Efficient parallel numerical solver for the elastohydrodynamic Reynolds–Hertz problem

M. Arenaz [a], R. Doallo [a,*], J. Touriño [a], C. Vázquez [b]

[a] *Department of Electronics and Systems, Universidade da Coruña, Campus de Elviña s/n, E-15071, A Coruña, Spain*
[b] *Department of Mathematics, Universidade da Coruña, Campus de Elviña s/n, E-15071, A Coruña, Spain*

Abstract

This work presents a parallel version of a complex numerical algorithm for solving an elastohydrodynamic piezoviscous lubrication problem studied in tribology. The numerical algorithm combines regula falsi, fixed point techniques, finite elements and duality methods. The execution of the sequential program on a workstation requires significant CPU time and memory resources. Thus, in order to reduce the computational cost, we have applied parallelization techniques to the most costly parts of the original source code. Some blocks of the sequential code were also redesigned for the execution on a multicomputer. In this paper, our parallel version is described in detail, execution times that show its efficiency in terms of speedups are presented, and new numerical results that establish the convergence of the algorithm for higher imposed load values when using finer meshes are depicted. As a whole, this paper tries to illustrate the difficulties involved in parallelizing and optimizing complex numerical algorithms based on finite elements. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Elastohydrodynamic lubrication; High-performance computing; Finite elements; Parallel numerical algorithm

1. Introduction

In many types of contact occurring in industrial devices the forces are transmitted through thin fluid films that prevent damage to the lubricated surfaces. Elastohy-

---

[*] Corresponding author.

E-mail addresses: arenaz@des.fi.udc.es (M. Arenaz), doallo@udc.es (R. Doallo), juan@udc.es (J. Touriño), carlosv@udc.es (C. Vázquez).

drodynamic lubrication is the part of tribology that deals with the lubrication of elastic surfaces. Thus, the elastohydrodynamic mathematical models are concerned with equations governing the elastic behaviour of the solids and the hydrodynamic displacement of the lubricant. A large number of surfaces appearing in industrial applications can be represented by means of an equivalent ball-bearing contact (Fig. 1). In this geometry, the surfaces consist of a sphere and a plane [16]. Much research has been devoted to the mathematical modelling of this physical situation for thin film flows and point contacts. Thus, for example, in [16] a formulation of the elastohydrodynamic point contact problem is posed; in [6] many elastohydrody- namic models can be found; and Reynolds' equation for the lubricant pressure as limit of Stokes' equation in the hydrodynamic case is rigorously justified in [3]. The model for the ball-bearing contact mainly involves Reynolds' equation for the lu- bricant pressure and the Hertzian contact law for the elastic deformation of the sphere. Additional aspects to be taken into account are the external load imposed on the device, the pressure–viscosity relation and the presence of air bubbles (*cavitation*) in the fluid. The inclusion of these new aspects provides very interesting mathe- matical problems, which are posed in terms of highly coupled nonlinear partial differential equations already studied in the literature (see [7,11,12,17], for example).

In order to simulate the real behaviour of industrial devices, mathematical analysis is an important previous step when designing robust numerical algorithms that are principally based on the finite element method. When cavitation is modelled by means of a variational inequality, different numerical simulation approaches to elastohydrodynamic point contacts have been developed: in [24] a finite difference strategy is chosen, in [27] adaptive finite elements based on error equidistribution are applied to a penalty method for the variational inequality and in [22] a multigrid solver is performed.

More recently, in view of performing a more realistic simulation of cavitation phenomena, Elrod–Adams' model has been introduced in [4]. The numerical solu- tion of the elastohydrodynamic coupled problems associated with this new model is the main goal of several works, such as [8,9]. More precisely, in [12], a complex numerical algorithm is proposed in order to solve the elastohydrodynamic point contact model which corresponds to a piezoviscous fluid when Elrod–Adams' model
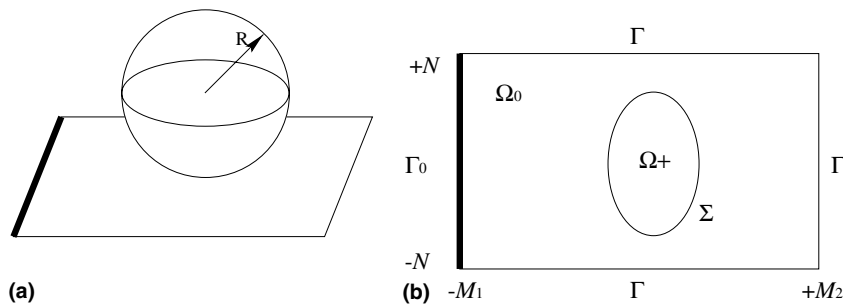

Fig. 1. Ball–bearing contact: (a) ball-bearing geometry; (b) two-dimensional domain of the problem (Ω).

is considered, and a given imposed load is applied to the ball-bearing device. As a result of the application of this algorithm to a real data set, the values of the pressure distribution, the gap profile of the contact and the cavitation region can be computed.

In order to obtain a more accurate approach to the different real magnitudes of the problem, it is interesting to handle finer finite element meshes. This mesh refinement involves a great increase in the storage cost and the execution time of the original sequential code. Therefore, the use of high-performance computing techniques is required in order to reduce the impact of this computational cost problem.

In a previous work [1], the authors developed an algorithm for the Fujitsu VP2400 vector processor. The percentage of vectorization of the source code was approximately 94%, but such high percentage could not be translated in terms of vector unit execution time. The vector processor execution time was not good enough because the most time-consuming computations could not be vectorized as they handle sparse data structures which involve complex subscripted subscripts.

The aim of the present work is not only to reduce execution time but also to test in practice the convergence of the finite element space discretization for several increasing values of the imposed load. For the example finite element meshes, *mesh*4 and *mesh*5 (see Table 2 in Section 3), the total execution time of the vector version was approximately 43.5 and 88.5 h, respectively. The high number of program executions required to tune the physical and numerical parameters to achieve realistic results, justifies the importance of a reduction in execution time. As a result, we concluded that the computational cost of the vector version was still too high. It was precisely this fact that motivated us to develop a parallel version of the algorithm.

In this paper, parallelization techniques are applied to the most costly parts of the original source code. Some blocks of the sequential code were also redesigned for the execution on a distributed-memory multiprocessor. In Section 2, the mathematical model for describing the industrial problem in tribology is outlined. In Section 3, the numerical algorithm for computing an approximate solution of the mathematical model is presented. In Section 4, the implementation of a parallel version of the algorithm is explained in detail. In Section 5, some execution time measurements obtained for the target machine Fujitsu AP3000 multicomputer, as well as speedup diagrams that illustrate the efficiency of our parallel algorithm, are shown. New numerical solutions of the elastohydrodynamic lubrication problem are also presented. Finally, in Section 6, the conclusions of the work are discussed.

2. The model problem

The mathematical model corresponds to the displacement of the fluid between a rigid plane and an elastic and loaded sphere. The equations are posed over a two-dimensional domain (Fig. 1(b)) which represents the mean plane of contact between the two surfaces in the ball-bearing geometry (Fig. 1(a)). In the following formulation, the goal is to determine the pressure p and the saturation θ of the lubricant

(introduced by the Elrod–Adams cavitation model), the gap h between the ball and the plane, and the minimum reference gap h_0. Thus, the set of equations of the model is given by:

$$\frac{\partial}{\partial x}\left(e^{-\beta p}h^3\frac{\partial p}{\partial x}\right) + \frac{\partial}{\partial y}\left(e^{-\beta p}h^3\frac{\partial p}{\partial y}\right) = 12sv\frac{\partial}{\partial x}h, \quad p > 0, \ \theta = 1 \text{ in } \Omega^+, \tag{1}$$

$$\frac{\partial}{\partial x}(\theta h) = 0, \quad p = 0, \ 0 \leqslant \theta \leqslant 1 \text{ in } \Omega_0, \tag{2}$$

$$e^{-\beta p}h^3\frac{\partial p}{\partial \vec{n}} = 12sv(1-\theta)h\cos(\vec{n},\vec{\imath}), \quad p = 0 \text{ on } \Sigma, \tag{3}$$

$$h = h(x,y,p) = h_0 + \frac{x^2+y^2}{2R} + \frac{2}{\pi E}\int_\Omega \frac{p(t,u)}{\sqrt{(x-t)^2+(y-u)^2}}\,\mathrm{d}t\,\mathrm{d}u, \tag{4}$$

$$\theta = \theta_0 \text{ on } \Gamma_0, \tag{5}$$

$$p = 0 \text{ on } \Gamma, \tag{6}$$

$$\omega = \int_\Omega p(x,y)\,\mathrm{d}x\,\mathrm{d}y, \tag{7}$$

where the unknown vector is (p, θ, h, h_0) and the two-dimensional domain Ω, the lubricated region Ω^+, the cavitated region Ω_0, the free boundary Σ, the supply boundary Γ_0 and the boundary at atmospheric pressure Γ appearing in the above equations are, respectively (see Fig. 1(b)),

$$\Omega = (-M_1, M_2) \times (-N, N) \quad \Sigma = \partial\Omega^+ \cap \Omega,$$
$$\Omega^+ = \{(x,y) \in \Omega/p(x,y) > 0\} \quad \Gamma_0 = \{(x,y) \in \partial\Omega/x = -M_1\},$$
$$\Omega_0 = \{(x,y) \in \Omega/p(x,y) = 0\} \quad \Gamma = \partial\Omega \setminus \Gamma_0,$$

where M_1, M_2 and N are positive constants. The input data (see Table 1) are the velocity field $(s, 0)$, the piezoviscosity coefficients v_0 and β, the unit vector $\vec{n}$ normal

Table 1
Input data of the numerical algorithm

Symbol	Physical parameter
$(s, 0)$	Velocity field
v_0	Piezoviscosity coefficient
β	Piezoviscosity coefficient
$\vec{n}$	Normal unit vector
$\vec{\imath}$	X-axis unit vector
E	Young equivalent modulus
R	Sphere radius
ω	Imposed load

to Σ pointing to Ω_0, the unit vector in the x-direction $\vec{\imath}$, the Young equivalent modulus E, the sphere radius R, and the load ω imposed on the device in a normal to the plane direction. Eqs. (1)–(3) model the lubricant pressure behaviour, assuming the Barus pressure–viscosity relation (see [5, Chapter 2] for details about Barus law). Eq. (4) establishes the relation between the gap and the lubricant pressure. Eq. (7) balances the hydrodynamic and the external loads. For further explanation about physical motivation and mathematical modelling [8,12] may be consulted.

3. Numerical algorithm

A complex numerical algorithm for approximating the solution of the previous set of equations without load constraint is proposed in [8]. The inclusion of the nonlocal constraint (Eq. (7)) on the pressure gives rise to an additional external loop that balances the imposed load (see [12] for further details). The resultant numerical algorithm (depicted in Fig. 2) combines regula falsi, fixed point techniques, finite elements and duality methods [10]. The algorithm consists of four nested loops referred to as load, gap, characteristic and multiplier loops, respectively. Each loop performs different computations which we have denoted as *Blocks*. For an initial value of the gap parameter h_0, the load loop is executed (this loop corresponds to the numerical solution of Eqs. (1)–(6)). Next, the gap loop initializes h to the value of a rigid sphere (i.e., for $p = 0$ in Eq. (4)) and performs the first computations (*Blocks* 1–3) in order to solve the hydrodynamic problem (Eqs. (1)–(3)) with the boundary conditions given by Eqs. (5) and (6) in the two innermost loops, namely, the characteristic and multiplier loops. In these loops the pressure p and the saturation θ are computed by means of a transport-diffusion algorithm and a duality method based on maximal monotone operators, and a linear Lagrange finite element discretization of linearized partial differential equations. The characteristic iteration comes from the adaptation of the transport-diffusion technique for a steady state problem (*Block* 4), while the multiplier iteration (*Blocks* 5–8) is closely related to the duality method and to the saturation variable. Once both innermost loops converge in pressure and concentration (*Blocks* 9–10), the gap is updated according to Eq. (4) in *Block* 11. This process is repeated in the gap loop until the convergence in h (*Block* 12) is achieved. Finally, the hydrodynamic load generated by the pressure of the fluid is computed in *Block* 13 by means of numerical integration in Eq. (7). If the computed integral is close to the imposed load (*Block* 14), the algorithm finishes. Otherwise, a new value h_0 is determined by a regula falsi or bisection technique and a new iteration in the load loop of the algorithm is performed.

From a computational point of view, the algorithm gives rise to a great amount of sparse and dense computations: the sparse ones come from the numerical solution of the linear systems associated with the finite element discretization of the problem; dense computations mainly appear in relation with the numerical integration procedures required in Eqs. (4) and (7). In particular, the costly process associated with Eq. (4) requires integration over the whole domain for each finite element node.
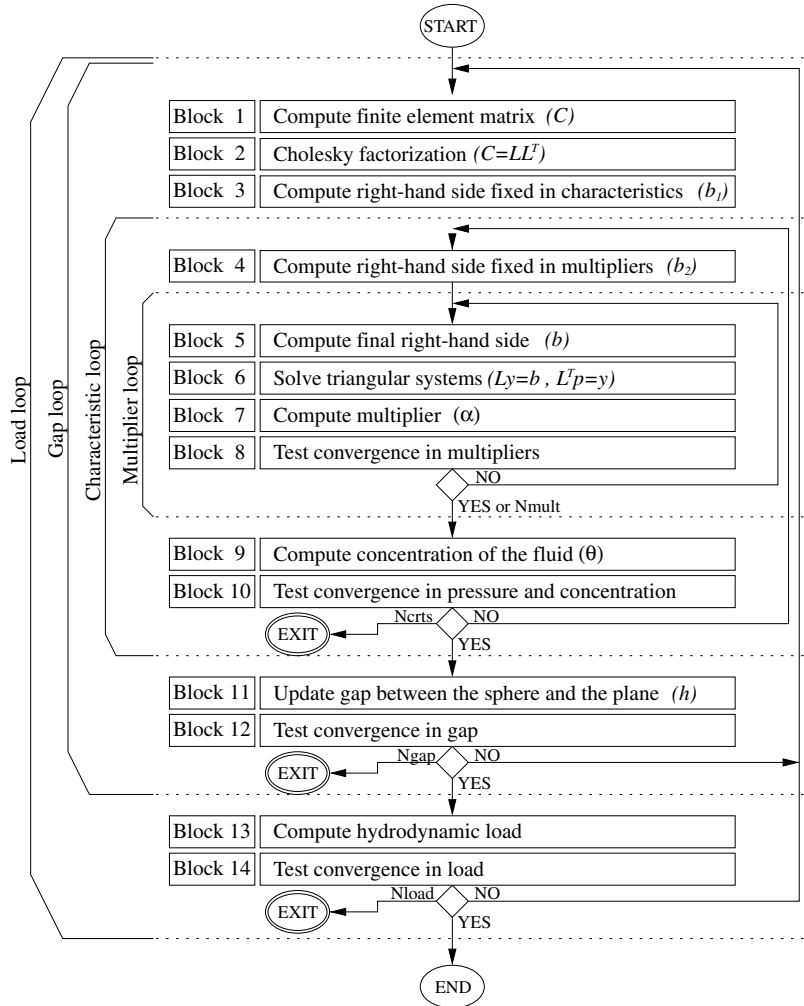
Fig. 2. Flowchart of the original sequential algorithm. The parameters *Nmult*, *Ncrts*, *Ngap* and *Nload* represent the maximum number of iterations for multiplier, characteristic, gap and load loops, respectively. When label *END* is reached, the numerical algorithm converges, while label *EXIT* indicates that it has not converged.

The number of iterations (ni) corresponding to load, gap, characteristic and multiplier loops is shown in Table 2 for an example test. In order to discretize the domain of the problem, four different uniform triangular finite element meshes were considered, namely, *mesh*1, *mesh*3, *mesh*4 and *mesh*5. The following parameters are specified for each mesh: number of finite elements (nfe), number of nodes (n), number of nodes on the supply boundary Γ_0 (nv), number of nonzero entries in the system matrix C (nz_C), and number of nonzero entries in the lower triangular matrix L resulting from the Cholesky factorization of C (nz_L).

Table 2
Number of iterations of each loop for an example test

	Number of iterations (ni)			
	$Mesh1$	$Mesh3$	$Mesh4$	$Mesh5$
Loop	$nfe = 768$	$nfe = 9,600$	$nfe = 38,400$	$nfe = 76,800$
	$n = 429$	$n = 4,961$	$n = 19,521$	$n = 38,961$
	$nv = 13$	$nv = 41$	$nv = 81$	$nv = 81$
	$nz_C = 2,053$	$nz_C = 24,481$	$nz_C = 96,961$	$nz_C = 193,681$
	$nz_L = 4,347$	$nz_L = 84,412$	$nz_L = 415,645$	$nz_L = 872,233$
Load	7	5	4	3
Gap	46	76	57	37
Characteristic	1,964	10,723	14,166	14,596
Multiplier	78,359	1,119,516	1,640,221	1,647,011

4. Parallelization process

The program is composed of four nested loops that are associated with different iterative algorithms. As the results computed in one iteration are used in the following iteration, and the total number of iterations is unknown a priori (because it depends on several convergence stopping criteria), we have centred in parallelizing the blocks shown in Fig. 2. Each block mainly involves the execution of a loop. Therefore, in this work we exploit loop-level parallelism where the iterations of a loop (associated with finite elements or with mesh nodes) are distributed among the processors.

We have experimentally observed that about 99% of the execution time of the sequential algorithm is concentrated on the multiplier loop (Fig. 2, *Blocks* 5–8) and on the updating of the gap (Fig. 2, *Block* 11). The parallelization of the multiplier loop is described in Section 4.1 except for the solution of the linear system which is described in Section 4.3, as it is especially important. The parallelization of the updating of the gap is explained in Section 4.4. The execution time of the characteristic loop (Fig. 2, *Blocks* 4 and 9–10) is negligible, but it has been parallelized in Section 4.2 as it improves the efficiency of the parallel algorithm in a significant manner.

4.1. Multiplier loop

The main tasks performed in this loop consist of the update of the right-hand side (from now on, rhs) of the linear system $Cp = b$ and the solution of this system. From the finite element method viewpoint, the rhs b is constructed by computing an *element rhs* for each finite element and combining all these elements rhs through an *assembly process* [23]. This process (also applied to matrix C in an outer loop) is usually implemented as an irregular reduction, also called histogram reduction [20]. Nevertheless, there exists another implementation that is more suitable for our parallel algorithm. Both alternatives are described and compared in Section 4.1.1,

where they are referred to as *finite element construction method* (FECM) and *node construction method* (NCM), respectively.

The computation of the rhs b is decomposed into three stages:

1. Calculation of the fixed component in the characteristic loop as $b_1 = f_1(h)$ (Fig. 2, *Block* 3); where f_1 is a function of the height h at which the points of the sphere are placed with respect to the plane of reference in the ball-bearing geometry.
2. Computation of b_2, the fixed component of b in the multiplier loop: $b_2 = b_1 + f_2(\theta)$ (Fig. 2, *Block* 4), where f_2 is a function dependent on the saturation of the lubricant fluid θ.
3. Updating in each iteration of the multiplier loop as $b = b_2 + f_3(\alpha)$ (Fig. 2, *Block* 5), f_3 being a function of the multiplier α introduced by the duality method.

The parallelization process corresponding to the computation of the final rhs b is explained in Section 4.1.1. In order to calculate b the value of α has to be obtained. Thus, parallel algorithms for the blocks *Compute multiplier* (Fig. 2, *Block* 7) and *Test convergence in multipliers* (Fig. 2, *Block* 8) are presented in Section 4.1.2. The parallel construction of b_2 is described in Section 4.2. The computation of b_1 was not parallelized as its computational cost is negligible compared with the execution time of the whole program.

4.1.1. Computation of the final right-hand side

First, we introduce some notations to describe the parallelization process for computing b (Fig. 2, *Block* 5). Our goal is to solve an elastohydrodynamic lubrication problem in the two-dimensional domain Ω shown in Fig. 1(b). In order to perform this task, we discretize Ω by using a set Φ of *nfe* triangular finite elements ϕ_k, $k = 1, \ldots, nfe$. Besides, we define:

$$\Phi_{P_j} = \{\phi_k \in \Phi / \phi_k \text{ is assigned to processor } P_j\}, \quad j = 0, \ldots, \rho - 1, \tag{8}$$

where ρ is the number of processors. We also define the whole set $\tilde{\Omega}$ of nodes and the set $\tilde{\Omega}_{\Phi_{P_j}}$ of nodes associated with Φ_{P_j} as follows:

$$\tilde{\Omega} = \bigcup_{\phi_k \in \Phi} \tilde{\Omega}_{\phi_k}, \quad \tilde{\Omega}_{\Phi_{P_j}} = \bigcup_{\phi_k \in \Phi_{P_j}} \tilde{\Omega}_{\phi_k}, \tag{9}$$

where

$$\tilde{\Omega}_{\phi_k} = \{\text{nodes of the finite element } \phi_k\}. \tag{10}$$

Finally, we denote the set of nodes assigned to each processor as:

$$\tilde{\Omega}_{P_j} = \left\{(x_i, y_i) \in \tilde{\Omega} / (x_i, y_i) \text{ is assigned to processor } P_j\right\}, \quad j = 0, \ldots, \rho - 1. \tag{11}$$

Let $b(i)$ be the component associated with the node (x_i, y_i). Vector b can be constructed in two equivalent ways: either by traversing the set of finite elements Φ (FECM) or by traversing the set of nodes of the mesh $\tilde{\Omega}$ (NCM). Both methods compute the value associated with a node (i.e., an element of b) by adding the partial contributions of all finite elements (denoted as $Contrib(\phi_k, (x_i, y_i))$), the difference

being the way in which those contributions are summed up. We will go on to explain why the first method cannot be parallelized efficiently, which has led us to change the sequential algorithm and implement the second method in our parallel version.

Finite Element Construction Method (*FECM*). This technique computes the value of vector b as follows:

$$\forall \phi_k \in \Phi, \quad \forall (x_i, y_i) \in \tilde{\Omega}_{\phi_k}, \quad b(i) = \sum Contrib(\phi_k, (x_i, y_i)). \tag{12}$$

Fig. 3(a) shows a pseudocode in which Eq. (12) is implemented as an irregular reduction. Each iteration k of the outer loop is associated with a finite element ϕ_k, being *nfe* the total number of finite elements (see Table 2). Each iteration v of the inner loop corresponds to a finite element vertex (i.e., an element of the set of nodes $\tilde{\Omega}_{\phi_k}$), being $NV(k)$ the number of vertices of element k (i.e., the cardinality of $\tilde{\Omega}_{\phi_k}$). Vector b is referenced via array f that, given a vertex v of a finite element k, stores the corresponding mesh node number. Array $CONTRIB(k, v)$, which represents the contribution of the vertex v of the finite element k, is directly accessed via the loop indices.

This algorithm can be parallelized by assigning a subset of finite elements (i.e., a subset of outer loop iterations) to each processor. Nevertheless, this method has an important drawback when attempting to parallelize it efficiently. Let P_1 and P_2 be two processors that have been assigned the sets of finite elements Φ_{P_1} and Φ_{P_2}, respectively. The construction of the mesh determines any node to be shared by at least two finite elements, which involves an overlap between the two sets $\tilde{\Omega}_{\Phi_{P_1}}$ and $\tilde{\Omega}_{\Phi_{P_2}}$ ($\tilde{\Omega}_{\Phi_{P_1}} \cap \tilde{\Omega}_{\Phi_{P_2}} \neq \emptyset$). Thus, for all nodes in the boundary $\tilde{\Omega}_{\Phi_{P_1}} \cap \tilde{\Omega}_{\Phi_{P_2}}$, it is necessary to perform some communications between P_1 and P_2 to compute the value of the global sum. Communication overhead is minimized (but never removed) when finite elements are assigned according to a block distribution. This is due to the fact that, as the mesh node numbering algorithm assures that finite element vertex numbers are as close as possible, this distribution supplies the smallest boundaries. As a conclusion, the communication overhead inherent in FECM is too costly for our tribology algorithm to be parallelized efficiently, but this drawback can be overridden by using NCM.

Node Construction Method (*NCM*). Vector b is computed according to the following expression:

$$\forall (x_i, y_i) \in \tilde{\Omega}, \quad \forall \phi_k \in \Phi / (x_i, y_i) \in \tilde{\Omega}_{\phi_k}, \quad b(i) = \sum Contrib(\phi_k, (x_i, y_i)) \tag{13}$$

Fig. 3(b) shows a pseudocode for Eq. (13). Each iteration i of the outer loop is associated with a mesh node (x_i, y_i), being n the total number of mesh nodes (see

```
do k = 1, nfe              do i = 1, n
  do v = 1, NV(k)            do fe = 1, NFE(i)
    i = f(v, k)                k = FE(i, fe)
    b(i) = b(i) + CONTRIB(k, v)  v = f'(k, i)
  enddo                         b(i) = b(i) + CONTRIB(k, v)
enddo                         enddo
          (a)              enddo        (b)
```

Fig. 3. Pseudocodes of the two different approaches for performing the assembly process: (a) finite element construction method; (b) node construction.

Table 2). Each iteration fe of the inner loop corresponds to a finite element that makes a partial contribution to the mesh node number i (i.e., an element of the set of nodes $\tilde{\Omega}_{\phi_k}$), being NFE(i) the number of finite elements for a given mesh node i. Note that vector b is now directly referenced via the outer loop index i. Nevertheless, array *CONTRIB* is accessed by means of array *FE*, which stores the finite elements associated with the mesh node i, and array f', which contains the local vertex number in the finite element k for the global node i.

In this case, it is the set of nodes $\tilde{\Omega}$ (i.e., outer loop iterations) that is distributed. This approach removes any dependence among the computations assigned to different processors. In FECM, dependences arise because array b is accessed indirectly. In NCM, the irregular reduction is removed and the array b is referenced directly via a loop index. Note that some vectors, different from b, are now accessed indirectly. These vectors are computed in our tribology algorithm either once at the beginning of the program or, at most, at the beginning of load or gap loops. Consequently, communication overhead is moved from the most costly to the less costly parts of the global algorithm. This property makes the efficiency of the parallelization process independent of the mesh node distribution; thus, the most appropriate distribution scheme can be chosen for the other stages of the algorithm. Specifically, we have used the block distribution described in Section 4.1.2.

4.1.2. Computation of the multiplier

The duality method introduces a vector of multipliers, α, in the algorithm (Fig. 2, *Block* 7). In each iteration of the multiplier loop, α is computed for each node of the mesh as follows:

$$
\begin{aligned}
&\forall (x_i, y_i) \in \tilde{\Omega} \\
&\qquad \text{IF } (x_i, y_i) \in \Gamma_r \text{ THEN} \\
&\qquad\qquad \alpha^{(r)}(i) = g_1\big(\alpha^{(r-1)}(left(i)), \quad h(left(i))\big) \\
&\qquad \text{ELSE} \\
&\qquad\qquad \alpha^{(r)}(i) = g_2\big(\alpha^{(r-1)}(i), \quad p(i)\big) \\
&\qquad \text{ENDIF}
\end{aligned}
\tag{14}
$$

where $\Gamma_r = \{(x_i, y_i) \in \tilde{\Omega} \,/\, x_i = +M_2\}$ represents the set of nodes located at the right-hand boundary (see Fig. 1(b)); $\alpha^{(r)}$ and $\alpha^{(r-1)}$ denote the value of α for iterations r and $(r-1)$ of the multiplier loop; g_1 and g_2 are functions for calculating the multiplier for a given node (x_i, y_i), and *left* is another function for identifying the left-adjacent node of (x_i, y_i). Taking into account that the nodes of the mesh are numbered up and down, left and right, we can formally define *left* as follows:

$$
left(i) = \begin{cases} i, & \text{if } (x_i, y_i) \in \Gamma_0, \\ i - nv, & \text{otherwise}, \end{cases}
\tag{15}
$$

where $nv = Card(\Gamma_r)$ represents the number of nodes that are located at the left-hand boundary of the domain (see Fig. 1(b)).

The pseudocode in Eq. (14) is cross-iteration independent except for the computations corresponding to those nodes belonging to Γ_r. In this case, a drawback arises if the computations corresponding to the node (x_i, y_i) and to its left-adjacent node $(x_{left(i)}, y_{left(i)})$ are assigned to different processors P_i and $P_{left(i)}$, respectively. The problem lies in the fact that P_i computes the correct value of $\alpha(i)$ only if $P_{left(i)}$ has already computed $\alpha(left(i))$. In other words, there is a true dependence between the iterations i and $left(i)$. This problem can be solved if the communications for coping with the dependence are performed, but this solution is inefficient as processor P_i must wait for $P_{left(i)}$ to finish its computations. Nevertheless, this communication overhead can be avoided by implementing a distribution of vectors α and b so that all nodes included in Γ_r and their left-adjacent nodes are assigned to the same processor. This requirement can be expressed as a constraint on the distribution. Thus, we have implemented the following block distribution:

$$\{(x_{n_{inc} * j+1}, y_{n_{inc} * j+1}), \ldots, (x_{n_{inc} * (j+1)}, y_{n_{inc} * (j+1)})\} \text{ for } P_j, \quad j = 0, 1, \ldots, \rho - 2,$$
$$\{(x_{n_{inc} * (\rho-1)+1}, y_{n_{inc} * (\rho-1)+1}), \ldots, (x_n, y_n)\} \text{ for } P_{\rho-1},$$

(16)

where $n_{inc} = \lceil \frac{n}{\rho} \rceil$, subject to:

$$\forall (x_i, y_i) \in \Gamma_r, \quad \exists^\bullet P_j / (x_i, y_i) \in \tilde{\Omega}_{P_j} \text{ and } (x_{left(i)}, y_{left(i)}) \in \tilde{\Omega}_{P_j}.$$

(17)

The stage *Test convergence in multipliers* (Fig. 2, *Block* 8) is associated with the computation of α. At this stage, local convergence tests are performed in parallel and the global result is calculated by a logical AND scalar reduction.

4.2. Characteristic loop

As explained in Section 4.1.2, the fixed rhs b_2 in the multiplier loop (Fig. 2, *Block* 4) is computed by function f_2. This function is decomposed in two steps:

(1) The part of the vector $f_2(\theta)$ due to the characteristic method is first computed. FECM was used in the sequential code, but in our parallel version, as in the computation of b, we have implemented NCM with the distribution described by Eq. (16) subject to Eqs. (17) and (18) (the latter explained below).

(2) The part of the vector $f_2(\theta)$ fixed for a lubrication problem (Neumann condition) is then created by FECM. In this case, only those nodes of the mesh located at the supply boundary Γ_0 are involved (see Fig. 1(b)). As in the computation of α, we have removed the communication overhead by imposing the following constraint on the data distribution:

$$\forall (x_i, y_i) \in \Gamma_0, \quad \exists^\bullet P_j / (x_i, y_i) \in \tilde{\Omega}_{P_j},$$

(18)

so that all nodes in Γ_0 are assigned to the same processor.

The concentration θ of the fluid (Fig. 2, *Block* 9) is obtained according to the following pseudocode:

$$\forall (x_i, y_i) \in \tilde{\Omega}$$

$$\text{IF } (x_i, y_i) \in \Gamma_0 \text{ THEN}$$
$$\theta(i) = \theta_0$$
$$\text{ELSE IF } (x_i, y_i) \in \Gamma_r \text{ THEN}$$
$$\theta(i) = \alpha(i) \tag{19}$$
$$\text{ELSE}$$
$$\theta(i) = \alpha(i) + \mu \, p(i)$$
$$\text{ENDIF}$$

Thus, for each node (x_i, y_i), this computation depends on p, α and θ_0 (the given concentration at Γ_0). The constant μ is a parameter of the algorithm. Once the multiplier loop converges, vectors p and α are distributed among the processors according to Eqs. (16)–(18). As the pseudocode presented in Eq. (19) is cross-iteration independent, its computations are distributed in the same manner to avoid data redistributions.

The stopping test in pressure and concentration (Fig. 2, *Block* 10) is performed according to the following expressions:

$$\text{test}_p = \frac{\sqrt{\sum (p^{(r)} - p^{(r-1)})^2}}{\sqrt{\sum p^{(r)^2}}}, \quad \text{test}_\theta = \frac{\sqrt{\sum (\theta^{(r)} - \theta^{(r-1)})^2}}{\sqrt{\sum \theta^{(r)^2}}}, \tag{20}$$

where test_p and test_θ pick up the values of the pressure and the concentration to determine the convergence. It should be noted that test_p depends on $p^{(r)}$ and $p^{(r-1)}$ and test_θ on $\theta^{(r)}$ and $\theta^{(r-1)}$. These vectors are distributed among the processors (see Eqs. (16)–(18)). Therefore, at this stage, local sum operations are performed in parallel and the global result is calculated by SUM scalar reduction operations.

Although the execution time of the characteristic loop is negligible, its parallelization improves the efficiency of the parallel algorithm (specifically, the multiplier loop) in a significant manner. This improvement can be quantified in terms of the number of scatter or gather operations on vectors of size $n_{inc} = \lceil \frac{n}{\rho} \rceil$ and scalar reduction operations. Let us take for this purpose the example test presented in Table 2 for *mesh*4. The parallelization of *Block* 4 allows us to perform one scatter operation (for b_1) per iteration in the gap loop (i.e., 57 scatters), while the sequential execution would require one scatter (for b_2) per iteration in the characteristic loop (i.e., 14,166 scatters). The parallel execution of *Blocks* 9–10 needs one gather operation (for p) and two SUM reductions per iteration in gap and characteristic loops, respectively (i.e., 57 gathers $+ 2 \times 14,166$ scalar reductions); on the other hand, their sequential execution would require two gathers (for p and α) per iteration in the characteristic loop (i.e., $2 \times 14,166$ gathers, much more costly than the scalar reductions). Fig. 4 summarizes the parallelization process of the sequential algorithm depicted in Fig. 2.

The minimization of communication overhead and the choice of appropriate communication primitives is critical to achieve good performance in our parallel
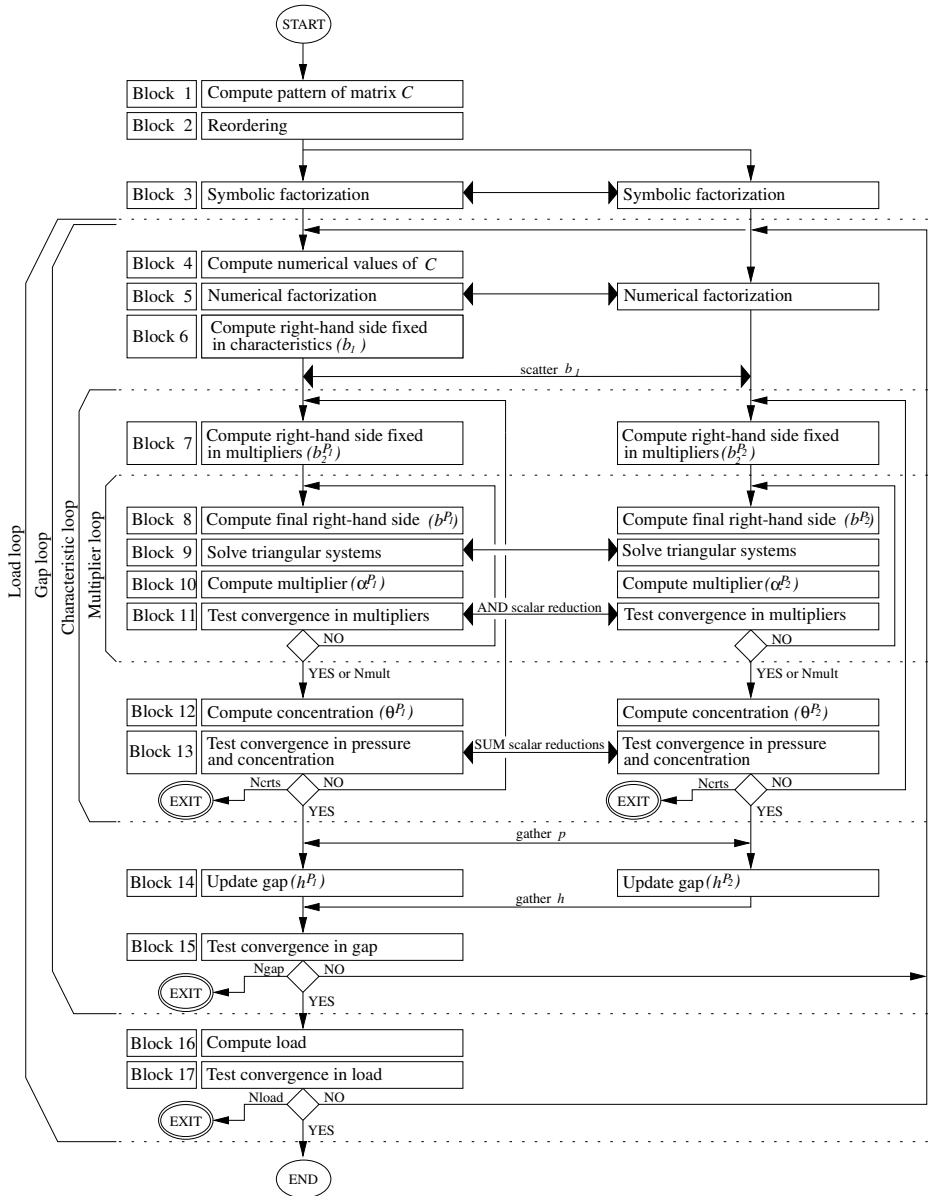
Fig. 4. Flowchart of the parallel algorithm for two processors (P_1 and P_2).

application. In [26], we evaluated and modelled the performance of the basic message-passing primitives (MPI, PVM and APlib libraries) provided by the AP3000 multicomputer, our target machine. Communication overheads were estimated using accurate models. More detailed results about the characterization of MPI collective

primitives were presented in [25] and taken into account in the design of this application.

4.3. Computation of the pressure

The approximation to the pressure of the lubricant fluid is computed by solving the linear system $Cp = b$ per multiplier iteration. The block *Compute finite element matrix C* (Fig. 2, *Block* 1) is devoted to matrix C construction. In the sequential code, FECM is used. As in the computation of b, the calculation of C could be parallelized by implementing NCM. Nevertheless, it is not worthwhile in our tribology algorithm for two reasons: first, the computational cost of this block is negligible and, second, its parallelization does not contribute to improve the performance of the program in a significant manner.

Matrix C is symmetric and positive definite. The order n of the matrix and the small number of nonzero entries (consult nz_C in Table 2), made compressed storages the most suitable in terms of computational and storage cost. We have used a Compressed Row Storage (CRS) format [2, Chapter 4]. Symmetric positive definite systems are usually solved via the *Conjugate Gradient* (iterative method) or via the *Cholesky Factorization* (direct method), the most adequate method being determined by the nature of the specific problem. Sparse Conjugate Gradient method basically needs two vector inner products and a sparse matrix-vector product per iteration. On the other hand, an appropriate preconditioner, as well as a suitable stopping criterion, must be added in order to increase the convergence rate. The sparse conjugate gradient is relatively simple to parallelize [15]. Nevertheless, we have found that, for our problem, its execution time is much higher than the parallel sparse Cholesky Factorization approach described next. It should be noted that the multiplier loop mainly updates b and solves a new system per iteration.

Cholesky Factorization decomposes $C = LL^{\mathrm{T}}$, L being a lower triangular matrix. The sparse Cholesky approach consists of four well-known stages: reordering $C' = \Pi C \Pi^{\mathrm{T}}$, to reduce fill-in during the factorization stage; symbolic factorization, to establish the nonzero pattern of L; numerical factorization, to compute the nonzero entries of L; and solution of triangular systems (forward and backward substitutions), to obtain the solution vector p. The permutation (Π) of vectors b and p due to the reordering phase is included in this last stage. The partition of the solver in several independent stages fits into our tribology problem very well as it makes possible to reduce computations. The pattern of C is constant during the execution of the whole program, so the stages *Reordering* (Fig. 4, *Block* 2) and *Symbolic factorization* (Fig. 4, *Block* 3) are computed only once at the beginning of the algorithm (the number of nonzeros in L is shown in Table 2). As the entries of L change once per iteration in the gap loop, *Numerical factorization* (Fig. 4, *Block* 5) is done at the beginning of this loop. Finally, the *Solve triangular systems* stage (Fig. 4, *Block* 9) is computed once per iteration in the multiplier loop in order to determine the solution vector p.

The *Numerical factorization* is the most time-consuming stage of a sparse Cholesky solver. Nevertheless, in our program it is the block *Solve triangular systems*

that is the most time-consuming. This is because the latter is executed once per iteration in the multiplier loop, while the former is calculated at the beginning of the gap loop. In the example test of Table 2, the number of numerical factorizations and triangular system solvings for *mesh*4 is 57 and 1,640,221, respectively.

From the parallel point of view, the performance of the solver could be increased if multiple right-hand side vectors were computed and, therefore, multiple linear systems were solved simultaneously. Unfortunately, in our program, a new vector *b* is calculated per multiplier loop iteration. Hence, only one linear system can be solved at the same time.

In the parallel program, a serial multilevel graph partitioning algorithm was used to carry out the fill-reducing reordering. Furthermore, this algorithm constructs the elimination tree associated with the sparse matrix *C* and distributes the data among the processors so that the remaining stages of the solver can be performed with minimum data movement. The symbolic factorization step uses an elimination tree guided algorithm to generate the data structures suitable for the numerical factorization. Parallelism is achieved by assigning portions of the elimination tree to the processors according to a load-balanced subtree-to-submesh assignment strategy. The numerical factorization uses an elimination tree guided recursive formulation of the multifrontal algorithm described in [21]. This algorithm is parallelized using the same subtree-to-submesh mapping used in the symbolic factorization. The final step in the process is to solve the triangular systems. Here, standard triangular systems solution algorithms were used for computing vector *p* in parallel. More information about the stages we have just described can be found in [13,14] where a parallel sparse Cholesky solver is proposed.

4.4. Updating of the gap

This stage computes the gap *h* between the sphere and the plane in the ball-bearing geometry (Fig. 2, *Block* 11). For each node, *h* is calculated according to Eq. (4). It should be noted that this algorithm is cross-iteration independent, which allows us to distribute computations in the most suitable manner in order to achieve load balancing. Data redistributions are avoided by distributing the nodes according to Eqs. (16)–(18). Eq. (4) shows that *h* involves numerical integration over the whole domain. As the integral depends on *p*, each processor needs the whole vector *p*. This problem is solved by means of an *allgather* operation on the array *p* after the convergence test in pressure and concentration.

Finally, it is not worth parallelizing the *Test convergence in gap* stage (Fig. 2, *Block* 12). For this reason, vector *h* is gathered at the end of this block.

5. Experimental results on the Fujitsu AP3000

The parallel program was tested on a Fujitsu AP3000 distributed-memory multiprocessor. The AP3000 consists of UltraSPARC-II microprocessors at 300 Mhz interconnected in a two-dimensional torus topology through a high-speed

communication network (AP-net). A detailed description of the AP3000 architecture can be found in [19]. The original Fortran77 algorithm was parallelized using the MPI library in order to make the application portable. The programming paradigm was Single Program Multiple Data (SPMD).

5.1. Performance results

In order to perform our tests, we have used several meshes, namely, *mesh*1, *mesh*3, *mesh*4 and *mesh*5 (see mesh parameters in Table 2). Tables 3 and 4 show experimental results for *mesh*4 and *mesh*5, respectively, when using up to an eight-processor array (the maximum number of processors available on our AP3000 configuration). Experimental results obtained for the coarser meshes, *mesh*1 and *mesh*3, have revealed that it is not worth executing them on a multiprocessor. The information presented in Tables 3 and 4 includes ti, the average execution time per loop iteration; $tt = ti \times ni$, the estimated total execution time of all iterations of the corresponding loop, ni being the number of iterations per loop (consult ni values in Table 2); $\%pt$, the percentage of the estimated total program time pt that tt represents; and sp, the resultant speedup. Note that, for 1, 2 and 4 processors, the total

Table 3

Execution times and speedups for *mesh*4 (ti: time per iteration, tt: total time, pt: program time, $\%pt = (tt/pt) \times 100$, sp: speedup)

*Mesh*4		Number of processors			
		1	2	4	8
Load loop	ti	0.013 s	0.013 s	0.013 s	0.013 s
	tt	0.052 s	0.052 s	0.052 s	0.052 s
	$\%pt$	$0.11E-3$	$0.19E-3$	$0.35E-3$	$0.58E-3$
	sp	1.00	1.00	1.00	1.00
Gap loop	ti	9 m:01 s	4 m:51 s	2 m:29 s	1 m:17 s
	tt	8 h:33 m:57 s	4 h:36 m:27 s	2 h:21 m:33 s	1 h:13 m:09 s
	$\%pt$	6.41	6.04	5.79	4.88
	sp	1.00	1.86	3.63	7.03
Characteristic	ti	0.043 s	0.029 s	0.018 s	0.013 s
loop	tt	10 m:09 s	6 m:51 s	4 m:15 s	3 m:04 s
	$\%pt$	0.13	0.15	0.17	0.20
	sp	1.00	1.48	2.39	3.31
Multiplier loop	ti	0.274 s	0.157 s	0.084 s	0.052 s
	tt	124 h:50 m:20 s	71 h:31 m:55 s	38 h:16 m:19 s	23 h:41 m:31 s
	$\%pt$	93.44	93.79	94.01	94.87
	sp	1.00	1.74	3.26	5.27
Remainder	tt	1 m:23 s	55 s	46 s	41 s
	$\%pt$	0.02	0.02	0.03	0.05
	sp	1.00	1.51	1.80	2.02
Program	pt	133 h:35 m:49 s	76 h:16 m:08 s	40 h:42 m:53 s	24 h:58 m:25 s
summary	sp	1.00	1.75	3.28	5.35

Table 4

Execution times and speedups for *mesh*5 (*ti*: time per iteration, *tt*: total time, *pt*: program time, %*pt* = (*tt*/*pt*) × 100, *sp*: speedup)

*Mesh*5		Number of processors			
		1	2	4	8
Load loop	*ti*	0.029 s	0.029 s	0.029 s	0.029 s
	tt	0.087 s	0.087 s	0.087 s	0.087 s
	%*pt*	0.09E − 4	0.15E − 4	0.28E − 4	0.47E − 4
	sp	1.00	1.00	1.00	1.00
Gap loop	*ti*	37 m:24 s	20 m:03 s	10 m:11 s	5 m:10 s
	tt	23 h:03 m:48 s	12 h:21 m:51 s	6 h:16 m:47 s	3 h:11 m:10 s
	%*pt*	8.23	7.77	7.37	6.26
	sp	1.00	1.86	3.67	7.24
Characteristic loop	*ti*	0.091 s	0.056 s	0.035 s	0.025 s
	tt	22 m:08 s	13 m:37 s	8 m:30 s	6 m:04 s
	%*pt*	0.13	0.14	0.17	0.20
	sp	1.00	1.62	2.60	3.65
Multiplier loop	*ti*	0.561 s	0.320 s	0.172 s	0.104 s
	tt	256 h:39 m:33 s	146 h:24 m:03 s	78 h:41 m:25 s	47 h:34 m:49 s
	%*pt*	91.61	92.05	92.41	93.46
	sp	1.00	1.75	3.26	5.39
Remainder	*tt*	5 m:41 s	3 m:44 s	2 m:46 s	2 m:21 s
	%*pt*	0.03	0.04	0.05	0.08
	sp	1.00	1.52	2.05	2.42
Program summary	*pt*	280 h:11 m:10 s	159 h:03 m:15 s	85 h:09 m:28 s	50 h:54 m:24 s
	sp	1.00	1.76	3.29	5.50

execution times *tt* and *pt* have been estimated on the basis of *ti*. Reliable values of *ti* were obtained by averaging a high enough number of iterations of each loop. Execution times were calculated this way because they exceeded the CPU time limit available in the job queues of our target machine.

The results in the tables are detached for each one of the loops described in Fig. 4. Each loop includes its corresponding computations, excluding the work of their inner loops. Thus, the load loop includes the computations of *Blocks* 16–17; the gap loop contains *Blocks* 4–6 and 14–15; the results of the characteristic loop are for *Blocks* 7 and 12–13; and, finally, the multiplier loop includes *Blocks* 8–11. The *Remainder* row represents additional computations that are performed before entering and after exiting the load loop for initialization and output purposes, respectively. As explained in Section 4.3, stages *Compute pattern of matrix C*, *Reordering* and *Symbolic factorization* (*Blocks* 1–3) are also included here.

The parameter %*pt* clearly indicates the critical stages of the algorithm. According to these tables, they are the computations associated with the multiplier loop and, to a lesser degree, with the gap loop. All of these computations represent, jointly, more than 99% of the whole execution time of the program. Note that this percentage is almost constant with regard to the size of the mesh and the number of processors.

On the one hand, the multiplier loop is the most time-consuming part of the algorithm. This is due to the fact that, although the average execution time per iteration ti is very low, the number of iterations ni is much higher than in the other loops. For instance, for the finer meshes *mesh*3, *mesh*4 and *mesh*5, it can be observed in Table 2 that ni of the multiplier loop is approximately two orders of magnitude greater than ni of the characteristic loop. On the other hand, the gap loop (specifically, the *Update gap block*) is the next most costly part. In contrast to the previous case, the execution time per iteration is the highest by far, but the number of iterations is extremely low.

We have also measured how the execution time of the multiplier loop is distributed when using *mesh*4 on up to eight processors. We have checked that the *Solve triangular systems* block is the most time-consuming, since it takes up a percentage in the interval 79–87% (the percentage increases with the number of processors). The *Compute final right-hand side* block takes up between 6% and 13% (the percentage decreases as the number of processors rises). Finally, the remaining functional blocks (*Compute multiplier* and *Test convergence in multipliers*) take up the rest of the time, about 7%.

It should be noted that, for a fixed mesh size, as the number of processors rises, the value of %*pt* for the gap loop diminishes, while %*pt* of the multiplier loop increases. As the speedups demonstrate, this effect can be explained by the fact that the parallelization of the gap loop is more efficient than that of the multiplier loop, which redounds in a faster reduction of the execution time of the gap loop than the one corresponding to the multiplier loop. Also, for a fixed number of processors, the increase in the mesh size has two different effects: the weight of the gap loop on the global execution time is slightly greater, while the weight of the multiplier loop is reduced in the same ratio. We have also checked this for *mesh*1 and *mesh*3. This is because, for each node, the gap is computed as an integral over the whole problem domain. Thus, the mesh refinement increases the computational cost of this stage to a large extent. Nevertheless, in the multiplier loop, computations are distributed by assigning a subset of mesh nodes to each processor.

In Fig. 5, speedups for *mesh*5 are depicted. The most efficient stage is the gap loop. However, as the multiplier loop is by far the most costly part of the algorithm, its speedup determines the whole program speedup. It is not important to obtain good speedups for the characteristic loop because its %*pt* is negligible. As a matter of interest, note that the execution times measured for the parallel program on an eight-processor array are lower than those obtained with the vectorized algorithm described in [1]. The ratio vector-time/parallel-time was approximately 1.75 and 1.73 for *mesh*4 and *mesh*5, respectively. Therefore, the goal of reducing the execution time of the vectorized version was achieved successfully.

Scalability discussion. An algorithm is scalable on a parallel architecture if the efficiency can be kept fixed as the number of processors is increased, provided that the problem size is also increased. We use the isoefficiency metric [18] to characterize the scalability of our algorithm. This metric determines the rate at which the problem size must increase with respect to the number of processors to keep the efficiency fixed.
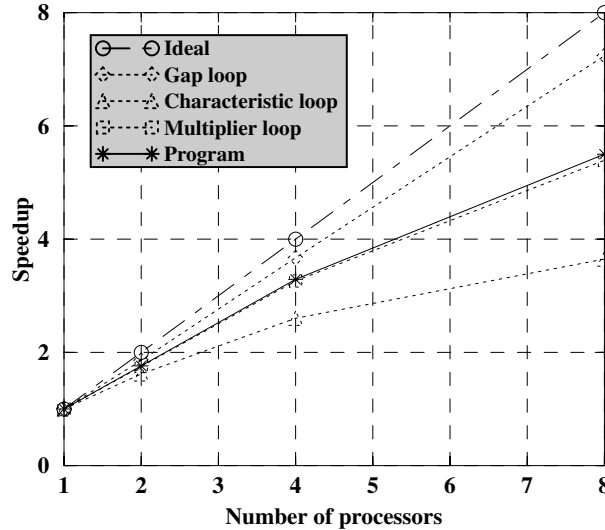
Fig. 5. Speedups for *mesh5*.

As the efficiency of our application is determined by that of the multiplier loop, the isoefficiency of the program can be reduced to that of the multiplier loop. We have also seen that this loop is dominated by the *Solve triangular systems* block. In the other blocks (*Compute final right-hand side*, *Compute multiplier* and *Test convergence in multipliers*) we balanced the computational load according to the distribution given by Eqs. (16)–(18) and we removed all communication overhead except a scalar reduction, whose cost per iteration is negligible ($90 \log_2 \rho - 15$ μs following our model [26]). Therefore, as these blocks represent a small percentage of the loop execution time, they hardly influence the global scalability (this influence would be positive in any case). As a conclusion, the isoefficiency of our program can be approximated by the isoefficiency of the sparse forward and backward triangular solvers discussed in Section 4.3.

In [14], it is shown that this kind of parallel sparse triangular solver is scalable for system matrices C generated in two- and three-dimensional finite difference and finite element methods (in which frame our tribology problem is numerically solved). Moreover, the isoefficiency of that solver is $\Theta(\rho^2)$ and it is asymptotically as scalable as its dense counterpart. The isoefficiency $\Theta(\rho^2)$ means that when the number of processors is increased from ρ to ρ', the problem size must be increased in a factor of $(\rho')^2/\rho^2$ to get the same efficiency as on ρ processors. The problem size in our tribology algorithm is determined by the dimension n of the system matrix, being n the number of mesh nodes. It is easy to derive that we should increase the problem size in a factor ≈ 4 to keep the efficiency fixed when ρ is doubled. Thus, reasonable speedups could be obtained on configurations larger than our eight-processor array using meshes of appropriate size.

5.2. Numerical results

The motivation of this work was not only to reduce computing time in a complex numerical algorithm by means of high performance computing techniques, but also to obtain more accurate approximations to the real magnitudes which constitute the solution of the mathematical model when higher imposed load values are used. In this work, numerical results for the finer finite elements meshes *mesh*4 and *mesh*5 (with 38,400 and 76,800 triangular elements, respectively) with imposed load $\omega = 3$ and $\omega = 5$ are presented. The parallel program allowed us to obtain new results (for
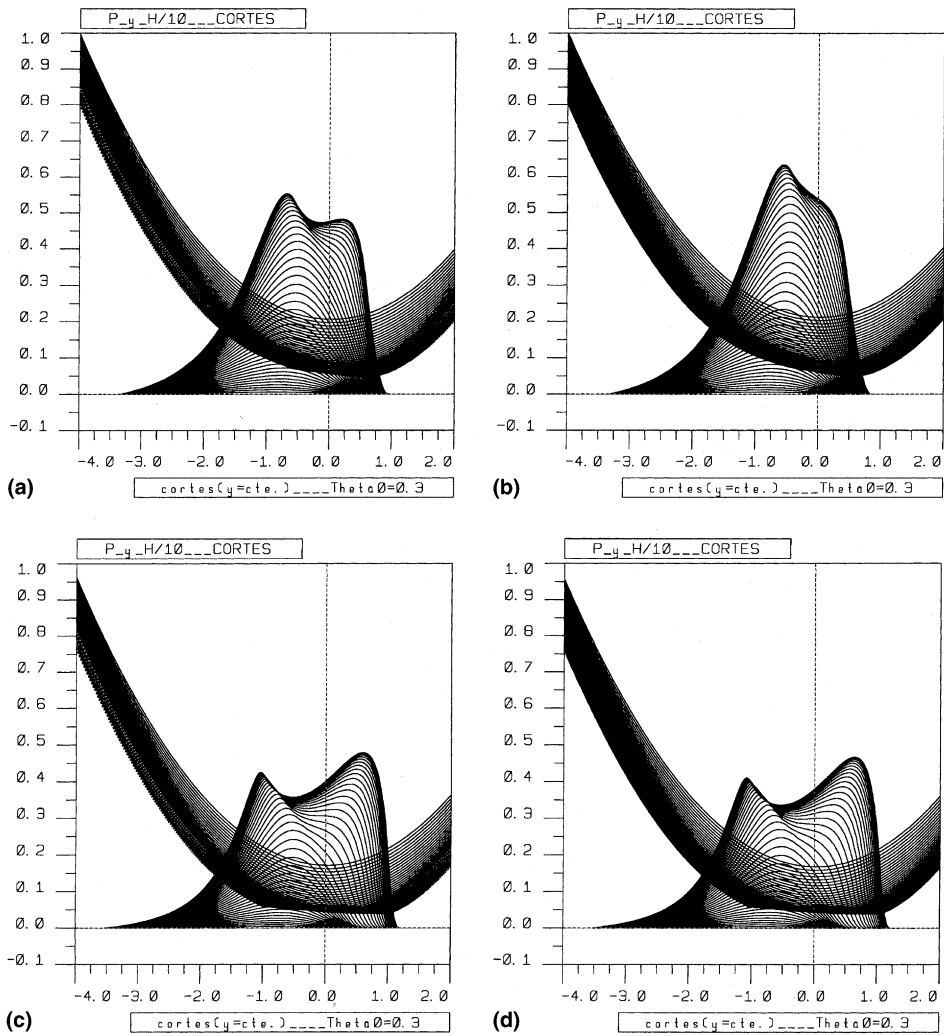
Fig. 6. Pressure and gap approximation profiles.

$\omega = 5$) and it is currently being used to validate the convergence of the numerical algorithm with even higher imposed load values.

In Figs. 6(a)–(d) the pressure and the gap approximation profiles for *mesh*4 and *mesh*5 are presented in an appropriate scale. More precisely, Figs. 6(a) and (b) show the pressure profiles (represented by hill-like curves) and the gap profiles (parabola-like curves with minimum) for *mesh4* and *mesh5*, respectively. The load imposed on the device takes the value $\omega = 3$ in both figures. Analogous results are presented for $\omega = 5$ in Figs. 6(c) and (d). The results illustrate the behaviour of the pressure and the gap when the load imposed on the device is increased from $\omega = 3$ to $\omega = 5$. In particular, for $\omega = 3$ the pressure has only one maximum near the contact point, while for $\omega = 5$ it has two maxima (one at the entry and another at the exit of the device). This is due to the fact that the increment of the load augments the deformations in the sphere, which deadens the increase in the pressure in turn. This phenomenon is typical in elastohydrodynamic problems. A rigid sphere would not become deformed, therefore the pressure would increase everywhere when increasing the applied load.

Although the theoretical convergence of this complex algorithm is an interesting and difficult open problem, the measure of numerical flux supports the flux conservation of the numerical method. More precisely, the numerical flux conservation level is about 93% for *mesh*3, and it is about 96% for *mesh*4 and *mesh*5.

6. Conclusions

In this work, we have described the parallelization of a numerical code for solving an elastohydrodynamic piezoviscous lubrication problem. The most time-consuming parts are multiplier loop, where the pressure of the fluid is calculated, and the computation of the gap (block included in gap loop) between the sphere and the plane in the ball-bearing geometry.

In our tribology algorithm, the computation of the pressure involves solving a sparse linear equation system in which the calculation of the rhs is decomposed into three stages. The FECM was used at these stages in the original sequential code. We showed that the direct parallelization of the irregular reduction associated with FECM has an inherent communication overhead that would significantly decrease the efficiency of our parallel program. An important contribution of our work was to redesign some rhs computation stages and implement the NCM, which moves communication overhead to the less costly parts of the program, i.e., the beginning or the outermost loops (load or gap). This property allowed us to distribute the computation of the rhs so that no data redistributions were needed before and after the execution of the parallel sparse Cholesky solver. A block data distribution subject to two constraints (Eqs. (16)–(18)) was used for this purpose. In practice, when the finer finite element meshes are used, it is clear that a standard block distribution always fulfills the constraints. As we are interested precisely in fine meshes, the data distribution could be simplified.

Taking into account the nature of the problem, the speedups obtained are very reasonable. Moreover, the execution times could be reduced even more by using BLAS routines specifically tuned for the target multicomputer. The results presented in this work were obtained by executing a general purpose BLAS implementation. The scalability of our parallel algorithm is determined by that of the parallel sparse forward and backward triangular system solvers. In terms of the isoefficiency, this solver is asymptotically as scalable as its dense counterpart. It should be noted that, although characteristic loop has a negligible execution time, its parallelization improved significantly the efficiency of the multiplier loop and, hence, the efficiency of the whole parallel program.

From the numerical point of view, the practical convergence validation of a new algorithm was performed. The accuracy of the approximation to the solution was increased with the introduction of finer finite element grids, which confirmed the expected results for the numerical algorithm. Moreover, the convergence of the algorithm when increasing the external imposed load was tested.

Currently, this parallel application is being used at the Department of Mathematics of the University of A Coruña for research purposes. The aim is to establish the practical convergence of the algorithm when varying some relevant input parameters (mainly, the mesh size and the load imposed on the device).

References

[1] M. Arenaz, R. Doallo, G. García, C. Vázquez, High performance computing of an industrial problem in tribology, in: Proc. VECPAR'98, Lecture Notes in Computer Science, vol. 1573, 1999, pp. 652–665 (Best Student Paper Award: First Prize).

[2] R. Barret, M. Berry, T. Chan, J.W. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. Van der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, PA, 1994.

[3] G. Bayada, M. Chambat, The transition between the stokes equation and the reynolds equation: a mathematical proof, Appl. Math. Optim. 14 (1986) 73–93.

[4] G. Bayada, M. Chambat, Sur quelques modélisations de la zone de cavitation en lubrification hydrodynamique, J. Theor. Appl. Mech. 5 (5) (1986) 703–729.

[5] A. Cameron, in: Basic Lubrication Theory, in: Ellis Horwood Series in Engineering Science, Wiley, Chinchester, 1981.

[6] D. Dowson, G.R. Higginson, Elastohydrodynamic Lubrication, Pergamon Press, Oxford, 1977.

[7] J. Durany, G. García, C. Vázquez, A mixed Dirichlet–Neumann problem for a nonlinear reynolds equation in elastohydrodynamic piezoviscous lubrication, Proc. Edinb. Math. Soc. 39 (1996) 151–162.

[8] J. Durany, G. García, C. Vázquez, Numerical computation of free boundary problems in elastohydrodynamic lubrication, Appl. Math. Model. 20 (1996) 104–113.

[9] J. Durany, G. García, C. Vázquez, An elastohydrodynamic coupled problem between a piezoviscous Reynolds equation and a hinged plate model, Math. Model. Num. Anal. 31 (4) (1997) 495–516.

[10] J. Durany, G. García, C. Vázquez, Numerical solution of a Reynolds–Hertz coupled problem with nonlocal constraint, in: Sydow (Ed.), Proceedings of Scientific Computation Modelling and Applied Mathematics, Wissenschaft and Technik Verlag, 1997, pp. 615–620.

[11] J. Durany, C. Vázquez, Mathematical analysis of an elastohydrodynamic lubrication problem with cavitation, Appl. Anal. 53 (1–2) (1994) 135–142.

[12] G. García, Mathematical study of cavitation phenomena in piezoviscous elastohydrodynamic lubrication, Ph.D. Thesis, University of Santiago de Compostela, Spain, 1996 (in Spanish).

[13] A. Gupta, G. Karypis, V. Kumar, Highly scalable parallel algorithms for sparse matrix factorizations, IEEE Trans. Parallel Distribut. Syst. 8 (5) (1997) 502–520.

[14] A. Gupta, V. Kumar, Parallel algorithms for forward and back substitution in direct solution of sparse linear systems, in: Proc. Supercomputing'95, San Diego, 1995.

[15] A. Gupta, V. Kumar, A. Sameh, Performance and scalability of preconditioned conjugate gradient methods on parallel computers, IEEE Trans. Parallel Distribut. Syst. 6 (5) (1995) 455–469.

[16] B.J. Hamrock, D. Dowson, Isothermal elastohydrodynamic lubrication of point contacts. Part 1 – Theoretical formulation, J. Lubricat. Technol. Trans. ASME (1976) 223–229.

[17] B. Hu, A quasivariational inequality arising in elastohydrodynamics, SIAM J. Math. Anal. 21 (1990) 18–36.

[18] K. Hwang, Z. Xu, Scalable Parallel Computing: Technology, Architecture, Programming, McGraw-Hill, New York, 1998.

[19] H. Ishihata, M. Takahashi, H. Sato, Hardware of AP3000 scalar parallel server, Fujitsu Sci. Technical J. 33 (1) (1997) 24–30.

[20] Y. Lin, D. Padua, On the automatic parallelization of sparse and irregular Fortran programs, in: Languages, Compilers and Run-Time Systems for Scalable Computers, LCR'98, Lecture Notes in Computer Science, vol. 1511, 1998, pp. 41–56.

[21] J.W.H. Liu, The multifrontal method for sparse matrix solution: theory and practice, SIAM Rev. 34 (1) (1992) 82–109.

[22] A.A. Lubretcht, The numerical solution of the elastohydrodynamic lubricated line and point contact problems using multigrid techniques, Ph.D. Thesis, Twente University, 1987.

[23] J.N. Reddy, An Introduction to the Finite Element Method, second ed., McGraw-Hill, NewYork, 1993.

[24] J.O. Seabra, Influence de l'ondulation des surfaces sur le comportement des contacts hertziens secs ou lubrifiés, Ph.D. Thesis, INSA of Lyon, 1988.

[25] J. Touriño, R. Doallo, Modelling MPI collective communications on the AP3000 multicomputer, in: Proc. EuroPVM/MPI'99, Lecture Notes in Computer Science, vol. 1697, 1999 pp. 133–140.

[26] J. Touriño, R. Doallo, Performance evaluation and modelling of the Fujitsu AP3000 message-passing libraries, in: Proc. Euro-Par'99, Lecture Notes in Computer Science, vol. 1685, 1999, pp. 183–187.

[27] S.R. Wu, J.T. Oden, A note on applications of adaptive finite elements to elastohydrodynamic lubrication problems, Comm. Appl. Numer. Meth. 3 (1987) 485–494.