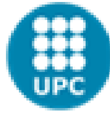
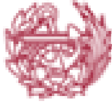

Coupling efficient parallel solvers for saddle-point problems with serial Matlab code

Treball realitzat per:
Arthur Lustman

Dirigit per:
Dr. S. Zlotnik
Dr. M. Giacomini

Màster en:
Erasmus Mundus M.Sc. in Computational Mechanics

Barcelona, **14 June 2019**

Departament de Laboratori de Cálcul Numèric

TREBALL FINAL DE MÀSTER

Contents

Abstract i

Acknowledgments ii

1 Introduction 1
1.1 Numerical solution of linear system of equation 1
1.1.1 Direct methods 1
1.1.2 Iterative methods 2
1.1.3 Solving linear systems in Matlab 3
1.1.4 Solving linear systems in hypre 3
1.2 Parallel Computing to treat large scale problems 3
1.3 Contributions of this Master thesis 5
1.4 Outline 6

2 The Uzawa Algorithm 7
2.1 Saddle-point problems 7
2.2 Krylov Subspace 9
2.3 Uzawa iteration 9
2.4 Derivation of the Uzawa algorithm 10
2.4.1 Initial Residual $\boldsymbol{r}^{(0)}$ 11
2.4.2 Conjugate direction $\boldsymbol{d}_1^{(i)}$ 11
2.4.3 Step size $\alpha^{(i)}$ 12
2.4.4 The update of the velocity $\boldsymbol{u}^{(i+1)}$ 12
2.4.5 The incremental residual $\boldsymbol{r}^{(i+1)}$ 12
2.5 A simple implementation of the Uzawa Algorithm 12
2.6 A parallel version of the Uzawa Algorithm 14
2.6.1 Hypre implementation 14
2.6.2 A scheme of the algorithm 16
2.7 Saddle-point problem solver with Matlab interface 17
2.7.1 GMRES 17
2.7.2 BoomerAMG 18
2.7.3 ParaSails 19

3 Results 20
3.1 Problem statement 20
3.2 Direct solver 20
3.3 Iterative solver 22
3.4 Hypre's code results 22
3.4.1 Strong scalability results 23
3.4.2 AMG results 25
3.4.3 Details of scalability 25

4 Conclusion 27

5 Further Improvements 28

References 29

Abstract

Linear systems with saddle-point structure arise from the discretization of several engineering problems, like incompressible flows as well as problems in elasticity and electromagnetics involving mixed finite element formulations. The peculiar structure of the resulting matrices makes classical iterative methods available in most common linear algebra libraries inefficient for large scale problems. This work aims to propose an efficient alternative to built-in Matlab iterative algorithms, providing to end-users a seamless access to a parallel linear solver tailored for saddle-point problems. More precisely, an interface between Matlab and an implementation of the Uzawa algorithm in hypre, the High Performance Preconditioners library developed at Lawrence Livermore National Laboratory (USA), is proposed.
The developed strategy allows Matlab users to run the Uzawa algorithm on up to 32 processors, exploiting efficient hypre solvers, e.g. preconditioned conjugate gradient (PCG), generalized minimal residual method (GMRES), algebraic multigrid (AMG), as well as hypre preconditioners like AMG and ParaSails. Validation of the methodology and scalability analysis are performed for moderate-sized systems, up to 1 million of unknowns, arisin from the Taylor-Hood discretization of a 3D Stokes problem with high variations of the viscosity coefficient.

Acknowledgments

I would like to thank every person who has contributed to the success of my thesis

First of all, I would like to express my deepest thanks to Dr. M. Giacomini and Dr. S. Zlotnik for their time dedicated to me, sharing their expertise and precious advices in regards to the completion of my thesis.

I would like to thanks my family who has always been encouraging me through those last two years to complete my master.

1 Introduction

The solution of linear system of equations is ubiquitous in engineering applications. Among them, linear system arising from finite element discretization have a sparse structure or other properties that can be taken advantage of. Large sparse symmetric and non-symmetric linear systems of equations appear in eigenvalue computation, solving discrete finite-element problems, device and circuit simulation, linear programming, chemical engineering, and fluid dynamics modeling[1]. Several techniques have been proposed for such problems.

This work focuses on saddle-point matrices which appears in most applications of scientific computing. These demanding and important applications can immediately benefit by any improvements in linear equation solvers. The goal is to provide an efficient parallel solver for large saddle-point problems that can be exploited with a seamless Matlab interface.

1.1 Numerical solution of linear system of equation

In the case of solving a classic linear system of equation,

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$ is a non-singular sparse matrix and $\boldsymbol{b} \in \mathbb{R}^n$ is a given vector, one can solve it with a direct or an iterative solver.

A simple and direct way to compute the solution $\boldsymbol{x}$ would be to invert the matrix A and multiply it by the forcing vector $\boldsymbol{b}$. The matrix A has the benefit of being sparse, meaning the non zero values takes only a small percentage of the information inside the matrix. Computing the inverse of a matrix is a computationally expensive operation that usually results in this case with storing a dense matrix A^{-1} which is not recommended. Solving the problem this way is not recommended as it would mean we are not using the sparsity of the matrix, a valuable property the could be taken advantage of. Another inconvenience is when the solver doesn't have access to the entire matrix at once, such case is in relation with the usage of parallel computing, which is a convenient way to speed up the computation of the solution.

1.1.1 Direct methods

A sparse linear system is frequently solved by sparse direct solvers, which usually rely on variations of Gauss elimination methods[2]. Permutation matrices are chosen to preserve sparsity and maintain stability in order to factorize the matrix A into two triangular matrices LU[3]. The two triangular systems $L\boldsymbol{y} = \boldsymbol{b}$ and $U\boldsymbol{x} = \boldsymbol{y}$ are solved to obtain the solution $\boldsymbol{x}$.

During the last 30 years, sparse direct solvers have been the subject of a lot of discussion and were developed to reach high efficiency. In the early times of computers, the sparsity of the matrices had to be exploited as a way to spare memory and time when solving large linear systems. A dense matrix of a million unknowns could never be stored in the computers available in 1985[4]. Nowadays the computers have grown bigger and more powerful, but the computation cost required by a direct solver for large scale problems still exceeds the available computing resources.

The various fields of science gave birth to different patterns and characteristics of sparse matrices. A vast amount of solvers have been developed and show strength of weaknesses depending on the properties of those matrices, some in parallel computing too. For example, MUMPS (Multifrontal Massively Parallel sparse direct Solver)[5] is a parallel sparse direct solver for which the matrix needs to be positive definite. Direct solvers are very efficient when dealing with small scale problem ($\sim 10^5$) but their usage can be criticized in terms of computational resources. For large problems, direct solvers require an amount of memory that is not available in existing computing facilities.

1.1.2 Iterative methods

On the other side, an efficient iterative solver reaches the solution using computationally cheap iterations. An iterative solver work by making a guess on the solution $\boldsymbol{x}_0$ and computing the residual of the solution $\boldsymbol{r} = A\boldsymbol{x}_0 - \boldsymbol{b}$. The iterative solution $\boldsymbol{x}_i$ is then updated depending on the method and the iteration stops when a tolerance over the residual is passed, reaching a stopping criteria.

A classical example of iterative algorithm is represented by Richardson's method[6]. It relies on a perturbation matrix E and a relaxation parameter $\alpha \in (0, 1)$. The equations of the method are written

$$B = A + E$$
$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \alpha B^{-1}\left(\boldsymbol{b} - A\boldsymbol{x}^k\right)$$

and could control the efficiency towards reaching a solution of the linear system of equation (1).

The convergence is defined by the method, and so, proof of convergence can be derived from the mathematical description of the solution. The convergence is described as the expression of the iterative error $\boldsymbol{e}^k = \boldsymbol{x} - \boldsymbol{x}^k$, where $\boldsymbol{u}$ is the exact solution and $\boldsymbol{x}^k$ is the solution computed at kth iteration of the solver. The convergence of Richardson method is assured as long as the condition number, defined by the operator ρ, of the matrix $\left(I - \alpha B^{-1}A\right)$ is smaller than 1

$$||\boldsymbol{e}^{k+1}|| \leq \rho\left(I - \alpha B^{-1}A\right)||\boldsymbol{e}^k||$$

For that reason, the matrix B is called a preconditioner and has the crucial role to decrease the condition number of the matrix A, improving the solvability of the system. As a result, in order to decrease the workload of the algorithm, we want a matrix B such as cheap (easy to parallelize and/or to inverse) and close to the matrix A. This way, the action of the preconditioner reduce the condition number $\rho\left(I - B^{-1}A\right)$, leading to a convergence acquired in less iterations.

The role of α is to control the size of the steps jumping to the next iterative solution $\boldsymbol{x}^{(i+1)}$. The rate of convergence is controlled by this parameter, which has to be the large enough to build a fast converging algorithm, but not too much to induce big changes, producing large errors within each iterations.

The solution of a system of equation is reached differently depending on the iterative method used. Their application and efficiency depends on their properties, this subject is part of the thesis and presented in section 2.

The iterative scheme to perform the solution of a saddle-point problem has been introduced by Arrow, Hurwicz and Uzawa [7]. In our case, the method used is the Uzawa algorithm coupled with the preconditioned conjugate gradient method (PCG), one of the most famous Krylov subspace methods. The Krylov methods are interesting for their computationally cheap operations as the linear subspace of variable is spanned by the multiplication of the exponent of a sparse matrix A by a vector $\boldsymbol{x}$. Meaning the iterations of such methods are performed with the same expense,

$$\mathcal{K}_k(A, \boldsymbol{x}) := \mathrm{span}\{\boldsymbol{x}, A\boldsymbol{x}, A(A\boldsymbol{x}), A\left(A(A\boldsymbol{x})\right), ...\}$$

as a single matrix-vector product. The Uzawa algorithm is later described in the subsection 2.3

The PCG method has the advantage to adapt the previously described *relaxation parameter* α within each iteration, hence the *steepest descent*: locally computed convergence to minimize the A norm of the error $\boldsymbol{u} - \boldsymbol{u}^{(k)}$ in the evaluated steepest direction. The iterative solver is often described with the usage of a preconditioner to increase the rate of convergence and the number of iterations. More about this subject is covered in section 2.

1.1.3 Solving linear systems in Matlab

When the dimension of the problem is large, the computational and memory constraints make is necessary to adapt the algorithm to a parallel environment. Matlab is a great software for prototyping numerical method and provides very efficient solvers for general systems. Unfortunately, it does not give you much control on the solver used, in particular for parallel solvers, and the performance for large problem is not optimal under default configurations.

1.1.4 Solving linear systems in hypre

In this paper, the task is to build an efficient solver for saddle-point problems by distributing subsets of the linear system of equation among different processing units. Since hypre is a library with iterative solvers implemented in parallel, the usage of such library makes it desirable.

Hypre is an open source library of linear solvers developed by the Lawrence Livermore National Laboratory since 1990s. The library offer a large panel of solvers and preconditioners to perform the solution of large sparse linear systems on massively parallel computers[8]. The open-source codes are written with different programming language using a parallel MPI and OpenMP environment.

The Uzawa algorithm is implemented on Hypre and run on a cluster. This implementation opens the path for solving other types of problem in a controlled parallel environment. The solvers implemented in the Hypre library and their efficiency are described in the result section 3.

1.2 Parallel Computing to treat large scale problems

The size of the sparse linear system of equation solvable has greatly increased since the first one solved using relaxation methods such as successive over-relaxation (SOR) or related techniques[4]. From a rough number of 200 in 1970 to more than a billion on 2010, this limit is pushed upon every day[4]. This is related to the performance of the machines, more powerful and more memory allocated and as a result of the time spent into development of algorithms and their efficient implementation.

Parallel computing is a type of computation in which many calculations or execution processes are carried out simultaneously by involving multiple processors. Larger problems can be divided into smaller ones which can then be solved at the same time. Parallelism has long been employed in high-performance computing, but it is gaining broader interest due to the physical constraints preventing frequency scaling. The frequency scaling is the technique to increase the performance of computers by increasing the frequency of the processors, thus the power consumption. It has been the dominant way to decrease run time, see Moore's law, from the mid-1980s until recently[9][10]. Moore's law, originated around 1970, predicted that the run time but more specifically, the integrated number of transistors, related to the processing power for computers will double about every 18-24 months. Although the law is still in effect after the end of frequency scaling, the focus of companies has been shifted in 2005 from single to multiple processing units, often called "cores", into a single chip[9]. As a result, the techniques for problem solving using modern computers have gradually translated from sequential to parallel instructions.

The strength of a parallel code lies in the fact that the algorithm can use more computational resources to further reduce the run time. The problem is split into independent parts that are individually solved each by a single processing unit. As a necessity to be suitable for a parallel execution, the algorithm must provide enough independent computations.

Due to the advancement in the scientific computing field, simulations are being more precise and the problems are growing larger every single day: there is a constant demand for computing resources. The consequence of such lead to the research of parallel high-performance machines such as a computer cluster.

A computer cluster is a system built up from server nodes, it consists of a large number of processors in proximity that interacts with each other. Clusters are now often used for parallel simulations due to their performances. Their usage, although possible for sequential codes, is adequate for algorithm that can partition the workload into several independent parts in order to speed up the run time. To obtain a parallel program, the algorithm must be written in a suitable programming language for which the parallel execution is controlled by specific run time libraries[9]. They can handle different programming languages and are necessary in order to make use of the multiple processors that are included in the machine. Their implementation and efficiency is based on the assumptions they make about the underlying memory architecture: shared memory, distributed memory or shared distributed memory.

Usually, when introduced to parallel computing, one of the first thought protocol is OpenMP, a multi-threaded parallel programming language. It is designed for multi-platform shared memory multiprocessing programming in C, C++ and Fortran. It is a very flexible interface for developing parallel applications that are based on the concept of multithreading, a method of parallelizing whereby a master thread forks a specified number of worker threads where the instructions are run sequentially to divide the total task.

When implementing a sequential code to parallel, the desire is to obtain a scalable algorithm, meaning the run time linearly decrease with the number of processors involved in the computation. The scalability of a complete algorithm cannot be satisfied by solely running portions of the code in a parallel scheme, i.e. like would an OpenMP implementation.

Additionally, the data availability performed by the OpenMP protocol can be a major drawback. Most variables are considered visible to all threads by default, meaning they are shared and stored on the memory. These information have to be grabbed by the speed-limited transmission of wires connected to the memory and could severely impact the performance of the algorithm. It is often a called a bottleneck in software engineering, the capacity of the computer to perform fast algorithms is limited by such component. In this case, the scalability of the parallel algorithm would be limited by the memory architecture.

The OpenMP implementation of an algorithm like Uzawa would consist on using multiple threads to perform computationaly expensive instructions while the rest of the code could be kept in a sequential environment. It represents a naive way to build a parallel code as the scalability would not be good for previously described reasons. In order to reach an appropriate scalability factor, the whole algorithm needs to be running in a complete parallel fashion and split the data directly to the processing unit.

The algorithm should be performed in a distributed memory environment where each processors owns their private memory. The data is split at the start and stored by each processes. The task is individually performed by each core that operates on their local data and a message-passing protocol is used when remote data is required. The most widely used message passing system is Message Passing Interface (MPI).

Hypre library has both OpenMP and MPI implemented in their functions. The code written solely uses the MPI environment, for which the implementation of the Uzawa algorithm is described at the end of section 2.

1.3 Contributions of this Master thesis

The aim of the master thesis is to provide an interface between Matlab and a library of parallel solvers with a special emphasis on saddle-point problem. The contributions can be split into multiple points:

- To implement an interface between Matlab and Hypre. The Matalb code would be running sequentially and send instructions to Hypre to solve a saddle-point problem in parallel for a variable number of processors
- To implement in Hypre a solver for saddle-point problems based on the Uzawa algorithm
- The usage of the solvers in Hypre should be transparent and configurable from the Matlab interface. In this case, the interface should allow the extension of other kind of specific solvers

1.4 Outline

The thesis starts by explaining the saddle-point problem to be solved. An example of Stokes problem leading to a saddle-point matrix is briefly introduced and the characteristics of the resulting linear system are described. The properties of Krylov subspace method is introduced, with the PCG method, in order to demonstrate the solving pattern of the Uzawa algorithm.

The rest of section 2 is focused towards the implementation of the Uzawa algorithm in the Hypre library and the Matlab interface. As a first step towards reaching this goal, a simple implementation of the Uzawa algorithm was developed in Matlab for testing and debugging purposes. This version has the advantages of being straightforward and sequential but it suffer from poor efficiency when dealing with large matrices. The parallel implementation of the algorithm is later discussed and a scheme of the Uzawa implementation in hypre is given. Finally, an explanation of the developed Matlab interface to solve saddle-point problems is given. To conclude this section, a scheme of the complete algorithm is presented.

Section 3 describes the numerical experiments performed. A set of 7 matrices with different dimensions are utilized. Small-scale problems that are affordable to solve in Matlab are used to test the solutions of the proposed routine. After that, the Hypre computation of the solution is completed with multiple processors to evaluate the scalability of the algorithm. Since the Hypre solve of the Uzawa algorithm has been implemented with multiple iterative solvers, this section is concluded by looking at their individual results.

The last section reports the results computed on the set of problems. For the performed simulations, the implemented methods with the best numerical performances can be deducted. A few paragraph dedicated to the strength and weaknesses of the code is given, with the further improvement that could be performed.

2 The Uzawa Algorithm

This section describes the theoretical knowledge towards solving saddle-point problems using an Uzawa algorithm coupled with the PCG method. The properties of saddle-point matrices are first explained and a small example is given through the Stokes problem. The Krylov subspace method is described as a structure to understand the PCG method involved in Uzawa. This section concludes by describing the implementation of the code through the library Hypre.

2.1 Saddle-point problems

The subject of this thesis is the focus on the construction of an efficient solver of a saddle-point problem appearing in the block of a 2×2 system of equation

$$\begin{bmatrix} A & B \\ B^\top & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} \tag{2}$$

It is important to note that this format is not the most general form of a saddle-point problem[1][11] but they are not the subject of discussion in this thesis.

Finding the solution $(\boldsymbol{u}_*, \boldsymbol{p}_*)$ of the saddle-point problem (2) becomes a first-order optimality conditions for the following equality-constrained quadratic programming problem[1]

$$\min J(\boldsymbol{u}) = \frac{1}{2}\langle A\boldsymbol{u}, \boldsymbol{u}\rangle - \langle \boldsymbol{f}, \boldsymbol{u}\rangle \tag{3a}$$

$$\text{subject to } B^\top \boldsymbol{u} = \boldsymbol{g} \tag{3b}$$

Where the operation $\langle \boldsymbol{a}, \boldsymbol{b}\rangle$ defines the inner product of two vectors

$$\langle \boldsymbol{a}, \boldsymbol{b}\rangle = \boldsymbol{a}^\top \cdot \boldsymbol{b} = a_i b_i \tag{4}$$

As for optimization problems with constraints, the Lagrangian functional can be introduced

$$\mathcal{L}(\boldsymbol{u}, \boldsymbol{p}) = J(\boldsymbol{u}) + \left(B^\top \boldsymbol{u} - \boldsymbol{g}\right)^\top \cdot \boldsymbol{p} \tag{5}$$

A saddle point problem is a point $(\boldsymbol{u}_*, \boldsymbol{p}_*) \in \mathbb{R}^{m+n}$ that satisfies

$$\mathcal{L}(\boldsymbol{u}_*, \boldsymbol{p}) \leq \mathcal{L}(\boldsymbol{u}_*, \boldsymbol{p}_*) \leq \mathcal{L}(\boldsymbol{u}, \boldsymbol{p}_*) \qquad \text{for any } \boldsymbol{u} \in \mathbb{R}^m \quad \text{and } \boldsymbol{p} \in \mathbb{R}^n \tag{6}$$

Or equivalently,

$$\min_{\boldsymbol{u}} \max_{\boldsymbol{p}} \mathcal{L}(\boldsymbol{u}, \boldsymbol{p}) = \mathcal{L}(\boldsymbol{u}_*, \boldsymbol{p}_*) = \max_{\boldsymbol{p}} \min_{\boldsymbol{u}} \mathcal{L}(\boldsymbol{u}, \boldsymbol{p}) \tag{7}$$

The reasoning behind those equation is that in a saddle-point problem, the two unknowns $\boldsymbol{u}$ and $\boldsymbol{p}$ need to be solved together and never be understood as two unrelated equations. This matter is discussed in the following subsections where we introduce the Uzawa algorithm.

This particular type of problem can be retrieved in the case of many different applications in engineering. For example, in the case of a Stokes problem with the saddle-point problem being a symmetric linear system. The incompressible Stokes equation can be written in the weak formulation as: find $(\boldsymbol{u}, p) \in V \times Q$ such that

$$a(\boldsymbol{w}, \boldsymbol{u}) + b(\boldsymbol{w}, p) = (\boldsymbol{w}, \boldsymbol{f}) \qquad \forall \boldsymbol{w} \in V, \tag{8a}$$

$$b(q, \boldsymbol{v}) = 0 \qquad \forall q \in Q \tag{8b}$$

Where V and Q are Hilbert spaces, a and b are bounded bi-linear forms on $V \times V$ and $V \times Q$ respectively and $(\boldsymbol{w}, \boldsymbol{f})$ is a bounded linear functional on V. After discretizing with stable finite element pair, a system of equation (8) can appear in the form of the block 2×2 linear systems (2)[12][13][14].

For the rest of the thesis, although not particularly specified for, the block structure equation to be solved is written as the solution of a Stokes problem. It serves as a starting point to visualize the variable but, the solver proposed is not restricted to such problem.

$$\begin{bmatrix} K & G \\ G^\top & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} \tag{9}$$

Where $K \in \mathbb{R}^{n\times n}$ is a symmetric positive definite matrix and matrix $G \in \mathbb{R}^{m\times n}$. The solution $(\boldsymbol{u}_*, \boldsymbol{p}_*)$ of the problem is a saddle-point of the corresponding energy functional where the velocity is a minimum and the pressure is a maximum for the energy functional[1][15].

A problem arising from the Navier-Stokes equations has not been considered in this thesis but is still introduced out of completeness. The case of an incompressible Navier-Stokes problem is more complex since non-linearity is induced. The convection matrix $C(\boldsymbol{u})$ is added to the K matrix. Such problem would require a nonlinear solver and a method to treat with the non-symmetric K and is out of the scope of this thesis.

The system of equation (9) is equivalent to the system of equation,

$$K\boldsymbol{u} + G\boldsymbol{p} = \boldsymbol{f} \quad , \tag{10a}$$

$$G^\top \boldsymbol{u} = \boldsymbol{g} \quad . \tag{10b}$$

From these equations, the solution can be interpreted as,

$$\boldsymbol{u} = K^{-1}(\boldsymbol{f} - G\boldsymbol{p}) \quad , \tag{11a}$$

$$G^\top K^{-1} G\boldsymbol{p} = G^\top K^{-1}\boldsymbol{f} - \boldsymbol{g} \quad . \tag{11b}$$

The last equation can be represented by the linear system of equation

$$S\boldsymbol{p} = \boldsymbol{t} \quad . \tag{12}$$

This equation originates from the usage of the *Schur complement* S that arise from a block Gaussian elimination. The unknown of this equation is the pressure, and the $\boldsymbol{t}$ vector is equal to the right hand side of the equation (11b).

The unique solvability of the saddle-point problem is assured by the non-singularity of the matrix of the *Schur complement* $S = G^\top K^{-1} G$. Due to the structure and properties of equation (9), the matrix S is non-singular if and only if G has full column rank$(G) = n$, n being the column dimension of the matrix[1][12].

The non-singular linear system of equation has a unique solution $(\boldsymbol{u}_*, \boldsymbol{p}_*)$ reached by using the Uzawa method. This algorithm originated from Hirofumi Uzawain in the context of concave programming and solves both equations at the same time. The algorithm performs an iterative process to reach a solution $\boldsymbol{p}^{(i)}$ through the application of the conjugate gradient method on equation (11b). The solution of velocity field $\boldsymbol{u}^{(i)}$ is then updated during the iterative process.

Since the Schur complement is a symmetric positive definite matrix, the requirements for the usage of the conjugate gradient method are met[13] and the Uzawa method can be performed. The variables are computed at each iteration i with increments and the solution for the velocity $\boldsymbol{u}^{(i)}$ and the pressure $\boldsymbol{p}^{(i)}$ are computed with the conjugate directions $\boldsymbol{d}_1^{(i)}$ and $\boldsymbol{d}_2^{(i)}$ respectively. Both $\boldsymbol{d}$ vectors form an orthogonal base serving as a direction vector towards the solution of the saddle-point problem $(\boldsymbol{u}_*, \boldsymbol{p}_*)$.

2.2 Krylov Subspace

For simplicity, the basics of Krylov subspace method is described for the unpreconditioned and non-singular systems. If an initial solution $\boldsymbol{u}^{(0)}$ is defined, the initial residual $\boldsymbol{r}^{(0)}$ is defined with the second equation $\boldsymbol{r}^{(0)} = G^\top \boldsymbol{u}^{(0)} - \boldsymbol{g}$. Krylov subspace methods are iterative methods whose ith iterative solution $\boldsymbol{u}^{(i)}$ is described as,

$$\boldsymbol{u}^{(i)} \in \boldsymbol{u}^{(0)} + \mathcal{K}_i(G^\top, \boldsymbol{r}^{(0)}) \qquad \text{for } i = 1, 2, ...$$

Where $\mathcal{K}_i$ denotes the ith Krylov subspace generated by $G^\top$ and $\boldsymbol{r}_0$ where clearly $\mathcal{K}_0 \subset \mathcal{K}_1 \subset \mathcal{K}_2 \subset \dots$. Every member is a matrix-product vector of the first i power of a matrix $A \in \mathbb{R}^{n\times n}$ and a vector $\boldsymbol{x} \in \mathbb{R}^n$.

$$\mathcal{K}_k(A, \boldsymbol{x}) \equiv \text{span}\{\boldsymbol{x}, A\boldsymbol{x}, A^2\boldsymbol{x}, ..., A^{k-1}\boldsymbol{x}\}$$

The utility for this class of methods lies in the observation that the linear subspace spanned with a sparse matrix A and any vector $\boldsymbol{x}$ is relatively cheap to compute : if $A \in \mathbb{R}^{n\times n}$ has at most l non-zeros entries in any rows then $A\boldsymbol{x}$ can be computed in only nl floating point operator. The result of the matrix-vector product is a vector $A\boldsymbol{x}$ and the further element of the Krylov subspace is computed with the matrix-vector product $A(A\boldsymbol{x})$ which makes the generation of every family-member within the same range of workload.

2.3 Uzawa iteration

The initial residual $\boldsymbol{r}^{(0)}$ and conjugate direction $\boldsymbol{d}^{(0)}$ are computed from the equation (10b) and updated in the loop. The iterative process starts to compute the solution of the pressure $\boldsymbol{p}^{(i)}$ from the conjugate direction $\boldsymbol{d}_2^{(i)}$ and the step size $\alpha^{(i)}$: the solution is reached by increments $\boldsymbol{p}^{(i+1)} = \boldsymbol{p}^{(i)} + \alpha^{(i)}\boldsymbol{d}_2^{(i)}$. The iterative velocity $\boldsymbol{u}^{(i)}$ is updated during this algorithm with the conjugate direction $\boldsymbol{d}_1^{(i)}$ which expression is described in equation (16).

The following instructions describe the equation of the Uzawa algorithm to perform the solution of a saddle-point problem (9).

A Consider an initial guess $\boldsymbol{p}^{(0)}$ for the pressure, generally equal to an array of zeros, in order to get the corresponding $\boldsymbol{u}^{(0)}$ from equation (10a)

$$K\boldsymbol{u}^{(0)} = \boldsymbol{f} - G\boldsymbol{p}^{(0)} \tag{13}$$

B Obtain the initial residual $\boldsymbol{r}^{(0)}$ associated with the equation (10b) and initialize the conjugate direction $\boldsymbol{d}_2^{(0)}$

$$\boldsymbol{r}^{(0)} = G^\top \boldsymbol{u}^{(0)} - \boldsymbol{g} \tag{14}$$

$$\boldsymbol{d}_2^{(0)} = \boldsymbol{r}^{(0)} \tag{15}$$

C Start the iterative process with $i = 0$

(a) Solve the following equation to obtain conjugate direction $\boldsymbol{d}_1^{(i)}$

$$K\boldsymbol{d}_1^{(i)} = G\boldsymbol{d}_2^{(i)} \tag{16}$$

(b) Evaluate the step size $\alpha^{(i)}$

$$\alpha^{(i)} = \frac{\langle \boldsymbol{r}^{(i)}, \boldsymbol{r}^{(i)} \rangle}{\langle G\boldsymbol{d}_2^{(i)}, \boldsymbol{d}_1^{(i)} \rangle} \tag{17}$$

(c) Update the variables of the velocity, pressure and residual $\boldsymbol{u}^{(i)}$, $\boldsymbol{p}^{(i)}$ and $\boldsymbol{r}^{(i)}$

$$\boldsymbol{p}^{(i+1)} = \boldsymbol{p}^{(i)} + \alpha^{(i)}\boldsymbol{d}_2^{(i)} \tag{18}$$

$$\boldsymbol{u}^{(i+1)} = \boldsymbol{u}^{(i)} - \alpha^{(i)}\boldsymbol{d}_1^{(i)} \tag{19}$$

$$\boldsymbol{r}^{(i+1)} = \boldsymbol{r}^{(i)} - \alpha^{(i)}G^{\top}\boldsymbol{d}_1^{(i)} \tag{20}$$

(d) Test of convergence with the norm of the new residual $||\boldsymbol{r}^{(i+1)}||$ and the conjugate direction $||\boldsymbol{d}_1^{(i)}||$

(e) Compute the optimization direction $\boldsymbol{d}_2^{(i)}$ with the parameter $\beta^{(i)}$

$$\beta^{(i)} = \frac{\langle \boldsymbol{r}^{(i+1)}, \boldsymbol{r}^{(i+1)} \rangle}{\langle \boldsymbol{r}^{(i)}, \boldsymbol{r}^{(i)} \rangle} \tag{21}$$

$$\boldsymbol{d}_2^{(i+1)} = \boldsymbol{r}^{(i+1)} - \beta^{(i+1)}\boldsymbol{d}_2^{(i)} \tag{22}$$

The algorithm is composed of 2 instructions (A and B) outside the loop and 5 instructions (C.(a)-(e)) inside. The iterative process produce a vector $(\boldsymbol{u}^{(i+1)}, \boldsymbol{p}^{(i+1)})$ that converge towards the solution of the saddle-point problem.

The stopping criteria are subject of a tolerance ϵ imposed by the user that define the precision of the solution reached. The solution is obtained only when the two test of convergence are successfully passed:

- The relative residual of the solution, for which the initial residual is given from equation (14)

$$\frac{||\boldsymbol{r}^{(i)}||}{||\boldsymbol{r}^{(0)}||} \leqslant \epsilon \quad . \tag{23}$$

- The iterative solution for the conjugate direction $||\boldsymbol{d}_1^{(i)}||$. Since the solution $\boldsymbol{u}^{(i+1)}$ is updated by the factor $\alpha^{(i)}\boldsymbol{d}_1^{(i)}$, it serves as an indicator to how close are we getting to the solution. From equation (19),

$$\boldsymbol{u}^{(i+1)} - \boldsymbol{u}^{(i)} = -\alpha^{(i)}\boldsymbol{d}_1^{(i)}$$

The second test performed is done over the relative increments $\alpha^{(i)}\boldsymbol{d}_1^{(i)}$,

$$\frac{||\alpha^{(i)}\boldsymbol{d}_1^{(i)}||}{||\boldsymbol{u}^{(i+1)}||} \leqslant \epsilon \tag{24}$$

The Uzawa iterations are based on the conjugate gradient algorithm for the transformed problem (11). The algorithm is constructed in a manner which avoids the need to explicitly calculate K^{-1} and to construct the Schur complement S. Instead, the instructions A and C.(a) require a linear solver. The choice of solver, between direct or any iterative method, usually depends on the size of the problem, the properties of the matrix and the flexibility of the software.

2.4 Derivation of the Uzawa algorithm

The Uzawa algorithm equations is the result of the conjugate gradient method applied to the second equation of the transformed problem (11). The algorithm of the preconjugate gradient method for equation (12) is written

Algorithm 1: The Preconditioned Conjugate Gradient (PCG) method for $S\boldsymbol{p} = \boldsymbol{t}$[13]

1. Choose initial solution $\boldsymbol{p}^{(0)}$
2. Compute initial residual $\boldsymbol{r}^{(0)} = \boldsymbol{t} - S\boldsymbol{p}^{(0)}$
3. Solve $M\boldsymbol{z}^{(0)} = \boldsymbol{r}^{(0)}$
4. Set $\boldsymbol{d}^{(0)} = \boldsymbol{z}^{(0)}$
5. **while** $i = (0, i_{max})$ *until convergence* **do**
6. $\quad\alpha^{(i)} = \langle \boldsymbol{z}^{(i)}, \boldsymbol{r}^{(i)} \rangle / \langle S\boldsymbol{d}^{(i)}, \boldsymbol{d}^{(i)} \rangle$
7. $\quad\boldsymbol{p}^{(i+1)} = \boldsymbol{p}^{(i)} + \alpha^{(i)} \boldsymbol{d}^{(i)}$
8. $\quad\boldsymbol{r}^{(i+1)} = \boldsymbol{r}^{(i)} - \alpha^{(i)} S\boldsymbol{d}^{(i)}$
9. $\quad\langle$ Test for convergence $\rangle$
10. $\quad\beta^{(i)} = \langle \boldsymbol{z}^{(i+1)}, \boldsymbol{r}^{(i+1)} \rangle / \langle \boldsymbol{z}^{(i)}, \boldsymbol{r}^{(i)} \rangle$
11. $\quad\boldsymbol{d}^{(i+1)} = \boldsymbol{z}^{(i+1)} - \beta^{(i+1)} \boldsymbol{d}^{(i+1)}$
12. **end**

The role of the preconditioner M can be rendered ineffective if set to the identity matrix. If so, the line 3 is ignored and the variable $\boldsymbol{z}^{(i)}$ is equal to the residual $\boldsymbol{r}^{(i)}$.

The PCG method has been developed by Hestenes & Stiefel [16] and is interesting for the cheap computational work: the workload of a single iteration is two inner products, three vector updates and one matrix-vector product. The test performed for the convergence is done by checking if the relative residual $\boldsymbol{r}^{(i)}$ is smaller than a tolerance ϵ imposed by the user. In this case, the initial residual $\boldsymbol{r}^{(0)}$ is equal to the right-hand side of the problem $S\boldsymbol{p} = \boldsymbol{t}$ due to the usual initial choice of the solution $\boldsymbol{p}^{(0)}$ to a 0 vector.

Most of the equations in the Uzawa algorithm are directly derived from the PCG method. But, the PCG method solves the equation containing the schur complement S which we do not want to explicitely construct. Since we have an equivalent value coming from equation (11b), the lines 2, 6 and 8 in the algorithm 1 are using S and need to be modified. Additionally, an equation has to be implemented to compute the iterative velocity $\boldsymbol{u}^{(i)}$.

2.4.1 Initial Residual $\boldsymbol{r}^{(0)}$

From line 2, the initial residual $\boldsymbol{r}^{(0)}$ becomes,

$$\begin{aligned}
\boldsymbol{r}^{(0)} &= \boldsymbol{t} - S\boldsymbol{p}^{(0)} \\
&= \left(G^\top K^{-1}\boldsymbol{f} - \boldsymbol{g}\right) - G^\top K^{-1} G\boldsymbol{p}^{(0)} \\
&= G^\top \underbrace{K^{-1}\boldsymbol{f}}_{\boldsymbol{u}^{(0)}} - \boldsymbol{g} - \cancelto{0}{G^\top K^{-1} G\boldsymbol{p}^{(0)}} \\
&= G^\top \boldsymbol{u}^{(0)} - \boldsymbol{g}
\end{aligned}$$

Under the assumption that the initial solution $\boldsymbol{p}^{(0)}$ is equal to 0.

2.4.2 Conjugate direction $\boldsymbol{d}_1^{(i)}$

The PCG algorithm is performed on the equation (12) for which the unknown is the pressure and the direction $\boldsymbol{d}_2^{(i)}$ is obtained. A conjugate direction is computed for the velocity, hence $\boldsymbol{d}_1^{(i)}$ coming from the following equation

$$K\boldsymbol{d}_1^{(i)} = G\boldsymbol{d}_2^{(i)}$$

This is the foundation for the following equations in order to make sure of the non-usage of the inverse of the sparse matrix K. This step of the algorithm would be solved with an appropriate direct or iterative solver.

2.4.3 Step size $\alpha^{(i)}$

Consider $\boldsymbol{d}^{(i)} = \boldsymbol{d}_1^{(i)}$, from line 6 of the CG algorithm we transform only the denominator of $\alpha^{(i)}$

$$\begin{aligned}
\langle S\boldsymbol{d}^{(i)}, \boldsymbol{d}^{(i)} \rangle &= \langle S\boldsymbol{d}_2^{(i)}, \boldsymbol{d}_2^{(i)} \rangle \\
&= \langle G^\top K^{-1} G\boldsymbol{d}_2^{(i)}, \boldsymbol{d}_2^{(i)} \rangle \\
&= \langle G^\top \underbrace{K^{-1} G\boldsymbol{d}_2^{(i)}}_{\boldsymbol{d}_1^{(i)}}, \boldsymbol{d}_2^{(i)} \rangle \\
&= \langle G^\top \boldsymbol{d}_1^{(i)}, \boldsymbol{d}_2^{(i)} \rangle \\
&= \langle G\boldsymbol{d}_2^{(i)}, \boldsymbol{d}_1^{(i)} \rangle
\end{aligned}$$

The last step is done by using the properties of the inner product operator.

2.4.4 The update of the velocity $\boldsymbol{u}^{(i+1)}$

The velocity field $\boldsymbol{u}^{(i+1)}$ is updated by using the equation (11a)

$$\begin{aligned}
\boldsymbol{u}^{(i+1)} &= K^{-1} \left(\boldsymbol{f} - G\,\boldsymbol{p}^{(i+1)} \right) \\
&= K^{-1} \left(\boldsymbol{f} - G(\boldsymbol{p}^{(i)} + \alpha^{(i)} \boldsymbol{d}_2^{(i)}) \right) \\
&= \underbrace{K^{-1} \left(\boldsymbol{f} - G\,\boldsymbol{p}^{(i)} \right)}_{\boldsymbol{u}^{(i)}} - \alpha^{(i)} \underbrace{K^{-1} G\boldsymbol{d}_2^{(i)}}_{\boldsymbol{d}_1^{(i)}} \\
&= \boldsymbol{u}^{(i)} - \alpha^{(i)} \boldsymbol{d}_1^{(i)}
\end{aligned}$$

2.4.5 The incremental residual $\boldsymbol{r}^{(i+1)}$

Finally, the $(i+1)$th residual $\boldsymbol{r}^{(i+1)}$ is slightly different, from line 8 of the CG algorithm

$$\begin{aligned}
\boldsymbol{r}^{(i+1)} &= \boldsymbol{r}^{(i)} - \alpha^{(i)} S\boldsymbol{d}_2^{(i)} \\
&= \boldsymbol{r}^{(i)} - \alpha^{(i)} G^\top \underbrace{K^{-1} G\boldsymbol{d}_2^{(i)}}_{\boldsymbol{d}_1^{(i)}} \\
&= \boldsymbol{r}^{(i)} - \alpha^{(i)} G^\top \boldsymbol{d}_1^{(i)}
\end{aligned}$$

2.5 A simple implementation of the Uzawa Algorithm

A simple Matlab implementation of the Uzawa algorithm was developed for testing and debugging purposes. The algorithm can be tested for direct or iterative solvers for solving equations (13) and (16).

The following algorithm written is an exact transcription of the section 2.3 in order to compute the solution of a saddle-point problem of variable size. The results obtained with different saddle-point matrices are discussed in the section 3.

The data matrices loaded are of format equation (9). The additionnal terms A and $\boldsymbol{b}$ are the matrix and vector equal to the left and right-hand-side of the block system of equation. The unknown $\boldsymbol{x}$ is the vector of both velocity and pressure unknowns.

$$A = \begin{bmatrix} K & G \\ G^\top & 0 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix}$$

The saddle-point problem can be directly solved using the direct solver `mldivide` "\" in Matlab with the linear system of equation $A\boldsymbol{x} = \boldsymbol{b}$. This solution `Vsol` is used to display the relative $\mathcal{L}_2$ error `conv_rel` with the solution computed by the Uzawa algorithm.

$$\mathrm{rel}_{\mathcal{L}_2} = \frac{||\boldsymbol{x} - \boldsymbol{x}^{(i+1)}||}{||\boldsymbol{x}||} \tag{25}$$

The stopping criterias are based on the previously discussed relative residual equation (23) and the relative increment equation (24)

```
load('matrices');

% computing the solution with a direct solver
Vsol = A\b;

final = 200; % number of maximum loops
error_rel = zeros(1,final);
incr_rel = zeros(1,final);
conv_err = zeros(1,final);
tol = 1E-4;

% arrays used in the algorithm
d_1 = zeros([length(f),final]);
d_2 = zeros([length(g),final]);
u = d_1;
p = d_2;
r = d_2;
alpha = 0;
beta = 0;

%UZAWA algorithm starts here
u(:,1) = K\f;
r(:,1) = G'*u(:,1) - g;
d_2(:,1) = r(:,1);
for i = 1:final-1
    d_1(:,i) = K\(G*d_2(:,i));
    alpha = (r(:,i)'*r(:,i))/(d_2(:,i)'*G'*d_1(:,i));
    p(:,i+1) = p(:,i) + alpha*d_2(:,i);
    u(:,i+1) = u(:,i) - alpha*d_1(:,i);
    r(:,i+1) = r(:,i) - alpha*G'*d_1(:,i);
    conv_err(i) = norm(Vsol - [u(:,i+1);p(:,i+1)])/norm(Vsol);
    error_rel(i) = norm(r(:,i+1))/norm(r(:,1));
    incr_rel(i) = norm(alpha*d_1(:,i))/norm(u(:,i+1));
    disp(['Iteration ',num2str(i),': Error of ',num2str(error_rel(i))])
    disp(['Directional error of ', num2str(incr_rel(i))])
    disp(['Convergence error of ', num2str(conv_err(i))])
    if error_rel(i) < tol AND incr_rel(i) < tol
        break
    end
    beta = (r(:,i+1)'*r(:,i+1))/(r(:,i)'*r(:,i));
    d_2(:,i+1) = r(:,i+1) + beta*d_2(:,i);
end
```

2.6 A parallel version of the Uzawa Algorithm

Hypre librairies uses the parallel environment `MPI` which is triggered with a `MPI_Init` and closes at the complete end of the algorithm on a `MPI_Finalize` command. There is no possibility to shut down an restart the parallel `MPI` implementation during the algorithm to perform operations in a sequential way.

The way the `MPI` parallel implementation works is: the same program runs on all processes which can all be distinguished by a unique identifier called *rank*. All variables in a process are local, there is no concept of shared memory. The program has to be written in a sequential language supported by `MPI`, like Fortran, C or C++. The communication performed between each processor is done via calls to an appropriate library. All messages transferred between processes carries data which could be a simple item or a complicated distributed structure. Although `MPI` is not the only message passing protocol, other specialized languages have been developed[17] but their comparison is beyond the scope of this thesis.

Since there are no notion of shared data in `MPI`, the matrices and vectors have to be split before the start of the algorithm. The functions developed in the library are dependant on the scattering of the data, which might be conducted in a column or row wise decomposition. Although most of the operations used in the Uzawa algorithm are embarrassingly parallel, data communication between processors is needed for the **matrix-vector product** and **inner product**. Their behavior is explained in the following subsection.

2.6.1 Hypre implementation

Hypre is a very powerful object oriented library of parallel solvers with a large amount of tools implemented. The manipulation of the data structure is split among 4 different conceptual interfaces depending on the grid-based interfaces of the domain. The objective is to build an iterative solver for a saddle-point linear system of equations, for that we do not have access to information about the domain nor the grid structure. Thus, the Linear-Algebraic System Interface (`IJ`) was picked since it is a very generic interface.

The `IJ` interface performs a row-wise data decomposition for the reason that it is the only scalable approach for assembling matrices on thousands of processes[8]. The matrices are distributed by block of rows where each processor $i \in (0, P-1)$ owns the sub-matrix A_i,

$$\begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{P-1} \end{bmatrix}$$

where $A \in \mathbb{R}^{m \times n}$. A component of the split matrix is written A_{ij} with the row partition $i \in (m_{i-1}, m_i)$ for which the whole column $j \in (0, n)$ can be retrieved.

The implementation scheme to initialize the sub-matrix in each processor has to be done on a collective call, each processes builds their own sub-matrix in a synchronized scheme by reading a text file provided. The sub-matrices are created as an object type `HYPRE_PARCSR`, the only object type supported in the implemented functions. Once initialized, the routine `SetValues()` sets matrix values for some number of rows (`nrows`) and some number of columns in each row (`ncols`), in a Compressed Column (CC) format for storing sparse matrices.

The same split among the processes occurs for the vectors, each processes owns a sub-vector $\boldsymbol{b}_i$, for $i \in (0, P-1)$ which is a portion of the full vector $\boldsymbol{b}$. Their object type and routines are similar to those of the matrices.

Each processor contains a row portion of the matrices and vectors. The matrices and vectors are not completely accessible at once by each processor and if needed, such information has to be fetched from the owner of the data. Simpler functions like vector updates and scalar-vector product could be locally computed and does not require inter-processor communication. They are respectively performed by the functions `hypre_ParVectorAxpy` and `HYPRE_ParVectorScale` .The two other operations in the Uzawa algorithm are dependent of the data owned by the other processes, their behavior is determined by the row-wise data decomposition :

- **The inner product** of the owned sub-vectors is performed to obtain a local inner product. Then a reduction routine is done in a summation way by calling the `MPI` library's function `MPI_AllReduce` . The resulting inner product is the summation of all local inner product of each processes. This operation is performed by the function `HYPRE_ParVectorInnerProd` .
- Due to the row-wise data decomposition, the **matrix-vector product** needs data about the subvectors of the other processes. This operation could be time-consuming if not adequately implemented, see the note on *edgce-cut* from graph theory[18]. This is being taken care of by the function `HYPRE_ParCSRMatrixMatvec` and `HYPRE_ParCSRMatrixMatvecT` for the product with the transpose of the matrix.

Notes on the edge-cut notion :
The linear system of equation is split among all the different processes in different sub-domains or subsets. An edge is an operation linking different data of the subsets. Sometimes, the edge is cut by the subsets boundary when the data are not included in the same subsets, and is part of a *cut-set* : the set of edges that have one endpoint in two different subsets. An edge-cut is important as it represents data not locally available that need to be reached from other processes.

A function that generates no edge-cut performs operations between locally owned data on the processor. For example, a vector update is an embarrassingly parallel function because it does not require communication and therefore, the *cut-set* is empty. On the other hand, a matrix-vector product function in a row-wise data decomposition is an operation written

$$\sum_i A_{ij}\boldsymbol{x}_j \qquad \text{for } A_{ij} \in P_i$$

for which each non zeros component $\boldsymbol{A}_{ij}$ has to be coupled with the vector component b_j. Since the vector is also split between processors, the component b_j might not be locally owned and would represent an addition to the *cut-set.* Each cut represents a component $\boldsymbol{b}_j$ that needs to be fetched in the adequate subset for the matrix-vector product to correctly work.

Figure 1: Parallel implementation of matrix-vector product

The Figure 1 is an example of an MPI parallel implementation of a matrix-vector product with 3 processors. The matrix A and vector $\boldsymbol{b}$ are split among 3 processors P^i with $i \in (0, 1, 2)$ in a row-wise decomposition. Due to the sparsity of the matrix, the product of $A(P^1)$ with the vector $\boldsymbol{b}$ needs the following components: $\boldsymbol{c_1} = \{0, 1, 2, 3, 4\}$.

A portion of the vector is locally stored and so, only the edge-cut components are to be fetched in the respective processors $\boldsymbol{ec}_1 = \{0, 1, 4\}$. Instead of just reaching for the whole vector $\boldsymbol{b}$, the components have to be individually reached in order to maintain efficiency and to not induce overhead.

2.6.2 A scheme of the algorithm

The Uzawa algorithm implemented in Hypre follows step by step the subsection 2.3. Uzawa iteration described earlier in this section. The matrix splitting is not being taken care by this algorithm, this part is completed by the Matlab interface. The matrices and vectors are split and saved on a folder accessible by Hypre. The Hypre's implementation of Uzawa striclty performs the solution of the saddle-point problem and writes the solution back in the folder.

The parallel implementation of Hypre's algorithm is described on Figure 2. As a preliminary step, the matrices and vectors are initialized and the memory is allocated on each processors. Each processor P^i starts by loading their data and performing the Uzawa algorithm sequentially. Communication can occur at each instruction and is not restricted to a neighbor processor. After reaching their portion of the solution, a convergence test in the form of equations (23) and (24) is ran. If failed, the Uzawa algorithm restarts for each processor at instruction C until the solution is obtained.

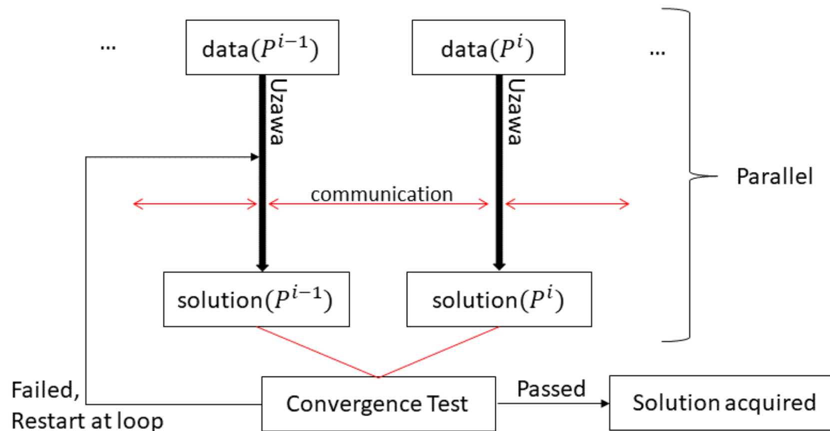

Figure 2: Hypre parallel implementation of UZAWA algorithm

Some range of flexibility has been left to the user as the iterative solver can be chosen from the ones implemented. The performances of the different iterative solvers used in the Uzawa algorithm are written in section 3. Results.

Since the matrix splitting is not being taken care of, the hypre algortihm is rendered useles without the proper pre-processing of the data. A Matlab interface is considered to serve as a user-friendly communicator with hypre and can supply the desired pre-process algorithm. The complete process solving a saddle-point problem through a Matlab interface is explained in the following subsection.

2.7 Saddle-point problem solver with Matlab interface

This subsection explains the usage of the saddle-point problem solver implemented in hypre through a Matlab interface. The interface initiates the complete process and communicates with the saddle-point solver through a folder called `hypre4matlab` installed within the hypre library. A scheme of the complete process is pictured on the following figure.

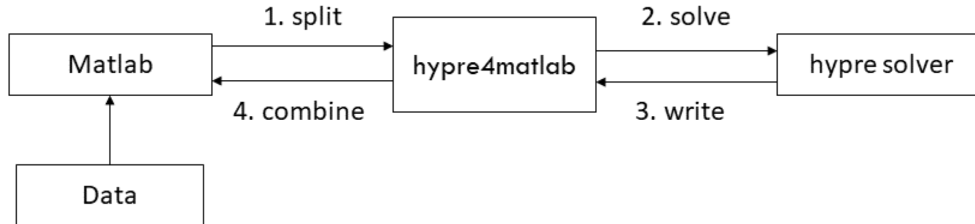

Figure 3: Scheme of saddle-point problem solver with Matlab interface

When the code within the interface is launched, the matrices and vectors corresponding to the saddle-point problem are split into `nOfProc` parts corresponding to the different number of processors chosen. This data is numbered accordingly to the `rank` of the processor in charge and sent to the solver folder `hypre4matlab`. The Uzawa algorithm is initiated and the solution to the saddle-point problem is computed using the iterative solver chosen by the variable `solver_id` within the Matlab interface. The scattered solution is then written within the folder `hypre4matlab`, loaded back and combined in Matlab to perform post-process analysis.

The iterative solver of use could be modified by changing the variable `solverID` to the 4 different possibilites.

	PCG	GMRES
AMG	0	20
ParaSails	1	21

Table 1: `solverID` parameter

If the algorithm is to be executed on a cluster, it usually involves a job-scheduler in the form of a shell file. The complete process can the be executed by scheduling the task on the cluster.

The solver and preconditioner implemented in the hypre algorithm are described in the following subsections.

2.7.1 GMRES

The generalized minimum residual method is an algorithm developed by Saad & Schultz[6] to perform the solution of a non-symmetric system of linear equations. The method is a generalization of the MINRES method developed by Paige & Saunders[19] a robust algorithm for indefinite coefficient matrices as well as positive-definite matrices.

The GMRES method is derived by replacing the symmetric Lanczos occurrence used to build the solution $\boldsymbol{x}^{(i)}$, by the Arnoldi variant for non-symmetric matrices[13]. The Lanczos occurence is closely related to the PCG method and gave birth to the MINRES method, which is adequate for symmetric indefinite systems. The usage of the Armoldi variant lead the path to built the solution

for non symmetric problems.

The algorithm is constituted of two loops which iterates k times to build up the orthonormal basis $\boldsymbol{v}^k$ of the Krylov space $\mathcal{K}_k(A, \boldsymbol{v}^{(1)})$. After reaching a stopping criteria the solution $\boldsymbol{x}^{(i)}$ of the linear system of equation is assembled.

$\boldsymbol{v}^{(1)}$ is the unit vector of the initial residual $\boldsymbol{r}^{(0)}$

$$\boldsymbol{v}^{(1)} = \frac{\boldsymbol{r}^{(0)}}{||\boldsymbol{r}^{(0)}||} \tag{26}$$

on which the orthonormal basis is constructed.

$$\left\{\boldsymbol{v}^{(1)}, \boldsymbol{v}^{(2)}, \cdots, \boldsymbol{v}^{(k)}\right\}$$

The additional vector $\boldsymbol{v}^{(k+1)}$ is added to the basis by checking orthogonality with all previously constructed vectors $\{\boldsymbol{v}^{(j)}\}_{j=1}^{k}$. Due to this, the required work and storage is proportional to the amount of iterations. There exist a point to where the computational cost of an iteration is too large, this is restricted by a restart parameter m. Once $k > m$, the GMRES method stops, computes the temporary solution $\boldsymbol{x}^{(i)}$ and restarts with it as to compute the initial residual $\boldsymbol{r}^{(0)}$. However, this may put a break on the rate of convergence of the solution compared to a full GMRES method[13].

2.7.2 BoomerAMG

BoomerAMG is the parallel implementation of the Algebraic Multigrid Method which is part of the multigrid methods[20][21], a class of very efficient iterative solvers designed originally for discrete Poisson problems. The strength of those methods comes from the fact that the computational work is linearly proportional $\mathcal{O}(N)$ to the dimension of the discrete problem.

The essence of multigrid is to decompose the grid function into its component and split them among multiple subspaces to quickly reach the solution. The relaxation scheme, or smoother, used by the multigrid method is efficient with high frequency error modes but does not do well with the smooth part of the error. The smoother part of the error stands out more in coarser grid, from which the error is approximated using the relaxation scheme. The error is later corrected in the finer grid.

The error components is rapidly reduced in the coarser subspace which contains much less degrees of freedom. Although this method originated from a *two-grid* algorithm, modern methods use a serie of coarser grids to gradually reduce the problem size. The coarsening strategy, responsible for the way the coarser grid is build, is at the root of the efficiency of the iterative solver.

The faster method of the multigrid class is the geometric multigrid method but it is not frequently used due to its lack of flexibility. On the other side, AMG has a wide range of applicability and a very robust solution method best developed for symmetric positive definite problems[22]. The coarsening of the AMG is completed solely on the algebraic information.

The coarsening strategy are heuristics methods to select which component is required for a coarser grid. They evaluate the strength of a node regarding the number of links it has with its neighbors. A node i strongly depends on a node j if

$$-a_{ij} \leq \theta \max_{k \neq i}(-a_{ik})$$

Where θ is a user-defined threshold. If $\theta = 0$ is selected, then there are no coarsening.

The complexity of a coarser grid can be further reduced by using an *aggressive coarsening*. The choice of nodes is more selective by introducing the notion of long-range connections, gradually removing the number of entries in the coarser grid. The coarsening will results in less number of nodes selected, less communications to be done in parallel and so affect the rapidity of the algorithm. On the other side, it will impact substancially impact the complexity of the preconditioner[22].

2.7.3 ParaSails

ParaSails is a parallel implementation of a sparse approximate inverse preconditioner[8][23]. The function evaluate the sparsity pattern through the usage of the least-squares minimization. The preconditioner can handle symmetric positive definite matrices and non-symmetric and/or indefinite matrices depending on the parameters used in the function in hypre.

ParaSails approximates the sparsity pattern of the inverse of the matrix A. Since A is sparse, the inverse of the matrix is dense. The sparsification of the matrix A is performed by dropping all entries in a symmetrically diagonally scaled when a value is smaller than the parameter `thresh`[8]. The accuracy and cost of ParaSails is defined by the parameters to select the sparsity pattern. The storage spend to build the preconditioner is usually smaller than the space required to store the original matrix.

The preconditioner P is built with a least-squares minimization (Frobenius norm).

$$||AP - I||_F \tag{27}$$

Developing the preconditioner in column vectors $P = [\boldsymbol{p}_1, ..., \boldsymbol{p}_n]$, the Frobernius norm can be written in euclidean norm

$$||AP - I||_F^2 = \sum_{i=1}^{n} ||A\boldsymbol{p}_i - \boldsymbol{e}_i||_2^2 \tag{28}$$

where $\boldsymbol{e}_i$ is the euclidean vector.

Since the matrix A is sparse, the least square problem is reduced to the non-zero entries S_i of $\boldsymbol{p}_i$ and T_i of $A(:, S_i)$.

$$||A\boldsymbol{p}_i - \boldsymbol{e}_i||_2 = ||A(T_i, S_i)\boldsymbol{p}_i(S_i) - \boldsymbol{e}_i(T_i)||_2 \tag{29}$$

3 Results

The results performed on the Hypre implementation of the Uzawa algorithm are described in this section. This chapter starts with a short description of the problems and its dimensions. A set of 7 matrices is split into two categories, small-scale and large-scale problems, over which the efficiency of solvers is compared. The matter starts by using Matlab's direct solver which has the advantages of being very fast and rigid. However, the direct solver's time computation does not linearly scale with the number of unknowns and it cannot compete with an iterative solver for large-scale matrices. The usage of iterative solvers is a necessity for larger matrices.

The study of the efficiency of the Uzawa algorithm over all set of matrices with different iterative solvers concludes this chapter.

3.1 Problem statement

The set of matrices are obtained from the discretization of a $\mathbb{R}^3$ Stokes problem in a cube domain $\Omega \in (0,1)^3$ with a varying number of elements used for the mesh in each direction. The elements are cubes with 20 nodes for the velocity and 8 for the pressure. The number of unknowns for the velocity is multiplied by 3 since we are working in three dimensions.

The viscosity functions used in the definition of the Stokes problem are not constant but have a spatial variation of up to 4 orders of magnitude. This represents a challenge for the solver and requires the use of preconditioners.

The number of unknowns and nodes in each matrices of the set is displayed in the Table 2. The line between small and large-scale problems is drawn by the drop of efficiency from the direct solver of Matlab for matrices, around 200,000 of unknowns. From now on, references to the matrices will be done by their number of unknowns.

elements	*# nodes for* $\boldsymbol{u}$	*# unknowns*
$5^3 = 125$	1,331	4,208
$10^3 = 1,000$	9,261	29,113
$15^3 = 3,375$	29,791	93,468
$20^3 = 8,000$	68,921	216,023
$25^3 = 15,625$	132,651	415,528
$30^3 = 27,000$	226,981	710,733
$35^3 = 42,875$	357,911	1,120,388

Table 2: Size of the Stokes problem

The equations of the Uzawa algorithm are written in the previous section 2.3. Only both instruction A and C(a) for solving a linear system of equation require the usage of a direct or iterative solver. The complete Uzawa algorithm run time and efficiency can be compared with the usage of different solvers.

The instruction C(a) is inside a loop, which means, depending on the saddle-point problem, it could be repeated over more than a hundred times: the solver plays an important role in the algorithm. It is a potential bottleneck, the efficiency would be severely impacted for the usage of an inappropriate solver.

3.2 Direct solver

This subsection is focused on the Matlab's direct solver to compute the solution of a Stokes problem. The operator `mldivide` "\" solves the linear system of equation $A\boldsymbol{x} = \boldsymbol{b}$ using a direct solver

of choice depending on the dimensions and format of A and $\boldsymbol{b}$.

The direct solver takes advantages of the properties of the matrix, such as symmetry, by dispatching an appropriate solver. This approach aims to minimize computation time[24]. `Mldivide` works like a decision tree and chooses a different direct solver depending of the properties of the matrix. In the case of solving the presented saddle-point problem, the matrix at hand is symmetric positive definite and so, the direct solver used is a LDL or Cholesky.

The average time for solving different matrices of the saddle-point problem has been written in the table 3.

# unknowns	*average time*
4,208	$\sim$ 0.5 sec
29,113	$\sim$ 6 sec
93,468	$\sim$ 120 sec
216,023	$\sim$ 1100 sec

Table 3: Computational cost of a monolithic Matlab solver

The computational time rises up by an order of magnitude for each matrix. On top of that, Matlab's direct solver is a memory expensive operation and the memory requirements get unaffordable for larger matrices.

The average time to directly compute the solution of a monolithic system of equation is relatively quick for matrices up to 100,000 degrees of freedom. The previous results cannot be directly compared with an iterative solver as they are inadapted to perform the solution of a saddle-point problem in a monolithic scheme; the Uzawa algorithm is required.

For the Uzawa algorithm, the options are to use either a direct or iterative solver. The computational time to complete the algorithm can be evaluated and the number of loops serves as an indicator. Since the workload per iteration doesn't change, the number of iterations for a saddle-point matrix is similar and not related with the choice of the solver. The computational time to perform the Uzawa algorithm would roughly be equal to the number of iterations multiplied by a single iteration time.

The time completion of an Uzawa algorithm depends on the solver used. A simple way to compare the usage of a direct and iterative solver would be to compare their efficiency over the repeated instruction C.(a) inside the loop $K\boldsymbol{d}_1^{(i)} = G^\top \boldsymbol{d}_2^{(i)}$. The matrix on the left-hand side K is symmetric positive definite, a property that should be acknowledged by the user for choosing the adapted iterative solver and the appropriate preconditioner. The PCG method is an appropriate iterative linear solver in this case[13]. The Table 4 compares the time computation for solving small-scale linear system instruction C.(a) between the direct method and the `pcg` function of Matlab with an 10^{-6} tolerance.

# unknowns	*direct*	*PCG*($\epsilon = 10^{-6}$)
3,993	$\sim$ 0.2	$\sim$ 1
27,783	$\sim$ 4	$\sim$ 10
89,373	$\sim$ 70	$\sim$ 60
206,763	$\sim$ 1000	$\sim$ 300

Table 4: Computational time [s] spent on solving instruction C.(a)

Looking at the results obtained from Table 4, it is clear that an iterative solver is faster than the direct method for the Uzawa algorithm with large matrices. Additionally, the computational

time of the iterative solver could be reduced by either implementing a preconditionner or reducing the tolerance criteria to reach a solution. Modifying the tolerance has an impact of the speed of convergence and the accuracy of the solution. There exist a lower limit for the tolerance ϵ for which the Uzawa algorithm won't converge due to imprecise solutions in instruction C(a) being computed.

Although the time spent on the Uzawa algorithm between the direct and iterative solvers are comparable, we are still far away from the efficiency of the monolithic Matlab solver on the saddle-point problem. In order to see further improvement of the algorithm, we have to make usage of more computational resources and dig into parallel implementation.

3.3 Iterative solver

The Hypre implementation of Uzawa algorithm is very similar to Matlab but permits the usage of multiple processors to speed up the computation. Because there exist no direct solver in Hypre, the results of the monolithic direct solver, table 3, cannot be directly compared.

The iterative solver has a large impact of the scalability of the algorithm. The algorithm can be tested with the usage of multiple iterative solvers for which there exist many different parameters.

The system interface chosen for Hypre limits the possibility to a variety of dozen different solvers and preconditioners. Unfortunately, due to implementation problems not all of them could be used in this algorithm, the following table list the different combinations implemented

Method \ Preconditionner	none	BoomerAMG	ParaSails
PCG	✓	✓	✓
GMRES	✓	✓	✓
BoomerAMG	✓		

Table 5: Iterative Solver-Preconditioner imported in hypre implementation of Uzawa

The AMG method is a very powerful solver that require a proper amount of tuning to be competitive. PCG and GMRES are Krylov subspace methods for which each operations can be computed cheaply. This gives a great advantages in terms of consumption of computational resources. Results for the Uzawa algorithm with those methods are described in the following subsection.

3.4 Hypre's code results

This subsection runs all the tests to measure the efficiency of Hypre's implementation of Uzawa algorithm. The time for each iteration can be measured separately, splitting the influence of the iterative solver and the other instructions in order to test the scalability of the algorithm in each part of the code.

All size matrices were run over the two solvers with the ParaSails preconditioner. The following subsections will discuss about their time computation and scalability. This could allow us to justify our choice of most efficient iterative solver for solving the Stokes problem under analysis using Uzawa.

The AMG preconditioner is a very tunable function that deserve the adequate parameters in order to be efficient. However, in the context of this problem, the study over the parameters has not been further pushed and a few tests were run with the default's parameters. In this sense, the AMG preconditioner provided results that are not competitive with ParaSails. The results with BoomerAMG preconditioner are written in the subsection 3.4.2.

All the following results were computed with the same stopping criterias. The selected tolerance ϵ has been fixed prior to the relative $\mathcal{L}_2$ error equation (25). The tolerance used for the stopping criteria was fixed at 10^{-2} which resulted in an approximative 10^{-4} relative $\mathcal{L}_2$ error. The exact solution of the Stokes problem has been computed using the direct Matlab's solver, which could be performed over smaller-scalled matrices (until 710,783 unknowns).

Since the linear system of equation to be solved is similar in each iteration, there is no need to each time build a preconditioner. Although this forces us to keep the same iterative method over the completion of the Uzawa algorithm, which could be restrictive in the case of a non-linear system of equation.

3.4.1 Strong scalability results

The strong scalability of an algorithm is verified by checking the speedup factor, the ratio between the sequential and parallel algorithm time computation. This ratio measures the relative performance of two systems computing the same instructions with different computational resources. With P being the number of processors, the speedup factor is expressed

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \leq P \tag{30}$$

The computational time is decreasing by less than a 1/P factor, a straight line of coefficient 1 demonstrate the perfect scalability but is rarely achieved due to multiple reasons[10][9]. The later the curve diverges from the straight line, the better the scalability of the method is.

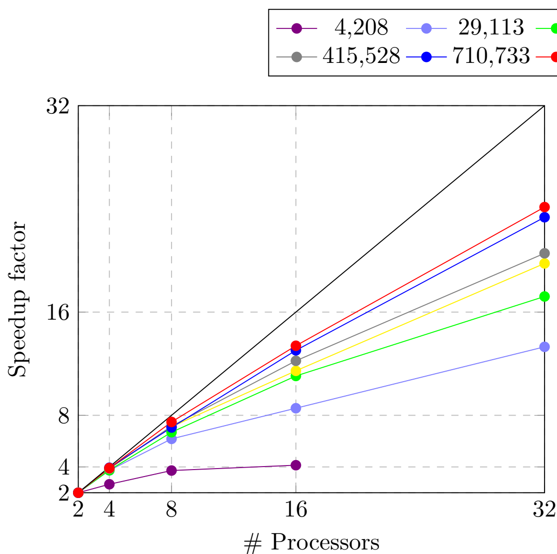

Figure 4: ParaSails-PCG

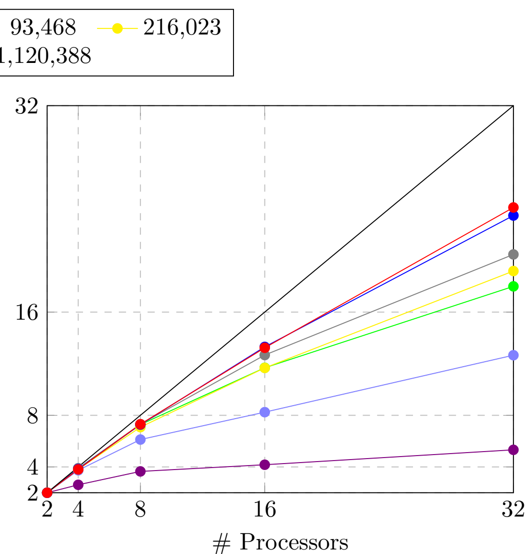

Figure 5: ParaSails-GMRES

The time spent is proportional to the number of processors and the two methods are giving similar results. Their strong scalability is getting better as the dimension of the problem increases: this is explained by the induced overhead introduced with small problems. In order to correctly scale, the amount of operations per processor should be adequate, which is not the case for small-scale problems. The amount of data per processor is too little and more time is spent in the communication between each processor[10][9].

The scalability is an important property of a parallel algorithm. The run time further decreases by using more processors, a resource that could be accessible. On modern parallel architecture,

the method that provides the best results for this problem is ParaSails-PCG which is close to a speedup factor of 24.14 with 32 processors over a saddle-point problem of 1,120,388 unknowns. Both methods provide very similar results as seen over the figures 6 and 7, a comparison of the strong scalability of both methods with the matrices of 710,733 and 1,120,388 unknowns.

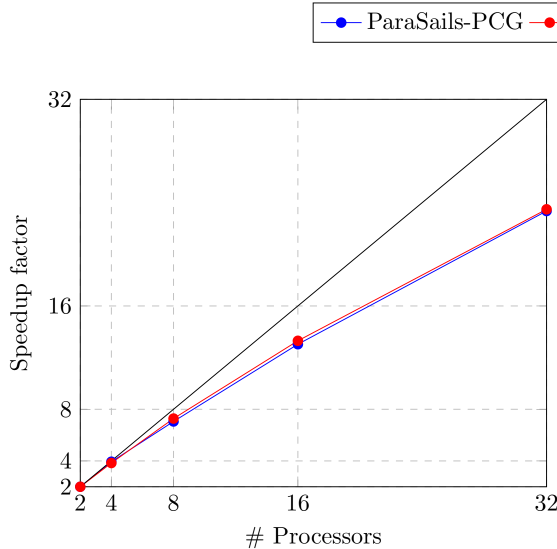

Figure 6: 710,733 unknowns

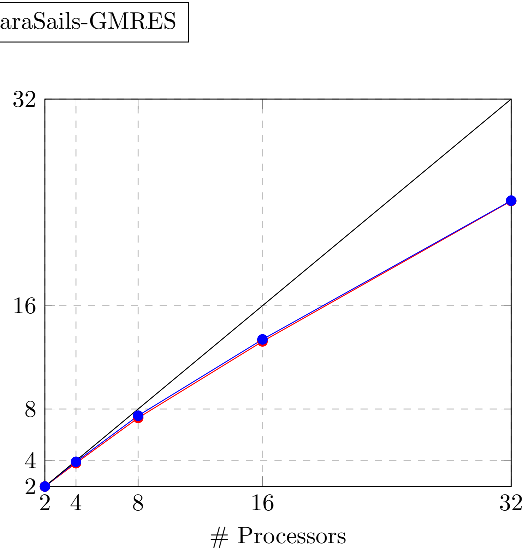

Figure 7: 1,210,388 unknowns

Rephrase

Another component that shouldn't be dismissed is the time computation. If the number of processors available in a cluster is fixed, the computational time cannot be further reduced by demanding more resources. This category is completely dominated by the ParaSails-PCG method, as it can be seen in the Table 6 with the computational time for the largest size matrix.

# unknowns	*ParaSails-PCG*	*ParaSails-GMRES*
4,208	9	10.6
29,113	51.2	58.8
93,468	200.3	274.9
216,023	295.5	327.8
415,528	543.4	630.2
710,733	924.6	1070.6
1,120,388	2797.2	5560.6

Table 6: Time computation [s] for all matrix sizes at 16 processors

As seen of the Table 6 in the context of this problem, the PCG method is more efficient than GMRES. Although the scalability is very close, there is a gap between their computational time that tends to increase with the size of the matrix.

The GMRES solver with the actual parameters lack efficiency over large scale matrices. This is explained by the increase of the computational cost induced by saving the constructed vector of the orthogonal base $\{\boldsymbol{v}^{(j)}\}_{j=1}^{k}$. As the matrices grow larger, the size of the vector increase and with it the memory requirements. Checking for orthogonality to build a new orthogonal vector $\boldsymbol{v}^{(j)}$ is computationally expensive. Although this problem could be solved by reducing the restart parameter for large matrices, it could have an impact on the rate of convergence. The GMRES

method should be the subject of a large series of test run over different restart parameters to deduce a trend and decrease the run time.

3.4.2 AMG results

In the current parameters for AMG, the preconditioner has been unsuccessful to give similar results to ParaSails. Due to the long time to process, the solution of saddle-point problems with the Uzawa algorithm has been limited up to matrices of 93,468 unknowns. A few tests were run with PCG and GMRES solver for saddle-point problem for the 3 smaller matrices. Although the scalability results are similar than the previous methods with a ParaSail preconditioner, the time completion is much higher.

# unknowns	*AMG-PCG*	*AMG-GMRES*
4,208	82	89
29,113	388	405
93,468	2261	2462

Table 7: Time computation [s] for small-scale matrices at 16 processors

The time completion for a matrix of 4,208 unknowns actually increased from the sequential time of 80 seconds, as expected of the scalability for small-scale matrices. Note that once again the GMRES time completion gives slightly longer results than the PCG method.

The slow time completion can be verified by looking at the time for the AMG-method to solve a single linear system of equation Table 10.

3.4.3 Details of scalability

On the side, smaller scalability test were run over two important sections of the code to give a better understanding of the previous results. A time measuring command has been introduced in the Hypre algorithm to give further details in the following parts of the code

- *Initialization of the matrices and vectors*
  This part contains the start of the algorithm; from allocating memory space to the different variables, to building the matrices and vectors with the text file provided. This is not a part of the instructions as it is a preliminary step required for the programming language, essential to Hypre but not necessary in Matlab.
- *Setup of the first iterative solver*, instruction A
  This method is instantaneous for unpreconditionned method but could be expensive in the case of setting up an abusive (but very effective) preconditionner. This matter can be verified by checking the time spent in this step.
- *Solving of the first iterative solver*, instruction A
  This step is heavily dependent on the previous, and the scalability of a solver is generally measured with the implication of both. But, the solvers are Krylov subspace methods, known to be scalable with or without the usage of a preconditionner.

A similar time measurement test was run over all other operations of the algorithm. Except for the matrix-vector product, all operations are embarrassingly parallel, thus they are not time consuming as they demand little to no communication between each processors. The computational time measured is around the smallest time unit of 0.01 second.

The initialization of the matrices and vectors are inherent operations in an Hypre algorithm. The results are shown in the following table for different problems dimension.

#Processors/#unknowns	4,208	29,113	93,468	216,023	415,528	710,733	1,120,388
2	0.39	1.63	5.1	12.01	23.88	41.24	66.56
4	0.29	0.98	2.85	6.48	12.38	21.44	34.49
8	0.26	0.64	1.74	3.44	6.45	12.6	17.65
16	0.28	0.51	1.04	2.07	3.67	6.59	9.42
32	0.43	0.62	1.03	1.64	2.9	4.49	6.34

Table 8: Time consumption [s] of matrix/vector initialization by the dimension of the problem

The initialization does not show great scalability results, the best speedup factor is around 10.5 for 32 processors. Thought it is an essential step that cannot be evaded and the time spent is relatively small compared to the time spent solving a single linear system of equation.

The appropriate preconditioner has to be chosen to fasten the rate of convergence. The last subsection analyzes the results of scalability of the different methods implemented to solve saddle-point problems. The conclusion obtained in the previous subsections should be retrieved here as the same results should be seen for solving a single linear system of equation.

Only the solving time computation is reproduced in each step since the preconditionner is reused. This gives us an approximation to how much time is being saved, as seen in the following table for different matrix sizes for both preconditioners ParaSails and AMG. The computational time for matrices larger than 89,373 unknowns were not done with AMG preconditioner due to the excessive computational time.

# unknowns	*ParaSails*	*AMG*
3,993	0.04	0.28
27,783	0.09	0.54
89,373	0.24	1.03
206,763	0.43	-
397,953	0.73	-
680,943	1.14	-
1,073,733	1.77	-

Table 9: Setup Precondiditioner time [s] for all matrix sizes at 16 processors

Since the solving of the linear system of equation has to be reproduced in each iteration, the difference in the computational time to perform Uzawa is explained by the results of the following table. This table shows the time completion over solving the first linear system of equation $K\boldsymbol{u} = \boldsymbol{f}$ with respect to their preconditioner-solver method.

# unknowns	*ParaSails-PCG*	*ParaSails-GMRES*	*AMG-PCG*	*AMG-GMRES*
3,993	0.01	0.01	0.28	0.33
27,783	0.06	0.08	0.74	0.74
89,373	0.48	0.61	7.03	7.03
206,763	0.87	1.01	-	-
397,953	2.12	2.82	-	-
680,943	4.42	5.59	-	-
1,073,733	14.7	28.99	-	-

Table 10: Solving time computation [s] for all matrix sizes at 16 processors

The ParaSails-PCG method gives small time results to solve a linear system of equation. The ParaSails-GMRES methods is close behind and the gap increase with the matrix size as expected from the previous results Table 6.

4 Conclusion

A Matlab interface is used to communicate and send instructions to a saddle-point solver in hypre. The Uzawa algorithm has been implemented in hypre and performs the solution in a parallel scheme. This code has a range of configurations to leave some flexibility to the user in running the solution. These parameters can be directly configured from the interface.

For the moment, the number of processors and the methods to solve the saddle-point problems are directly configurable from the Matlab interface. Matlab sequentially splits the matrices and vectors and provides them to the parallel solver. The Hypre code loads the data per processor and solves the scattered problem with the method chosen. Then it writes the split solution once finished. Information about the convergence is directly given through the Matlab interface and the solution can be loaded in order to perform post-process analysis. The preconditioner building has been dropped in each iteration, resulting into saved computational resources and reducing the run time.

A series of tests were run to solve saddle-point problems arising from Stokes equation with highly variable viscosity. The scalability of the code has been tested on a set of matrices over a large set of dimensions, from 4,208 to 1,120,388 unknowns. From the performed simulations, ParaSails-PCG and ParaSails-GMRES provided the best numerical performance. The optimal tuning of GMRES and AMG will be the subject of further investigations.

5 Further Improvements

This subsection is dedicated to remarks regarding the efficiency of the Hypre parallel implementation. Some points were notified through the thesis and can be addressed here.

The efficiency of GMRES over large scale matrices and the preconditioner AMG should be further analyzed. The method features a high number of parameters involved in the configuration that require proper tuning to obtain efficient results. This is especially important when considering non-symmetric matrices for which ParaSails may present limitations.

The context of implementation of the Uzawa algorithm in hypre was to perform the code on a small amount of processors. As the data of the split matrices and vectors are all saved on disk memory, being accessed by too many processors at the same time can affect the scalability of the algorithm and severely impact the performance of the software. For very large number of processors, the considered algorithm should be designed an run on distributed memory.

The split matrices and vectors are read by each corresponding processor but this is not performed as part of a Hypre preprocess. The splitting of the matrices and vectors over the folder can be time consuming. For large matrices and 32-processors computations, this splitting operation can represent more than 10% of the all run time. The scalability of the complete process of assembling the linear system and solving it is impacted if the splitting of the matrices is not being taken care of in a parallel scheme.

The matrix-vector product communication in a row-wise data decomposition depends on the sparsity of the matrix. The communication between processors can be a potential bottleneck and should be kept minimal. This communication could be reduced by switching lines/rows in the matrices via a pre-process function. Thus further reducing the runtime of the algorithm.

References

[1] M. Benzi, G. H. Golub, and J. Liesen, "Numerical solution of saddle point problems," *ACTA NUMERICA*, vol. 14, pp. 1–137, 2005.

[2] B. L. Buzbee, G. H. Golub, and C. W. Nielson, "On direct methods for solving poisson's equations," *SIAM Journal on Numerical analysis*, vol. 7, no. 4, pp. 627–656, 1970.

[3] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices.* Oxford University Press, 2017.

[4] J. Scott, *Sparse Direct Solvers 1: The Challenge.* 43th Woudschoten Conference, 10 2018. STFC Rutherford Appleton Laboratory and the University of Reading.

[5] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, "Mumps multifrontal massively parallel solver version 2.0," 1998.

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

[7] K. J. Arrow, L. Hurwicz, and H. Uzawa, "Studies in linear and non-linear programming," 1958.

[8] Lawrence Livermore National Laboratory, *hypre Reference Manual*, 2.11.2 ed. version 2.6.0b.

[9] T. Rauber and G. Rnger, *Parallel Programming: For Multicore and Cluster Systems.* Springer Publishing Company, Incorporated, 2nd ed., 2013.

[10] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers.* CRC Press, 2010.

[11] P. Ciarlet Jr, J. Huang, and J. Zou, "Some observations on generalized saddle-point problems," *SIAM J. Matrix Analysis Applications*, vol. 25, pp. 224–236, 03 2003.

[12] J. Donea and A. Huerta, *Finite Element Methods for Flow Problems.* Finite Element Methods for Flow Problems, Wiley, 2003.

[13] H. Elman, D. Silvester, and A. Wathen, *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics.* Numerical mathematics and scientific computation, Oxford University Press, 2014.

[14] F. Brezzi, "On the existence, uniqueness and approximation of saddle-point problems arising from lagrangian multipliers," *Publications mathématiques et informatique de Rennes*, no. S4, pp. 1–26, 1974.

[15] A. Quarteroni and S. Quarteroni, *Numerical models for differential problems*, vol. 2. Springer, 2009.

[16] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, vol. 49. NBS Washington, DC, 1952.

[17] F. Milicchio and W. A. Gehrke, *Distributed services with OpenAFS: for enterprise and education.* Springer Science & Business Media, 2007.

[18] S. Pemmaraju and S. Skiena, *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica®.* Cambridge university press, 2003.

[19] C. C. Paige and M. A. Saunders, "Solution of sparse indefinite systems of linear equations," *SIAM journal on numerical analysis*, vol. 12, no. 4, pp. 617–629, 1975.

[20] A. Greenbaum, *Iterative methods for solving linear systems*, vol. 17. Siam, 1997.

[21] S. Schaffer, "A semicoarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 228–242, 1998.

[22] K. Stuben, "Algebraic multigrid (amg): an introduction with applications," *Multigrid*, 2000.

[23] S. C. Hawkins and M. Ganesh, "Sparse approximate inverse preconditioners for electromagnetic surface scattering simulations," *ANZIAM Journal*, vol. 49, pp. 155–169, 2007.

[24] The Mathworks, *MATLAB Documentation*, version 2019a ed.