explain before, the assemble is the process in which we perform the calculation of the matrix and thus the linear system. The parallel version of algorithm 3 is show in algorithm 4.

During the assembly we need to cache or stash matrix entries before to perform the assembly in its final format, then there is a repetitive process of allocating memory and copying from the old to the new storage, which is a very expensive task. This explain the routine in line 1 in the algorithm 4. Before allocation we need to process the mesh in order to estimate the number of non-zeros entries per row of the matrix. In our problem context this is done by looping over the vertices, computing the number of neighbor vertices, which dictates the number of non-zero entries for the corresponding matrix row. Allocating the matrix can increase the performance significantly.

As a result of the mesh partition, each process has its own set of data to work on, this set has its own local numerations. Later on, to have the contributions of each element in the correct place in the global matrix, we need some way to associate the local numerations with his global numerations. Note that these are one-to-one relations. We will call this association a *mapping function*. Let be $I_{local} = 1 \dots n$ and $I_{global} = 1 \dots N$, where n is the local size assigned to each process and N the global size, next we define the *mapping function* as a injective function $\mathcal{F}$ such as $\mathcal{F} : I_{local} \rightarrow I_{global}$ (note that $n \leq N$). This is done in line 2. We define another usefull function $\mathcal{F}_{\mathcal{AB}}$ that takes a set of global numberings $\mathcal{A}$ and associates to them another global numbering $\mathcal{B}$.

Since we use PETSc objects for matrices and vectors, we are able to use a range of sparse matrix formats available in the package. The default format is the compressed sparse row format (CSR) see (SAAD, 1999). PETSc also supports additional formats like block compressed row, block diagonal storage, see (BALAY et al., 2009) for further details.

The rest of the routines are dedicate to calculate the elemental matrix and to add the

contributions. The element matrix computations depend on what matrix we are calculating.

Algorithm 4: Element-oriented algorithm for assembly - Parallel Version

1 *Allocate memory and create the sparse matrix A*
foreach *element* $e \in \omega_i$ **do**
construct elemental local matrix a of the element
2 $\mathcal{F}(I_{local}) = I_{global}$
foreach $i \in e$ **do**
foreach $j \in e$ **do**
add contribution to the global matrix A
$A_{\mathcal{F}(i)\mathcal{F}(j)} += a_{ij}$
end
end
end

3.7.2 Solve

Once we have the linear system mounted in the form $Ax = b$, we can proceed to its resolution. As we saw in algorithm 1, there are three linear systems to resolve, intermediate velocity $\hat{\mathbf{u}}^{n+1}$, pressure $\tilde{\mathbf{p}}^{n+1}$, and scalar concentrations $\tilde{\theta}^{n+1}$, line 3, 4 and 5 of algorithm 1 respectively. To resolve these large linear equations systems we have two approaches: direct and iterative. Direct methods perform a factorization of matrix,which in deep is based on the Gaussian elimination. They are robust, they would give exact solutions in the absence of round-off errors, and are chosen when reliability is the primary concern. Due to their robustness and reliability, which is required in certain applications, there is an effort to develop direct solvers in both distributed and shared memory computers. Example of this are MUMPs (AMESTOY et al., 1999), SUPERLU (DEMMEL et al., 1999; DEMMEL; GILBERT; LI, 1999; LI; DEMMEL, 2003) and UmfPack (DAVIS; DUFF, 1994) package, with which it is possible to solve linear systems of fairly large size efficiently in a reasonable amount of time.

Unfortunately direct methods require too many resources in terms of storage and time when the problem size increases, resulting in poorly scalable algorithms. The iterative methods relax the resources requirement by direct methods, at the price of less reliability and not always achieving the desired accuracy on the solution. The idea behind these method is to approximate the solution by a sequence of matrix-vector multiplications $\{Ax_k\}$. A metric of the quality is the number of iterations performed or the velocity of error reduction. In iterative methods, the kernel operations are the sparse matrix vector multiplications (SpMV), they are easy to parallelize, turning them suitable for parallel computing, in both shared and distributed environments.

Several iterative methods have been proposed (SAAD; VORST, 2000) (FREUND; GOLUB;

NACHTIGAL, 2008) (SIMONCINI; SZYLD, 2007). Some of them are the *conjugate gradient* (CG) (LANCZOS, 1952; HESTENES; STIEFEL, 1952; REID,) which has shown excellent convergence properties with symmetric positive definitive matrices (SPD), GMRes (SAAD; SCHULTZ, 1986) and BiCGStab (VORST, 1992), which can be seen as general solvers and are specially used to solve nonsymmetric problems. In our context problem, just the *conjugate gradient* is considered.

When the iterative methods are taking too many steps to attain convergence, modification on the original system are required. This is known as *preconditioning*, which when it is choosen correctly, accelerate dramatically the convergence. The action of a preconditioner is to modify the spectral properties of the coefficient matrix, i.e., producing spectra with almost all eigenvalues clustered around 1, (SAAD, 1999; DEMMEL, 1997). This is still an open and very important research area in scientific computing, many efforts are focused in developing a good general preconditioner instead of improving the existing iterative algorithms. We can state the problem of preconditioning as find a matrix M with the following properties,

- M^{-1} is an approximation in some sense of A,
- M^{-1} is easy to obtain.

Some common preconditioners are *diagonal* or *Jacobi*, *block Jacobi* (BJ), *additive Schwarz methods* (ASM), *incomplete Cholesky factorization* (ICC) for symmetric problems, *incomplete LU factorization* (ILU) for nonsymmetric problems, and domain decomposition based. Extense review of this can be founded in (SAAD, 1999; BENZI et al., 2005; DEMMEL, 1997). The preconditioners considered in this work are ILU, ICC, BJ and ASM. An ILU preconditioner is in general an LU factorization with some set of entries dropped during the factorization process. Different algorithm of discarding new nonzeros lead to differents level of ILU, thus an ILU of level zero, called ILU(0), is the simplest form of ILU preconditioner, it has the same sparsity pattern of the original matrix (SAAD, 1999). In the same way, we get an ICC factorization but applying Cholesky factorization instead of LU factorization. The *block Jacobi* preconditioner can be thougth as a generalizations of a diagonal preconditioner. The former is the simplest preconditioner which consist of just the diagonal entries of the matrix, so if $A = (a_{ij})$, the preconditioner will be $M = (a_{ii})$. Consequently, a block diagonal preconditioner which can be described as follow: let

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1k} \\ \vdots & \ddots & \vdots \\ A_{k1} & \cdots & A_{kk} \end{pmatrix}$$

be a block matrix, with diagonal square matrices A_{ii}. Then a block diagonal preconditioner has the form,

$$M = \begin{pmatrix} M_{11} & & \\ & \ddots & \\ & & M_{kk} \end{pmatrix},$$

choosing $M_{ii} = A_{ii}$ we obtain a block Jacobi preconditioner.

This preconditioner is specially interesting in parallel computing, since each block can be chosen to match the division of variables over processors. The idea using this preoconditioner is to perform an incomplete factorization (ILU or ICC) on each block, eliminating the fill-in between blocks, making it suitable for parallel computing since it does not need communications between subdomain.

There are more radical approaches to exploit the parallelism in incomplete factorization, which are based on renumbering of the unknowns. These techniques are known as *multicoloring* or *graph-coloring* (SAAD, 1999). Others techniques for parallel preconditioner are multi-elimination incomplete LU, distribute ILU, and domain decomposition approach, see (SAAD, 1999) for a review of this methods.

Ordering

Ordering consists on finding a permutation matrix P such the reordered matrix PAP^T has some desired properties in terms of ensuing factorizations. Sparse matrix ordering have been widely used in combination with direct methods to reduce fill-in (DUFF; ERISMAN; REID, 1989; GEORGE; LIU, 1981). On the other hand it has been shown, mostly in a experimental way, that the reordering in an incomplete factorizations tend to affect the rate of convergence of preconditioned Krylov subspace methods (BENZI; SZYLD; DUIN, 1999; BENZI; HAWS; TUMA, 2001; CHOW; SAAD, 1997),i.e., an incomplete factorization is sensitive to the ordering of the unknowns. Several reorderings have been proposed, in this work we tested four ordering algorithms: *nested dissection*, *one-way dissection*, *reverse Cuthill-McKee*, and *Quotient Minimum Degree*, descriptions of this algorithm can be founded in (GEORGE; LIU, 1981; SAAD, 1999; DUFF; ERISMAN; REID, 1989; GEORGE, 1971; GEORGE,).

Finally, we summarize solvers techniques used on each linear system,

- Intermediate Velocity(SPD) - Solver:CG + Preconditioner:BJ + Reordering on each block
- Pressure(Symmetric) - Solver:CG + Preconditioner:BJ + Reordering on each Block
- Scalar Concetrations (SPD) Solver:CG + Preconditioner:BJ + Reordering on each block

Results obtained with these solvers are showed in chapter 6.

CHAPTER 4

IMPLEMENTATION

This chapter present details about the implementations. First we explain the scope, the library used, and give a sequence call of the main program. We present all the implemented classes using the UML diagram, and describe the algorithm used on each method.

4.1 Scope

The implementation is written in C++ language, which use the object oriented programming paradigm (OOP). The design is originally based on the work realized in our work group, see (SHIN, 2009; ANJOS et al., 2007). The figure 4.1 shows the UML diagram for the code implementation. The original design is maintained but the underlying data structure was totally modified using using both the C++ STL (Standard Template Library) and the Boost Multiarray (http://www.boost.org/) libraries (KARLSSON, 2005), also all methods and data structure necessary for parallelism were added, taken mostly from PETSc. In the figure 4.1 classes with white colors were only changed the data structure, the algorithms were kept equals. On the other hands, the classes **Mesh** and **Simulator** were almost totally changed and adapted to run in parallel environment, and a new class **VTKFile** was added to support parallel I/O. The programming model used was the message-passing, since it is appropriate for cluster environment. It is important to mention that all functions and objects were taken from PETSc and we will reference with *italics* letters. The sequence of function called in the main programm is shown

next.

Algorithm 5: Sequence of Calling on the main program

```
1  Mesh mesh()
2  mesh.readVTK()
3  mesh.Distribute()
4  mesh.PartitionsElements()
5  mesh.MoveElements()
6  mesh.ScattersVertices()
7  mesh.GlobalRenumberVertex()
8  mesh.setCentroid()
9  mesh.GetVertexConectivity()
10 Simulator s(mesh)
11 VTKFile results()
12 SolverPetsc p
13 s.init()
14 for i ← 1 to Nsteps do
       s.applyBC()
       s.solveLinearSystem()
       results.write()
   end
```

The class **Mesh** is responsible for the preprocessing state,that is, read, partition and redistribute the data, and create the mapping for the different orderings used. **Simulator** class uses the informations of **Mesh** together with **FEMLinElement** and **FEMMiniElement** to generate matrix entries. The **SemiLagrangian** class computes the effect of the substantial derivative operator. The solutions of the linear systems are performed in the in the *Solver* class. The main methods implemented on each class will be described in detail below.

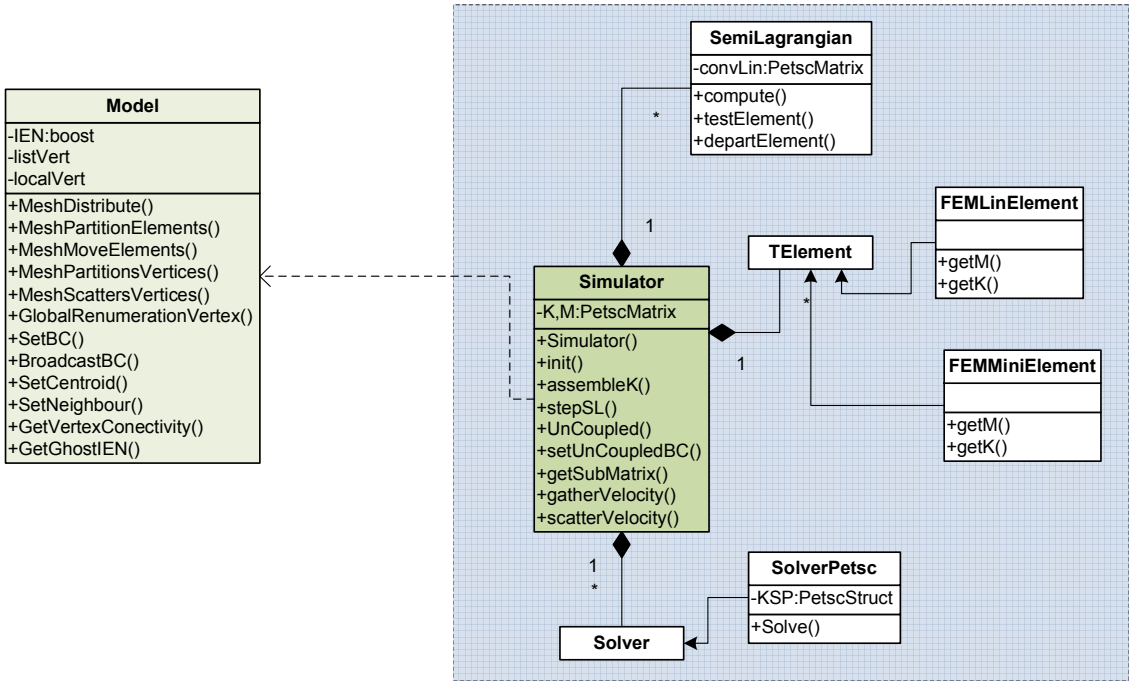


Figure 4.1: UML Diagram.

4.2 Preprocessing - Model Class

Class responsible to hold, build and manipulate all information about the mesh and boundary conditions. Implementations details of main methods to perform this task are give next.

4.2.1 Partitioning and Distribution Elements

In our approach the root process ($rank = 0$) reads the mesh from a file, which is in a VTK format file (KITWARE, 2010). It reads the total number of elements and vertices, and broadcast these values, so each process performs a simple calculation using equations 4.1, to determine how many values will have to receive. Next, the root process distribute linearly the vertices and elements to all process.

$$m_{local} = (numElem)/N + f(numElem, N) \tag{4.1a}$$

$$n_{local} = (numVert)/N + f(numVert, N) \tag{4.1b}$$

$$f(a, N) = \begin{cases} 1 & \text{if } a \ mod \ N > 0 \\ 0 & \text{if } a \ mod \ N = 0 \end{cases} \tag{4.1c}$$

Once all processes have their portion of mesh (vertices coordinates and vertices of elements), we have performed an initial distribution of the mesh. For a real scalable portion of IO code, each process must reads from binary file its portion of mesh, but we choose this as an initial approach. As illustrations of this process we give a small example shown in the figure 4.2.

In order to use a graph partitioner, we need to calculate the adjacency of each element. This job is performed by the root process, and then distributed this list to all other processors. Next, once all processes have their portion, they call collectively a function and create a parallel sparse matrix representing the adjacency list. The matrix does not has any value associated (BALAY et al., 2009).

Using this distributed matrix, we call a set of functions, on which we select the partitioner to use. A typical calling sequence is as follow,

Algorithm 6: Partition and Distribution of elements

Input: Mesh File in VTK format.

Output: Partitioned Mesh - *Each process will have its own list of elements, with globlal numbering of vertices(IEN array).*

1 Root process (rank=0) reads the mesh file and distribute linearly across the process
2 Compute the element neighbors and distribute
3 Mount a parallel adjacency matrix
4 Compute the partitioning
5 Redistribute the element to the correct processor

The ParMETIS library, is called in line 4 of algorithm 6, generating as a result a mapping of node number to processors. This process is illustrate in the figure 4.2, for $p = 2$ and a small mesh with 4 elements and 6 vertices. This is a preliminary information required to perform the redistribution of nodes to the correct processors.

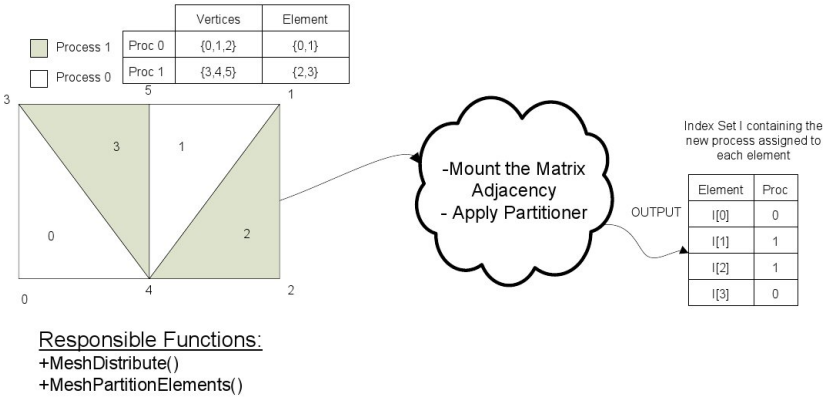
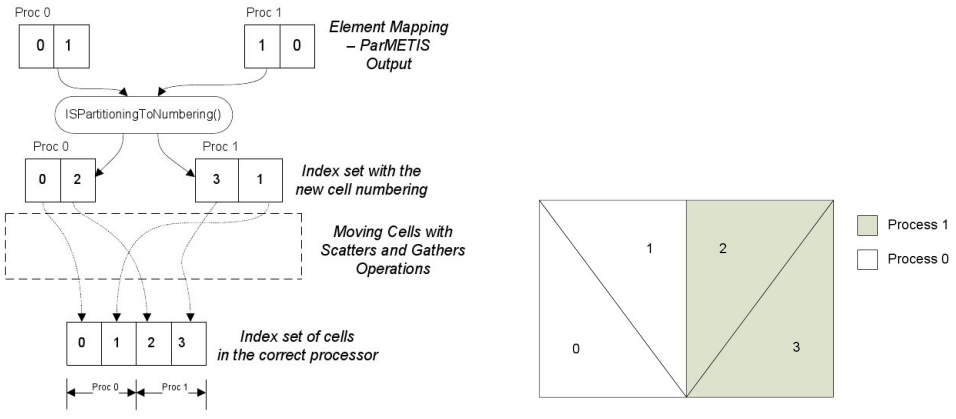


Figure 4.2: Partitioning Process.

As we see in the output index, elements $0, 3$ were assigned to process 0, and elements $1, 2$ were to assigned to process 1. The functions responsible for this task are MeshDistribute() and MeshPartitionElement() in class Model. At this point we don't have yet the necessary information to perform the element redistributions.

Besides the mapping to the new process, we need a new global cells (elements) numbering, this is automatically done with the functions *ISPartitioningToNumbering()*, which gives an index set with new numbering. With this information we can perform the scatters and gathers to move the element to the correct processor.

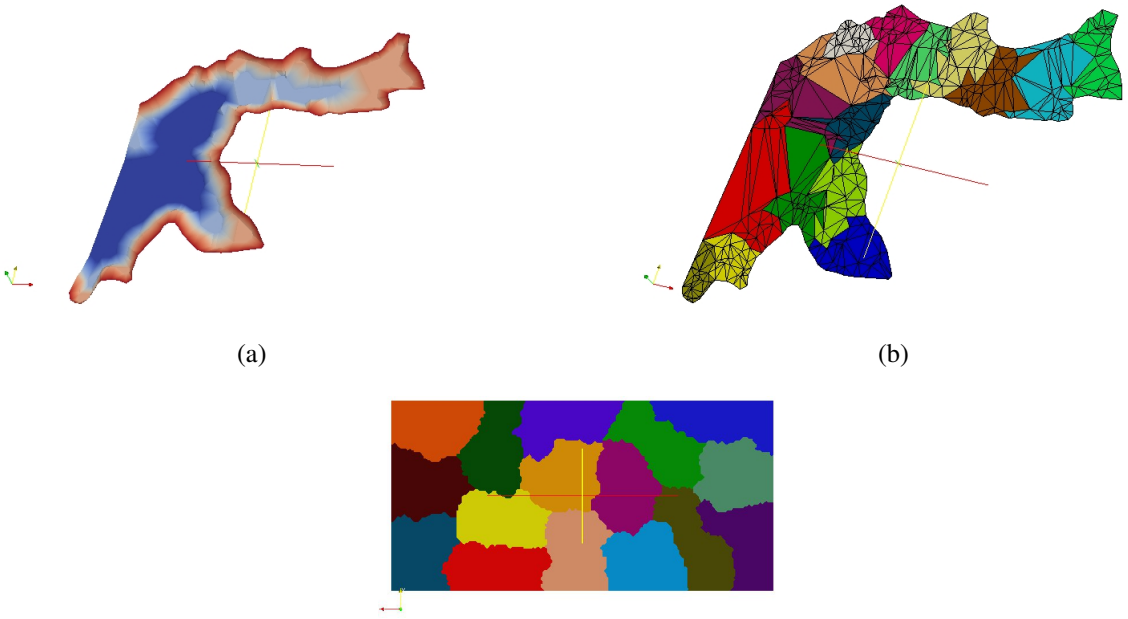
The movement process is done with the *VecScatter* object, which is basically a generalized 'struct' to perform scatters and gathers operations. It can work with both parallel and sequential *Vec* (PETSc's Object for vectors). In order to use this objects we need to setup two index sets, source and destination,i.e., we need to specify which indices to take, and where to put them. Following with our little example, we illustrate the redistributions process in the figure 4.3(a).



(a) (b)

Figure 4.3: Redistribution Process: (a) Simplified Set of Operations . (b) Redistribute Mesh with new ordering.

Here we scattered from a sequential to parallel vector, the figure 4.3(b) shows the final result of this operations, the final distribution of the mesh. These procedures are implemented in MeshMoveElements(). It is important to note that the final index set in 4.3(a) shows the global numerations of elements. This is a very simplified model, just to show how the methods works. In practice, each node represent a data block of size 3, containing the 3 vertices indices (in global numbering) that describe each element of the mesh. Once each process has its correct portions of the mesh we use local numbering to accessing it. The figure 4.4 shows examples of mesh decomposition with 16 processes using the implemented routine.



(a) (b)

(c)

Figure 4.4: Examples of Mesh Partitioning for different domains: (a) Example of a Reservoir Branch. (b) Mesh of the Reservoir Branch after the decomposition. (b) Rectangular Domain Partitioned with 16 processes.

4.2.2 Partitioning, Scattering and Renumbering Vertices

Once we have the list of owned elements, we need the vertices coordinates for each elements vertex, to perform computations in the next stages. As a first step we need to compute what vertices will be needed. Then we perform scatter operations in similar way made in the elements movement, with the difference that in this case we scatter from parallel to sequential

vector. The algorithm 7 show how to achieve these operations.

Algorithm 7: Scatter Vertices

Input: Element nodes array (IEN).

Output: Coordinates (X,Y) of requested vertices.

1 Copy all indices from IEN array to a *list vertices* array
2 Apply *sort* and *unique* algorithm to this *list vertices*
3 Mount a parallel vector with the coordinates obtained during the initial mesh distributions
4 Mount a index set with the indices of vertices needed
5 Perform the Scatter
6 Copy received coordinates (X,Y) in a more efficient container

Figure 4.5 outlines the behavior of the algorithm 7. As we can see, the communication pattern is random, and could become a bottleneck for very large problem. Further improvement in this phase can be a better initial partition to reduce the number of exchanged coordinates.

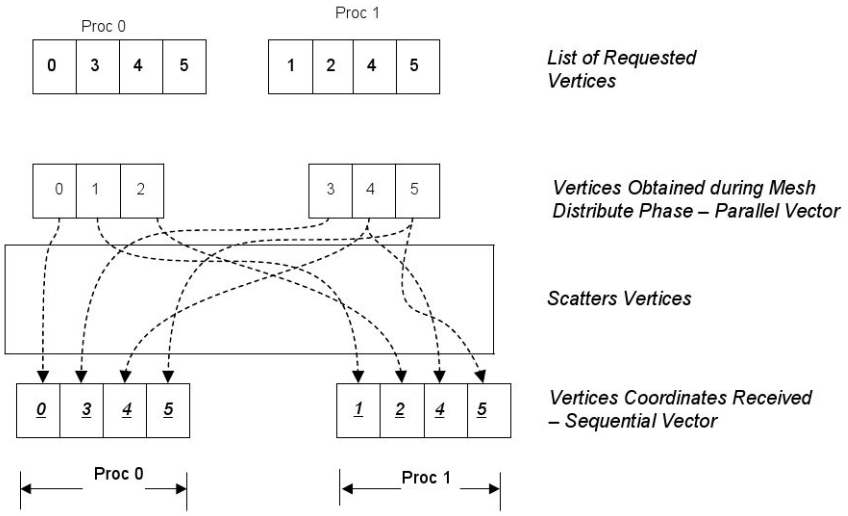


Figure 4.5: Sketch of the scatters vertices algorithm.

At the end of this stage, we have all vertices coordinates needed for computations. Taking into account the data layout used by PETSc see 3.4.1, we need to compute new vertices numerations, in order to generate most matrix entries locally. Following this idea, the set of indices of vertices in the process 0 are numerated first, then the next processor in the rank and so on. The approach was to first resolve the ownership vertices and to then create a function that maps the actual vertices numbering with the new more convenient numeration. In PETSc terminology this is called an *AO* (Applications Ordering), that basically is a struct which store both numerations. This can be expensive in term of memory use. First of all, we need to resolve the ownership, since to form the *AO* object we need that each process contributes with a unique

set of indices. The algorithm proposed is shown below (algorithm 8).

Algorithm 8: Resolving Vertices Ownership

Input: List of Vertices.

Output: List of Owned Vertices.

1 Get information about process neighbors.

2 **for** $p \leftarrow 1$ **to** $nprocs$ **do**

  if $rank > procs[p]$ **then**

    $I_k = I_k + I_p$

  end

end

3 Set difference between the *list of vertices* and I_k

4 Save the difference in a separate list, this will be the *local owned vertices*.

The algorithm uses the information from the neighborhood, to obtain this informations we use the function *ISLocalToGlobalMappingGetInfo()*, where we obtain the number of process connected to the current process, indices of vertices shared with them, and other information.

Once the vertices ownership is resolved, we are able to construct an applications ordering $\mathcal{F}_{\mathcal{AO}}$, to get the new numerations desired. If we applied this process to our example we obtain,

$$\{0, 3, 4, 5, 1, 2\} \xrightarrow{\mathcal{F}_{\mathcal{AO}}} \{0, 1, 2, 3, 4, 5\}$$
$$Proc\ 0\ \{0, 3, 4, 5\} \rightarrow Proc\ 0\ \{0, 1, 2, 3\}$$
$$Proc\ 1\ \{1, 2, 4, 5\} \rightarrow Proc\ 1\ \{4, 5, 2, 3\}$$

This information allows us to create other $\mathcal{F}_{\mathcal{AO}}$ to deal cleanly with different numberings for each degree of freedom. This is quite relevant, since this ordering will determine the data locality, which in turn impacts in the parallel assemble performance. Different approaches for this were taken and will be explained in 4.3.1. The task above corresponds to GlobalRenumerationVertex().

4.2.3 Boundary Conditions Treatment

For boundary conditions the process 0 reads the file that contain the information, and broadcast the whole data. Once all process have this data, each process replace the not owned indices by -1. Then when a vector that contain boundary indices is assemble, the flag *VEC_IGNORE_NEGATIVE_INDICES* is activated before setting any value. In the other hand, for matrix is not so easy to impose boundary conditions. The strategy used are explained in 4.3.2.

4.2.4 Vertices Connectivity and Ghost Element

As we explained in section 3.7.1, the matrix preallocation is crucial for performance. In order to do that, we need to estimate the number of non-zeros per row that the matrix will have. In our approach, we determine the maximum non-zeros. More specific calculation (which are in turn more efficient) can be done.

Unfortunately the strategy to compute the number of non-zeros (nnz) per row in a sequential matrix is different that for parallel matrix. This stems from the fact that for parallel matrix, we need to preallocate two matrices for process, i.e., each process has the portion of the global matrix separated in two matrices,a square submatrix and a rectangular submatrix. Then we need to determinate separately the non-zero entries for each matrix. These numbers are calculated by both function VerticesGetConnectivity() and GetGhostElement().

Algorithm 9: Getting Ghost Element

Input: IEN array.

Output: IEN Ghosted per process.

1 Get information about process neighbors.

2 **for** $i \leftarrow 1$ **to** n **do**

search I_n in IEN

save connectivity in $IEN_{Ghosted}$

end

3 Get the number of element ghosted.

4 *MPI_alltoall* to exchange how much element will be received from each process.

5 *MPI_Sendrecv* to exchange the ghosted element.

The VerticesGetConnectivity() function basically counts the number of neighbor on each vertex, which will be equal to the number of element neighbors. This information should be enough to preallocate a sequential matrix. In order to preallocate a parallel matrix, we need to estimate the nnz per row for both diagonal and off-diagonal matrix inner into a parallel matrix. That is way we need the second functions GetGhostElement(), to estimate the amount of off process values or nnz per row of off diagonal matrix. The algorithm 9 is used to this purpose.

4.3 Simulator Class

The **Simulator** class is responsible to perform all computations related to the simulations, like matrix assembling, boundary conditions applications and linear system setuping. Different assembling approaches were used, which will be explained in details below.

4.3.1 Techniques used for Matrix Assembling

Once we have the mesh partitioned in a number of sub-domains equals to the number of processes and if we want to obtain the same sparsity pattern shown in the figure 2.3, each sub-domain will contribute on different portions of matrix, i.e., each processor will generate entry values for unknown that do not to belong to them, resulting in an excessive communication between sub-domains. To avoid this,we propose and compare two different approaches for matrix assembling.

Method 1

Since in PETSc the matrix is distribute row-wise across the processors, we must be careful during assembly in maintaining the data locality, i.e., generate almost all values for the portion of the array that belongs to it. This is accomplished if the *i* row is mapped to the processor p, then so is the unknown *i*. Then it becomes necessary a set of previous steps before assembly the matrix, as shown in the algorithm 10.

Algorithm 10: Setting Mapping for Assembling

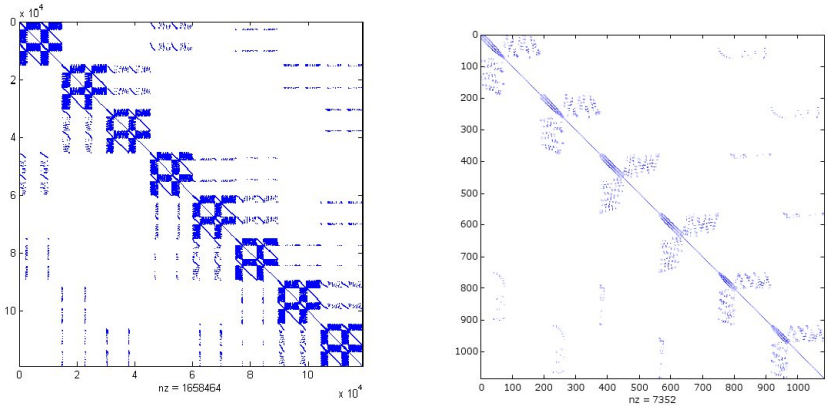
1 Each process counts the number of owned dofs for velocity.
2 Do MPI_Scan so that every process knows its starting offset.
3 Each process numbers owned dofs starting with this offset.
4 Create two $\mathcal{F}_{\mathcal{AO}}$ one for each degree of freedom per node.

$$\{Indices\ of\ Vertices\} \xrightarrow{\mathcal{F}_{\mathcal{AO}x}} \{Indices\ of\ rows\ for\ X\}$$

$$\{Indices\ of\ Vertices\} \xrightarrow{\mathcal{F}_{\mathcal{AO}y}} \{Indices\ of\ rows\ for\ Y\}$$

5 With the two $\mathcal{F}_{\mathcal{AO}}$ create one object $\mathcal{F}$.
6 Do the Assembly as explained in algorithm 3.

In this way the mapping object $\mathcal{F}$ is constructed such as almost all unknowns of the process are mapped to the matrix rows that belong to the process. The load balancing here will depend on the quality of vertices partition. Each mapping function is expensive in term of memory, since it is proportional to the number of total vertices, so we concatenate the both resulting mapping for *X* and *Y*, in one array and created a *ISLocalToGlobalMapping* struct to manage in one object all local to global conversions. The number of communications required in the assembling it proportional to the number of vertices in the interface on each sub-domain, which value was attempted to minimize with the graph partitioner. The sparsity resulted using this method is shown in the figure 4.6.



(a) (b)

Figure 4.6: Sparsity Patter of Matrix M and K: (a) Sparsity of Matrix K with 8 process. (b) Sparsity of Matrix M with 3 process.

Method 2

This second technique is focused in the matrix $\mathbf{B}$ and try to exploit the sparsity pattern, this sparsity is equal to the matrix $\mathbf{K}$ (figure 2.3).

The linear system $\mathbf{B}\hat{\mathbf{u}}^{n+1} = \boldsymbol{r}_u^n + \boldsymbol{bc}_2$ (line 3 of algorithms 1) includes both vertices nodes as well as centroids nodes. The influence of the centroids in the total size of matrix is remarkable, due to the numbers of elements is greater than the numbers of vertices. In a effort of reduce that communications we propose the algorithm 11, whose key step is the *static condensation* (KARDESTUNCER; NORRIE, 1987). Essentially we perform the elimination of dof's of the centroids, consequently we do not need to communicate these values to other processes,

expecting a considerable reduction of message exchanged.

Algorithm 11: Centroids dof's Eliminations

1 (l) Mount $\mathbf{B}_{local}$ on each subdomain

2 (l) Using local rhs of 2.27a and $\mathbf{B}_{local}$ mount the local linear system

$$\begin{pmatrix} k_{ss} & k_{si} \\ k_{is} & k_{ii} \end{pmatrix} \begin{pmatrix} \hat{u_s} \\ u_i \end{pmatrix} = \begin{pmatrix} f_s \\ f_i \end{pmatrix}$$

3 (l) Apply *static condensation* on each subdomain
- Calculate $K_{ss}^j = k_{ss} - k_{si}k_{ii}^{-1}k_{is}$
- Calculate $F_s^j = f_s - k_{si}k_{ii}^{-1}f_i$

4 (G) Mount K_{ss} and F_s
$K_{ss} = \sum_{j=1}^{n} K_{ss}^j$
$F_s = \sum_{j=1}^{n} F_s^j$
where n is the number of subdomains

5 (G) Solve the global system $K_{ss}u_s = F_s$ and gather $\hat{u_s}$

6 (l) Calculate on each subdomain $u_i = K_{ii}(f_i - K_{is}\hat{u_s})$

7 (G) Mount the global solution $\hat{\mathbf{u}}^{n+1}$

End

(l) indicates local operations, (G) indicates global operations

The matrix k_{ii}^{-1} is not expensive, because it is almost diagonal. The assembly in line 3 is done with a mapping similar to **Method 1**. Once we have the intermediate velocity, we can proceed as described in 1. In this manner, we have two methods to perform the linear system calculation, *method* 1 resolves all linear with a convenient dof's permutations, *method* 2 is equal to *method* 1 except that the intermediate velocity is calculated using algorithm 11.

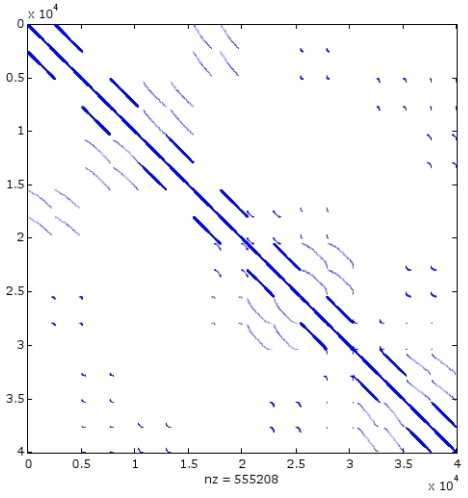


Figure 4.7: Sparsity of K_{ss} with 8 process.

4.3.2 Boundary Conditions

The Dirichlet boundary conditions are applied to the global linear system. The usual approach to avoid the loose of symmetry is to zero the *i*-column and and *i*-row associated to the *i* boundary conditions, put 1 in the *(i,i)* entries of the matrix and the *i*-column to the right-hand side (rhs). Since we use the CSR sparse format extract a column could be very expensive, in terms of CPU time. In this work we use an equivalent approach, which is to replace the *i*-row associated with the *i* boundary conditions, by rows of identity matrix and putting known value in the *i* entry of the rhs, using the function *MatZeroRows()*. Then to use symmetric solvers method we employ the *PCREDISTRIBUTE* preconditioner, which basically redistribute the matrix and solve a smaller system. In the following iterations before reassembling the matrix we activate the option MAT_NO_NEW_NONZERO_ENTRIES.

4.3.3 Solving The Linear System

Resolution of the linear system in PETSc library are based in the *KSP* object, the central part of the library implementations (figure 3.3), through this we have access to a wide variety of solvers and preconditioners, covering parallel, sequential, direct and iterative solvers, as well as an interface to external packages solvers, see (BALAY et al., 2009) for full list of options.

An attractive feature of the implementations is the algorithm independence, i.e., we can change or choose particular solvers and/or precondiotioners at runtime between runs of the program and even inside the program, passing an argument by command line and/or a file, and inside the programm this is passed to the corresponding context, this is useful when we are trying to compare different methods for a given problem. A similar paradigm is employed with most object in the package, which enables a runtime customization. This is done by maintaining a database of options, which basically have three possible inputs: file, environmental variable and command line. Others features of PETSc design are reported in (MATHEMATICS,) (BALAY et al., 1997) and (GROPP, 1998).

In our model we solve three linear system, to choose the algorithms and other options for each linear system we use the function *KSPSetOptionsPrefix()* in the constructor of each object **SolverPetsc**. With this functions we assign a name to each object (e.g. "velocity","pressure","scalar") allowing to control all KSP options database by command line, with just putting the name of the object in front (e.g. "-velocity_ksp_type cg"). Note that in our case we have three **SolverPetsc** objects, one for each linear system.

A typical sequence of function call in PETSc is shown below. Actually this calling is made inside a **SolverPetsc** object.

1 *KSPCreate()* // Create the object KSP, all processor must call this. This is part of the constructor in SolverPetsc class.

2 *KSPSetOperators()* // Associate the matrix of the linear system with the KSP object.

3 *KSPGetPC()* // Get a pointer to the preconditioner object.

4 *PCSetType()* // Set a preconditioner.

5 *KSPSetTolerances()* // Set stopping criteria.

6 *KSPSetFromOptions()* // This reads database options for the KSP objects and will override any options specified above.

7 *KSPSolve()* // Solves the linear system.

The call 3,4 and 5 inside the program are optional, since all options can be controlled by command line and set in call 6. An example of a runtime option setup could be "-velocity_ksp_type cg -velocity_pc_type ilu", this means that we chose the CG algorithm for the solver and ILU for the preconditioner.

4.4 Postprocessing

As mentioned in section 3.5, the storage could be another performance bottleneck if just one process would collect all the results in one file. Then, we implemented a new class **VTKFile** where each processor could store data independently of each others. This is done using the XML format of Visualization ToolKit (VTK), which has parallel format support, with the ability to save local data into a separate file without knowledge of the entire global dataset (KITWARE, 2010), reducing dramatically the communications between process.

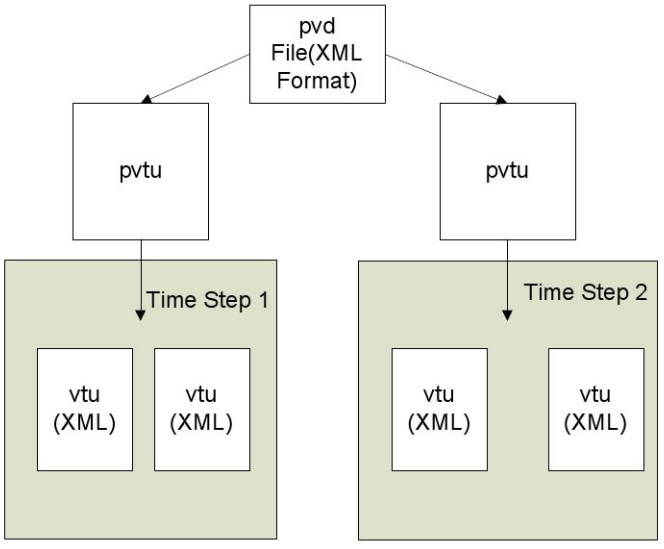


Figure 4.8: VTK Files Hierarchy .

The parallel format of VTK specifies a set of files, with a hierarchy shown in the 4.8. Each processor saves its local data in the vtu file, in a serial format. The pvd and pvtu does not store any data, they only points to the set of serial vtu files, to keep track of time-varying data.

CHAPTER 5

COMPUTATIONAL PLATFORM

In this chapter we introduce the hardware architecture and the software library used in the numerical experiment. We give a brief overview of the multicore hardware and present some benchmark results performed, in order to characterize the performance in the cluster.

5.1 Hardware Platform

Our platform consist on fours nodes with multi-core processor, interconnected by a 48 port Gb Ethernet switch. Figure 5.1 shows a simplified diagram of the cluster.

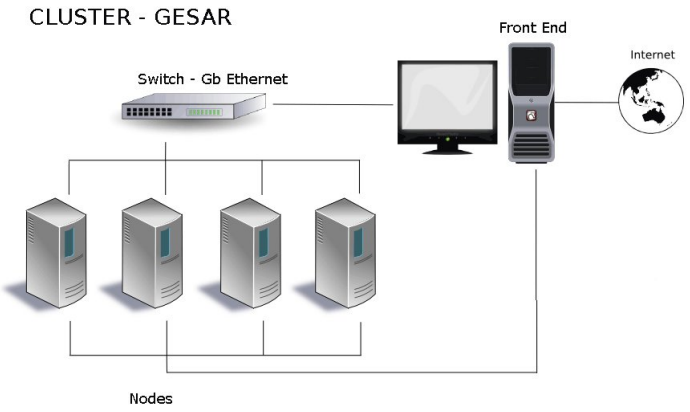


Figure 5.1: Diagram of the Cluster.

Each node has two Intel Quad-Core Xeon (Harpertown) $E5410$. Table 5.1 give a summary of the main features of each node.

Table 5.1: Architectural summary

Intel Quad-Core Xeon E5410	
Clock(Ghz)	2.33
L2 Cache	12MB
Bus Speed (Mhz)	1333
GFLOP/s per core	9.3
System	
# Sockets	2
Cores/Socket	4
L2 Cache	4x6 (Shared by 2)
DRAM Bus	DDR2-667
RAM Available	16GB
Hard Disk	500GB
Operating System - Kernel	Ubuntu 9.04 - 2.6.28-18-server

The Intel Xeon E5410 processor is based on the Intel Core microarchitecture, including on each core a 256KB L1 cache and each two cores (one chip) has shared 6MB L2 cache. The whole system has 74.7 GFlop/s peak performance, but will likely never attain the peak due to the memory bandwidth limitations. The figure 5.2 shows the microarchitecture for this processor.

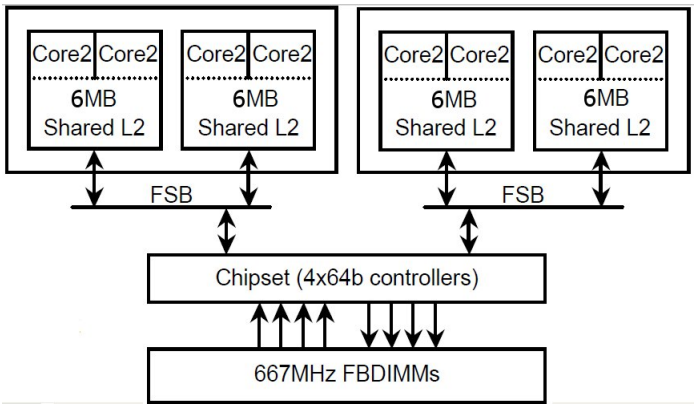


Figure 5.2: Architectural overview for dual-socket×quad-core Intel Harpertown.

5.2 Libraries and Compilers Used

Three main libraries were used the Intel Math Kernel Library version 10.2 Update 4 (**10.2.4.032**), MPICH2 version **1.3a1** and PETSc version **3.1-p1**. The compilers used were the Intel C++ and Fortran compiler for Linux version 11.1 Update 5 (**11.1.069**).

The MPICH2 was configured with **nemesis** (BUNTINAS; MERCIER; GROPP, 2006) as communications subsystem, and compiled with icc, icpc and ifort with options "-O3 -static-intel". This release includes a new default process manager HYDRA, replacing the MPD.

The PETSc library was configure with the followings options:

- –download-parmetis=1 –with-parmetis=1
- –with-clanguage=C++ CC=icc CXX=icpc FC=ifort
- CXXOPTFLAGS="-O3 -static-intel"
- COPTFLAGS="-O3 -static-intel"
- FOPTFLAGS="-O3 -static-intel"
- –with-blas-lapack-dir=/opt/intel/mkl/10.2.4.032

With the first option ParMetis (Version 3.1.1) was installed automatically by PETSc and the last option enabled us to, use the core math functions from MKL in PETSc, such as BLAS and LAPACK library.

5.3 Some Benchmark Results

5.3.1 Memory Bandwidth - STREAMS

The STREAM (MCCALPIN, 1991-2007) benchmark (version 5.9) was used to measured the memory bandwidth. This is a synthetic benchmark, written in standard Fortran 77 and C, which measures the performance of four long vector operations,i.e., the length of the array used is defined larger than the cache of the machine and the code is made such as the data re-use is avoided (MCCALPIN, 1995).

The fours kernel operations are: "Copy","Scale", "Sum" and "Triad". Each of them give independent informations about the memory access behavior. The table 5.2 shown what is performed on each operations.

Table 5.2: STREAMS Kernel Operations

Name	Kernel	Iteration	
		Bytes	Flops
COPY:	$a(i) = b(i)$	16	0
SCALE:	$a(i) = q * b(i)$	16	1
SUM:	$a(i) = b(i) + c(i)$	24	1
TRIAD:	$a(i) = b(i) + q * c(i)$	24	2

In order to evaluate memory access behavior on each node two version of STREAMS benchmark for multiple processors was tested, one using OpenMP directives and the other using MPI calls. The compilation was made as follows,

Table 5.3: STREAMS Results (TRIAD operations)

Operations	Memory Bandwidth Achieved (MB/s) with Cores/Thread							
	1	2	3	4	5	6	7	8
TRIAD-MPI	3725.8553	4318.0892	5345.1903	6022.9636	5866.398	6018.7922	6062.9061	6124.5151
TRIAD-Thread	3726.7519	5677.7290	5408.5158	6270.2938	5881.6691	6036.2364	5957.2012	6122.8853

- OpenMP: icc -O2 -openmp -D_OPENMP stream.c -o stream_openmp
- MPI: ifort -O -I/usr/local/include -L/usr/local/lib \
  -lmpich -lpthread -lrt stream_mpi.f mysecond.o -o stream_mpi

The results for these runs are shown in figure 5.3, and in particular for TRIAD kernel are shown in table 5.3.

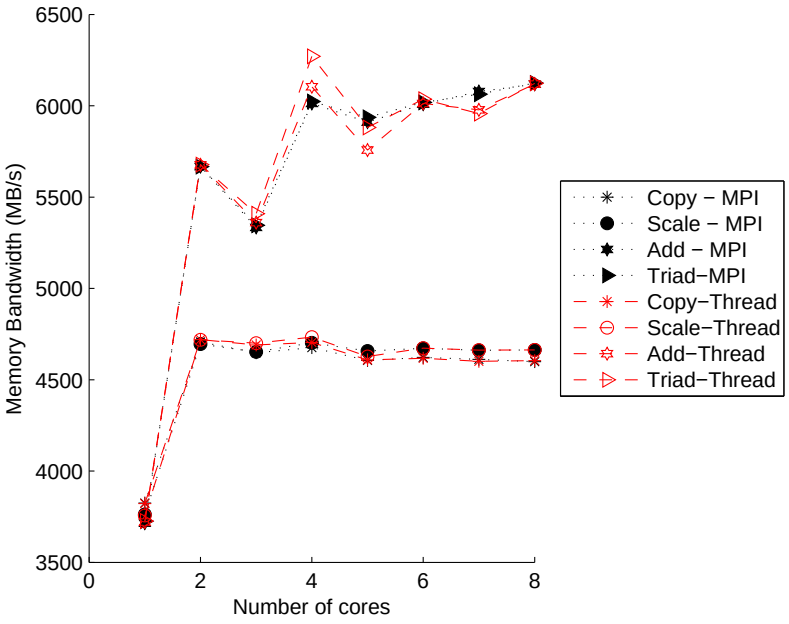


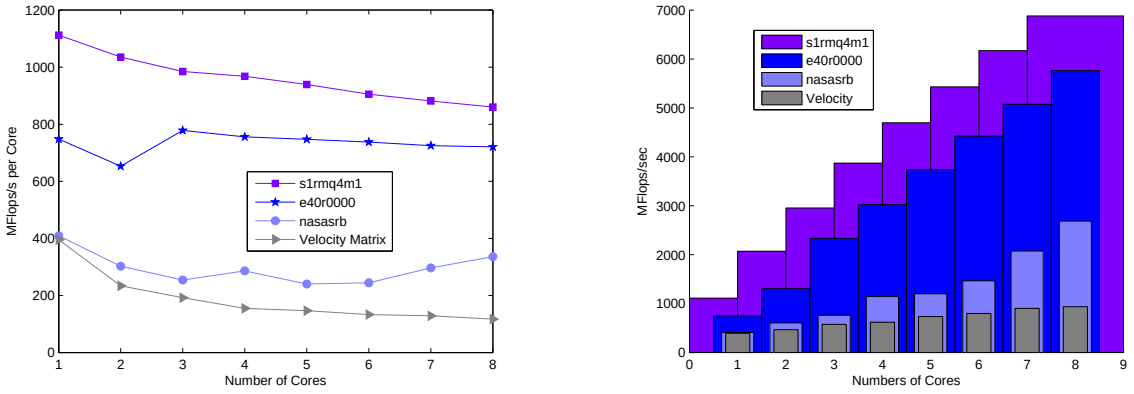
Figure 5.3: Stream Benchmark Results for both MPI and Thread Version.

As we can see both version give similar results. The figure 5.3 shows the memory bandwidth saturation with the increase number of threads or processes, because they share the limited resources as cache and memory bandwidth, see figure 5.2. In consequence, memory-intensive applications can suffer a constrain in the scalability and will not scale as expected.

Sparse Matrix-Vector Product (SpMV)

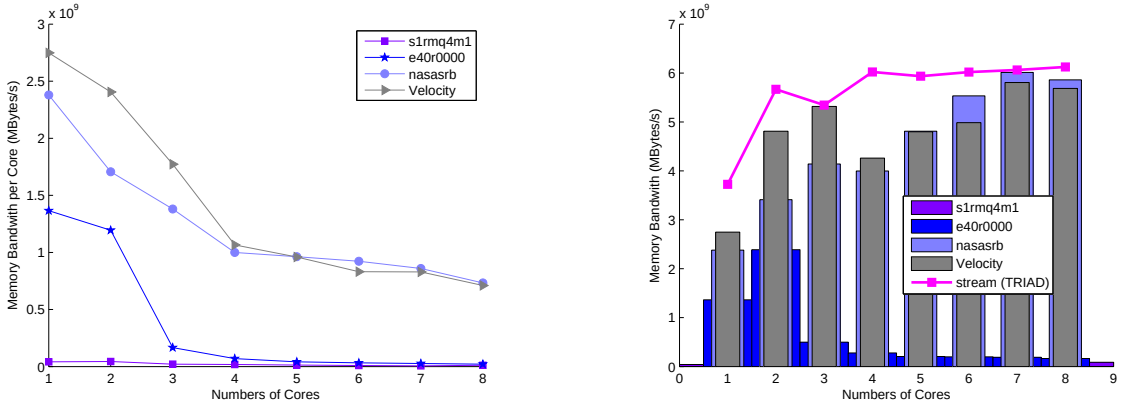
Since the SpMV operations are the kernel of most Krylov subspace methods, and has been shown that the performance of these operations in a give platform is sensitive to the mem-

ory bandwith, since it is a memory intensive operations (GROPP et al., 1999; GROPP et al., 2000). In order to characterize the matrix-vector product in one node, we perform a small test for 4 different sparse matrices, arising from different problems and with different sizes, running tests from 1 to 8 processes. The first three matrices were taken from matrix market (BOISVERT et al., 1997) (s1rmq4m1,e40r0000,nasasrb) and the fourth matrix was the resulting for the intermediate velocity in our problem. Each matrix size was chosen in increasing order, and developed a small code with PETSc objects and functions, to perform the mat-vec product. The MFlops/s and memory bandwidth (MB/s = 1e+06Bytes/s) results are given in figures 5.4 5.5.



(a) (b)

Figure 5.4: MFlops Achieved for each matrix: (a) MFlops per Core Achieved for different Matrices. (b) Total MFlops Achieved.



(a) (b)

Figure 5.5: Memory Bandwidth Achieved for each matrix: (a) Memory Bandwidth Measured per Core. (b) Total Memory Bandwidth Measured with core increase.

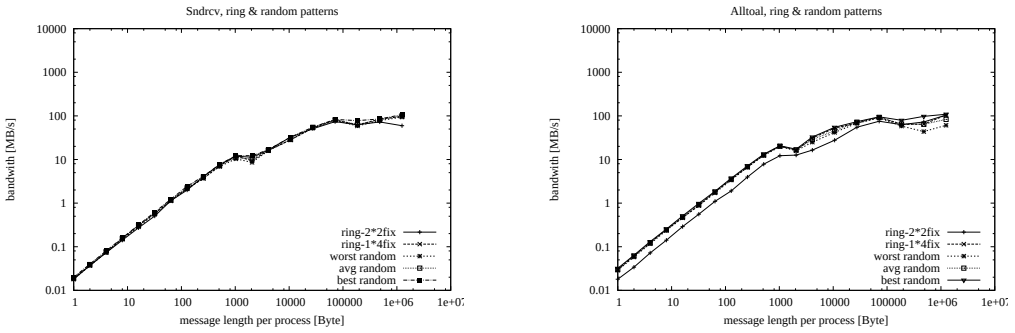
We can see (figure 5.4(a)) a reduction of MFlops/sec per core with the increase of problem, showing the competition of each process for the limited memory bandwidth. This

is demonstrated more clearly in figures 5.5(a) and 5.5(b), where with process increases, they suffers a reduction of their individual memory bandwidth. Finally, we can not expected full scalability with the core/process increases in our platform, when performing memory intensive computation, like mat-vec product.

5.3.2 Communication bandwidth and latency - b_{eff} (Effective Bandwidth Benchmark)

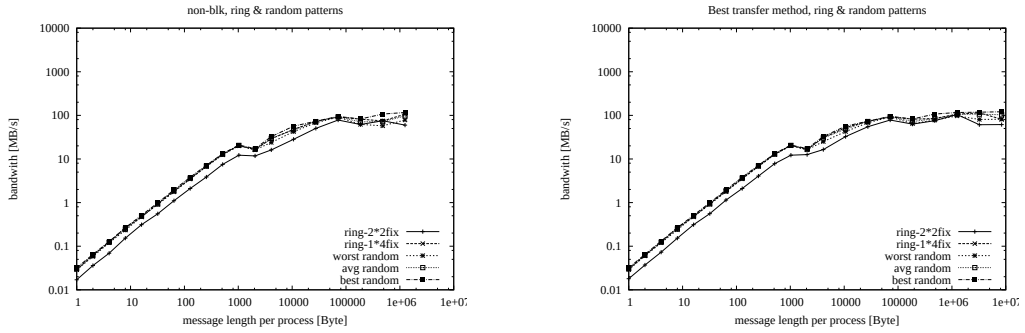
The Effective Bandwidth Benchmark (b_{eff}) (Version 3.5) (RABENSEIFNER; KONIGES,) was used, this measures the accumulated bandwidth of communications network of parallel or distributed computer. Using several communications patterns and message sizes this benchmark give a single number which represent the *effective bandwidth* in the cluster. Basically, the communication patters is based on rings and on random distributions and the communications was implemented with different approaches of MPI (MPI_Sendrecv,MPI_Alltoallv and non-blocking functions). Finally (b_{eff}) is computed as a logarithmic average of the results of both ring patterns and random patters, detail of this calculation and algorithms can be seen in (SOLCHENBACH, 1999) (RABENSEIFNER; KONIGES,).

The results for 4 process (1 process per node) are shown in the figure 5.8. In our test jus two ring patterns are possible, the first is two rings of size 2 and the second is one ring including all process.



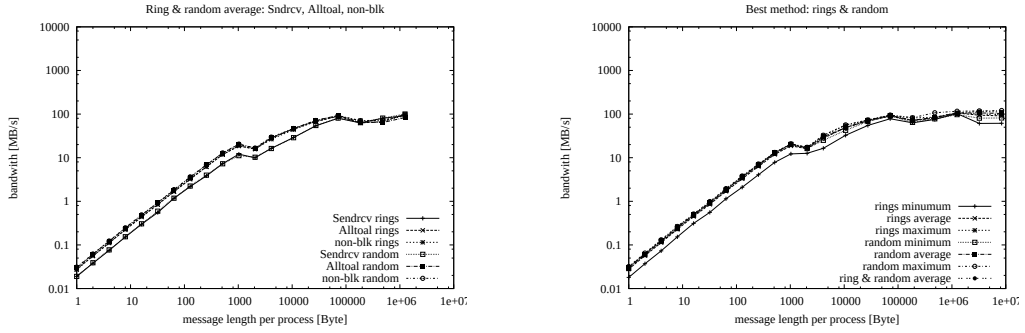
(a) (b)

Figure 5.6: beff Results for different MPI functions and communications pattern: (a) Sendrecv, Ring and Random patterns . (b) Alltoal, Ring and Random patterns .



(a) (b)

Figure 5.7: beff Results for different MPI functions and communications pattern: (a) non-blk, Ring and Random patterns . (b) Best Transfers method,Ring and Random patterns .



(a) (b)

Figure 5.8: beff Results for different MPI functions and communications pattern: (a) Average, Ring and Random patterns for Sendrcv,Alltoall,non-blk. (b) Best Method.

The main results for our cluster configurations, that is, the effective bandwidth measured and latency for different patters are shown in the table 5.4.

The results are consistent with the architecture of the platform, since the switch is a standard gigabit and the communications cards are standard gigabit ethernet cards.

Table 5.4: Effective Bandwidth Benchmark b_{eff} Main Results

b_{eff} **= 145.393 MB/s** = 36.348 * 4 PEs with 1024

	number of processors	b_{eff} MByte/s	Lmax	b_{eff} at Lmax rings& random MByte/s	b_{eff} at Lmax rings only MByte/s	Latency rings& random mircosec	Latency rings only microsec	Latency ping-pong microsec	ping-pong bandwidth MByte/s
accumulated	4	145	8 MB	388	364	34.013	35.817	50.418	123
per process		36	97	91					

Ping-Pong result (only the processes with rank 0 and 1 in MPI_COMM_WORLD were used):
Latency: 50.418 microsec per message Bandwidth: 122.508 MB/s (with MB/s = 10^6 byte/s)

CHAPTER 6

RESULTS

In this chapter the main results obtained are presented. Tests were performed considering two problem. The performance results for preprocessing and solve are reported. Next, comparison between method 1 and 2 described in chapter 4 are made. Finally we compare the flops necessary to solve each linear system.

6.1 Numeric Problem

The implementations were tested using a classic geometry used by both experimental and numerical research. This geometry with the boundary conditions used is shown in the figure 6.1. The no-slip conditions were imposed in all the walls for both x and y components of velocity and zero pressure in the outflow. In the inflow of canal we prescribe boundary conditions only for the x component of velocity and the scalar concentration.

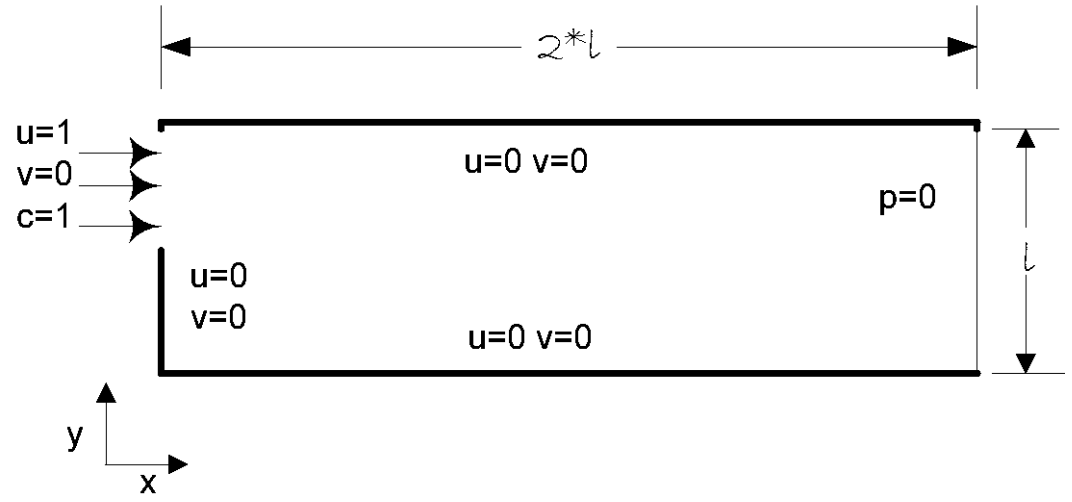


Figure 6.1: 2D Domain with Boundary Conditions.

We test with two different mesh sizes. The number of vertices and elements are shown in the table 6.1, along the dimensions and the number of nonzero elements obtained for each matrix.

Table 6.1: Problems Summaries

	Mesh		Velocity		Pressure		Scalar	
	Vertices	Elements	size	nnz	size	nnz	size	nnz
Problem 1	20000	39402	118804	1658464	20000	408810	20000	138802
Problem 2	200000	395802	1191604	16648864	200000	4109222	200000	1391602

6.2 Performance Measures

To study the performance of the implementations we have used a number of metrics to measured the benefits from parallelism. The first metric was the *executions time* or *cpu time*, defined as the time elapsed from the moment a parallel computations starts to the moment the last processing element finish execution. The second was the *relative speedup* or just *speedup* for our context, defined as the ratio of the elapsed time of the parallel algorithm on one processor to elapsed time of the parallel algorithm on p processors, and can be expressed as follows,

$$speedup = \frac{t_1}{t_p}$$

where t_1 is the elapsed time on one processor, and t_p is the elapsed time for p processors.

The third was the *efficiency ratio* (SAAD; SOSONKINA; ZHANG, 1998) defined as the ratio of the CPU time spent on computing the preconditioner to that on computing the solutions by the

preconditioned solver. This gives a measure of how expensive is to construct the preconditioner relative to the iterations phase. The fourth was the iteration count, which is the number of steps taken by a linear solver to get a solutions under defined stopping criteria. The last two metrics are specially applied to the solving stage.

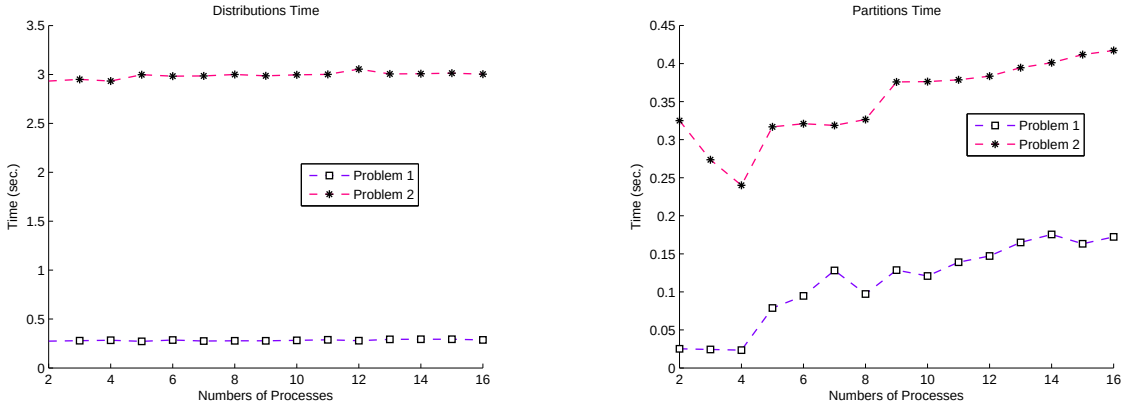
6.3 Running Strategy

In each nodes was allocated a maximum of fours processes, in such way that the intra-node communications were preferred, e.g., if we run 6 processes, the first 4 processes were allocated in the first node, and the rest 2 processes were allocated in the the second node. The runs was performed using the topology-aware allocation "topo:sockets,cores" to avoid processes sharing a core. This is possible using the process manager HYDRA (NADA, a).

6.4 Preprocessing

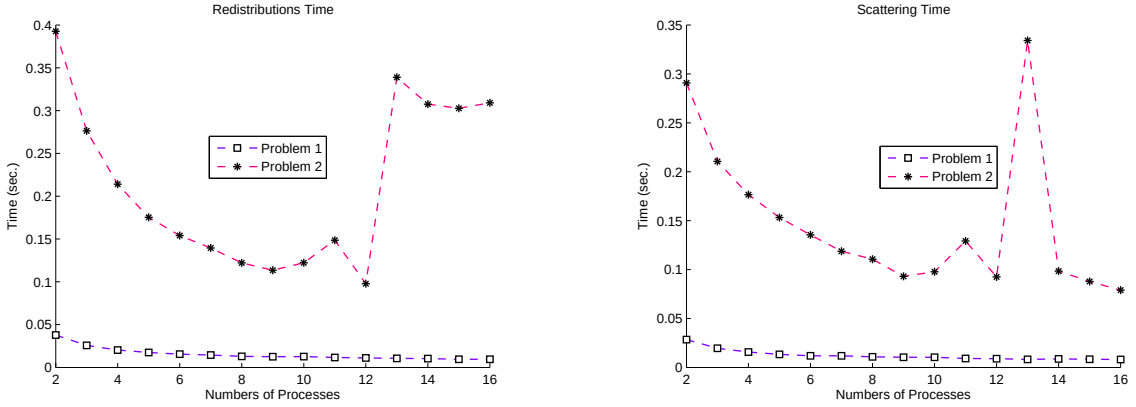
The preprocessing stage, as we saw above, consist in read, distribute, partition, redistribute elements and vertices, i.e., do all necessary steps to get the mesh ready to use. For both problem 1 and 2, it was measured the cpu time on each stage and the results are shown in figures 6.2- 6.5. Figure 6.2 gives the distribute time which shown a random behavior with a slight tendency to increase, but this behavior is associate to how the processes were allocated in the cluster, that is, in some cases we have a non-uniform number of processes doing intra-node and inter-node communications.

The graph partitioning and redistribution stages were measured in order to determine any communications bottleneck. The results in figures 6.3 and 6.5(b) shown that the redistribution for both elements and vertices scale well up to a certain number of processes. This could be explained by looking to figure 6.4 were we get an accentuated load imbalance as a results of the mesh partitions stage, which increases the communications time. On the other hand, we observe almost no speedup in the partitions process, figures 6.2(b) and 6.5(a). However thanks to the parallelism using PARMETIS this stage remains cheap respect to the total execution time, even when the number of processes increases.



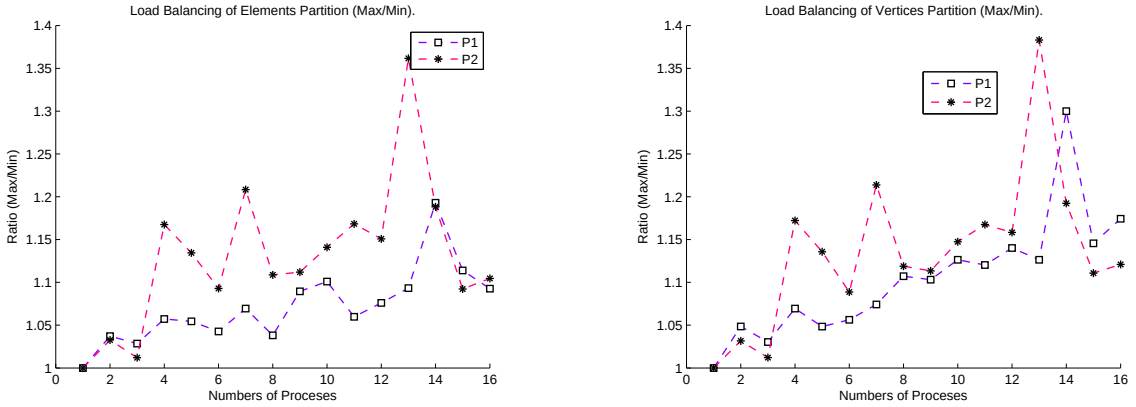
(a) (b)

Figure 6.2: Preprocessing Time: (a) Distributions . (b) Partitions .



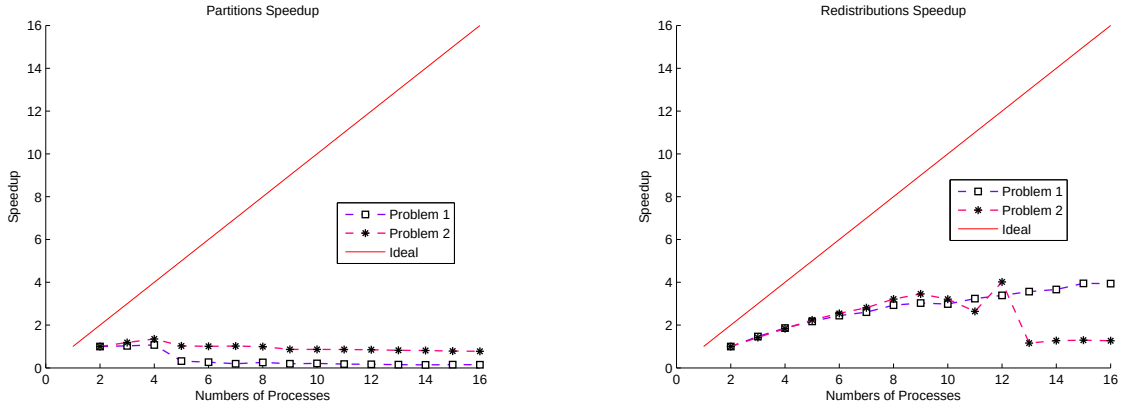
(a) (b)

Figure 6.3: Preprocessing Time: (a) Redistributions. (b) Scattering .



(a) (b)

Figure 6.4: Load Balancing: (a) Load Balancing of Elements . (b) Load Balancing of Vertices .

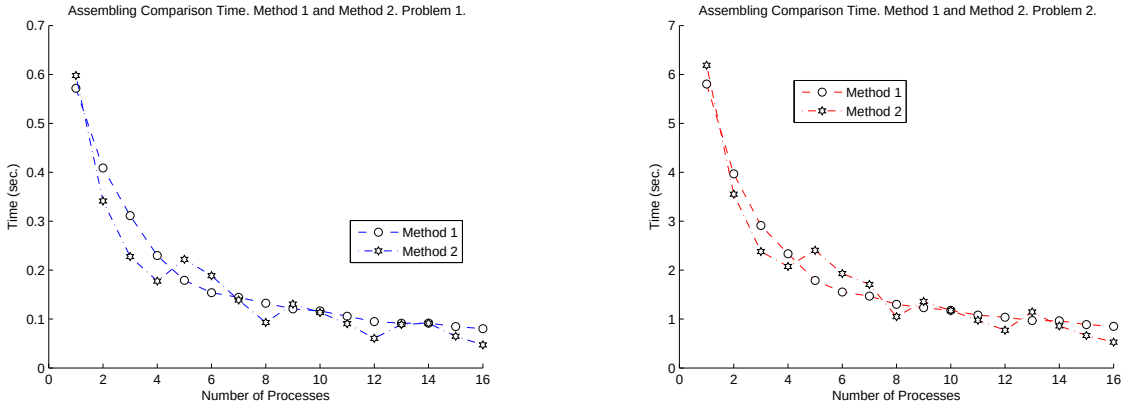


(a) (b)

Figure 6.5: Partitions and Redistributions Speedup: (a) Partitions . (b) Redistributions .

Assembling

In order to verify the bottleneck and the gain from parallelism we also measured the CPU time in the assembling stage, which covers the preallocation process, the builds of all involved matrices, which involves some communications between domains to exchange boundary values. The results for both methods are shown in figure 6.6, the elapsed time is reduced with the number of processes increase, therefore the performance in this stage was improved. It is important to note that, since the partitions are made by element, a set of vertices are ghosted on each subdomain, whose entries in the matrix must be updated generating serial communications on each subdomain or process, hence it is admissible a performance decrease. The speedup result was shown in figure 6.7. From these figures, we can see that Method 2 was aided by the allocations strategy.



(a) (b)

Figure 6.6: Assembling Time (sec.): (a) Problem 1. (b) Problem 2.

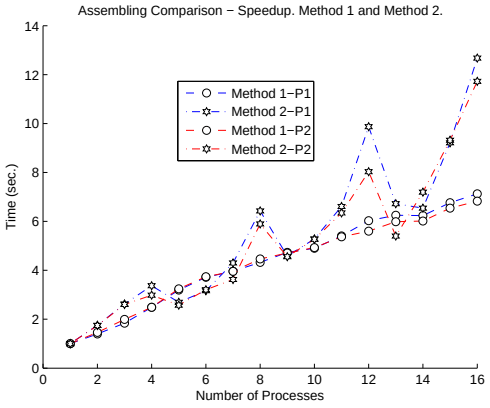


Figure 6.7: Assembling Speedup.

6.5 Solver

In this section we present the results of the solvers with the three linear system. For all matrices we used the conjugate gradient method, and the *Block-Jacobi* (BJACOBI) and Additive Schwarz Methods as preconditioner. For the block Jacobi preconditioner we use the incomplete Cholesky Factorizations with 0 fill-in (ICC(0)) and test different ordering algorithms as natural, nested dissection (nd), one-way dissection (1wd), and the reverse Cuthill McKee (RCM). For the ASM preconditioner we use the restrict type with ILU(0) on each block, and three different overlap 1, 2 and 4. The tolerance used was relative of 1e-08, absolute of 1e-50 and divergence of 10000.

6.5.1 Results for Intermediate Velocity - System $\mathbf{B}\hat{\mathbf{u}}^{n+1} = \boldsymbol{r}_u^n + \boldsymbol{bc}_2$

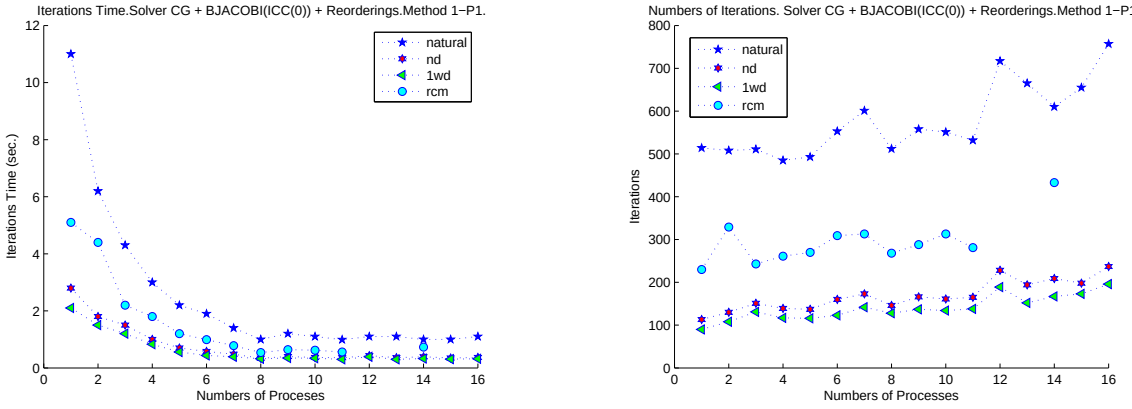
In this subsection we show the results for each methods used and the results for each problem, finally we compare both methods.

6.5.1.1 Method 1

We present and discuss results for both problem 1 and problem 2 using method 1.

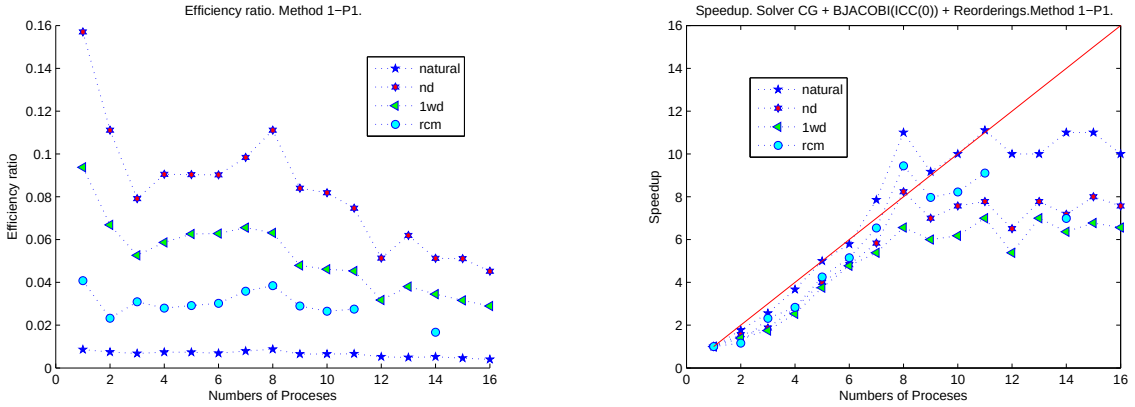
Problem 1

The results for the problem 1 using method 1 are shown in figures 6.8-6.11. We can observe in figures 6.8(a) and 6.8(b) that the best performance obtained was using one-way dissection algorithm, this is the smaller iterations counts, hence the the time was reduced. Note that runs with reverse Cuthill McKee did not converge for some number of process (p=12,13,15). The efficient ratio results are shown in figure 6.9(a), were aside the natural, the cheapest preconditioner was obtained with reverse Cuthill McKee. The figure 6.9(b) shows that the solver scales well until 11 process, where it become flat. It could be due the problem become too smaller from this number of process, therefore the overhead dominate the communications.



(a) (b)

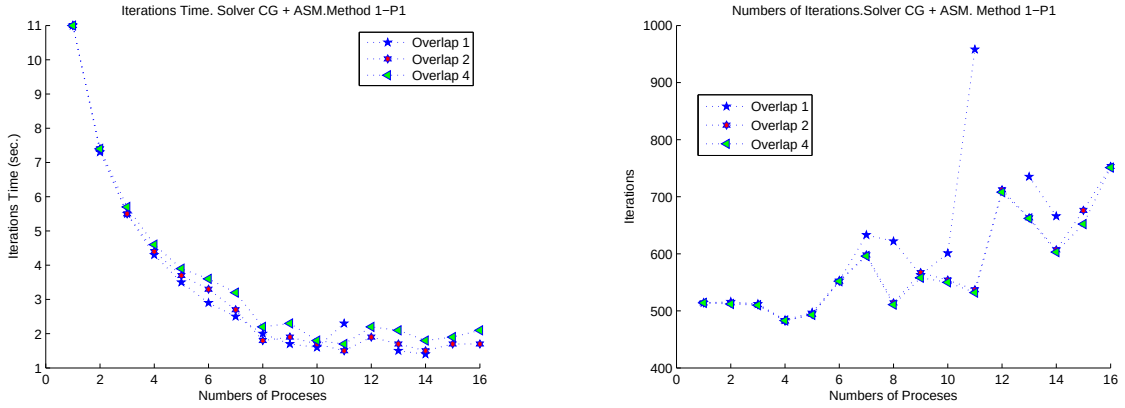
Figure 6.8: Number of Iterations and Time for Method 1. Problem 1: (a) Iterations Time in seconds. (b) Number of Iterations.



(a) (b)

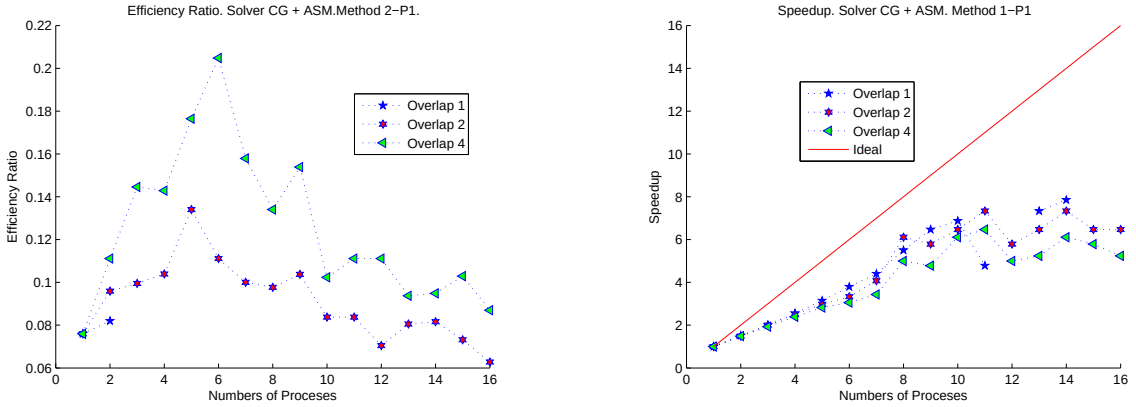
Figure 6.9: Efficiency Ratio and Speedup for Method 1 - Problem 1: (a) Efficiency Ratio. (b) Speedup.

The results for ASM preconditioner are shown in figures 6.11-6.10, compared with the the ICC(0) we do not get better performance with this preconditioner, evidenced by the number of iterations resulting, see figure 6.10(b). The speedup resulted are shown in figure 6.11(b).



(a) (b)

Figure 6.10: Time and Iterations Time for Solver CG with ASM preconditioner Method 1- Problem 1 : (a) Iterations Time in seconds. (b) Numbers of Iterations.

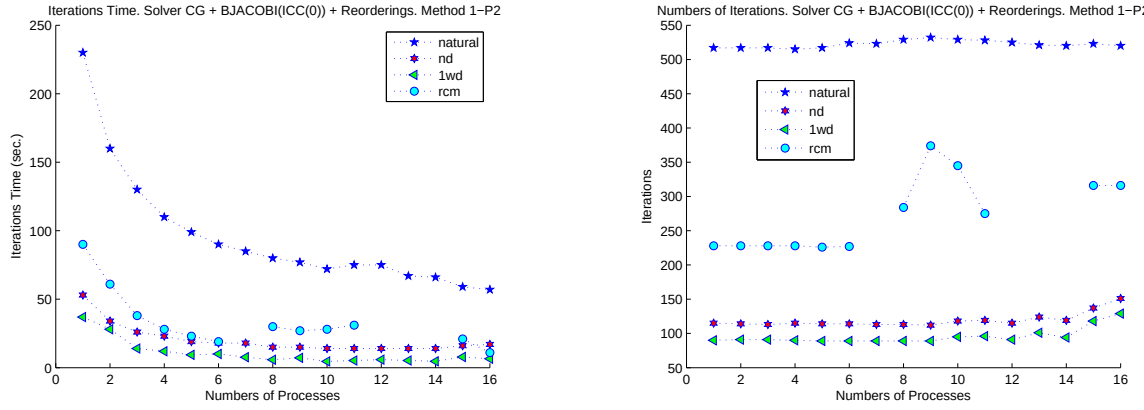


(a) (b)

Figure 6.11: Speedup and efficiency for CG+ASM. Method 1-Problem 1 : (a) Efficiency Ratio. (b) Speedup.

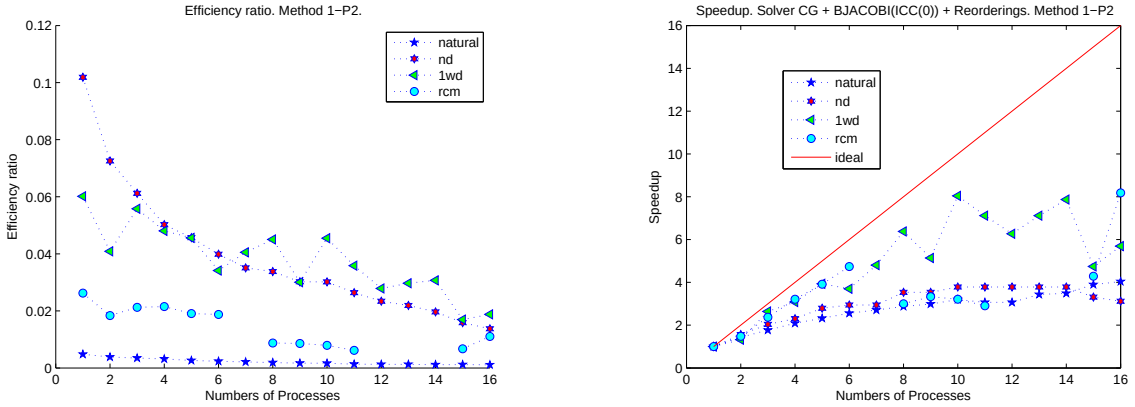
Problem 2

The same set of runs was repeated for problem 2, and the results shown in figures 6.12-6.15. Here, with the ICC(0) preconditioners we get a similar behavior that in problem 1, hence the one-way dissection represent the best iteration count resolution time. We note a variation respect to problem 1 in the efficient-ratio (figure 6.13(a)) and the speedup (6.13(b)), here the one-way dissection suffers a cost increase with the number of process. Respect to the scaling we observe a better scaling only for the one-way dissection, and since the time of the solver is dominated by the matrix-vector product and the number of iterations (figure 6.12(b)) remains almost constant (except for reverse Cuchill McKee), we attribute this scaling behavior to the inter- and intra-node communications pattern.



(a) (b)

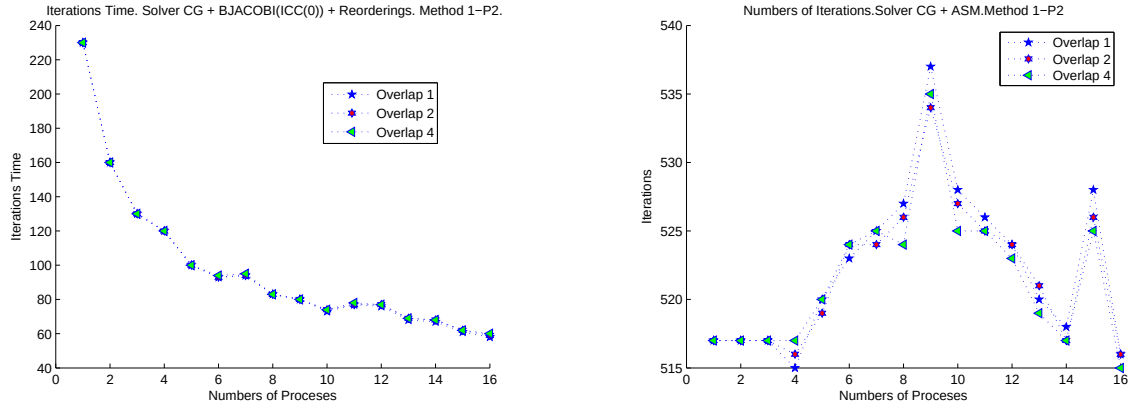
Figure 6.12: Number of Iterations and Time for Method 1. Problem 2: (a) Iterations Time in seconds. (b) Number of Iterations.



(a) (b)

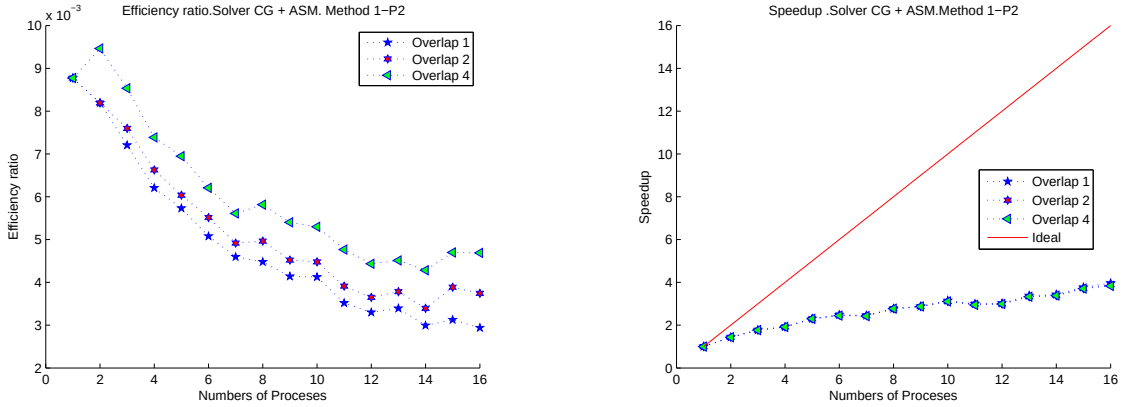
Figure 6.13: Efficiency Ratio and Speedup for Method 1 - Problem 2: (a) Efficiency Ratio. (b) Speedup.

Again, as in problem 1 the results for the ASM preconditioner are not better than the ICC(0) preconditioner. Results are shown in figures 6.14-6.15. Note the almost identical behavior with different overlap values.



(a) (b)

Figure 6.14: Time and Iterations Time for Solver CG with ASM preconditioner Method 1- Problem 2 : (a) Iterations Time in seconds. (b) Numbers of Iterations.



(a) (b)

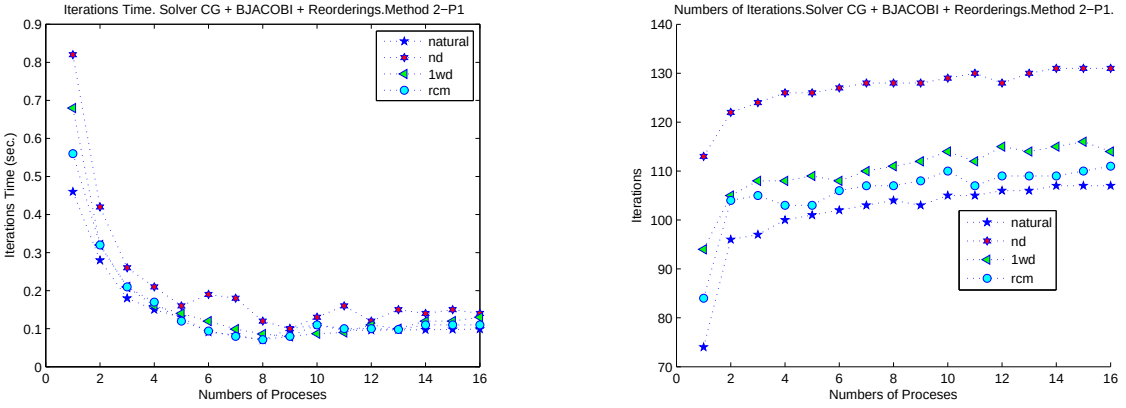
Figure 6.15: Speedup and efficiency for CG+ASM. Method 1-Problem 2: (a) Efficiency Ratio. (b) Speedup.

6.5.1.2 Method 2

We present and discuss results for both problem 1 and problem 2 using method 2.

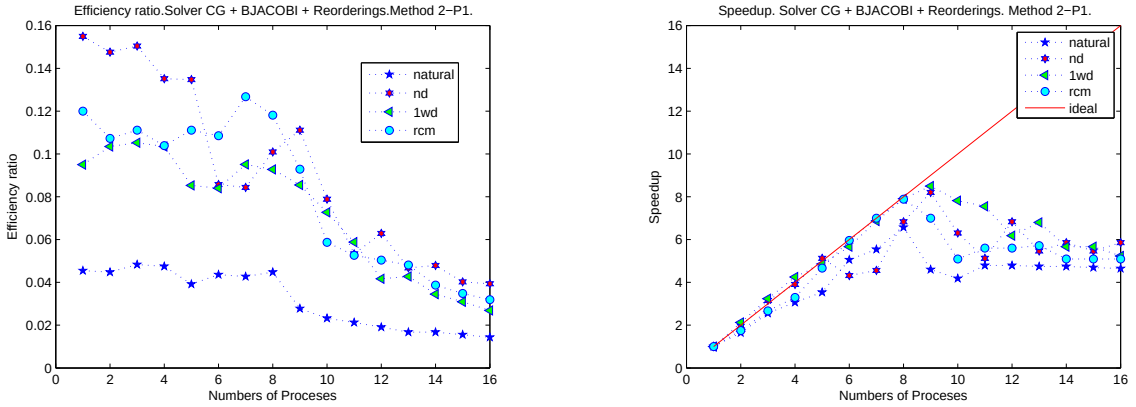
Problem 1

The results are shown in figures 6.16-6.19. Here, unlike the method 1, the best time figure 6.16(a) and number of iterations figure 6.16(b) was achieved using natural ordering, hence this preconditioner is the better one. The figure 6.17(a) shows a good scalability up to eight processes, where it become almost flat, which was probably due to the small problem size.



(a) (b)

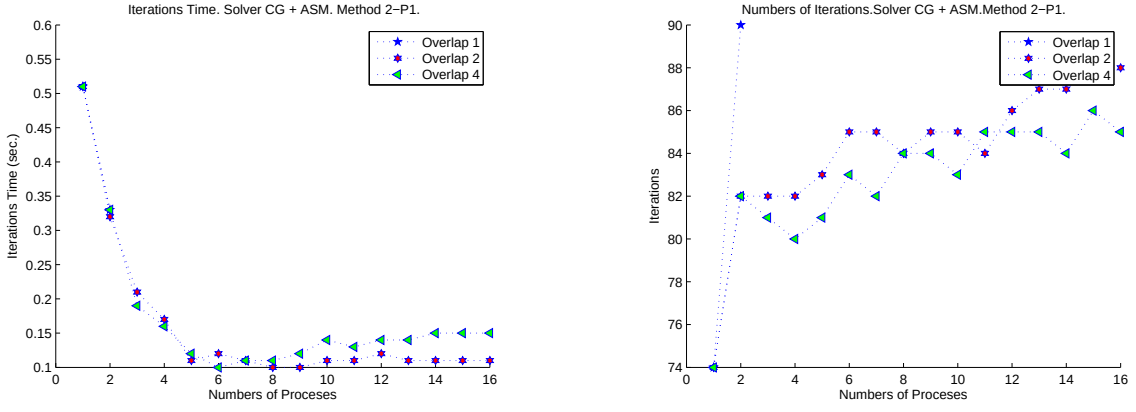
Figure 6.16: Number of Iterations and Time for Method 2. Problem 1: (a) Iterations Time in seconds. (b) Number of Iterations.



(a) (b)

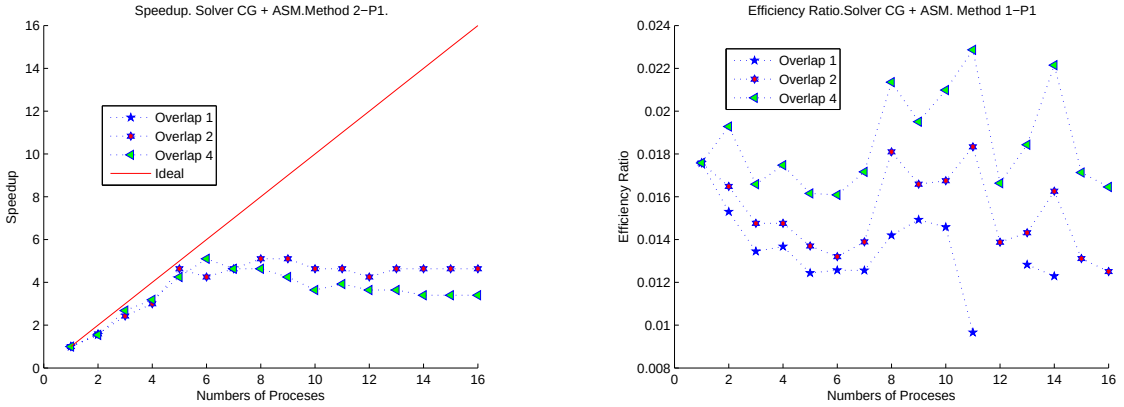
Figure 6.17: Speedup and Efficiency. Method 2 - Problem 1: (a) Efficiency Ratio. (b) Speedup.

The results for ASM preconditioners are shown in the figures 6.18-6.19. Note that with overlap 1 and for number of process larger than 2 the solver did not converge. The best time was achieved using overlap 2, but again this is not as good as the one obtained with ICC(0). The resulting speedup and efficiency with this preconditioner are shown in figure 6.19(a) and 6.19(b).



(a) (b)

Figure 6.18: Number of Iterations and time. Method 2 - Problem 1: (a) Time in seconds. (b) Iterations.

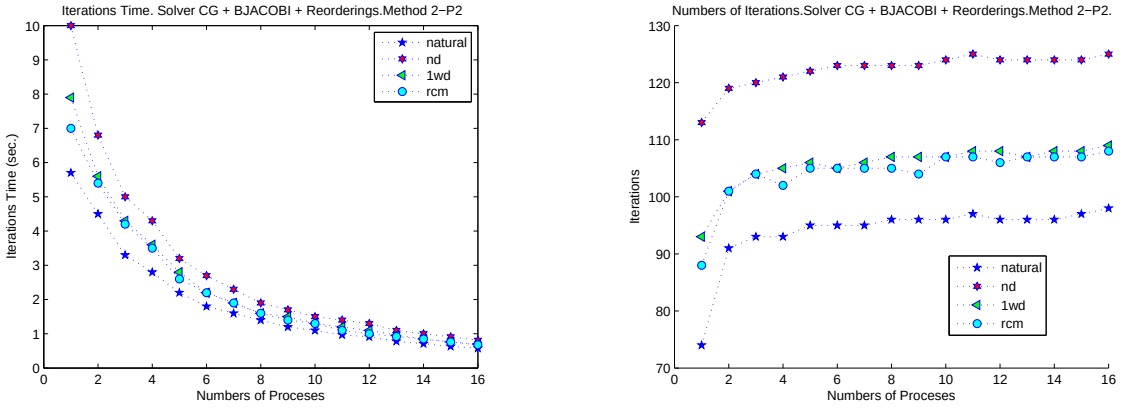


(a) (b)

Figure 6.19: Speedup and Efficiency with Solver CG + ASM. Method 2 - Problem 1: (a) Speedup. (b) Efficiency.

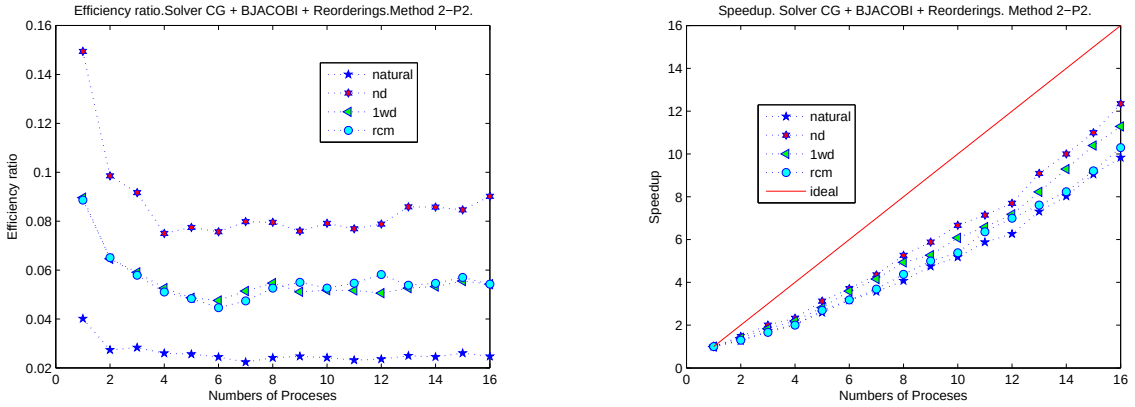
Problem 2

The same options of runs in the problem 1 were repeated for problem 2, the results are shown in figures 6.20-6.23. Agains the natural ordering wins in this case, note that for process increases the executions times (figure 6.20(a)) for different preconditioner are close to each other. This problem with method 2 in particular shows a better scalability (figure 6.21(a)) compared with the obtained in method 1 - problem 2 (figure 6.13(a)), which is probably due to the reduction of the number of nonzero of the off-diagonal matrix with the centroids eliminations, which could reduce the inter-process communications during the matrix-vector product.



(a) (b)

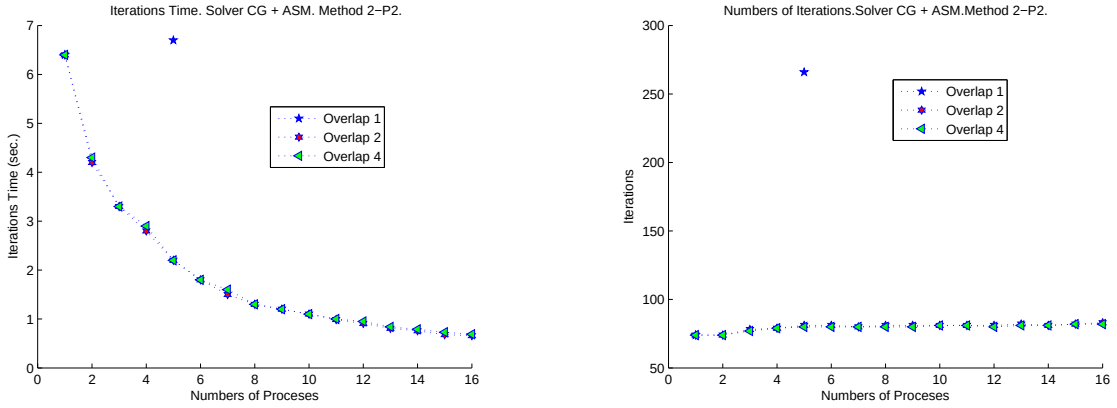
Figure 6.20: Number of Iterations and Time for Method 2. Problem 2: (a) Iterations Time in seconds. (b) Number of Iterations.



(a) (b)

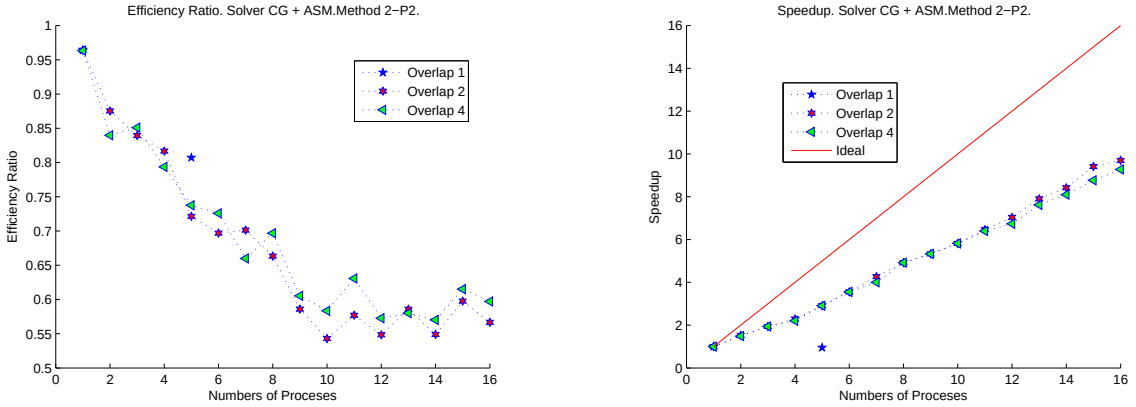
Figure 6.21: Speedup and Efficiency. Method 2 - Problem 2: (a) Efficiency Ratio. (b) Speedup.

Results for ASM preconditioner are shown in figures 6.22-6.23. As in problem 1 the ASM with overlap 1 did not converge for process larger than 2, and the ICC(0) is still better. The resulting speedup is shown in 6.23(a), also we get a better scalability in this case.



(a) (b)

Figure 6.22: Number of Iterations and time. Method 2 - Problem 2: (a) Time in seconds. (b) Iterations.

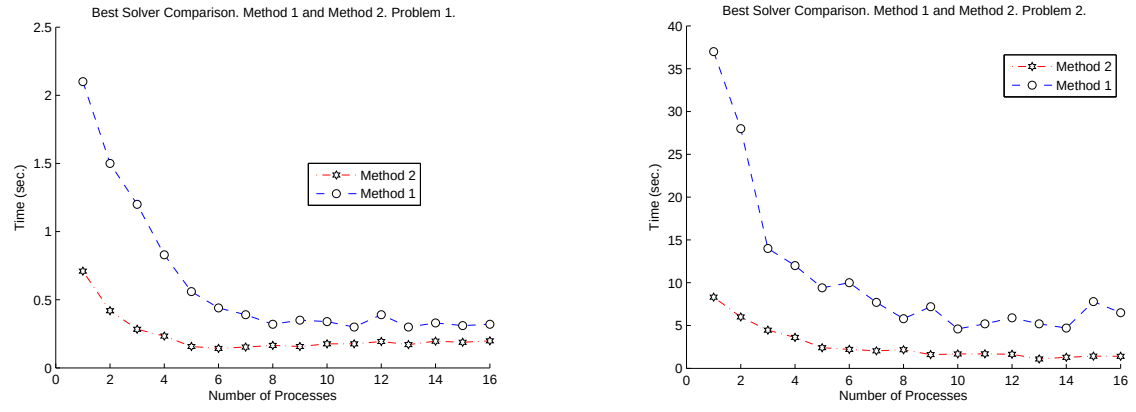


(a) (b)

Figure 6.23: Speedup and Efficiency with Solver CG + ASM. Method 2 - Problem 1: (a) Efficiency Ratio. (b) Speedup.

6.5.1.3 Comparison between Method 1 and Method 2

In figures 6.24 we compare the two proposed methods. That is, we take the best time obtained for methods 1 which is using CG and ICC(0) with one-way dissection ordering, and take the best time obtained for method 2 which is CG and ICC(0) with natural ordering and add the time spent in line 5, 6 and 7 of algorithm 11. This represent the total time to get a complete solution.



(a) (b)

Figure 6.24: Time Comparison between Method 1 and Method 2: (a) Results for Problem 1. (b) Results for Problem 2.

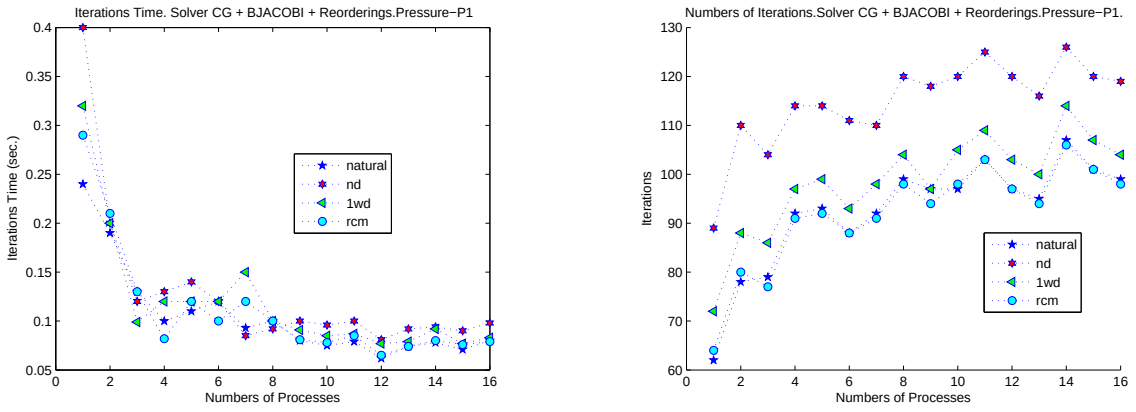
For both problem 1 and problem 2 the method 2 was better, and the difference become superior for the large one.

6.5.2 Results for Pressure - System $\mathbf{D}\widetilde{\mathbf{B}}^{-1}\mathbf{G}\tilde{\mathbf{p}}^{n+1} = -\mathbf{D}\hat{\mathbf{u}}^{n+1} + \boldsymbol{bc}_1$

Here, we present the results for the pressure linear system referenced in the line 4 of algorithm 1, for each problem solved we show the execution time, number of iterations and speedup for each precondiotioner. Only the ICC(0) block jacobi results were presented. Results from ASM preconditioner were discarded because this preconditioner did not converge for most cases.

Problem 1

Figures 6.25(a) and 6.25(b) show the CPU time and the iteration count, respectively. In this case it is not so clear which ordering algorithm was more efficient. The potencial candidates are the natural and the reverse Cuchill McKee. For this problem we get speedup up to 4 process as shown in the figure 6.26, from this value on, it become almost flat, which probably is due small size of the matrix.



(a) (b)

Figure 6.25: Execution Time and Iteration count for Pressure in Problem 1: (a) Execution Time (sec.). (b) Iteration count.

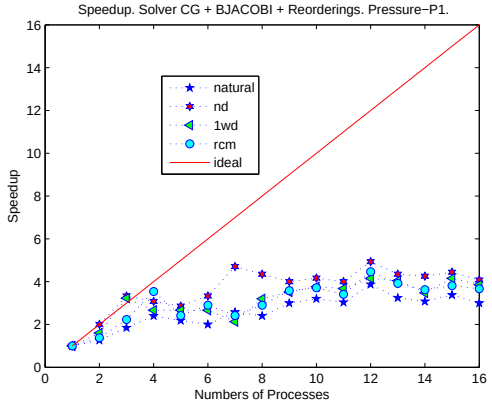
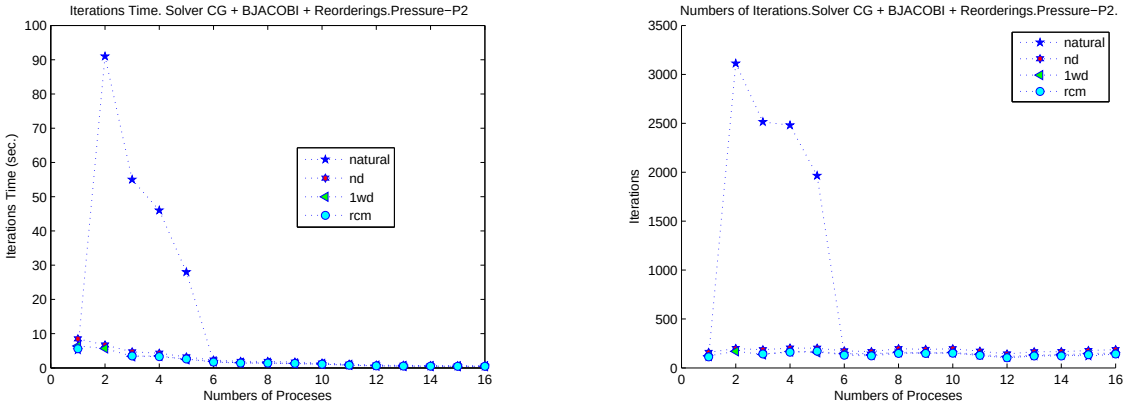


Figure 6.26: Speedup

Problem 2

For this problem we get an abnormal convergence behavior for natural ordering, thus we can say that reverse Cuchill McKee gives a better result for iterations count. Hence the execution time also are better for reverse Cuchill McKee, see figures 6.27(a) and 6.27(b). Unlike the problem 1, this problem size presents a good scalability (figure 6.28).



(a) (b)

Figure 6.27: Execution Time and Iteration count for Pressure in Problem 2: (a) Execution Time (sec.). (b) Iteration count.

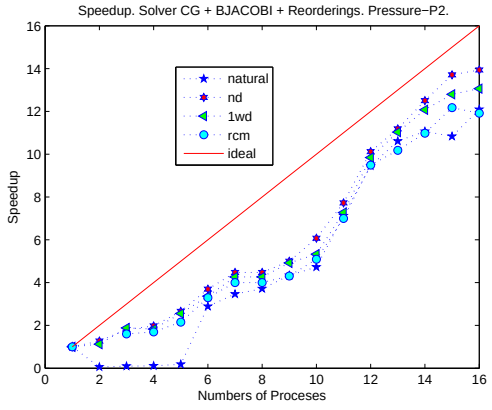


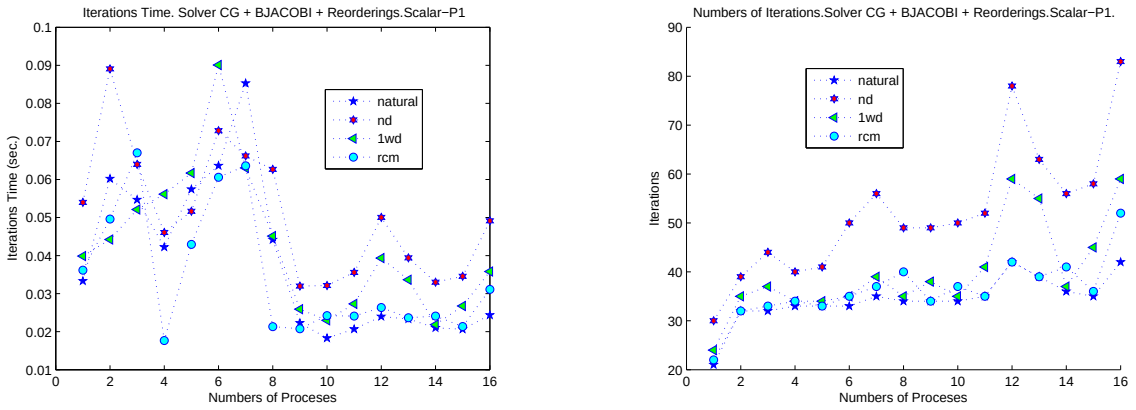
Figure 6.28: Speedup

6.5.3 Results for Scalar Concentrations - System $\mathrm{B}_\theta \tilde{\theta}^{n+1} = \boldsymbol{r}_\theta^n + \boldsymbol{bc}_3$

Results for linear system referenced in line 5 of algorithm 1 are presented in figures 6.29-6.32. For each problem solved we present the behavior of the solver using the only the ICC(0) preconditioner.

Problem 1

Due to the small size of the scalar concentration system we do not get gain with parallelism, but a least the cost of resolve the linear system remains cheap compared with the others linear system (figures 6.29(a) and 6.29(b)). Also, in this case the possible candidates are natural and reverse Cuchill McKee algorithm. As expected the speedup is not good for this problem size (6.30).



(a) (b)

Figure 6.29: Execution Time and Iteration count for Scalar Concentrations in Problem 1: (a) Execution Time (sec.). (b) Iteration count.

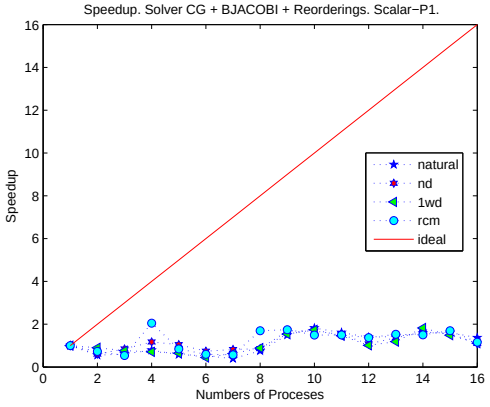
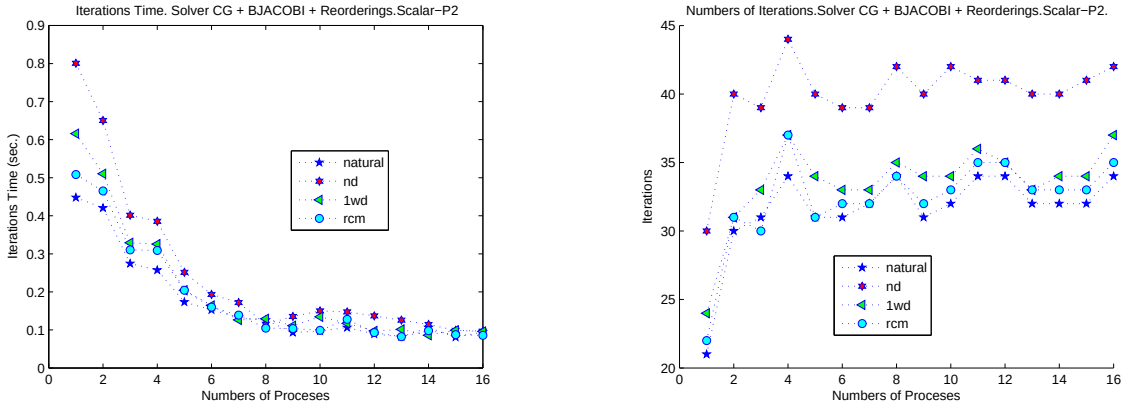


Figure 6.30: Speedup

Problem 2

With this problem size we get more advantage from parallelism, as can be see in figure 6.32. Observing figures 6.31(a) and 6.31(b) we can say that the natural ordering give better performance.



(a) (b)

Figure 6.31: Execution Time and Iteration count for Scalar Concentrations in Problem 2: (a) Execution Time (sec.). (b) Iteration count.

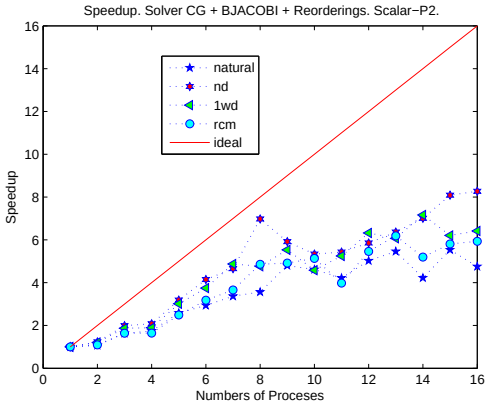


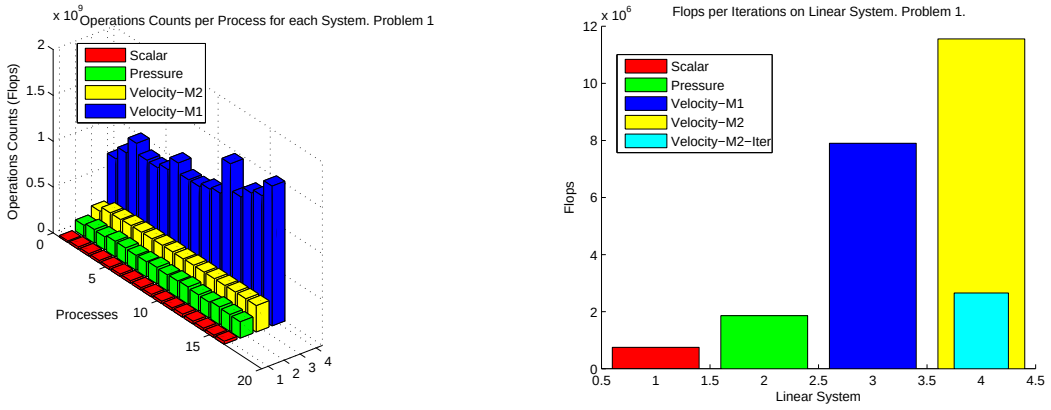
Figure 6.32: Speedup

6.5.4 Flops Comparison for Each Linear System

In this part we show a comparison of Flops (floating point operation count) for each linear systems. In this context we define 1 flop as 1 real operation of type multiply,divide,add or subtract. Results are shown in figures 6.33 and 6.34, one figure for each problem, and were taken the best results for each linear system. That is, we took the one-way dissection for velocity in method 1, the natural for velocity in method 2, reverse Cuchil McKee for pressure and natural for scalar concentration. We divide the results of each problem in two subfigures part(a) and part (b). Part (a) shows the total flops required for each solver to get the final solutions, the dependency on the numbers of iterations to get this solutions explains the variations of flops for different process. Part (b) shows the total flops required for one iterations.

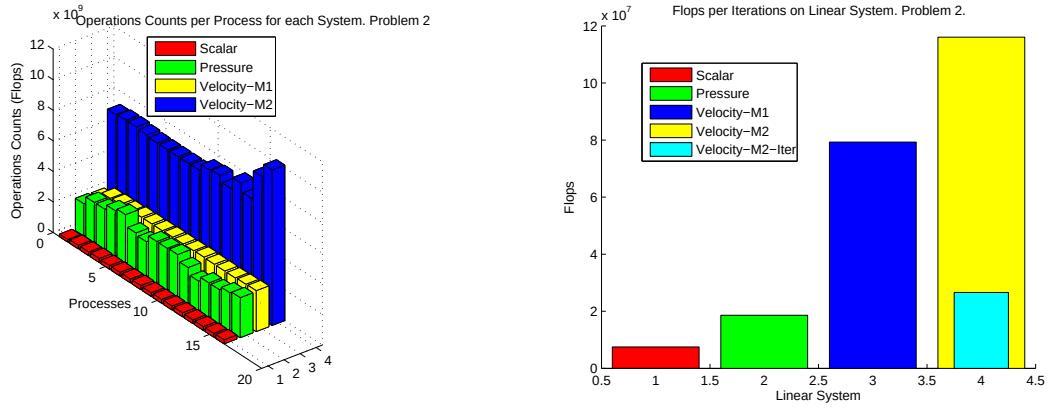
Note that the flops required for method 2 for one iteration is larger than the required

by method 1 (figures 6.33(b) and 6.34(b)), but the total flops to get a solutions in method 2 is less than the required by method 1 (figures 6.33(a) and 6.34(a)). This is because in method 2 the solver is performed in a reduced matrix and the flops required for one iteration (labeled as "Velocity-M2-Iter"), the rest of the flops are used to get the condensed matrix. That is, the intensive kernel operations, during the solver process, accumulate the flops for this reduced matrix, which in the end results in a total flops reduction .



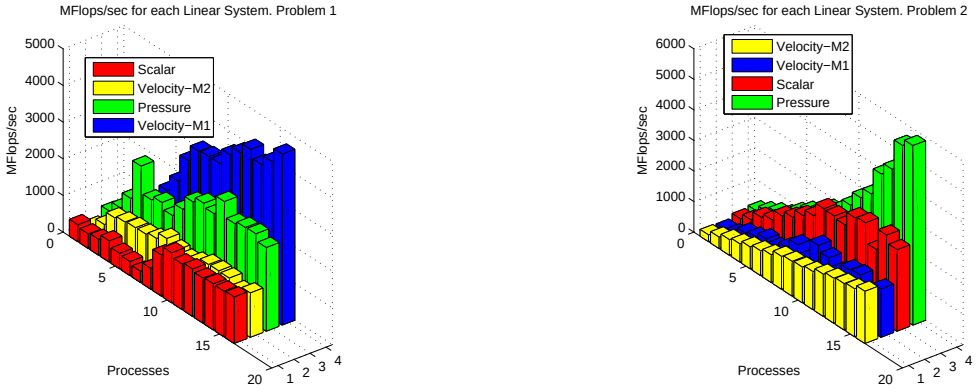
(a) (b)

Figure 6.33: Operations Counts Measured in Problem 1: (a) Total FLOPs. (b) Flops per Iterations.



(a) (b)

Figure 6.34: Operations Counts Measured in Problem 2: (a) Total FLOPs. (b) Flops per Iterations.



(a) (b)

Figure 6.35: MFlops/sec in Problem 1 and 2: (a) Problem 1. (b) Problem 2.

Finally figure 6.35 shows the MFlops/sec achieved for each linear system. These were calculate using the following expression,

$$MFlops/sec = 10^{-6} * (\sum_{i=0}^{p} f_i)/(t_{max})$$

Where f_i is the flops on process i, p the number of process and t_{max} is the maximum time measured over all processes.

6.6 Total Time

Table 6.2 gives a summary of total time achieved. That is, the elapsed time in the pre-processing stage and the solver for the three linear system.

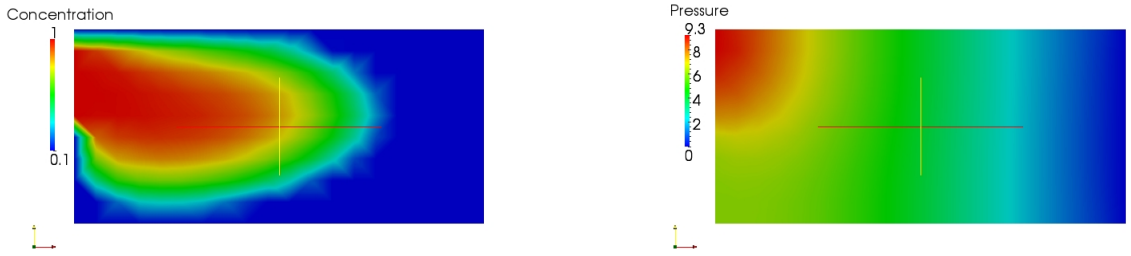
Table 6.2: Results Summary

Problem	Method	Processes				
		1	2	4	8	16
P1	M1	4.4138	3.4992	2.0417	1.3351	1.2591
	M2	3.1173	2.4837	1.4825	1.2028	1.1528
P2	M1	63.3070	49.7690	27.0765	16.3059	14.9857
	M2	15.3952	13.0751	10.3741	7.5806	5.9422

We note that for small problem the runs with method 1 and method 2, are almost closely. But for larger problem the the method 2 become superior.

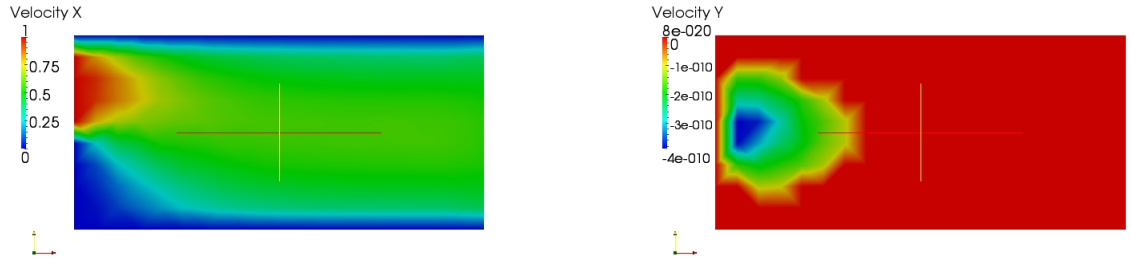
6.7 Simulations

Figures 6.36 and 6.37 shows an instants of the simulations. The parameters used for this runs were $Re = 100$ and $Sc = 20$ and boundary conditions described above.



(a) (b)

Figure 6.36: Snapshot of the Simulations: (a) Scalar Concentrations. (b) Pressure.



(a) (b)

Figure 6.37: Snapshot of the Simulations: (a) Velocity - Component X. (b) Velocity - Component Y.

CHAPTER 7

SUMMARY

In this work, the parallelization of finite element code for numerical simulations of water reservoirs is investigated and implemented using PETSc and MPICH libraries.

Chapters 1 - 3 give an introduction to the problem and a review of the main concepts necessary to develop this work. Implementation details about partitioning, elements and vertices distributions, and the mapping of degree of freedom were given in chapter 4. The mapping technique and the static matrix allocations were crucial to improve the assembling process. Implementing a complete finite element code with PETSc represent a great investment of effort in coding, but we can easily perform runs to tests different solver and preconditioner algorithm in order to determine the best solver for a given problem. In chapter 5 we see that memory bandwidth is a serious limitations of the scalability when performing sparse matrix-vector multiplications (SpMV). In chapter 6 we were able to determine the best preconditioner among the preconditioners tested, which were for intermediate velocity: one-way dissection reordering in method 1 and natural reordering in method 2. For pressure and scalar were the reverse Cuchil McKee and natural respectively. Respect to the method 1 and method 2 tested we can conclude that the method 2 (Static Condensation) was superior in time respect to the method 2, at a cost of increase the used memory. Further improvement can be done in this part, by avoiding the copy of part of the matrix when performing the factorization. In almost all part of the code we have enough room for tuning and get better speedup.

In general the computational time was reduced with the parallel implementations compared with the serial code, however more and longer simulations are needed in order to fine tune the implementations, in all stages.

7.1 Future Research Area

Some future research can be given in order to improve the implementations.

- In the preprocessing stage a geometric partitioner can be used to get a better speedup. Other parallel partitioning libraries can be tested and compared.
- In the assembling process the exact amount of non zeros of the matrix can be calculated, resulting in a perfect matrix allocation. Also setting up values by blocks in the matrix can result in a more efficient assembling.
- In the solver stage copy of the matrix can be avoided during the static condensation, resulting in a reduction of memory used. Also, more general factorization can be performed.
- Test other software implementations like OPEN-MX and GAMMA to improve the message passing performance over the Ethernet network.
- Implement the code using hybrid programming models.
- Extend the code to the tri-dimensional case.
- Implement the semi-lagrangian scheme in parallel.

REFERENCES

AMESTOY, P. R. et al. *A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling*. 1999.

ANJOS, G. R. d. et al. Numerical modeling of the hydrodynamic field coupled to the transport of chemical species through the finiteelement. In: *6th International Congress on Industrial and Applied Mathematics (ICIAM 2007)*. Zurique: 6th International Congress on Industrial and Applied Mathematics, 2007.

ANL/MSC. *MPI Implementations*. 2010. Http://www.mcs.anl.gov/research/projects/mpi/implementations.html.

ANL/MSC. *MPICH2: High-performance and Widely Portable MPI*. 2010. Http://www.mcs.anl.gov/research/projects/mpich2.

ASHCRAFT, C.; GRIMES, R. Spooles: An object-oriented sparse matrix library. In: *In Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*. [S.l.: s.n.], 1999. p. pages.

BALAY, S. et al. *PETSc Users Manual*. [S.l.], 2010.

BALAY, S. et al. *PETSc Web page*. 2009. Http://www.mcs.anl.gov/petsc.

BALAY, S. et al. Efficient management of parallelism in object oriented numerical software libraries. In: ARGE, E.; BRUASET, A. M.; LANGTANGEN, H. P. (Ed.). *Modern Software Tools in Scientific Computing*. [S.l.]: Birkhäuser Press, 1997. p. 163–202.

BATCHELOR, G. *An introduction to fluid dynamics*. UK: Cambridge University Press, 2000.

BECKER, E. B.; CAREY, G. F.; ODEN, J. T. *Finite Element Method*. [S.l.]: Prentice-Hall, 1981.

BENZI, M. et al. Numerical solution of saddle point problems. *Acta Numerica*, v. 14, p. 1–137, 2005.

BENZI, M.; HAWS, J.; TUMA, M. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM Journal on Scientific Computing*, Citeseer, v. 22, n. 4, p. 1333–1353, 2001.

BENZI, M.; SZYLD, D.; DUIN, A. V. Orderings for incomplete factorization preconditioning of nonsymmetric problems. *SIAM Journal on Scientific Computing*, Citeseer, v. 20, n. 5, p. 1652–1670, 1999.

BOISVERT, R. et al. The Matrix Market: A web resource for test matrix collections. *Quality of Numerical Software, Assessment and Enhancement*, Citeseer, p. 125–137, 1997.

BUNTINAS, D.; MERCIER, G.; GROPP, W. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*. [S.l.: s.n.], 2006. v. 1.

BURNS, A.; WELLINGS, A. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. USA: Addison-Wesley Educational Publishers Inc, 2009. ISBN 0321417453, 9780321417459.

BUTENHOF, D. R. *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-63392-2.

CHARNEY, J.; FJÖRTOFT, R.; NEUMMAN, J. von. On a numerical method of integrating the barotropic vorticity equation. *Tellus*, p. 179–194, 1952.

CHOW, E.; SAAD, Y. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, Citeseer, v. 86, n. 2, p. 387–414, 1997.

DAVIS, T. A.; DUFF, I. S. *An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization*. [S.l.], 1994.

DEMMEL, J. *Applied numerical linear algebra*. [S.l.]: Society for Industrial Mathematics, 1997.

DEMMEL, J. W. et al. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, v. 20, n. 3, p. 720–755, 1999.

DEMMEL, J. W.; GILBERT, J. R.; LI, X. S. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, v. 20, n. 4, p. 915–952, 1999.

DUFF, I.; ERISMAN, A.; REID, J. *Direct methods for sparse matrices*. [S.l.]: Oxford University Press, USA, 1989.

FOSTER, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*: Parallelism and computing. [S.l.]: Addison Wesley, 1995.

FREUND, R.; GOLUB, G.; NACHTIGAL, N. Iterative solution of linear systems. *Acta Numerica*, Cambridge Univ Press, v. 1, p. 57–100, 2008.

GAREY, M. R.; JOHNSON, D. S.; STOCKMEYER, L. Some simplified np-complete problems. In: *Annual ACM Symposium on Theory of Computing archive.Proceedings of the sixth annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1974.

GEORGE, A. An automatic one-way dissection algorithm for irregular finite element problems. *Numerical Analysis*, Springer, p. 76–89.

GEORGE, A.; LIU, J. Computer solution of large sparse positive definite systems. Prentice Hall, 1981.

GEORGE, J. *Computer implementation of the finite element method*. 1971.

GOCKENBACH, M. S. *Understanding and Implementing the Finite Element Method*. Houghton, Michigan: SIAM, 2006.

GRAMA, A. et al. *Introduction to Parallel Computing,*. 2nd. ed. [S.l.]: Pearson Education Limited, 2003.

GRAMA, A.; GUPTA, A.; KUMAR, G. K. andVipin. *Introduction to parallel computing*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

GROPP, W. *Exploiting Existing Software in Libraries: Successes, Failures, and Reasons Why*. 1998.

GROPP, W. et al. Toward realistic performance bounds for implicit CFD codes. In: CITESEER. *Proceedings of Parallel CFD*. [S.l.], 1999. v. 99.

GROPP, W. et al. Latency, bandwidth, and concurrent issue limitations in high-performance CFD. In: CITESEER. *First MIT Conference on Computational Fluid and Solid Mechanics, Cambridge, MA (US), 06/12/2001–06/14/2001*. [S.l.], 2000.

HENDRICKSON, B. *Chaco: Software for Partitioning Graphs*. January 2010. `http://www.sandia.gov/~bahendr/chaco.html`.

HEROUX, M. e. a. An overview of trilinos. *Special issue on the Advanced CompuTational Software (ACTS) Collection*, v. 31, n. 3, p. 397 – 423, 2005.

HESTENES, M.; STIEFEL, E. Methods of conjugate gradients for solving linear systems. *J*, 1952.

HUGHES, C.; HUGHES, T. *Professional Multicore Programming Design and Implementation for C++ Developers*. [S.l.]: Wiley Publishing, Inc., 2008.

HUGHES, T. *The finite element method*: Linear static and dynamic finite element analysis. New York: Dover Publications, 2000.

J., R. A. A stable numerical integration scheme for the primitive meteorogical equations. *ATMOSPHERE-OCEAN*, v. 19, n. 35-46, 1981.

J.S., S. A semi-lagrangian method of solving the vorticity advection eqution. *Tellus*, p. 336–342, 1963.

KARDESTUNCER, H.; NORRIE, D. *Finite element handbook*. [S.l.]: McGraw-Hill, Inc. New York, NY, USA, 1987.

KARLSSON, B. *Beyond the C++ standard library*. [S.l.]: Addison-Wesley Professional, 2005.

KARYPIS, G.; KUMAR, V. *Parallel Multilevel Graph Partitioning*. [S.l.], 1995.

KITWARE. *VTK File Formats for VTK Version 4.2*. 2010. Http://www.vtk.org/VTK/img/file-formats.pdf.

KSHEMKALYANI, A. D.; SINGHAL, M. *Distributed Computing Principles, Algorithms, and Systems*. [S.l.]: Cambridge University Press, 2008.

KUNDU, P. K.; COHEN, I. M. *Fluid Mechanics*. 2nd. ed. London and San Diego, CA: Academic Press, 2002.

LAB, K. *ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering*. January 2010. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

LANCZOS, C. Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Standards*, Citeseer, v. 49, n. 1, p. 33–53, 1952.

LAYTON, J. B. Parallel platters: File systems for hpc clusters. *Linux Magazine*, 2007.

LEE, M. J.; OH, B. D.; KIM, Y. B. Canonical fractional-step methods and consistent boundary conditions for the incompressible navier-stokes equations. *Journal of Computational Physics*, v. 168, p. 73–100, 2001.

LI, X. S.; DEMMEL, J. W. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, v. 29, n. 2, p. 110–140, June 2003.

LU, Y. *Optimum Flow Domain Partitioning of the Three-Dimensional Water Flow for Parallel Computation*. Dissertação (Master of Science Thesis) — KTH Computer Science and Comunication, Stockholm, Sweden, 2008.

MATHEMATICS, W. G. *Why we Couldn't Use Numerical Libraries for PETSc*.

MCCALPIN, J. D. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Charlottesville, Virginia, 1991–2007. A continually updated technical report. http://www.cs.virginia.edu/stream/. Disponível em: <http://www.cs.virginia.edu/stream/>.

MCCALPIN, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, p. 19–25, dez. 1995.

OPENMP. *The OpenMP API specification for parallel programming*. 2010. Http://openmp.org.

PELLEGRINI, F. *Scotch §PT-Scotch*. January 2010. `http://www.labri.fr/perso/pelegrin/scotch`.

PEROT, J. B. An analysis of the fractional step method. *Journal of Computational Physics*, v. 108, p. 51–58, 1993.

RABENSEIFNER, R.; KONIGES, A. Effective communication and file-i/o bandwidth benchmarks. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, p. 24–35.

REID, E. On the method of conjugate gradients for the solution of large sparse systems of linear equations. *Work*, v. 501, p. 39911.

SAAD, Y. *Iterative Methods for Sparse Linear System*. 2nd. ed. [S.l.]: SIAM, 1999.

SAAD, Y.; SCHULTZ, M. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, v. 7, n. 3, p. 856–869, 1986.

SAAD, Y.; SOSONKINA, M.; ZHANG, J. Domain decomposition and multi-level type techniques for general sparse linear systems. *Contemporary Mathematics*, Citeseer, v. 218, p. 174–190, 1998.

SAAD, Y.; VORST, H. V. D. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, Elsevier, v. 123, n. 1-2, p. 1–33, 2000.

SHIN, H. H. *A Methodology of Study of Three Dimensional Stratified Turbulent Fluid Flow for Hydroelectric Power Plant Reservoir Simulation*. Dissertação (Mestrado) — Faculdade de Engenharia Mecânica, 2009.

SIMONCINI, V.; SZYLD, D. Recent computational developments in Krylov subspace methods for linear systems. *Numerical Linear Algebra with Applications*, Citeseer, v. 14, n. 1, p. 1, 2007.

SOLCHENBACH, K. Benchmarking the balance of parallel computers. In: *SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems, Wuppertal, Germany*. [S.l.: s.n.], 1999.

STERLING, T.; SALMON, J.; SAVARESE., D. B. D. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*: Parallelism and computing. [S.l.]: MIT Press, 1999.

VORST, H. A. van der. Bi-cgstab: a fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 13, n. 2, p. 631–644, 1992. ISSN 0196-5204.

WALSHAW, C. *JOSTLE Ů graph partitioning software*. January 2010. `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle`.

WALSHAW, C.; CROSS, M. Parallel mesh partitioning on distributed memory systems. In: *Parallel and Distributed Processing for Computational Mechanics. Saxe-Coburg Publications*. [S.l.: s.n.], 1999.

WIIN-NIELSEN, A. on applications of trajectory method in numerical forecasting. *Tellus*, p. 180–196, 1959.

ZIENKIEWICZ, O.; TAYLOR, R. *The finite element method*: Volume 3: Fluid dynamics. Ma: Butterworth-Heinemann, 2000.

ZIENKIEWICZ, O.; TAYLOR, R. *The finite element method*: Volume 1: The basis. Ma: Butterworth-Heinemann, 2000.