Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXX, págs. 3021-3032 (artículo completo)
Oscar Möller, Javier W. Signorelli, Mario A. Storti (Eds.)
Rosario, Argentina, 1-4 Noviembre 2011

PARALLEL IMPLEMENTATION OF THE PARTICLE FINITE ELEMENT METHOD - SECOND GENERATION

Juan M. Gimenez[a] **and Norberto M. Nigro**[a,b]

[a]*International Center for Computational Methods in Engineering (CIMEC), INTEC-UNL/CONICET, Guemes 3450, Santa Fe, Argentina, http://www.cimec.org.ar*

[b]*Facultad de Ingeniería y Ciencias Hídricas - Universidad Nacional del Litoral. Ciudad Universitaria. Paraje "El Pozo". Santa Fe. Argentina. http://www.fich.unl.edu.ar*

Keywords: PFEM-2, HPC, particles, CFD, parallel computing.

Abstract. Particle Finite Element Method - Second Generation (PFEM-2) is a numerical method written in a Lagrangian formulation that uses both particles and a mesh to solve the physics equations in an uncoupled way. The method uses fractional-steps with explicit and implicit calculations as needed. The difference with the original form, is that in this second generation a temporal integration scheme based on streamlines integration is added while for the spatial discretization a point-colocation method is adopted. PFEM-2 has proved to obtain accurate and stable results, but it is necessary to make an efficient implementation to get a competitive code saving computing time.

Given that the particle based methods are inherently parallelizable, the present work is focused on the analysis, the comparison and the selection of the optimal tools and techniques to be used to solve, in an efficient way, each algorithm's critical stages in each time steps. Those critical stages are the particle trajectory and acceleration computation through streamlines time integration, the remeshing and the equation system solver on the mesh (necessary for implicit calculations).

Finally, results obtained with this implementation are presented, allowing to simulate, with accuracy, robustness and at a reasonable computational cost, large numbers of particles in problems with scalar unknowns, such as scalar transport problems, and vectorials unknows, such as fluid-dynamic problems.

1 INTRODUCTION TO PFEM-2 METHOD

In computational mechanics the need for decreasing execution times is overriding. There are different ways to reduce costs for solve an specific problem (with fixed geometry, fixed physical parameters such as viscosity, external forces and so on): the first possibility would be replace the hardware with another faster trying to take advantage of parallel processors and clusters existing nowdays, but this option depends on the available resources; another way tries to solve the model equations using new numerical approachs. Combining these options the numerical method called Particle Finite Element Method - Second Generation (PFEM-2) was developed and this paper discusses its parallel implementation. PFEM-2 can be used to solve problems with scalar unknows such as scalar transport problems, and vectorial unknows such as fluid-dynamic problems, but this paper focus on the last one.

Navier-Stokes equations describes the behavior of newtonian and incompressible fluids, its formulation is based on local momentum balance and must be coupled with the equation for the local mass balance, giving the equation system that is presented in Equations 1 and 2.

$$\rho\frac{D\mathbf{v}}{Dt} = -\nabla p + \mu(\nabla\mathbf{v}^T + \nabla\mathbf{v}) + \mathbf{f} \qquad (1)$$

$$\nabla\cdot\mathbf{v} = 0 \qquad (2)$$

It should be noted that the left term includes a material derivative instead the typical local derivatives for the local balance ($\frac{D\mathbf{v}}{Dt} = \frac{\partial\mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}$). Lagrangian Formulation is used to simplify some calculations (especially the non-linear convective term) normally used in Eulerian Formulation and specially because its natural adaptation to particles methods.

PFEM-2 is a particle-based method that uses a segregated approach to solve the fluid-dynamic equation system called *fractional-step*, which consists in three generic steps: first a velocity prediction, second a pressure calculation and last a velocity updating.

In PFEM-2, the velocity prediction is calculated using particles with an explicit approach called streamline-integration that allows to use a Courant number greater than one without lossing the stability of the solution. The second step is mesh-based and solves the Poisson equation for the pressure implicitly using Finite Element Method (FEM), and the last step is again particle-based and consists on using the new pressure to update the velocity field at each particle explicitly.

These approaches require a particle cloud with a background mesh to solve each time-step. In PFEM-2, the relation between particles and nodes is taken a 1-1, which requires remeshing the new particle positions after each streamline-integration.

Others mathematical features of the method and the differences with the original PFEM are not presented here and might be obtained from (Idelsohn et al., 2011). The Algorithm 1 presents a simplified description to understand the necessary steps to be implemented (Gimenez, 2011).

Algorithm 1 - Particle Finite Element Method Second Generation

```
1. Particles-Nodes in position $\mathbf{x}_p^n$.
    (a) Evaluate new particles position in $n+1$ and the fractionary velocity following the streamlines:
        $\mathbf{y}_p^{n+1} = \mathbf{y}_p^n + \int_0^{\Delta t} \mathbf{v}^n(\mathbf{y}_p^\alpha)\, d\alpha$.
        $\rho\hat{\mathbf{v}}^{n+1}(\mathbf{y}_p^{n+1}) = \rho\mathbf{v}^n(\mathbf{y}_p^n) + \int_0^{\Delta t}[\nabla\cdot\boldsymbol{\sigma}^n(\mathbf{y}_p^{n+\alpha}) + \mathbf{f}^{n+\alpha}(\mathbf{y}_p^{n+\alpha})]\, d\alpha$.
    (b) Remeshing:
        $M(\mathbf{x}_p^{n+1}) \approx M(\mathbf{y}_p^{n+1})$.
2. Particles-Nodes in position $\mathbf{x}_p^{n+1}$.
    (a) Find the pressure value solving the Poisson equation system using FEM:
        $\rho\nabla\cdot\hat{\mathbf{v}}^{n+1}(\mathbf{y}_p^{n+1}) = \frac{\Delta t}{2}\Delta[\delta p^{n+1}(\mathbf{y}_p^{n+1})]$
    (b) Actualizate the velocity value with the new pressure $\mathbf{v}_p^{n+1}$:
        $\rho\mathbf{v}^{n+1}(\mathbf{x}_p^{n+1}) = \rho\hat{\mathbf{v}}^{n+1}(\mathbf{y}_p^{n+1}) - \frac{\Delta t}{2}(\nabla p^{n+1}(\mathbf{y}_p^{n+1}) - \nabla p^n(\mathbf{y}_p^{n+1}))$
```

2 IMPLEMENTATION DETAILS

Historically conventional numerical programs has been developed using procedural programation paradigm through programming languages such as `Fortran` or `C`. There, simple data structures and reduced logic operation are used. Although in their application scope they have a high efficiency in computational time and can optimize CPU and memory resources. This optimization is problem-dependent: they solve particular problems and every time when some conditions change in the program or in the program extensions it requires not only the knowledge of the aspects to be improved, but also a deep knowledge about all the code. Last decreases the efficiency, the scalability and increase the initial effort to obtain results in this paradigm (Bretones et al., 1995).

Using Object Oriented Programming (OOP), with languages such as `C++`, the information is stored in objects with more complex structure to represent abstractions of the problem and it allows to solve any problem with a succession of high level instructions. The high scalability of this paradigm increases the number of problems that can be solved, preserving the same computational time because it is as efficient as any procedural paradigm (Cary et al., 1997). Then, the language chosen for this implementation is `C++` using a `OOP` approach. More details about `OOP` implementation may be obtained from author's previous work (Gimenez, 2011).

Nowadays, computers that include two or more cores and the recent upcoming of new Graphics Processing Units (GPU) give new calculation possibilities. Sequential codes, that do not have these parallel computing capabilities must be updated making use of parallel programing used intensively in such kind of hardware. Shared memory and distributed memory techniques are the two approachs to run programs in parallel: the first one is used in multi-core CPUs (is necessary that each process shares the same memory) and the second one can be used in a multi-node architecture (one multi-core PC or between various PCs). Finally exists an hybrid approach that use distributed memory techniques between nodes and shared memory techniques inside each multi-core node.

In the present implementation, shared memory with Open Multi-Processing (OpenMP) is chosen, that allows, in an easy way (only including some compiler directives) to transform sequential code into parallel code. However, not all numerical methods or algorithms are plausible to be coded and to be run in parallel, also problems such as the race condition (processes that modify the same data) or the reentrancy (dependency from past values) must be taken carefully.

2.1 Streamline Integration

In streamline integration each particle moves from initial position $\mathbf{x}_p^n$ to final position $\mathbf{x}_p^{n+1}$ through fixed streamlines at time n. For this task some sort of projection should be done. At each element some constant-element values (such as pressure gradient ∇p and tau divergence $\nabla \cdot \boldsymbol{\tau}$) should be transferred to the nodes, using an standard technique similar to gradient recovery. Also, each particle must know the elements in which is contained, calculate the area coordinates to know the acceleration and velocity at that position verifying if this particle has gone through its boundary to a neighbour element (or out of the domain), and update these data.

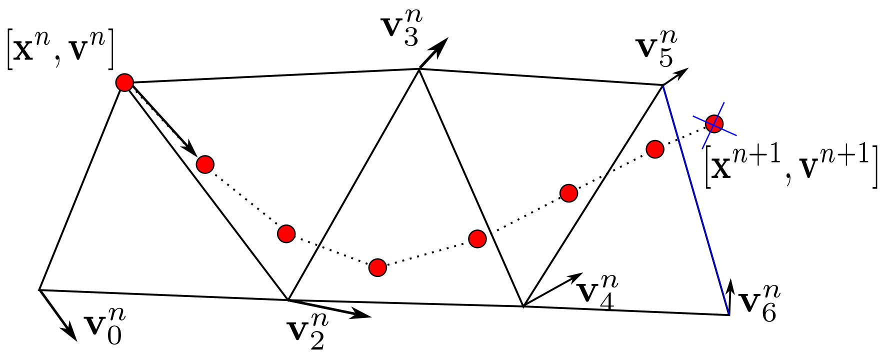


Figure 1: Streamline Integration Example.

Although this step includes many tasks, both particles and elements instances can calculate their data independently from other particles or elements respectively, so it is possible to parallelize it with shared memory techniques without race condition risks.

2.2 Remeshing

Before solving the Poisson step, it is need to regenerate the mesh with the new particle positions. Previously, some particles must be removed (those which are very close each other or making small internal angles that will generate bad-quality elements), and new particles must be seeded in those elements that exceed some minimum reference area; this task is called *particle updating*, and in order to optimize it, pointer-linked containers such as `std::List` must be used, which disables its parallelization, without affecting the computational cost significatively because the cost for operations such as `remove` and `append` are small (about $O(1)$).

After moving the particles and remove or seed them, is necessary to generate the new mesh. Taking only the new particles position, a Delaunay triangulation is used. Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the tesselation; they tend to avoid skinny triangles, which is a necessary feature for all FEM calculation.

Exists many libraries that implements that algorithm, in this implementation `msuite` (Calvo, 2005) is chosen, because is stable, fast, written in `C` and developed by the same investigation group, making easier the support.

The library msuite runs in sequential mode, for this reason a new approach to Delaunay triangulation is needed. A parallel implementation using `pthreads` has been developping, but unfortunately without stable results.

2.3 Poisson Step

Poisson Step is inherently implicit, meaning that for solving an appropiate numerical method should be adopted. PFEM-2 uses FEM to do this task. So, each element matrix must be calculated using an appropiate data-structure that simplifies this job and, further, to generate the equation system, to find the unknow vector using a fast *solver*.

2.3.1 Elemental Assembly

Each instance of class `Element` calculates its own finite element matrix, called elemental matrix K^e, and its own residue vector r^e. For that, fast data-structures must be selected. In this implementation *LTensor* library is chosen, because is almost as fast as `c-arrays` and allows perform operations using an indexation similar to Einstein summation convention (Limache and Fredini, 2010).

Each local assemble can be parallelized with `OpenMP` without risks: each `Element` is able to calculate its own matrix and residue, so the loop for all elements can be done in parallel. In Algorithm 2 the implementation is presented: the outer loop is parallelized and for each instance of `Element` the elemental matrix and residue vector are stored in `LTensor` `Ke` and `Re` objects, and their coefficients are easily calculated using index notation for the operations.

Algorithm 2 - Elemental matrices and residue in parallel using `OpenMP` and `LTensor`

```
int iele;
#pragma omp parallel for default(shared) private(iele)
for(iele=0;iele<this->numel;iele++){
  this->eles[iele].assembleElemental(var,param);
}
...
void Element::assembleElemental(char* var, Parameters& param){
  for(int ipg=0;ipg<pg.npg;ipg++){
    ...
    Ke(iG,kG) += (dt/(2*rho)+tau)*W * dN(iG,jG)*dN(kG,jG);
    Re(iG) += dN(iG,jG)*PI(jG)*tau*W;
    Re(iG) += dN(iG,jG)*gp(jG)*(dt/2)*W;
    Re(iG) -= dN(iG,jG)*gp(jG)*(dt/(2*rho)+tau)*W;
    Re(iG) -= div_u*N(iG)*W;
    ...
  }
}
```

2.3.2 Global Assembly

Assembling the global matrix consists on performing the matricial operation $K^g = \sum_{e=1}^{N} K^e$ and $R^g = \sum_{e=1}^{N} R^e$, where K^g and R^g are the global matrix and the global residual vector respectively. The matrix K^g generated with FEM is initialized primarily with zeros. Due to compact support of shape functions, it is necessary to store their coefficients in an efficient data-structure called *sparse matrix*. Compressed Sparse Row (CSR) format is chosen with a row-major upper triangular storage: the matrix is compressed row-by-row and for symmetric matrices only non-zero elements in the upper triangular half of the matrix are stored. Then the storage is specified by three arrays: values, columns, and rowIndex and this format is used in Pardiso solver (Intel MKL, 2005c).

To parallelize this assembly, each node will generate a row in the matrix that will represent its own equation, and these coefficients can be calculated independently node by node, here structures such as `std::map` are used to facilitate the ordering of the column by index number and the accumulation of the coefficient values. Also, in this parallel task, Dirichlet conditions are imposed in the selected nodes generating a row with only an one in the diagonal and zeros for the rest.

After that each row was calculated in parallel, a non-parallel task is done, where the three global arrays are constructed ordering that data by node (row) number.

2.3.3 Solving equation system

After the global matrix and the global residue vector were assembled, the linear equation system $K\phi = r$ must be solved. For choosing the best option to make this task, some known parallel solvers were compared, specifically their performance solving a laplacian equation system, similar to the equation system found in PFEM-2.

The parallel solvers compared were

- Kyrlov Subspace Library (KSP) from Portable, Extensible Toolkit for Scientific Computation (PETSc) (Balay et al., 2010) that implements the iterative solver Preconditioned Conjugated Gradient (PCG) and is parallelized with distributed memory programming using Message Passing Interface (MPI).

- Intel Math Kernel Library - Parallel Sparse Direct Linear Solver (MKL-PARDISO) (Intel MKL, 2005a) that implements a direct solver with Gaussean Elimination for solving large sparse symmetric and nonsymmetric linear systems of equations on shared memory multiprocessors with OpenMP.

- Intel MKL Iterative Sparse Solver (MKL-IIS) (Intel MKL, 2005b) that allows controls the parallel reversing communication interface to implement the PCG.

- Matlab backslash operator that works in parallel too.

The figure 2 shows the computation times for each solver necessary to find the solution vector ϕ in a problem $K\phi = r$ where the size $N = \sqrt{n_{rows}}$ of the matrix K changes. The matrix K was assembled using the laplacian finite difference operator in 2D present in the Poisson Step in PFEM-2. The result allows to choose the Pardiso solver, because it has demonstrated solving each system in the lowest time and its scalability (when N grows) is better than the others solvers.

2.4 Velocity Updating

This is the simplest step. It consists on take each particle and modify its state with the new pressure data. This step is explicit and it is able to be parallelized easily. Subsets of particles are sent to each processor using `OpenMP`.

In Algorithm 3 the implementation of this step is presented. The only code-line necessary to parallelize it is the `pragma` directive, that indicates to the compiler that the next `for` instruction must be *forked* (the load splitted among each processor) and *joined* (gather results from each processor). This code implements the step 2-b) of the Algorithm 1.

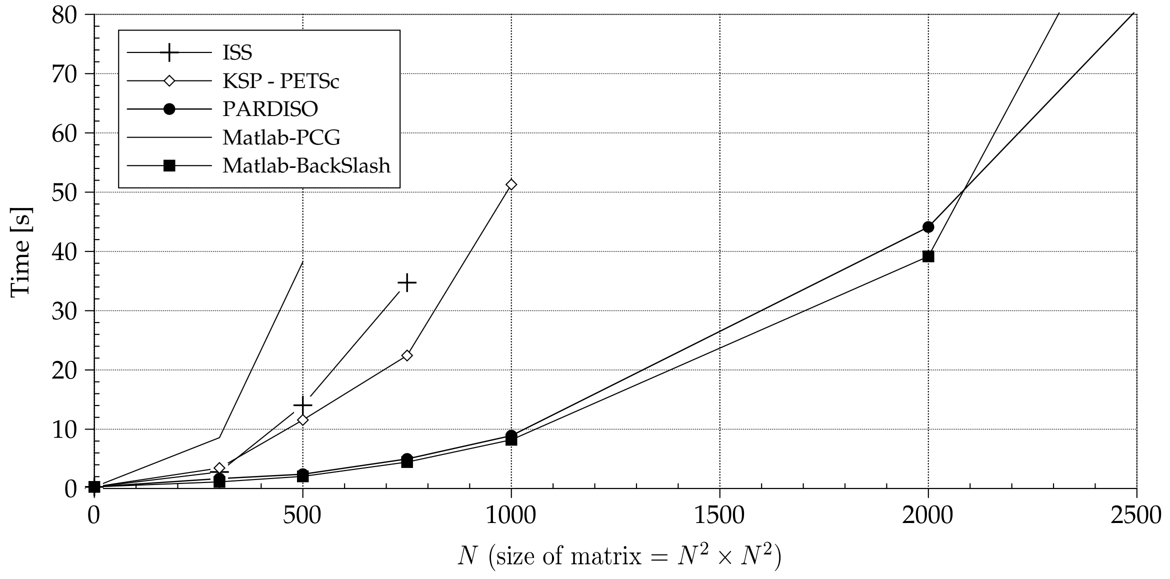


Figure 2: Solvers Comparison in Intel(R) Xeon(R) CPU E5335 - 2.00GHz (8 cores)

Algorithm 3 - Code used to parallelize velocity actualization

```
Node* pNode;
int i;
#pragma omp parallel for default(shared) private(i, nodePtr)
for(i=0;i<this->mesh.numnp;i++){
 pNode = &(this->mesh.nodes[i]);
 pNode->state(vG) -= (param.dt/2)*
       (pNode->gp_new(vG)-pNode->gp_old_on_new(vG))
}
```

2.5 Implementation Overview

Taking into account the previous subsections, the implementation can be summarized in the Algorithm 4. Finally, in this first implementation only shared memory techniques with `OpenMP` are used.

Algorithm 4 - Time Step resolution in PFEM-2

```
int PFEM2::doOneStep()
{
  if(itime%param.nplot==0)
    this->printStatistics();
  param.ttime = param.ttime + param.dt;
  cout << "Time: " << param.ttime << "[s]" << endl;

  //first step: velocity predictor
  this->doubleProjection();
  this->streamlineIntegration();
  this->updateParticles();
  this->remeshing();

  //second step: pressure calculation
  this->poissonStep();

  //third step: velocity actualization
  this->actualizeVelocity();

  return 0;

}
```

3 TESTS

3.1 Lid-Driven Cavity

A first set of comparison is against the Ghia references of the lid-driven cavity (Ghia et al., 1982). The geometry used is presented in Figure 3. A fixed velocity of $v_x = U = 1\frac{m}{s}$ is applied on the top, so as is well known a big vortex is developed within the cavity. Comparisons were done at x and y centerlines, with coordinates $(x, 0.5)$ and $(0.5, y)$ taking the case for $Re = 1000$.

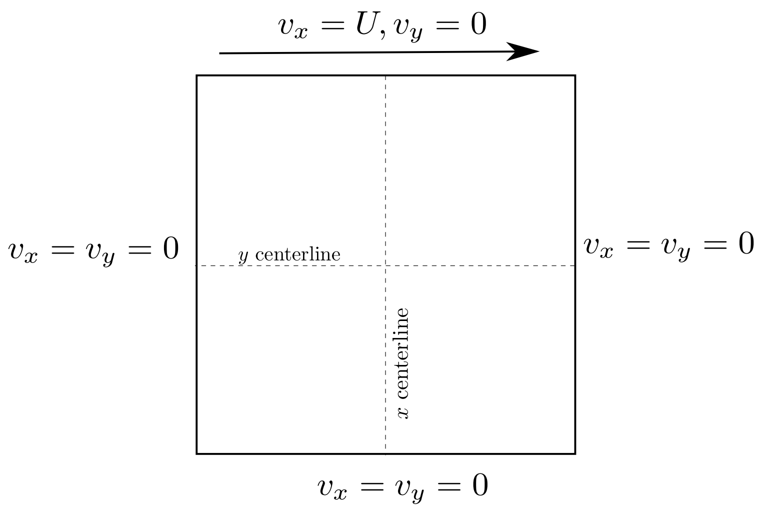
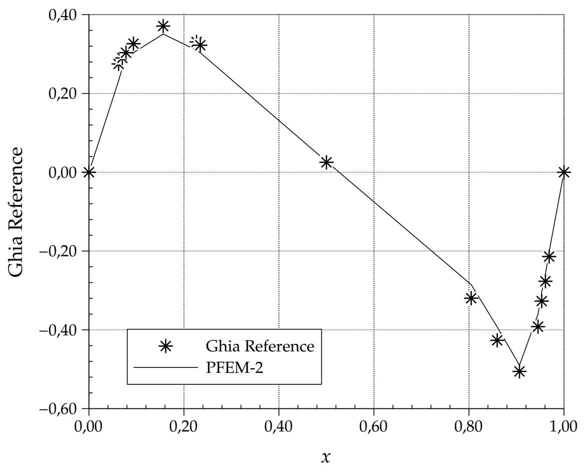
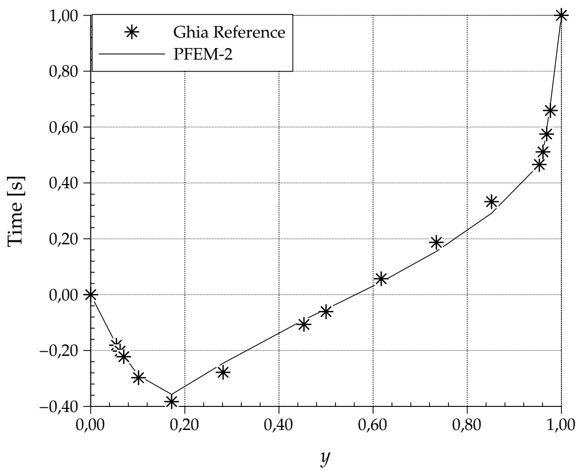


Figure 3: Geometry in Lid-Driven Cavity

Simulations were carried out using 75×75 initial particles, with a maximum Courant number about $2 < Co_{max} < 4$. The Figures 4a and 4b present the results of accuracy comparing the reference data against those obtained with PFEM-2.



(a)



(b)

Figure 4: Comparison vs. Ghia reference for x and y centerlines respectly (using $Re = 1000$).

Another results of interest are those concerning the computation times for each step of the algorithm when one or several processes are used. In this comparison the differents times taken for each step are shown, allowing find the implementation bottlenecks. A lid-driven test was used with a initial seeding of 150×150 particles and the execution is from $t = 0[s]$ to $t = 2.5[s]$ with $\delta t = 0.01[s]$. The processor used was Intel-Core i7 2600K 3.40GHz and the results are presented in Figure 5.

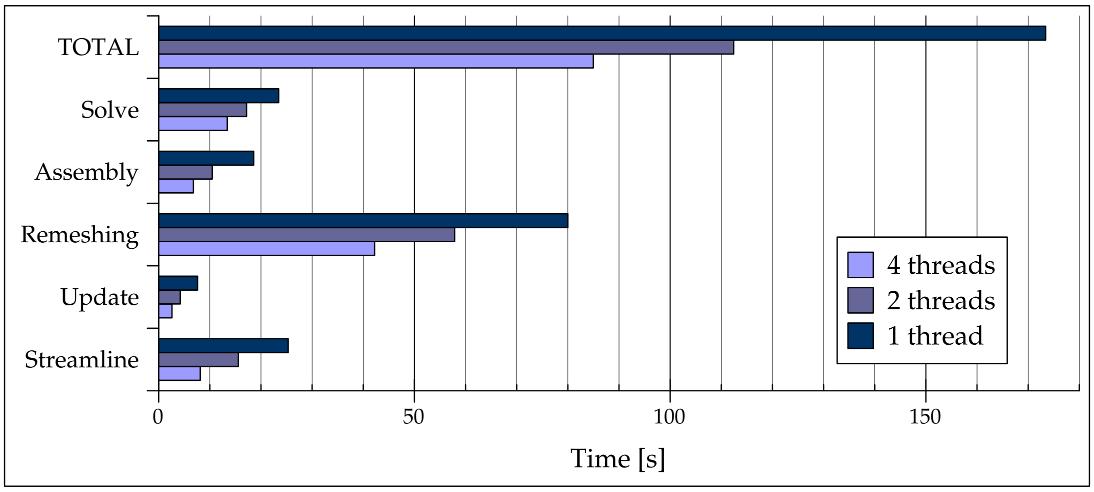


Figure 5: Times in Lid-Driven Cavity for different # of threads

Some conclusions can be extracted from those times results. First, the *remeshing* step is the bottleneck, it spend about 50%, and its reduction of time when more threads are used is because the constructor of the class `Mesh` is parallelized, therefore, a large-constant time for construct the new mesh is need. The *solve* step shows that, although Pardiso is a parallel solver, not all the method can be parallelized, only some algebraical operations between rows, columns of the matrix; for this, the decrease is not n-threads-times, as it happens, for example, in *streamline integration* step, where computation time decrease to about 31% when 4 threads are used, due to the aforementioned parallel capacity of the particle methods.

3.2 Backward Facing Step

The second test that was carried out was the Backward Facing-Step. It was modelled in laminar regime. This flow allows to compare prediction of separated flow like is developed along the step in geometry.

Laminar case was compared to experimental results from Armaly given by (Chiang and Sheu, 1999), taking the case of $Re = 389$. Simulation was carried out in 2D and geometry (Figure 6). Dimensions h, S, $Ld = 55h$ and upstream length $Lu = h$ were selected following guidelines given by (Marquez Damián and Nigro, 2010). The initial seeding was 550×60 particles (x-refinement $\frac{10}{h}$ and y-refinement $\frac{30}{h}$), running with $1.5 <= Co_{max} <= 6$ aproximatelly.

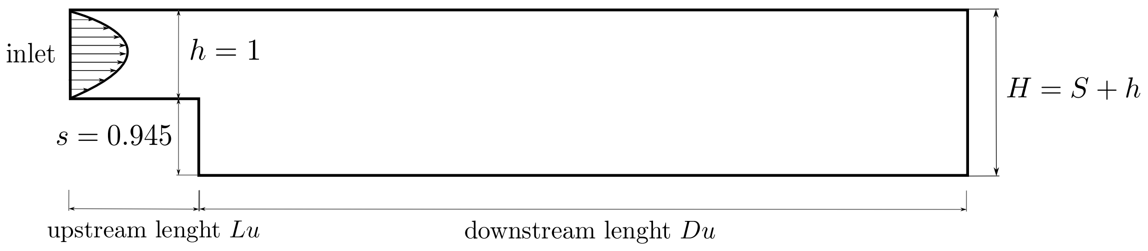


Figure 6: Geometry in Backward Facing Step

The results are presented in Figure 7.

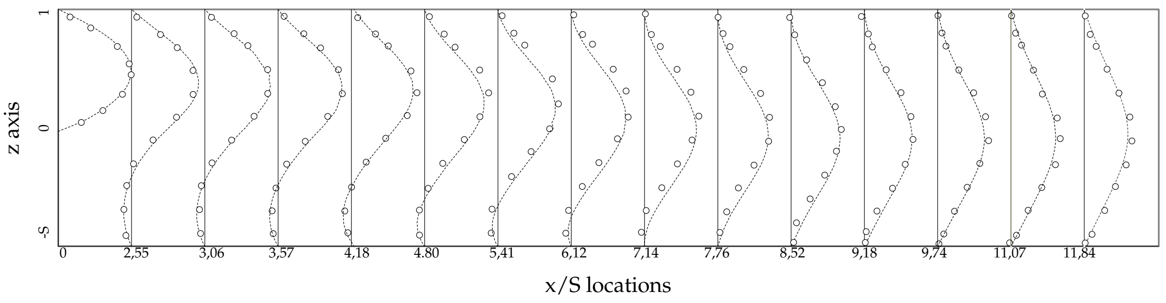


Figure 7: Results in Backward Facing Step compared with Armaly References. Circles represents Armaly reference values and dotted lines show PFEM-2 values.

An interesting result shows the evolution of the Courant numbers throught the simulation. Although the method contains explicit calculations, is able to use $Co > 1$ without affect the stability and the accuraccy. This feature is presented in the Figure 8.

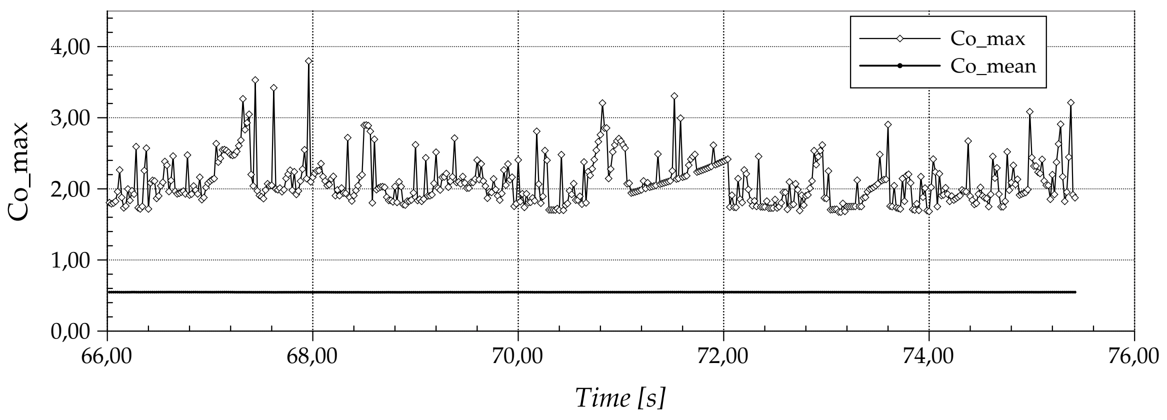


Figure 8: Evolution of the Courant numbers (maximum and mean)

4 CONCLUSIONS

The actual implementation of the method PFEM-2, was developed in `C++` following the paradigm `OOP`, using libraries such as `msuite` to generate a new mesh in each time step, `LTensor` to assembly each elemental matrix and `PARDISO` to solve the equation system; parallelized with shared memory technique provided by `OpenMP` where it was possible to do.

This first implementation reaches the accuracy reported in others PFEM-2 papers (Idelsohn et al., 2011), giving a promizing future in terms of computation times. To complete the shared memory implementation, it remains to complete the parallelization of the Delaunay tesselation in 2D, however this approach (shared memory) is not very scalable and it is needed to move to distributed memory approach, mainly to run this implementation in multi-nodes enviroments such as clusters. An option that recycles the actual code, is to move to hybrid computation (a combination of shared and distributed approaches), but this option is harder and requires larger implementation times than the other.

Finally, PFEM-2 is a new method to solve physics equations, it has accuracy and stability and here it was rewritten to be parallelized in an easy way, reaching in this paper its first parallel implementation.

REFERENCES

Balay S., Brown J., , Buschelman K., Eijkhout V., Gropp W.D., Kaushik D., Knepley M.G., McInnes L.C., Smith B.F., and Zhang H. PETSc users manual. Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

Bretones M., Ferran A., and Huerta A. La programación orientada al objeto aplicada al cálculo por elementos finitos. *Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingenieria*, 11:423–449, 1995.

Calvo N. *Generación de mallas tridimensionales por métodos duales*. Ph.D. thesis, Santa Fe, Argentina, 2005.

Cary J., Shasharina S., and Cummings J. Comparison of c++ and fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 105(1):20–36, 1997.

Chiang T. and Sheu T. A numerical revisit of backward-facing step flow problem. *Physics of Fluids*, 11(4):862–874, 1999.

Ghia U., Ghia K., and Shin C. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of Computational Physics*, 48:387–411, 1982.

Gimenez J.M. Desarrollo e implementación del método pfem-2. *Degree Thesis - Facultad de Ingeniería y Ciencias Hídricas, Universidad Nacional del Litoral*, 2011.

Idelsohn S., Nigro N., Limache A., and Oñate E. Large time-step explicit integration method for solving problems with dominant convection. *Computer Methods in Applied Mechanics and Engineering*, (submitted), 2011.

Intel MKL. Intel math kernel library, linear solvers basics. Document Number: 308659-001, 2005a.

Intel MKL. Iterative sparse solvers based on reverse communication interface (rci iss). http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/index.htm, 2005b.

Intel MKL. Sparse Matrix Storage Formats. 2005c. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/appendices/mkl_appA_SMSF.htm.

Limache A. and Fredini P.R. Ltensor: A high performance tensor library based on index notation. http://code.google.com/p/ltensor/, 2010.

Marquez Damián S. and Nigro N. Comparison of single phase laminar and large eddy simulation (les) solvers using the openfoam suite. *MECOM*, XXIX:3721–3740, 2010.