Parallelization of a Finite Element CFD code using PSPARSLIB: Application to Three-dimensional Free Surface Flows [1]

by

Azzeddine SOULAIMANI

Permanent address: Ecole de technologie supérieure, Département de génie mécanique
1100 Notre-Dame Ouest, Montreal,
PQ, H3C 1K3, CANADA

Temporary address: Center for Aerospace structures, University of Boulder,
Boulder, CO 80309-0429, U.S.A.

Yousef SAAD

University of Minnesota, Department of Computer
Science and Engineering
4-192 EE/CSci Building, 200 Union Street S.E., Minneapolis, MN 55455

and

Abdessamad QADDOURI
Centre de Recherches Mathématiques, Université de Montréal,
Montréal, PQ, H3C 3J7, CANADA

Abstract. This paper discusses finite element solution methods for simulating unsteady three-dimensional free surface flows. Parallelization of the code is carried out using PSPARSLIB and the MPI message passing libraries. This code is used to simulate three-dimensional incompressible flows in dynamic meshes. The governing equilibrium equations are written in the framework of the Arbitrary Lagrangian-Eulerian kinematic description to which the variational formulation is applied. The fluid flow and the dynamics of the mesh are solved in a coupled fashion using Newton's method. The resulting set of nonlinear equations is solved by a robust preconditioned GMRES algorithm. Parallel implementation issues such as data structures for the finite element method using domain decomposition and their impact on iterative solvers will be discussed. Finally, the computational performance is assessed on a practical three-dimensional hydraulic problem.

[1] Work supported by ARPA/NIST under grant NIST 60NANB2D1272, by NSF under grant CCR-9618827, by NSERC of Canada, and by the Minnesota Supercomputer Institute

1 Introduction

In this paper we discuss a finite element method for solving three-dimensional incompressible flows. Our ultimate objective is to develop a code capable of solving various free surface problems encountered in hydraulic structure design (e.g. penstocks, spillways...) as well as in some MHD processes. For this purpose, we choose to use the Arbitrary Lagrangian-Eulerian (ALE) concept for the kinematic description, the finite element method for space discretization, an implicit scheme for time discretization and finally all the equations are solved in a coupled fashion; the primary unknowns being the pressure, the fluid velocity and the mesh velocity. The computational performance issue is addressed through a parallel implementation on shared-memory parallel computers of the finite element data structure and the iterative solver. The PSPARLIB and MPI libraries are used for this purpose.

Domain decomposition is a general and convenient paradigm for solving partial differential equations on parallel computers. Typically, a domain is partitioned into several sub-domains and some technique is used to recover the global solution by a succession of solutions of independent subproblems associated with the entire domain. Each processor handles one or several subdomains in the partition and then the partial solutions are combined, typically over several iterations, to deliver an approximation to the global system. All domain decomposition techniques rely on the fact that each processor can do a big part of the work independently.

When the matrix arises from a finite element method and there is only one unknown per grid point then the adjacency graph and the finite element mesh are identical. One of the main implementation issues is to define general and useful data structures for 'distributed sparse matrix computations' and to draw efficient solution techniques from the domain decomposition framework. In order to implement domain decomposition techniques efficiently and for realistic problems we need to keep in mind the following requirements.

1. In order to be able to deal with the most complex problems that arise in realistic applications, it is vital to provide automatic tools for performing the many tasks that would otherwise make the implementation impractical.

2. Flexibility is critical in a general purpose Domain Decompositionimplementation. For example, the domains, or subgraphs, must be allowed to overlap if necessary and the data structures should take this into account.

3. The message passing programming paradigm represents one of the best possible environments for Domain Decompositiontechniques. In addition, Domain Decompositiontechniques can also benefit from heterogeneous computing environments.

The outline of the paper is as follows. In section 2, the ALE kinematic description is reviewed. The Navier-Stokes equations are written for the referential representation using respectively a Cartesian frame. In section 3, we discuss a possible choice of the referential motion. In section 4, the variational method using the Galerkin Least Squares (GLS) method is presented. In section 5, we discuss the use of the GMRES solution algorithm along with the ILUT preconditioning. In section 6, we present the parallel data structure and its impact on the iterative algorithm. In section 7, some numerical results are presented in order to assess the methods and algorithms proposed.

2 The Arbitrary Lagrangian-Eulerian kinematic description

Arbitrary Lagrangian-Eulerian kinematic description (or Referential) has proved to be very suitable for simulating some free boundary problems encountered in fluid and solid mechanics (free surface flows, large deformation problems,...). Unlike the Lagrangian and the Eulerian description, the computational domain

may be animated with its proper motion. In this situation it is possible to track the free boundary motion while maintaining a fairly regular mesh.

Let $\mathcal{B}$ denote the material body which occupies an open region of $\mathbf{R^{nd}}$, where nd is the space dimension (Figure 1). A fixed Cartesian reference system $(\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_{nd})$ is chosen such that the Cartesian coordinates of points of $\mathcal{B}$ are noted by $\boldsymbol{X} = (X_1, X_2, \ldots, X_{nd})$. The notation Ω stands for another open and bounded part of $\mathbf{R^{nd}}$ with smooth boundary Γ.

A motion of $\mathcal{B}$ is expressed in terms of a position function ϕ and the spatial velocity and acceleration vectors are then respectively defined by:

$$\boldsymbol{u}(\boldsymbol{x}, t) = \frac{\partial \phi}{\partial t}(\boldsymbol{X}, t) \tag{2.1}$$

and

$$\boldsymbol{a}(\boldsymbol{x}, t) = \frac{d\boldsymbol{u}}{dt}(\boldsymbol{x}, t) \tag{2.2}$$

with $\boldsymbol{x} = \phi(\boldsymbol{X}, t)$ being the position of point $\boldsymbol{X}$ at time t.

Any point $\hat{\boldsymbol{x}}$ of Ω will occupy a position $\boldsymbol{x} = \lambda(\hat{\boldsymbol{x}}, t)$ at time t following the referential motion λ, with velocity:

$$\boldsymbol{w}(\boldsymbol{x}, t) = \frac{\partial \lambda}{\partial t}(\hat{\boldsymbol{x}}, t).$$

We will use the following convention in our notation: every function defined over Ω will be denoted by a superposed hat; for instance, $\boldsymbol{w}(\boldsymbol{x}, t) = \boldsymbol{w}(\lambda(\hat{\boldsymbol{x}}, t), t) = \hat{\boldsymbol{w}}(\hat{\boldsymbol{x}}, t)$.

An updated referential configuration $\lambda_t(\Omega)$ is then completely determined by the velocity field $\hat{\boldsymbol{w}}(\hat{\boldsymbol{x}}, t)$ or by the gradient deformation tensor $\hat{\boldsymbol{F}}(\hat{\boldsymbol{x}}, t)$, whose components are given by:

$$\hat{F}_{ij} = \frac{\partial x_i}{\partial \hat{x}_j} \ .$$

Any point $\boldsymbol{x}$ of $\lambda_t(\Omega) \cap \phi_t(B)$ is a new position of two points $\boldsymbol{X}$ and $\hat{\boldsymbol{x}}$ being related by the following relations:

$$\hat{\boldsymbol{x}} = \lambda^{-1}(\phi(\boldsymbol{X}, t), t) = \Psi(\boldsymbol{X}, t). \tag{2.3}$$

The mapping $\Psi(\boldsymbol{X}, t)$ represents in fact the relative motion of $\mathcal{B}$ over Ω. Equation (2.3) may also be written as follows:

$$\boldsymbol{x} = \lambda(\Psi(\boldsymbol{X}, t), t) \tag{2.4}$$

a straightforward differentiation of (2.4) with repect to t leads to:

$$\boldsymbol{u}(\boldsymbol{x}, t) = \boldsymbol{w}(\boldsymbol{x}, t) + \hat{\boldsymbol{F}}(\hat{\boldsymbol{x}}, t) \cdot \bar{\boldsymbol{c}}(\hat{\boldsymbol{x}}, t) \tag{2.5}$$

where $\bar{\boldsymbol{c}}(\hat{\boldsymbol{x}}, t)$ represents the relative velocity; the components of which are expressed in the moving reference system, which is defined as $(\hat{\boldsymbol{e}}_1, \hat{\boldsymbol{e}}_2, \ldots, \hat{\boldsymbol{e}}_{nd})$ with $\hat{\boldsymbol{e}}_i = \frac{\partial x_j}{\partial \hat{x}_i} \boldsymbol{e}_j$, by:

$$\bar{c}_i(\hat{\boldsymbol{x}}, t) = \frac{\partial \Psi_i}{\partial t}(\boldsymbol{X}, t)$$

We also need the expression of the vector acceleration of the point $\boldsymbol{X}$ at time t as a function of $\hat{\boldsymbol{x}}$. It is derived as follows:

$$\boldsymbol{a}(\boldsymbol{x}, t) = \hat{\boldsymbol{a}}(\hat{\boldsymbol{x}}, t) = \frac{d\hat{\boldsymbol{u}}}{dt}(\hat{\boldsymbol{x}}, t)$$

and using the chain rule differentiation, we get,

$$\hat{\boldsymbol{a}}(\hat{\boldsymbol{x}}, t) = \frac{\partial \hat{\boldsymbol{u}}}{\partial t}(\hat{\boldsymbol{x}}, t) + (\bar{\boldsymbol{c}}(\hat{\boldsymbol{x}}, t) \cdot \hat{\boldsymbol{\nabla}}) \hat{\boldsymbol{u}}(\hat{\boldsymbol{x}}, t) \tag{2.6}$$

Using (2.5), equation (2.6) is then rewritten as:

$$\hat{\boldsymbol{a}}(\hat{\boldsymbol{x}},t) = \frac{\partial \hat{\boldsymbol{u}}}{\partial t}(\hat{\boldsymbol{x}},t) + [(\hat{\boldsymbol{F}}^{-1}\cdot(\hat{\boldsymbol{u}}-\hat{\boldsymbol{w}}))\cdot\hat{\boldsymbol{\nabla}}]\ \hat{\boldsymbol{u}}(\hat{\boldsymbol{x}},t) \tag{2.7}$$

where $\hat{\boldsymbol{\nabla}}$ stands for the gradient operator with respect to the referential $\hat{\boldsymbol{x}}$ coordinates.

2.1 Conservation equations in the Cartesian inertial frame

We now consider the formulation of conservation principles using the referential concept. The momentum material equilibrium is expressed in the spatial description by the classical relation,

$$\rho(\boldsymbol{x},t)\ \boldsymbol{a}(\boldsymbol{x},t) = \rho(\boldsymbol{x},t)\ \boldsymbol{f}(\boldsymbol{x},t) + \mathrm{div}_x\ \boldsymbol{\sigma}(\boldsymbol{x},t) \tag{2.8}$$

where ρ is the density, $\boldsymbol{f}$ is the body force and $\boldsymbol{\sigma}$ is the Cauchy stress tensor.

Since the actual configuration of the material body at time t may be unknown, it is then more convenient to rewrite equation (2.8) in a well defined domain such as the referential one. Making a change of coordinates, the above equation can be expressed in terms of $\hat{\boldsymbol{x}}$ by:

$$\hat{\rho}(\hat{\boldsymbol{x}},t)\ \hat{J}\ \hat{\boldsymbol{a}}(\hat{\boldsymbol{x}},t) = \hat{\rho}(\hat{\boldsymbol{x}},t)\ \hat{J}\ \hat{\boldsymbol{f}}(\hat{\boldsymbol{x}},t) + \mathrm{div}_{\hat{x}}\ \hat{\boldsymbol{P}}(\hat{\boldsymbol{x}},t) \tag{2.9}$$

where $\mathrm{div}_{\hat{x}}$ denotes the divergence operator with respect to $\hat{\boldsymbol{x}}$, $\hat{J}$ is the determinant of $\boldsymbol{F}$; and $\hat{\boldsymbol{P}} = \hat{J}\ \boldsymbol{\sigma}\ (\hat{\boldsymbol{F}}^{-1})^t$ is the Piola-Kirchhoff stress tensor of the first kind. This, however, does not have the same meaning as in the material description.

By substituting (2.7) into (2.9), the momentum conservation equations in the referential kinematic description become:

$$\hat{\rho}\ \hat{J}\ \Big(\frac{\partial \hat{\boldsymbol{u}}}{\partial t} + \bar{\boldsymbol{c}}\cdot\hat{\boldsymbol{\nabla}}\hat{\boldsymbol{u}}\Big) = \hat{\rho}\ \hat{J}\ \hat{\boldsymbol{f}} + \mathrm{div}_{\hat{x}}\ \hat{\boldsymbol{P}} \tag{2.10}$$

It is also possible to derive the expression of the mass conservation equation in a referential kinematic description, which reads:

$$\hat{J}\ \frac{\partial \hat{\rho}}{\partial t} + \mathrm{div}_{\hat{x}}\ \big(\hat{\rho}\ \hat{J}\ \hat{\boldsymbol{F}}^{-1}\ \hat{\boldsymbol{u}}\big) = 0\ . \tag{2.11}$$

There are many possibilities for choosing the referential domain Ω:

- Total ALE Formulation: the referential domain is defined by the initial mesh configuration.
- Incremental ALE Formulation: the referential domain is defined by the mesh configuration at a previous time $t-s$.
- Updated ALE Formulation: the referential domain is defined by the mesh configuration at the current time t.

2.2 Constitutive equations

It still remains to define the constitutive laws of the continuum. We restrict ourselves to incompressible ($\rho = constant$) and Newtonian fluids for which the Cauchy stress tensor is linear with respect to the velocity gradient tensor:

$$\boldsymbol{\sigma}(\boldsymbol{x},t) = -p\,\boldsymbol{I} + \mu\big(\boldsymbol{\nabla u} + (\boldsymbol{\nabla u})^t\big) \tag{2.12}$$

where p represents pressure, μ is the viscosity and $\boldsymbol{I}$ is the identity tensor. Making use of the change of coordinates, equation (2.12) is rewritten as,

$$\hat{\boldsymbol{\sigma}}(\hat{\boldsymbol{x}},t) = -\hat{p}\,\boldsymbol{I} + \mu\big(\hat{\boldsymbol{\nabla}}\hat{\boldsymbol{u}}\cdot\hat{\boldsymbol{F}} + (\hat{\boldsymbol{\nabla}}\hat{\boldsymbol{u}}\cdot\hat{\boldsymbol{F}})^t\big) \tag{2.13}$$

3 Definition of the referential motion

At the boundary Γ, the velocity $\boldsymbol{w}$ is either explicitly specified (if known) or it is determined for the free boundary Γ_f by imposing the kinematic boundary condition:

$$(\boldsymbol{u} - \boldsymbol{w}) \cdot \boldsymbol{n} = 0$$

where $\boldsymbol{n}$ is the outward normal vector at Γ. The above condition is a constraint on the normal component of $\boldsymbol{w}$ only. In the course of the boundary motion, all or some of the points inside the domain Ω move with a velocity $\boldsymbol{w}$ which is formulated in such a way as to maintain a relative regularity of the finite elements mesh of Ω.

A general statement may be presented as follows: find the referential velocity $\hat{\boldsymbol{w}}(\hat{\boldsymbol{x}}, \boldsymbol{t})$ such that

$$\beta(\hat{\boldsymbol{w}}(\hat{\boldsymbol{x}}, t)) = 0 \qquad \text{on} \quad \Gamma \tag{3.1}$$

$$\pi(\hat{\boldsymbol{w}}(\hat{\boldsymbol{x}}, t)) = 0 \qquad \text{in} \quad \Omega \tag{3.2}$$

Equation (3.1) is an additional condition required to define the position of the boundary (such as the kinematic boundary condition) and the operator π (algebraic or differential) is chosen arbitrarily to distribute the displacement of the boundary inside Ω in such a way to enhance the numerical solution (e.g. to preclude collapse or large mesh distortions).

For instance, in the case of free surface flows, the problem defined by equations (3.1) and (3.2) is reduced to seek for the referential velocity such that the free/moving part Γ_f of Γ follows the material motion in the normal direction to the boundary, i.e. equation (3.1) is specified as follows:

$$\beta(\hat{\boldsymbol{w}}) = (\boldsymbol{u} - \boldsymbol{w}) \cdot \boldsymbol{N} = 0 \qquad \text{and} \qquad \hat{\boldsymbol{w}}_i = 0 \text{ (for } i = 1, .., nd-1); \qquad \text{on } \Gamma_f \ ,$$

and

$$\beta(\hat{\boldsymbol{w}}) = \hat{\boldsymbol{w}} = 0 \qquad \text{at} \quad \Gamma/\Gamma_f$$

where $\boldsymbol{N} = \left(-\frac{\partial h}{\partial x_1}, \ldots, -\frac{\partial h}{\partial x_{nd-1}}, \ 1\right)$ is the outward vector at the free boundary which has a spatial parametric representation $h(x_1, \ldots, x_{nd-1}, \ t)$. In other words, the equation defining the free surface positions reads simply

$$\frac{\partial h}{\partial t} + u_1 \frac{\partial h}{\partial x_1} + u_2 \frac{\partial h}{\partial x_2} - u_3 = 0. \tag{3.3}$$

The operator π may be simply defined as being the Laplace operator, and only the nd^{th} component of $\hat{\boldsymbol{w}}$ could be allowed to vary inside the domain, that is

$$-\hat{\boldsymbol{\nabla}} \cdot k(\hat{\boldsymbol{\nabla}} \hat{\boldsymbol{w}}_3) = 0 \tag{3.4}$$

where k is a stiffness parameter which depends on the local element size. In order to avoid small elements to degenerate, k is defined as

$$k(e) = \frac{maxvol}{vol(e)}$$

where $vol(e)$ is the volume of element e and $maxvol$ is the maximum volume of the mesh elements. At the free surface $w_3 = \frac{\partial h}{\partial t}$.

4 Variational formulation

We are now interested in looking for weak solutions of the equilibrium equations (2.10)-(2.11) and for the dynamic mesh (3.3)-(3.4) respectively, assuming that the fluid is incompressible and we have appropriate boundary conditions. A Stabilized Galerkin formulation reads: for all test functions $\hat{\boldsymbol{v}}(\hat{\boldsymbol{x}},t)$, $q(\hat{\boldsymbol{x}},t)$, δh and δw_3 one has

$$\int_\Omega \hat{\rho}\hat{J}\hat{\boldsymbol{v}}\cdot\frac{\partial\hat{\boldsymbol{u}}}{\partial t}\,d\Omega+\int_\Omega \hat{\boldsymbol{\nabla}}\hat{\boldsymbol{v}}:\,\hat{\boldsymbol{P}}^t\,d\Omega\,+\int_\Omega \hat{\rho}\hat{J}\hat{\boldsymbol{v}}\cdot\left[(\hat{\boldsymbol{F}}^{-1}\cdot(\hat{\boldsymbol{u}}-\hat{\boldsymbol{w}})\cdot\hat{\boldsymbol{\nabla}})\hat{\boldsymbol{u}}\right]d\Omega-\int_\Gamma \hat{\boldsymbol{v}}\cdot\hat{\boldsymbol{P}}\cdot\hat{\boldsymbol{n}}\,d\gamma$$
$$+\int_\Omega \tau_1\rho\hat{J}\left[(\hat{\boldsymbol{F}}^{-1}\cdot(\hat{\boldsymbol{u}}-\hat{\boldsymbol{w}})\cdot\hat{\boldsymbol{\nabla}})\hat{\boldsymbol{v}}+\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{\nabla}}q\right]\cdot\left[(\hat{\boldsymbol{F}}^{-1}\cdot(\hat{\boldsymbol{u}}-\hat{\boldsymbol{w}})\cdot\hat{\boldsymbol{\nabla}})\hat{\boldsymbol{u}}+\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{\nabla}}p\right]d\Omega$$
$$+\int_\Omega \hat{q}\hat{\boldsymbol{\nabla}}\cdot(\hat{J}\hat{\boldsymbol{F}}^{-1}\hat{\boldsymbol{u}})\,d\Omega=\int_\Omega \hat{\rho}\hat{J}\hat{\boldsymbol{v}}\cdot\hat{\boldsymbol{f}}\,d\Omega \tag{4.1}$$

where $\hat{\boldsymbol{P}}\cdot\hat{\boldsymbol{n}}=\hat{\boldsymbol{f}}_s(\hat{\boldsymbol{x}})$ is the traction vector and $\hat{\boldsymbol{n}}$ is the unit outward normal vector along Γ

$$\int_\Omega[\delta\hat{h}+\tau_2(u_1\frac{\partial\delta h}{\partial x_1}+u_2\frac{\partial\delta h}{\partial x_2})][\frac{\partial h}{\partial t}+u_1\frac{\partial h}{\partial x_1}+u_2\frac{\partial h}{\partial x_2}-u_3]\,d\Omega=0 \tag{4.2}$$

and

$$\int_\Omega \hat{\boldsymbol{\nabla}}\hat{w}_3\cdot k(\hat{\boldsymbol{\nabla}}\hat{\boldsymbol{w}}_3)=0\,d\Omega \tag{4.3}$$

with $\tau_1=[(4*\nu/h^2)^2+(2||\boldsymbol{u}-\boldsymbol{w}||/h)^2]^{-0.5}$, $\tau_2=h/(2||\boldsymbol{u}||)$, h is an element size measure and ν is the physical viscosity, Furthermore, for strong varying flows, an artificial viscosity is used as $\nu_{art}=c_0h^2||Curl\ \boldsymbol{u}||$, with c_0 a tuning parameter. We note at this point that any function involved in the equilibrium equations and in the variational formulation is expressed in terms of the referential coordinates.

5 Solution Algorithms

5.1 Introduction

Using time and space discretizations, the variational problem (4.1-4.3) is equivalent to solving a linear set of algebraic equations of the form:

$$\mathcal{J}(\{\boldsymbol{U}\}^{n,i-1})\{\boldsymbol{\delta U}\}^{n,i}=-\{\mathcal{R}\}(\{\boldsymbol{U}\}^{n,i-1}) \tag{5.1}$$

where $\{\boldsymbol{U}\}^{n,i}$ is the global vector of the degrees of freedom at time step n and iteration i, $\{\mathcal{R}\}$ is the global residual vector and $\mathcal{J}=\frac{\partial\{\mathcal{R}\}}{\partial\{\boldsymbol{U}\}}$ is the Jacobian matrix .

The solution algorithm for the unsteady nonlinear problem (4.1-4.3), based upon the (exact) Newton method coupled with an implicit time discretization, is then summarized as follows:

ALGORITHM **5.1 Newton**

1. *Given an initial solution* $\{\boldsymbol{U}\}^0$ *Do:*
2. *For* $n=1,2,\ldots$ *do:*
3. $\{\boldsymbol{U}\}^{n,0}=\{\boldsymbol{U}\}^{n-1}$,
4. *For* $i=1,\ldots,$ *Do*
5. *Compute* $\mathcal{J}(\{\boldsymbol{U}\}^{n,i-1})$ *and* $\{\mathcal{R}\}(\{\boldsymbol{U}\}^{n,i-1})$,

6. *solve the linear system (5.1)*
7. *update* $\{\boldsymbol{U}\}^{n,i}$, $\{\boldsymbol{U}\}^{n,i} = \{\boldsymbol{U}\}^{n,i-1} + \{\boldsymbol{\delta U}\}^{n,i}$
8. *update the coordinates,*
9. *if the convergence criteria are satisfied goto 11,*
10. *EndDo*
11. *EndDo*

The dominant cost factor in the above algorithm in terms of computational and storage requirements, is no doubt the way in which the linear system (5.1) is solved. The Jacobian matrix $\mathcal{J}$ is sparse and nonsymmetric. The analytical expressions of its components can be derived from the approximate variational problem (4.1-4.3). For large scale systems, iterative methods based on Krylov subspaces, such as the GMRES algorithm, are known to be better suited than direct methods. Next we take up the issue of preconditioning the Jacobian linear system.

5.2 ILUT preconditioning

An important factor of the success of iterative methods is the way in which the system is preconditioned. Typically, preconditioning consists of replacing the original linear system (5.1) by, for example, the equivalent system

$$\mathcal{J}M^{-1}M\{\boldsymbol{\delta U}\} = -\{\mathcal{R}\} \tag{5.2}$$

which is then solved by a Krylov subspace method. We note that the matrix $\tilde{\mathcal{J}} = \mathcal{J}M^{-1}$ need not be computed explicitly. To apply $\tilde{\mathcal{J}}$ to an arbitrary vector, we only need to be able to perform a matrix - by vector product with $\mathcal{J}$ and to solve a linear system with M. Therefore one requirement is that solving a system with M should be inexpensive. In the simplest instance of incomplete LU preconditioners, the matrix M is of the form $M = LU$ where L is a sparse lower triangular matrix and U a sparse upper triangular matrix, such that L and U have the same structure as the lower and upper triangular parts of $\mathcal{J}$ respectively. This incomplete factorization referred to as ILU(0), is obtained by performing the standard LU factorization of $\mathcal{J}$ and dropping all fill-in elements that are generated during the process. There are more accurate factorizations denoted by ILU (k) and IC (k) which allow a limited amount of fill-in to take place.

Incomplete factorizations in this class are based on dropping elements depending on their *level of fill*, an integer which is obtained from the pattern of the matrix only. The fact that the level of fill depends only on the structure of $\mathcal{J}$ can cause some difficulties for realistic problems that arise in some applications. A number of alternatives have been developed in the literature. One of the simplest ones consists of taking a sparse direct solution algorithm (or code) and modify it by adding lines of code which will ignore small elements that are introduced during the elimination. This tends to be an expensive approach because of the nature of sparse direct solvers.

Another approach is based on using a threshold strategy combined with what is referred to as the I,K,J variant of Gaussian Elimination. A sketch of the general structure of one such algorithm is described next. Details can be found in Saad[13].

ALGORITHM **5.2 ILUT**
1. *Do i=1, n*
2. *Do k=1,i-1*
3. $\mathcal{J}_{i,k} := \mathcal{J}_{i,k}/\mathcal{J}_{k,k}$
4. *If* $|\mathcal{J}_{i,k}|$ *is not too small then Do* $j = k+1, n$
6. $\mathcal{J}_{i,j} := \mathcal{J}_{i,j} - \mathcal{J}_{i,k} * \mathcal{J}_{k,j}$
5. *EndDo*
6. *EndDo*
7. *Drop small elements in row* $\mathcal{J}_{i,*}$

8. EndDo

The implementation in [13] uses two parameters. The first parameter is a drop tolerance which is used mainly in Line 4, to avoid doing an elimination if the pivot is too small. The second parameter is an integer k which controls the number of entries that are kept in line 7.

By using a large fill-value k and a small drop tolerance ϵ we can get arbitrarily accurate incomplete factorizations. However, the storage and computational cost required to obtain the factorization increases. In general, if several systems must be solved with the same matrix $\mathcal{J}$ then it is cost effective to obtain an accurate factorization as the cost is amortized over several linear systems.

If we wish to use an ILUT type preconditioning, then the issue as to when to recompute the preconditioning is important. Indeed as was seen before, the preprocessing cost to compute the L and U factors of ILUT can be significant. It is therefore important to recompute them as infrequently as possible. The linear systems that arise in Newton's method are not too unrelated to one another. As a result it is sometimes sufficient to compute only one ILUT factorization and use it for all steps of the Newton iteration. This is a form of Secant method, in which the derivative is frozen at that obtained at the first Newton step.

Efficient heuristics here will essentially be based on the cost as measured by the number of iterations required for Newton algorithm to converge. For example, a strategy would be to recompute the ILUT factorization whenever the maximum number of iterations is reached in line 9 of Algorithm 5.1. A rather simple criterion would be to freeze the preconditioner and its factorization for a number of time steps given in advance.

ALGORITHM **5.3 Preconditioned Newton-GMRES**

1. *Select an initial solution* $\{\boldsymbol{U}\}^0$.
2. *For* $n = 1, 2, \ldots,$ *Do:*
3. $\quad \{\boldsymbol{U}\}^{n,0} = \{\boldsymbol{U}\}^{n-1}$,
4. $\quad$ *Compute* $\mathcal{J}$ *and its ILUT factorization if necessary,*
5. $\quad$ *For* $i = 1, \ldots, max - iteration,$ *Do*
6. $\quad\quad$ *Compute* $\tilde{\mathcal{J}}(\{\boldsymbol{U}\}^{n,i-1})$ *and* $\{\tilde{\mathcal{R}}\}(\{\boldsymbol{U}\}^{n,i-1})$,
7. $\quad\quad$ *Run GMRES to solve the system (7.2)*
8. $\quad\quad$ *update* $\{\boldsymbol{U}\}^{n,i}$, $\{\boldsymbol{U}\}^{n,i} = \{\boldsymbol{U}\}^{n,i-1} + \{\boldsymbol{\delta U}\}^{n,i}$
9. $\quad\quad$ *Update the coordinates,*
10. $\quad\quad$ *If convergence criteria are satisfied goto 12*
11. $\quad$ *EndDo*
12. *EndDo*

An important feature of Krylov subspace methods is that they only require the action of the Jacobian matrix $\mathcal{J}^{n,i-1}$ times a vector $\{v\}$. As this action is repeated several times in the GMRES algorithm, $\mathcal{J}^{n,i-1}$ times $\{v\}$ is never explicitly computed, it can be well approximated by a difference quotient of the form

$$\mathcal{J}(\{\boldsymbol{U}\}^{n,i-1})\{v\} \approx \frac{\{\mathcal{R}\}(\{\boldsymbol{U}\}^{n,i-1} + \sigma\{v\}) - \{\mathcal{R}\}(\{\boldsymbol{U}\}^{n,i-1})}{\sigma}, \tag{5.3}$$

where σ is some small scalar given by : $\sigma = \epsilon(\|\{\boldsymbol{U}\}^{n,i-1}\| + \epsilon)$, with $\epsilon = 10^{-5}$.

6 Parallel implementation

In order to implement a domain decomposition approach we need a number of numerical and non-numerical tools for performing the preprocessing tasks required to decompose a domain and map it into processors, as

well as to set up the various data structures. PSPARSLIB is a portable library of parallel sparse iteratives solvers, whose goal is to address this need.

Assume that we have a convenient partitioning of the graph, as defined by a certain node to processor mapping. A number publically available partitioners, such as METIS [2], can be used for this purpose. Initially, we assume that each subdomain is assigned to a different processor. We need to set up a local data structure in each processor (or subdomain, or subgraph) which will allow us to perform the basic operations such as computing local matrices/vectors, the assembly of interface coefficients, and preconditioning operations. The simplest option is to duplicate the whole matrix in each processor but this is wasteful and it does not obviate the need for communicating the variables across processors. The driving criterion for the decomposition given in the above argument is the 'physical node'. For general sparse matrices under consideration, this will be translated by a restriction to map rows (equations) as well as associated unknowns to the same processor: If row number i is mapped into processor p then so is the unknown i. i.e., the matrix is distributed row-wise across the processors according the distribution of the variables. Also, if there is an obvious blocking of the unknowns then it should be exploited. Thus, our mapping algorithms should deal with a reduced adjacency graph corresponding to a physical grid rather than with the original adjacency graph of the matrix. Another important assumption we will make here is that the graph is undirected, i.e., the matrix has a symmetric pattern. Once more this restriction is only made for simplicity and because we wish to exploit symmetric communication across boundaries.

6.1 The boundary information

The first step in setting up the local data-structure prior to executing an iterative algorithm for a distributed sparse matrix is to have each processor determine the set of all other processors with which it must exchange information when performing matrix-vector products. Although these are not necessarily physical neighbors, they hold subdomains that are adjacent to the subdomain that is mapped to them. For simplicity, and by analogy with the name used in the actual codes that are run on each processor, we will refer to the label of a given processor in which a copy of the (same) code is executed on each processor as *myproc*. The set of neighbors of processor *myproc* will be referred to as *myneighbors*.

The information needed to find these neighboring processors is by using the global node-to-processor array. For simplicity we will assume for this description that there is no overlap, i.e., any node j belongs to only one processor, namely processor $map(j)$. The local rows are inspected one by one and for each nonzero a_{ij} with $map(j) \neq myproc$, where *myproc* is the label of the current processor, we add $map(j)$ to the list of neighboring processors if it is not already listed. In the overlapping case, $map(j)$ can actually be a set having more than one element, so the linked list structure for node j must be crossed each time. We store the labels of the neighboring processors in an array $proc(1:nproc)$ where *nproc* is the number of processors found.

In this phase, each processor *myproc* will also determine for each of its neighboring processors Q the list of nodes that are coupled with nodes of that processor. We refer to these nodes are *local interface nodes*. When performing a matrix-by-vector product, neighboring processors must exchange values of their adjacent interface nodes. In order to perform this data exchange operation efficiently, it is important to group these nodes processor by processor. Thus, we list first all those nodes that must be sent to $proc(1)$, followed by those to be sent to $proc(2)$ etc.. Two arrays are used for this purpose, one called ix which lists the nodes as indicated above and a pointer array ipr which points to the beginning of the list for $proc(i)$.

Therefore, the boundary exchange information consists of the following items.

1. `nproc` – The number of all adjacent processors, i.e., processors with which processor `myproc` will be exchanging information.

2. `proc(1:nproc)` – List of the `nproc` adjacent processors;

3. `Ix` – The list of (local) interface nodes, i.e., nodes whose values must be exchanged with neighboring processors. The list is organized processor by processor using a pointer-list data structure.

4. `ipr` – the pointer to the beginning of the list in array `ix` of each of nproc neighboring processors.

This information is extracted by examining the adjacency graph as well as the partitioning. It is performed in each processor independently if the adjacency graph is available in each node.

6.2 Local data structure of a distributed sparse matrix

Once the boundary exchange information is determined, we need to build the local representations of the distributed matrices in each processor, using a suitable data structure. For instance, in order to perform a matrix-by-vector product with a matrix that is distributed in the manner described earlier, we need to multiply the matrix consisting of rows that are local to a given processor by some global vector v. Some components of this vector will be local, and some components must be moved to the current processor for the operation to complete. These external variables correspond to interface points belonging to adjacent subdomains. Let A_{loc} be the local matrix, i.e., the (rectangular) matrix consisting of all the rows that are mapped to *myproc*. We will call B_{loc} the 'diagonal block' of A located in A_{loc}, i.e., the submatrix of A_{loc} whose nonzero elements a_{ij} are such that j is a local variable. Similarly, we will call B_{ext} the 'off-diagonal' block, i.e., the submatrix of A_{loc} whose nonzero elements a_{ij} are such that j is *not* a local variable.

In our implementation, all local variables are relabeled from their absolute labeling (used in the original system) to a local labeling, in the range 1 – *nloc*. When an external variable is referenced, it is immediately known from the 'boundary information' described above to which processor it belongs. The ix array which contains the list of external variables processor by processor uses local labeling.

In addition to the local data structure which describes the boundary information described in Section 6.1, we also need the data structure for the local matrix. The information consists of the following.

- `nloc` – The total number of nodes mapped to processor `mynode`;
- `nbnd` – Pointer to the start of the interface nodes;
- `aloc, jaloc, ialoc` – data structure for the two matrices B_{loc} and B_{ext} stored together as one matrix in a CSR format.

We used the CSR format for storing the local matrix for the purpose of illustration only. In reality the local matrices can be stored in any format that can optimize performance on the processor architecture.

6.3 Implementation of the Preconditioned GMRES Algorithm

The computational requirements of the various conjugate gradient like algorithms that have been developed are essentially identical. Here, we would like to illustrate the implementation of these methods with only one such technique, namely the GMRES algorithm. In the simplest case, if a a block-Jacobi iteration is used as a preconditioner (additive Schwarz), in which the blocks correspond to the different subdomains, then we solve each system associated with a subdomain by an iterative process.

To enhance flexibility, we found it extremely helpful to include an additional feature referred to as a "reverse communication mechanism" whose goal is to avoid data structures. When calling a standard FORTRAN subroutine implementation of an iterative solver, we normally need to pass a list of arguments related to the matrix $\mathcal{J}$ and to the preconditioner. This can be a heavy burden on the programmer since it is nearly impossible to find a data structure that will be suitable for all possible cases. The solution is not to pass the matrices in any form. Whenever a matrix – vector product or a preconditioning operation is needed, we can simply exit the subroutine and have the subroutine caller perform the desired operation.

The calling program should call the iterative routine again, after placing the result of the matrix-vector operation in one of the vector arguments of the subroutine. We need a code parameter to help determine the type of operation that is requested by GMRES. Thus, a typical execution of GMRES routine with reverse communication would be as follows:

```
   icode = 0
 1 continue
   call gmres (n,im,rhs,sol,i,vv,w,wk1, wk2,eps,maxits,iout,icode)
   if (icode .eq. 1) then
   call  precon(n, wk1, wk2)  <--- user's preconditioning operation
   goto 1
   else if (icode .eq. 2) then
   call  matvec (n,wk1, wk2)  <--- user's matrix vector product
   goto 1
   endif
```

The icode parameter in the above program segment, is an indicator of the type of operation needed by the subroutine. If it is set to one then we need to apply a preconditioning operation to the vector $wk1$, put the result in $wk2$ and call GMRES again. If it is equal to two then we need to multiply the vector $wk1$ by the matrix A, then put the result in $wk2$ and call GMRES again. Reverse communication enhances the flexibility of the GMRES routine enormously.

The only other functions we need to implement for having a complete version of GMRES are vector operations, specifically DAXPY's and DDOTS. Because the vectors are all mapped in the same way across processors, the vector updates (DAXPY) are translated into ince the vectors x and y are distributed in the same manner among the processors, this vector operation will be translated into p independent vector updates, requiring no communication. If *nloc* is the number of variables local to a given processor *myproc*, this processor will simply execute the vector loop

```
y(1:nloc) = y(1:nloc) + a * x(1:nloc)
```

along with other active processor. Unlike vector updates, dot products require global communication between processors. More specifically, we need to compute the inner product $t = x^T y$ of two distributed vectors x and y and then make the result t available in each processor. Indeed, this result is needed to perform vector updates in each node in the Krylov acceleration routine. For a large number of processors, this sort of operations may be expensive in terms of communication costs. However, many parallel computer manufacturers have become aware of their importance and are starting to provide hardware and software support for performing *global reduction operations* efficiently. Reduction operations include global sums, global max/min calculations, etc.. A commonly adopted calling convention is to provide one subroutine for all these operations, e.g., 'reduce', and pass the type of operation (add, max, min, multiply, ...) as a parameter. In MPI, a distributed dot-product function can be programmed roughly as follows.

```
      function distdot(nloc, x, incx, y, incy)
      integer nloc
      real*8 x(nloc), y(nloc)
      ....
c
      tloc = DDOT (nloc, x, incx, y, incy)
      call MPI_allreduce(tloc,distdoc,1,MPI_real8,MPI_sum,MPI_COMM_WORLD,ierr)
      ...
      return
```

In the above function ddot performs the usual BLAS-1 dot product operation. The reduce operation is called with 'MPI_sum' as the parameter for the operation-type, and as a result it will sum all the variables 'tloc' from each processor and put the result in the variable *distdot* in each processor.

At this point we can make the following important observation. If reverse communication is used, no matrices are explicitly invoked in the GMRES routine and the only difference between a sequential and a parallel implementation of GMRES lies in the dot product. A standard dot-product (function ddot) is to be replaced by a global one (function distdot). As was shown above the calling sequences associated with the two types of dot products are identical. As a result, the parallel and the sequential versions of GMRES will be essentially identical. This is achieved by (1) hiding the communication involved in the dot product from the user and (2) hiding the data structures related to the matrices thanks to the reverse communication implementation.

7 Experiments

In this section we report on some numerical tests which are carried out to assess the performance of the formulations described earlier. The problem chosen consists of a flow around and in a penstock of a projected dam in Quebec (figures 1-2). Here, we only point out some numerical aspects revealed in this application. Figure 3 shows the unstructured mesh used involving 142, 462 tetrahedra and 28, 528 nodes, the total number of equations generated is 109, 448. The steady solution of this problem is obtained by a time marching procedure. We compare the performance of the sequential finite element code with the parallel one. The parameters used in the computations are: Krylov space dimension of 8, a parameter of fill-in in ILUT algorithm of 5 to 30, 4 Newton iterations per time step, and the Jacobian matrix is recomputed and factored every 10 time steps. The code has been run on shared memory machines (a 165 Mhz Sun-Enterprise-6000 and a 195 Mhz SGI-Origin-2000). We conducted many tests in order to assess the convergence behavior of the Block-Jacobi preconditioner used. We found that for the problems we solved, the most important parameter which affects the convergence is the fill-in parameter used in ILUT factorization. For instance, figures (3-4) present the convergence history with a fill-in parameter of 15 and 25. However, with a low fill-in parameter of 5, the parallel code could not converge while the sequential one did. Figures (5-6) present the CPU performance obtained for 10 time steps.

8 Conclusion

A finite element method for solving some free surface hydrodynamic problems was presented. In contrast with most codes used in hydraulic simulations, the standard assumption of hydrostatic pressure distribution is not made a priori. This permits to enlarge the field of applications since the Navier-Stokes equations are solved entirely. Robustness of the proposed approach stems from the fact that the fluid and mesh motions are solved implicitly and in a coupled fashion. All nonlinearities are solved accurately using the nonlinear version of the GMRES algorithm. For illustration purposes we have used only a simple Block-Jacobi preconditioner though there are existing alternatives which can be superior, see, e.g., [15]. Block Jacobi or additive Schwarz, behaves quite well thanks to the stabilized Galerkin formulation and to the fill-in used in the ILUT factorization of the consistent Jacobian matrix. Combining these algorithms it is now possible to tackle large scale 3D problems on available commercial parallel platforms.

References

[1] J. Donéa. Arbitrary Lagrangian-Eulerian Finite Element Methods. *Computational Methods for Transient Analysis.*, Vol. 1, pp. 473-516 (1983).

[2] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, Accepted for publication, 1997. Also available on WWW at URL http://www.cs.umn.edu/~karypis.

[3] T.J.R. Hughes, W.K. Liu and T.K. Zimmermann. Lagrangian-Eulerian Finite Element Formulation for Incompressible Viscous Flows. *Computer Methods in Applied Mechanics And Engineering*, 29, 329-349 (1981). University, Ithaca, N.Y., Aug. 7-11 (1978).

[4] A. Soulaimani, M. Fortin, Y. Ouellet and G. Dhatt. Finite Element Simulation of Two- and Three-dimensional Free Surface Flows. *Computer Methods in Applied Mechanics And Engineering* 3, 265-296 (1991).

[5] T.J.R. Hughes, L.P. Franca and G.M. Hulbert. A new finite element formulation for computational fluid dynamics: VIII. The Galerkin-least-squares method for advective-diffusive equations. *Computer Methods in Applied Mechanics and Engineering*,73, 173-189 (1989).

[6] D.N. Arnold, F. Brezzi and M. Fortin. A stable finite element for the Stokes equations. *CALCOLO*, 21, 337-344 (1984).

[7] A. Soulaimani, M. Fortin, Y. Ouellet, G. Dhatt anf F. Bertrand. Simple continuous pressure elements for two- and three- dimensional incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 62, 47-69 (1987).

[8] F. Brezzi, M.O. Bristeau, L.P. Franca, M. Mallet and G. Rogé. A relationship between stabilized finite element methods and the Galerkin method with bubble functions . *Computer Method in Applied Mechanics and Engineering*, 96, 117-129 (1992).

[9] A. Soulaimani and Y. Saad An Arbitrary Lagrangian-Eulerian Finite Element Method for solving three-dimensional free surface flows *in press, Computer Methods in Applied Mechanics and Engineering*

[10] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[11] W.G. Habashi, M, Robichaud, V.N. Nguyen, W.S. Ghali, M. Fortin and J.W.H. Liu . Large-scale computational fluid dynamics by the finite element method. *International Journal for Numerical Methods in Fluids*, vol.18, 1083-1105 (1994).

[12] L.C. Dutto, W.G. Habashi, M.P. Robichaud and M. Fortin. A method or finite element parallel viscous compressible flow calculations. *International Journal for Numerical Methods in Fluids*, vol. 19, 275-294 (1994).

[13] Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[14] Y. Saad and A.V. Malevsky. PSPARSLIB: A Portable Library of Ditributed Memory Sparse Iterative Solvers *Report, Computer Science, University of Minnesota.*

[15] Y. Saad and M. Sosonkina. Distributed Schur complement techniques for general sparse linear systems. Technical Report UMSI 97/159, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1997.