ELSEVIER

Advances in Engineering Software 33 (2002) 427–443

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

A parallel finite element program on a Beowulf cluster

V.E. Sonzogni*, A.M. Yommi, N.M. Nigro, M.A. Storti

International Center for Computer Methods in Engineering (CIMEC), INTEC—Universidad Nacional del Litoral—CONICET, Güemes 3450, 3000 Santa Fe, Argentina

Received 14 November 2000; accepted 1 July 2002

Abstract

Some experiences on writing a parallel finite element code on a Beowulf cluster are shown. This cluster is made up of seven Pentium III processors connected by Fast Ethernet. The code was written in C++ making use of MPI as message passing library and parallel extensible toolkit for scientific computations. The code presented here is a general framework where specific applications may be written. In particular CFD applications regarding Laplace equations, Navier–Stokes and shallow water flows have been implemented. The parallel performance of this application code is assessed and several numerical results are presented.

Keywords: Beowulf clusters; Parallel computing; Finite element method; Computational fluid dynamics; Distributed computations; Computational mechanics

1. Introduction

CIMEC is an academic research center where numerical methods are applied to some engineering problems in areas such as CFD, solid materials modeling, dynamics of mechanisms and structures, metal casting, forming processes and biomechanics. Computational requirements for some problems exceed the capabilities of the available equipment (workstations). Attempts to improve our computer power led us to exploit distributed computation on network of workstations (NOWs) and latter on a Beowulf cluster.

In the history of computers there was a time when the computer power was in the hands of some supercomputers, of sophisticated technology and very high prices. The need for growing performances led to parallel computing. On the other hand, the increase in microprocessors technology made the gap between these and supercomputers to become shorter and shorter. The performance/price ratio attracted attention to the use of microprocessors on a parallel environment. In 1994 a machine composed of microprocessors connected by channel bonded Ethernet, gave place to the so-called Beowulf class cluster computers [1].

The Beowulf clusters share some characteristics; the hardware is constructed by the users themselves making use of commodity of the shelf (COTS) components. Usually microprocessors (Intel Pentium II, P6) of ease availability are connected through Fast Ethernet and Fast Ethernet switches. The operating system adopted has been Linux, which is widely available; GNU compilers have also been used [2]. Software tools for message passing such as PVM [3] or MPI [4] are also part of the Beowulf clusters.

There exists a world wide Beowulf community (like the Linux community) which is composed of independent researchers and developers, who share their experiences. This is a very attractive proposal for our university, which has had usually limited resources. In this context, we began the construction of a cluster, which then had 7 nodes, and is continuously growing and we expect it to reach 20 nodes very soon. These are Pentium III, diskless processors connected by Fast Ethernet. One of them has a server disk. We have MPI and PVM as message passing libraries.

In this work we present some results regarding the implementation of a general purpose, multi-physics, FEM code oriented toward CFD applications running on Beowulf clusters and based on the MPI message passing library. We have also used the parallel extensible toolkit for scientific computations (PETSc). This one provides an environment for distributed programming allowing different levels of compromise to the programmer with parallelism. One may let PETSc perform the entire parallel tasks (allocation of data, solving systems, etc.) or may program explicitly the parallel tasks.

We have developed FEM programs to study: Laplace,

* Corresponding author. Tel.: +54-342-4559175; fax: +54-342-4550944.
E-mail address: sonzogni@intec.unl.edu.ar (V.E. Sonzogni).

0965-9978/02/$ - see front matter
PII: S0965-9978(02)00059-5

shallow water and incompressible Navier–Stokes equations. The incompressible Navier–Stokes equations are solved by two different numerical approaches, one is the so-called 'fractional-step' strategy and other is that introduced by Tezduyar called SUPG–PSPG. One of the strongest restrictions for solving such a system comes from the incompressibility condition that inhibits the use of explicit integration algorithms, which are more common for compressible flows and shallow water. In this sense the fractional step and related algorithms treat to decouple this system in a certain number of linearized and scalar substeps. In the first step, called predictor, the pressure gradient is treated explicitly giving rise to a set of three advection diffusion equations for the three components of the velocity, which is stabilized with the well-known SUPG method. The linear system corresponding to this substep is a typical mass matrix plus diffusion term, and may or may not include an advection term depending on whether the advection term is treated explicitly or implicitly. However, even in the case of not including an advection term, the system is non-symmetric due to the SUPG 'biased' weighing. In the following substep a Poisson equation is solved for the pressure–time increment and, finally, in the third step a projection (mass matrix) system is solved. The last two substeps have symmetric positive definite matrices. Even though the computational cost is drastically reduced unfortunately the solution is highly dependent on the time step being critical in its selection. On the other hand the SUPG–PSPG strategy is a formulation based on an equal order interpolation of velocity and pressure variables that circumvents the Brezzi–Babuska condition by the addition of two perturbation functions to the standard Galerkin formulation. In this way it is feasible to avoid not only the checkerboard modes but also the typical oscillations in advection dominated problems. In both the approaches the choice of the solver for each time step is linked to the parallelism strategy.

The aim of this work was to conduct some experiments with parallel implementations of a finite element code on our Beowulf cluster, focusing on the task of solving the linear equations systems. Several linear equations solvers have been tested. They include calls to PETSc tools or to some solvers that were developed specifically for this program. The program has been written in C++. Topics like scalability and load balancing have been considered.

2. Beowulf cluster

Parallel computing was traditionally a matter of very expensive hardware and software. In the late 1980s it was very common to use cluster of workstations (COWs). This allowed people at universities to run jobs in parallel in a NOWs at night, when the user load is low. In most cases the workstations ran some kind of Unix OS, and most of the message passing libraries (PVM, MPI) were developed oriented toward the Unix world.

With the massive arrival of PCs many people attempted to use them for parallel applications. Today Intel processors have the best processing speed/cost ratio. But in order to use the software developed for scientific computations, a version of Unix for PCs was needed. In 1991, the GNU/Linux OS was born. Nowadays, an array of Intel processors running Linux is perhaps the cheapest way to access the parallel computing world with the best speed/-performance ratio (arrays of DEC/Alphas running Linux are also a popular choice). From Ref. [6] a 'Beowulf cluster' is "A cluster of mass-market commodity off-the-shelf (M^2COTS) PCs interconnected by low cost LAN technology running an open source code Unix-like OS and executing parallel applications programmed with and industry standard message passing model and library." The 'Beowulf Project' was originally developed at the Goddard Space Flight Center (GSFC).

Numerical experiments have been performed with a cluster built with seven Intel Pentium III processors, with 128 MB RAM each, (256 MB for the server) linked through a switched Fast Ethernet network (100 Mbit/s, latency = O(100)) supported by a 3COM SuperStack 3300 switch in full duplex mode. The configuration of the cluster is 'disk-less', i.e. the nodes do not have a hard disk. At start, they boot from a floppy diskette, loading the Linux RedHat 5.2 kernel (kernel version 2.0.36) and then, they send a RARP request to the server, who answers telling them their IP (according to their MAC addresses) and their root directory (actually a subdirectory of `/tftpboot` in the server). Then, the nodes mount via NFS their root directories on the server and proceed to the rest of the boot sequence. Some of the performance issues related to the disk-less configuration are the following ones:

- It represents a saving because the nodes do not need a hard disk.
- The cluster is easier to manage, since a change in configuration is performed locally on the node root directories, (subdirectories of `/tftpboot` on the server) and the kernel on the floppies. For instance, a new node can be easily added to the cluster by inserting a copy of the boot floppy and running a script, which takes charge of creating the root directory in `/tftpboot` for the new node.
- Nodes do not have swap memory. This could be considered a drawback but, swap memory degrades performance so much that it is useless, and moreover, memory for Intel based clusters is so cheap that calculations are in most cases CPU bound. Anyway, if some amount of swap memory is desired, it is better to have a local (i.e. per-node) hard disk only for swap and some local data files, and leave the root directory, software and user accounts on a global (i.e. on the server) partition.

- When starting a program (with `mpirun`), the nodes access the server through the network and read the entire code to the node RAM. This can lead to a bottleneck for very large systems but, for the codes we are running, this delay is usually negligible with respect to the average execution time.

3. PETSc-FEM: a general finite element code

3.1. The PETSc-FEM library

PETSc-FEM is not a FEM code but a layer of routines that enables an 'application writer' to develop a typical finite element application code to be used by the 'end user' [7]. However, specific application codes for Navier–Stokes, Euler, shallow water, Laplace and advection diffusion have been developed and included in the distribution in order to be used by end users or, as well, to serve as a basis to application writers for the development of other codes. This involves usually two complimentary parts:

- The routine that describes the main algorithm (i.e. linear/non-linear strategy, steady/unsteady, etc.), where the application writer calls a set of routines that read the mesh, assembles PETSc vectors and matrices, check convergence, etc.
- The element routines compute for each element that how a specific term (residual, stiffness matrix, etc.) is computed.

Solution of the resulting system of equations is done by appropriated calls to the PETSc. Typically, element routines receive a list of elements, and should return a list of residual vectors, matrix elements or anything else *per element*. These are afterwards assembled in the global vector or matrix by the PETSc-FEM layer. Meshes are partitioned by using METIS, and once this is done, PETSc-FEM takes charge of passing the appropriate elements to each processor. Eventually, this list of elements may be internally splitted in 'chunks'. At the element level, calculations may (but it is not required) be done with the help of the Newmat C++ matrix library. Newmat is very interesting since it gives an environment similar to Matlab, but may be inefficient for small matrices computations.

PETSc-FEM is written with an Object Oriented Programming (OOP) philosophy, but keeping in mind the target of efficiency. The basic building object is an 'elemset', i.e. a list of elements that are similar (same number of nodes, same model, same properties). In this way, very large elemsets can be processed efficiently, since the application writer can perform *outside the loop over the elements* a certain amount of common tasks, as decoding the element properties, flags, etc.

Element properties are passed via general hash tables (string → string), so that it is very easy to add new properties to a given element type.

3.2. The PETSc-FEM application codes

As it was mentioned in Section 3.1, the users who want to write an application code, should provide the main algorithm and the element routine. We will refer in this paper to two codes: Navier–Stokes and Laplace. Problem data are given in a file: the dimension of the problem, the degrees of freedom per node, coordinates, type of elements, physical properties, connectivity tables and boundary conditions.

Data input and partition of the mesh, called *Read Mesh* stage, is carried out at the same time by all processors; so, the elapsed time in this stage will be the same in one or p processors. This redundancy could be avoided, using the software PAR-METIS to do the mesh partition in parallel.

In order to reduce the load unbalance, partition of the mesh could be carried out by assigning to each processor a load proportional to their relative speed.

Next, each processor compute the number of non-null values of their own rows of the matrix and allocate the memory needed. This is the stage we call *Matrix Structure*. Then, an SLES context set the iterative method to be applied, the type of preconditioning and other options such as tolerances and parameters.

Later, in the stage *Assemble Matrix*, the elementary matrices are computed and assembled. In the same way, computation of the residual has been identified as the stage *Assemble Residual*.

In the *Navier–Stokes* code, the stage *Matrix Structure* is carried out once, keeping the sparse profile of the matrix. At each time step, only one loop of Newton method is carried out. Linear equations system is solved at each non-linear iteration, using the GMRES iterative method with restart equal 50 and with *Jacobi* preconditioning.

In the *Laplace* code, there is only one loop, and the system of equations is solved with the *Conjugate Gradient* method and with *Jacobi* preconditioning.

In both the programs, the solution stage was named *System Solution*. In all the stages that were defined, a high degree of parallelism is observed, except in the solution stage, where the processors need to exchange more information.

4. Formulation of the Navier–Stokes code

Viscous flow is well represented by Navier–Stokes equations. The incompressible version of this model includes the mass and momentum balances that can be written in the following form

$$\nabla\cdot\mathbf{u} = 0 \qquad \text{in } \Omega\times(0,T) \tag{1}$$

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right)-\nabla\cdot\boldsymbol{\sigma}=\mathbf{0}\qquad \text{in } \Omega\times(0,T), \tag{2}$$

with ρ and $\mathbf{u}$ the density and velocity of the fluid and $\boldsymbol{\sigma}$ the stress tensor, given by

$$\boldsymbol{\sigma}=-p\mathbf{I}+2\mu\varepsilon(\mathbf{u}) \tag{3}$$

$$\varepsilon(\mathbf{u})=\tfrac{1}{2}(\nabla\mathbf{u}+(\nabla\mathbf{u})^t) \tag{4}$$

where p and μ are pressure and dynamic viscosity, $\mathbf{I}$ represents the identity matrix and the deformation tensor. The initial and boundary conditions are

$$\Gamma=\Gamma_g\cup\Gamma_h \tag{5}$$

$$\Gamma_g\cap\Gamma_h=\emptyset \tag{6}$$

$$\mathbf{u}=\mathbf{g}\qquad \text{on } \Gamma_g \tag{7}$$

$$\mathbf{n}\cdot\boldsymbol{\sigma}=\mathbf{h}\qquad \text{on } \Gamma_h \tag{8}$$

4.1. Discretization

The spatial discretization adopted is equal order for pressure and velocity and is stabilized through the addition of two operators. Advection at high Reynolds numbers is stabilized with the well-known SUPG operator, while the PSPG operator proposed by Tezduyar et al. [5] stabilizes the incompressibility condition, which is responsible for the checkerboard pressure modes.

The computational domain Ω is divided into n_{el} finite elements Ω_e, $e=1,\ldots,n_{\text{el}}$, and let $\mathscr{E}$ be the set of these elements, and H^{1h} the finite dimensional space defined by

$$H^{1h}=\{\phi^h|\phi^h\in C^0(\bar{\Omega}),\phi^h|_{\Omega^e}\in P^1,\forall\Omega^e\in\mathscr{E}\}, \tag{9}$$

with P^1 representing polynomials of first order. The functional spaces for weight and interpolation are defined as

$$S_u^h=\{\mathbf{u}^h|\mathbf{u}^h\in(H^{1h})^{n_{\text{sd}}},\mathbf{u}^h\doteq\mathbf{g}^h\text{ on }\Gamma_g\} \tag{10}$$

$$V_u^h=\{\mathbf{w}^h|\mathbf{w}^h\in(H^{1h})^{n_{\text{sd}}},\mathbf{w}^h\doteq\mathbf{0}\text{ on }\Gamma_g\} \tag{11}$$

$$S_p^h=\{q|q\in H^{1h}\} \tag{12}$$

where n_{sd} is the number of spatial dimensions. The SUPG–PSPG is written as follows: find $\mathbf{u}^h\in S_u^h$ and $p^h\in S_p^h$ such that

$$\begin{aligned}&\int_\Omega \mathbf{w}^h\cdot\rho\left(\frac{\partial\mathbf{u}^h}{\partial t}+\mathbf{u}^h\cdot\nabla\mathbf{u}^h\right)+\int_\Omega\varepsilon(\mathbf{w}^h)\\&:\boldsymbol{\sigma}^h\,\mathrm{d}\Omega+\underbrace{\sum_{e=1}^{n_{\text{el}}}\int_\Omega\boldsymbol{\delta}^h\cdot\left[\rho\left(\frac{\partial\mathbf{u}^h}{\partial t}+\mathbf{u}^h\cdot\nabla\mathbf{u}^h\right)-\nabla\cdot\boldsymbol{\sigma}^h\right]}_{\text{(SUPG term)}}\\&+\underbrace{\sum_{e=1}^{n_{\text{el}}}\int_\Omega\boldsymbol{\epsilon}^h\cdot\left[\rho\left(\frac{\partial\mathbf{u}^h}{\partial t}+\mathbf{u}^h\cdot\nabla\mathbf{u}^h\right)-\nabla\cdot\boldsymbol{\sigma}^h\right]}_{\text{(PSPG term)}}+\int_\Omega\\&\times q^h\nabla\cdot\mathbf{u}^h\,\mathrm{d}\Omega\\&=\int_{\Gamma_h}\mathbf{w}^h\cdot\mathbf{h}^h\,\mathrm{d}\Gamma\end{aligned} \tag{13}$$

$$\forall\mathbf{w}^h\in V_u^h,\ \forall q^h\in V_p^h$$

The stabilization parameters are defined as

$$\boldsymbol{\delta}^h=\tau_{\text{SUPG}}(\mathbf{u}^h\cdot\nabla)\mathbf{w}^h, \tag{14}$$

$$\boldsymbol{\varepsilon}^h=\tau_{\text{PSPG}}\frac{1}{\rho}\nabla q^h, \tag{15}$$

$$\tau_{\text{SUPG}}=\frac{h}{2\|\mathbf{u}^h\|}z(Re_u) \tag{16}$$

$$\tau_{\text{PSPG}}=\frac{h^{\#}}{2\|\mathbf{U}^{\text{glob}}\|}z(Re_U^{\#}) \tag{17}$$

Here, Re_u and Re_U are Reynolds numbers based on element properties, namely

$$Re_u=\frac{\|\mathbf{u}^h\|h}{2\nu} \tag{18}$$

$$Re_U=\frac{\|\mathbf{U}^{\text{glob}}\|h^{\#}}{2\nu} \tag{19}$$

where $\mathbf{U}^{\text{glob}}$ is a global characteristic velocity.

The element size h is computed with the expression

$$h=2\left(\sum_{a=1}^{n_{\text{en}}}|\mathbf{s}\cdot\nabla w_{\text{a}}|\right)^{-1} \tag{20}$$

where w_{a} are the functions associated to node a, n_{en} is the number of nodes connected to the element and $\mathbf{s}$ is streamline oriented unit vector. The length $h^{\#}$ is defined as the diameter of a circle (sphere in 3D) equivalent to the element area. Function $z(Re)$ used in Eq. (16) is defined as

$$z(Re)=\begin{cases}Re/3, & 0\le Re<3\\ 1, & 3\le Re\end{cases} \tag{21}$$

Spatial discretization leads to the following equation system

$$\begin{aligned}&(\mathbf{M}+\mathbf{M}_\delta)\mathbf{a}+\mathbf{N}(\mathbf{v})+\mathbf{N}_\delta(\mathbf{v})+(\mathbf{K}+\mathbf{K}_\delta)\mathbf{v}\\&-(\mathbf{G}-\mathbf{G}_\delta)\mathbf{p}=\mathbf{F}+\mathbf{F}_\delta\end{aligned} \tag{22}$$

$$\mathbf{G}^t\mathbf{v} + \mathbf{M}_\varepsilon\mathbf{a} + \mathbf{N}_\varepsilon(\mathbf{v}) + \mathbf{K}_\varepsilon\mathbf{v} + \mathbf{G}_\varepsilon\mathbf{p} = \mathbf{E} + \mathbf{E}_\varepsilon \tag{23}$$

where

$$\mathbf{v} = \text{Array}\{\mathbf{u}^h\} \tag{24}$$

$$\mathbf{a} = \dot{\mathbf{v}} \tag{25}$$

$$\mathbf{p} = \text{Array}\{\mathbf{p}^h\} \tag{26}$$

are the vectors of velocities, accelerations and pressures, whereas the matrices are

$$\mathbf{M} = \int_\Omega \mathbf{w}^h \rho \mathbf{w}^h \, \mathrm{d}\Omega \tag{27}$$

$$\mathbf{M}_\delta = \int_\Omega \boldsymbol{\delta}^h \rho \mathbf{w}^h \, \mathrm{d}\Omega \tag{28}$$

$$\mathbf{M}_\varepsilon = \int_\Omega \boldsymbol{\varepsilon}^h \rho \mathbf{w}^h \, \mathrm{d}\Omega \tag{29}$$

$$\mathbf{K} = \int_\Omega \tfrac{1}{2}(\nabla\mathbf{w}^h + \nabla(\mathbf{w}^h)^t) : \mu(\nabla\mathbf{w}^h + \nabla(\mathbf{w}^h)^t)\mathrm{d}\Omega \tag{30}$$

$$\mathbf{K}_\delta = -\int_\Omega \boldsymbol{\delta}^h \cdot \nabla \cdot (2\mu\varepsilon(\mathbf{w}^h))\mathrm{d}\Omega \tag{31}$$

$$\mathbf{K}_\varepsilon = -\int_\Omega \boldsymbol{\varepsilon}^h \cdot \nabla \cdot (2\mu\varepsilon(\mathbf{w}^h))\mathrm{d}\Omega \tag{32}$$

$$\mathbf{G} = \int_\Omega q^h \nabla \cdot \mathbf{w}^h \, \mathrm{d}\Omega \tag{33}$$

$$\mathbf{G}_\delta = \int_\Omega \boldsymbol{\delta}^h \cdot \nabla q^h \, \mathrm{d}\Omega \tag{34}$$

$$\mathbf{G}_\epsilon = \int_\Omega \varepsilon^h \cdot \nabla q^h \, \mathrm{d}\Omega \tag{35}$$

$$\frac{\partial}{\partial \mathbf{v}}\mathbf{N}(\mathbf{v}) = \int_\Omega \mathbf{w}^h \cdot \rho\mathbf{u}^h \cdot \nabla\mathbf{w}^h \, \mathrm{d}\Omega \tag{36}$$

$$\frac{\partial}{\partial \mathbf{v}}\mathbf{N}_\delta(\mathbf{v}) = \int_\Omega \boldsymbol{\delta}^h \cdot \rho\mathbf{u}^h \cdot \nabla\mathbf{w}^h \, \mathrm{d}\Omega \tag{37}$$

$$\frac{\partial}{\partial \mathbf{v}}\mathbf{N}_\epsilon(\mathbf{v}) = \int_\Omega {}^h \cdot \rho\mathbf{u}^h \cdot \nabla\mathbf{w}^h \, \mathrm{d}\Omega \tag{38}$$

Vector **F** arises from the imposition of Dirichlet and Neumann boundary conditions, whereas vector **E** arises form the Dirichlet conditions only.

5. Assessment of the parallel performance

Performance analysis of PETSc-FEM was based on the study of some tests for Laplace and Navier–Stokes programs.

The aim of the performance analysis was to get tools for predicting and estimating how the PETSc-FEM codes would behave while increasing the number of processors. Therefore, elapsed, CPU and communication times have been measured; speedup and efficiency have been computed; and features of parallel processing, like the reduction in processing time for a given problem or the size of a problem to solve on a given time, have been addressed.

5.1. General features of parallel processing

A general description of the cluster has been done in Section 2. Each node is composed of one processor and one local memory. Nodes are connected through a Fast Ethernet network. This configures a MIMD distributed memory environment on which the natural way of programming is that of message passing.

In this computing environment programming efficient codes rely on factors such as

- the number of processors and the capacity of their local memories;
- the processors inter-connection;
- the ratio of the computation and communication speeds.

Besides, performance of distributed memory architectures depends greatly on the network features:

- Topology: how the nodes are connected;
- Latency: time required to initiate the communication;
- Bandwith: maximum speed for data transfer.

5.2. Time measures

Elapsed and CPU times have been measured by means of the PETSc functions PetscGetTime() and PetscGetCPUTime(), respectively. The last one does not include communication time.

PETSc makes use of the message passing interface (MPI) library for communication. The user has no need to be involved in the management of the message passing. It was not possible to go into the PETSc routines for explicitly measuring the communication times. Nevertheless, profiling options have been added which report the total number of messages sent; the average number of messages per processor; the total and average length of the messages; among other data. These information allowed to estimate the total communication time and the communication time per processor, for a given problem.

In general, the time for a message transfer between two processors may be given by [10]

$$t_{\text{comm}} = \alpha + \beta n \tag{39}$$

α being the latency or start-up time; β the time needed to transmit 1 byte, and n the message length (in bytes). $\theta = 1/\beta$ is the bandwidth. Experience has shown that, for Beowulf clusters based on Fast Ethernet, bandwidth is in the order of 12 MB/s while latency is near 200 μs [6].

From previous measures performed with parallel virtual machine (PVM) as communication tools, we have computed

Table 1
Linpack performance for the Beowulf cluster

Node	100×100	1000×1000
1	68.8	33.2
2	69.8	35.5
3–7	68.7	38.8

for the cluster

$$\alpha = 264\ \mu s \qquad \text{and} \qquad \theta = 10.97\ \text{MB/s} = 92\ \text{Mbit/s}$$

The bandwidth, obtained for a message length of 1 MB, is 8% lower than the theoretical value. The communication time may be estimated as

$$t_{comm} = 2.64 \times 10^{-4} + 0.0911n \tag{40}$$

Another measure of interest is

$$n_{1/2} = \frac{\alpha}{\beta} = \alpha\theta$$

that is the length of a message for which both terms of the t_{comm} equation are equal. In our cluster $n_{1/2} = 3042$ bytes. Messages of length much shorter than $n_{1/2}$ would be dominated by latency, whereas messages of length much longer than $n_{1/2}$ would be dominated by bandwidth [6].

5.3. Computational speed

The performance of a processor is measure in megaflops: millions of floating point operations (flops) per second. Linpack Benchmark [11] was used to estimate the speed of the nodes in the Beowulf cluster. This package performes the LU decomposition with partial pivoting [8], and uses that decomposition to solve a given system of linear equations. The results obtained in double precision are 33.2 mflops in node 1; 35.5 mflops in node 2 and 38.8 mflops in nodes 3–7 for matrix of order 1000×1000. For matrix of order 100×100, 69 mflops are reached in all the nodes (Table 1). As we could see, when the size of the problem grows, the speed of the nodes decreases, because the matrix can no longer be fully contained in the cache, and parts must be reloaded from main memory.

If we know the speed of calculus (C) and the number of operations to carry out (M), the CPU time may be estimated as

$$t_{comp} = \frac{M}{C} \tag{41}$$

From the cluster description, one can observe that differences exist among the speeds of the nodes. To measure these differences, the Laplace program for a problem with 122,850 degree of freedom was implemented. The results obtained, show that the second node is 20% faster than the server (Node 1), and the nodes 3–7, all homogeneous, are 26.8% faster than Node 1.

5.4. Parallel measurement

Since the goal of parallel processing is to reduce the elapsed time, we compare the performance of parallel programs with the calculation of some of the following measures. Let t_1 be the time to execute a given problem with one processor, and t_p the time needed to execute the same problem with p processors. Then the *Speedup* is the relationship among the elapsed times using 1 and p processors:

$$S_p = \frac{t_1}{t_p} \tag{42}$$

This measure is a function of the number of processors, although it also turns out to be a function of the problem size. If we use p processors, we expect that the parallel time will be nearly $1/p$ of that corresponding to only one processor. This yields an upper bound equal p for S_p.

The *Efficiency* is defined as the speedup but relative to the number of processors,

$$E_p = \frac{S_p}{p} = \frac{t_1}{pt_p} \tag{43}$$

In an ideal situation, an efficiency equal 1 would be expected.

Another way to express the efficiency is the following:

$$E_p = \frac{1}{1+\omega} \tag{44}$$

where ω represents the 'generalized' overhead; that is, the communication to computation ratio. The most important sources of parallel overheads are [9]

- *Communications and coordinations*. The parallel execution time t_p with p processors, may be represented in the following manner:

  $$t_p = t_{coor} + t_{comm} + t_{comp} \tag{45}$$

  where $t_{comp} = t_1/p$, t_{coor} is the coordination overhead and t_{comm} the communication overhead. Thus, the speedup and the efficiency may be expressed as follows:

  $$S_p = \frac{1}{\frac{1}{p} + \frac{t_{coor} + t_{comm}}{t_1}} \qquad \text{and}$$

  $$E_p = \frac{1}{1 + \frac{t_{coor} + t_{comm}}{t_{comp}}}$$

  therefore, from Eq. (44) we get

  $$\omega = \frac{t_{coor} + t_{comm}}{t_{comp}}$$

- *Redundancy*. This type of overhead takes place when a parallel algorithm performs the same computations on many processors.

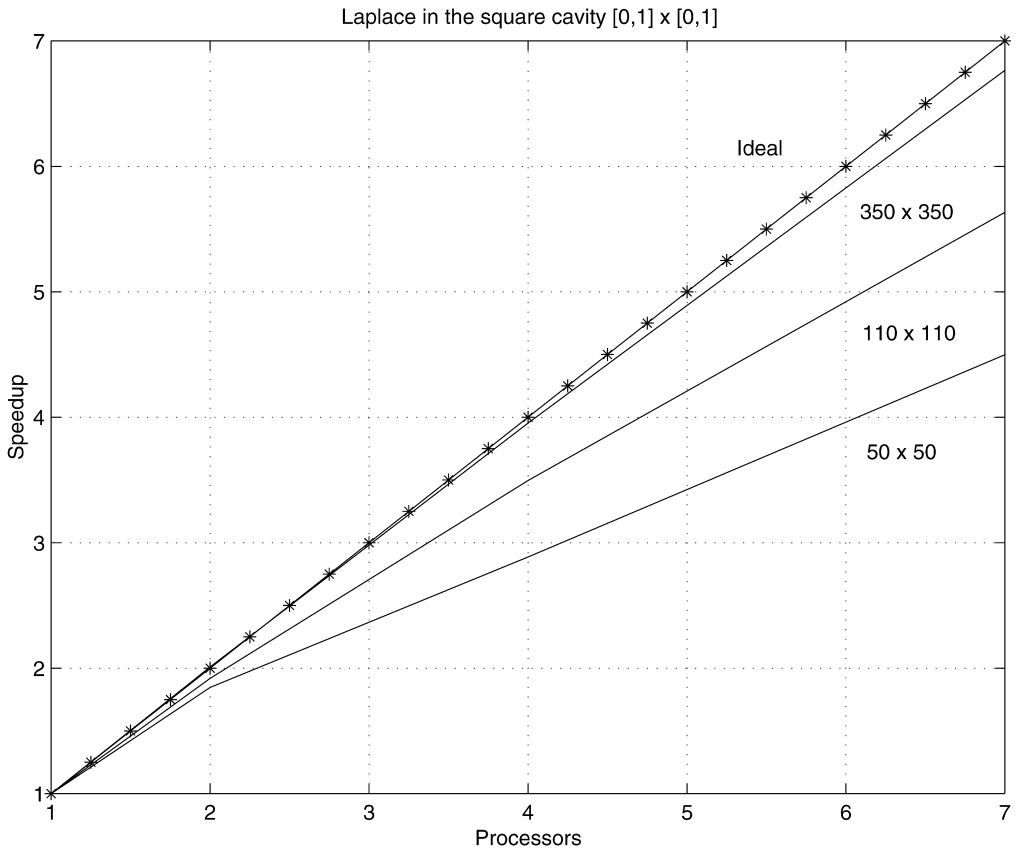

Fig. 1. Speedup on a heterogeneous network for a Laplace problem.

- *Load unbalance*. This overhead measures the extra time spent by the slowest processor to do the assigned tasks relative to the time needed by the other processors. So, the elapsed time is dictated by the slowest processor.
- *Extra work*. They are parallel computations that does not take place in a sequential implementation.

6. Analysis of the Laplace code performance

This program was implemented for different cases corresponding to two-dimensional structured meshes defined in $[0,1]\times[0,1]$. Homogeneous square elements were used.

In each case the number of elements by side N was specified and the solution in only one side of the square was fixed: $u(x,0)=1$. The program finishes when the tolerance arrives to 10^{-6}.

In Laplace, there is only one degree of freedom per node, so that, if there are N elements per side, there are $N+1$ nodes by side and a total of $(N+1)^2$ nodes, $N+1$ nodes being fixed. Therefore, the degrees of freedom are $(N+1)N$. We consider values of N varying from 3 up to 350; that is to say, from 12 up to 122,850 degrees of freedom. All the applications were solved in 1, 2, 4 and 7 processors.

6.1. Speedup and efficiency

The stage *Read Mesh* of PETSc-FEM was ignored while computing the speedup, because it was carried out by all processors in a redundant manner. We assume it is matter for ulterior modifications of the program. Instead of the sequential time, that of the parallel code with one processor has been used to compute the speedup:

$$S_p = \frac{(T_{\text{global}} - T_{\text{ReadMesh}})_1}{(T_{\text{global}} - T_{\text{ReadMesh}})_p} \tag{46}$$

The time in one processor was taken in Node 1, when the complete network of seven processors was used, and it was taken in Node 4 when the homogeneous network composed of the last four processors was used.

In Fig. 1 the speedup as a function of the number of processors is shown. It is observed that the speedup is very high because the *Read Mesh* stage was excluded, and because the 'sequential' time was obtained in the slowest processor. This motivated to carry out the calculations in the homogeneous subset of four processors, being obtained the curves in Fig. 2. In these as well as the rest of the figures, it should be kept in mind that computations have been performed just with 1, 2, 4 or 7 processors. Therefore, the reference to other number of processors in the abscissa axis is meaningless.

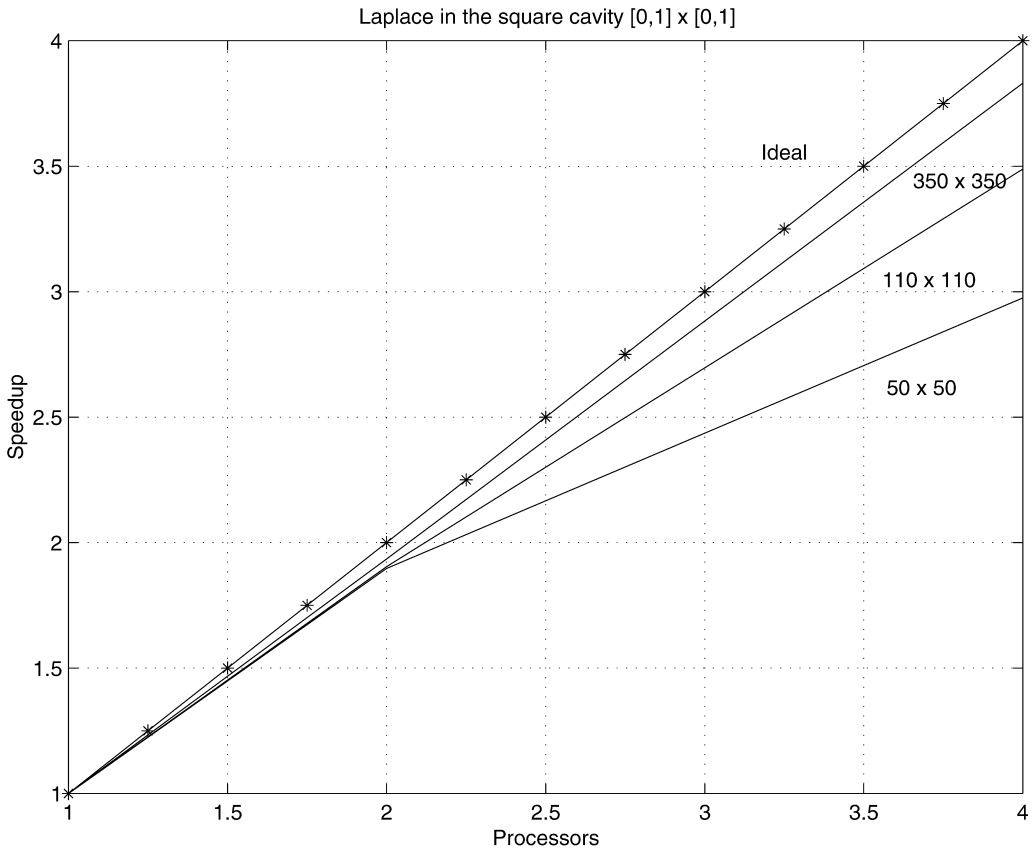

Fig. 2. Speedup on a homogeneous network for a Laplace problem.

In both the cases, parallel speedup grows as the size of the problem increases; nevertheless, this increase in speedup decreases when the number of processors is increased. A similar behavior was observed for the efficiency. For the heterogeneous network, the efficiency with 4 and 7 processors exceed 80% when the problem size is 110×110 or larger. For smaller problems it is efficient to implement 2 processors. The homogeneous network is more efficient, but the number of available processors decrease here from 7 to 4.

6.2. Number of iterations

The rate of convergence in Conjugate Gradient is proportional to $1/\sqrt{\kappa}$, where κ is the condition number of the matrix. On the other hand, the condition number for a structured mesh is $k = \alpha N^2$, being N the number of elements by side. Therefore, the number of iterations needed to achieve the specified tolerances is proportional to N. Fig. 3 shows the number of iterations carried out as a function of the size of the problem.

6.3. Times of execution and parallel aspects

Fig. 4 shows the elapsed time as a function of the problem size, in 1, 2, 4 and 7 processors. How the time is spent at the different stages of the program is shown in Fig. 5. The values for each processor have been taken for all the cases, as the average of the percentage of elapsed time of each stage. It is observed that the percentage of time dedicated to the stage *Read Mesh* (which is performed redundantly by all processors) grows as the number of processors increase, being 10% with one processor and 37% with seven. The stages *Matrix Structure*, *Assemble Residual* and *Assemble Matrix* tends to decrease very slowly as the number of processors increase, while the stage *System Solution* remain unchanged. The average CPU and communication time on 1, 2, 4 and 7 processors as percentage of the total time are given in Fig. 6. The communication time were measured taking into acount the average number of messages (n_p) sent by each processor, and the average length of each message (l_p), being

$$t_{\text{comm}} = \alpha n_p + \beta n_p l_p \tag{47}$$

It can be seen that the CPU time decreases as the number of processors grows, while the communication time increases and even more increases the remaining time. The last one can be considered as an overhead time, possibly by task synchronization, coordination and load unbalance. The percentage of time spent in computation, communication and coordination is given in Table 2.

The rate r of the maximum to the minimum values are given for reduction operations, length and number of messages. A ratio of approximately one indicates that

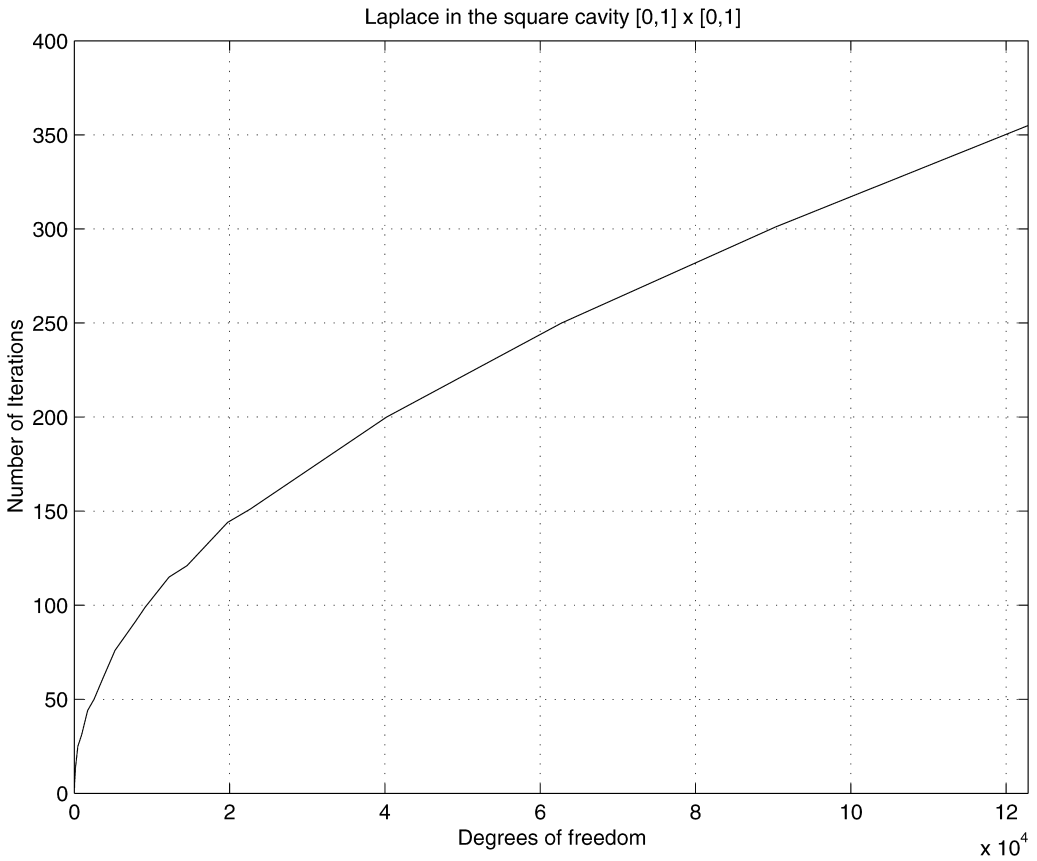

Fig. 3. Laplace problem: iterations.

computations within a given phase are well balanced among the processors. This happens with the reduction operations in 2, 4 and 7 processors. With two processors, the program is perfectly well balanced. With 4 nodes, a little unbalance in the length and the number of messages is observed; and with 7 processors $r > 2$, indicates that the number and length of the messages of some processor are duplicated by another one. This can be reflected in a loss in efficiency.

When the number of nodes increases, the average number of messages that each processor must send grows, but the length of these becomes smaller (less than 3350 bytes). So, the time to send a message will be dominated by the latency, specially with 4 and 7 processors.

In the systems solution stage the program performs 17 mflops with 1 processor, 33 mflops with 2, 63 mflops with 4 and 104 mflops with 7, for the case compiled without optimization. For the optimized compilation of the program we got 24, 41.5, 79.3, and 145 mflops, respectively. This figures correspond to the case of the finest mesh which can be run on one processor; this is not the larger problem that can be solved in the cluster.

Table 2
Percentage of time spent in Laplace problem

Processor	t_{comp}	t_{comm}	t_{coor}
1	99.44	0	0.56
2	93.23	1.86	4.91
4	81.83	3.25	14.92
7	75.41	4.24	20.35

7. Analysis of the Navier–Stokes code performance

The lid-driven square cavity is a well-known benchmark for NS equations. This problem was solved in a unit square domain, using structured meshes of quadrangular elements. In each case, the number N of nodes per side was given, having a total of $(N-1)^2$ elements. There are 3 degrees of freedom per node: velocity components and pressure. The first component of the velocity was fixed to in a non-zero value only on one side of the square: $u(x,1) = 1$; and $u \equiv 0$ in the other sides. The second component is zero in all the sides and the pressure was fixed in one node. Then, we have $8N - 7$ fixed nodes and $3N^2 - 8N + 7$ degrees of freedom. The values taken for N were: 10, 30, 60, 100, 150 and 200, having problems with 227 up to 118,407 degrees of freedom. The tolerance fixed for each case was 10^{-3} and, of course, a maximum number of iterations for the temporal step was fixed. In all cases, two values for the Reynolds number: $Re = 1000$ and $Re = 100$ were considered; ($Re = UL/\nu$, U is the velocity in the upper face and L is the length of the square). We propose a Courant number equal to 10. The value for the time step was taken in

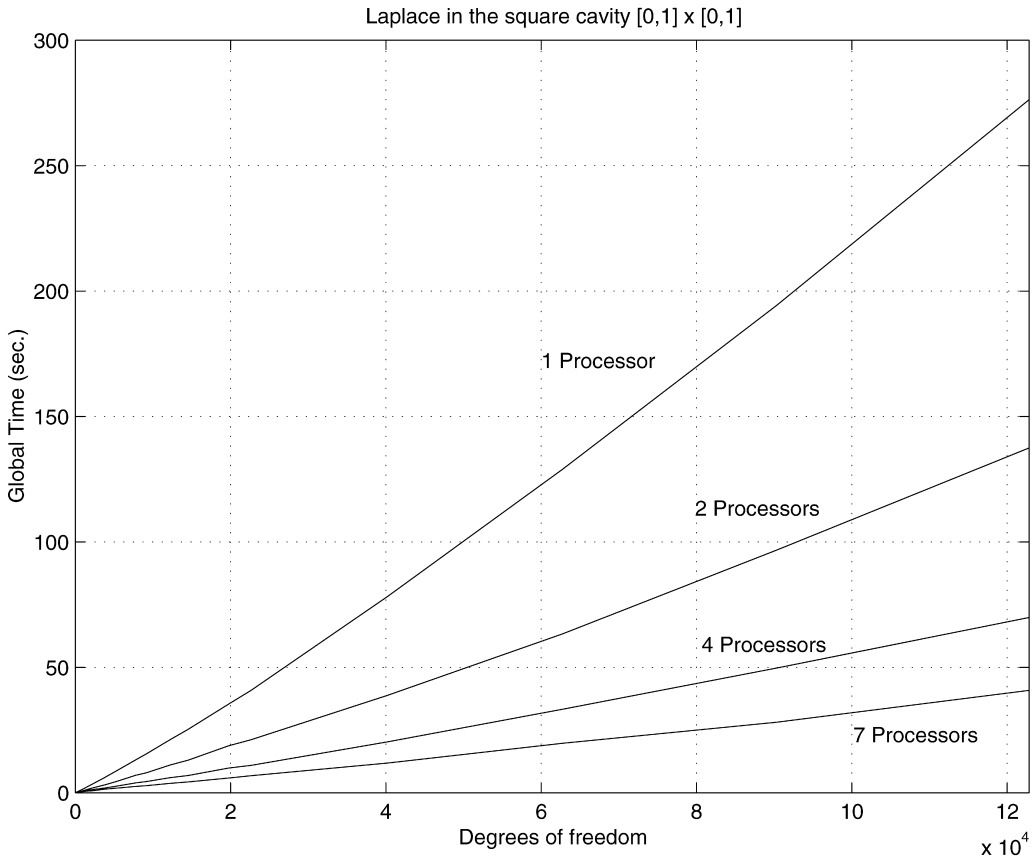

Fig. 4. Laplace problem: elapsed time.

function of the elements length by

$$\Delta t = h_{\min} \times 10 \tag{48}$$

The viscosity is the reciprocal of the Reynolds number. All the cases were solved with 1, 2, 4 and 7 processors.

7.1. Speedup and efficiency

Like in the Laplace solver, the stage *Read Mesh* was excluded from the speedup; and similar considerations were taken about the sequential time. We only show details about the cases with $Re = 1000$.

Table 3
Time in seconds for the time step in Navier–Stokes problem with $Re = 1000$

N	Unknowns	Processor			
		1	2	4	7
10	227	0.65	0.47	0.46	0.48
30	2467	8.5	5	2.5	1.7
60	10,327	37.5	18.1	9.7	5.5
100	29,207	105.5	53.2	26.5	15
150	66,307	249	122.5	58.4	33.1
200	118,407	428.3	–	106.3	69.6

In Fig. 7, the speedup as a function of the number of processors are shown for three cases, observing that one of them has overcome the ideal speedup. This happen because the sequential time was measured in the slowest processor. Identical curves were obtained with $Re = 100$.

In the homogeneous network, the curves for the cases $N = 100$ and 200, shows speedups greater than 1.9 with 2 processors and 3.8 with 4 processors, as we can see in Fig. 8. This indicates that the elapsed time decrease almost in half with 2 processors, and the parallel time with 4 processors represent 26% of the sequential time.

The program becomes more efficient as the size of the problem increases. For problems with smaller size, the efficiency is acceptable only with 2 processors. In the homogeneous network, the efficiency exceed 90% in problems with more than 29,000 degrees of freedom. Similar results were obtained with $Re = 100$.

7.2. Number of time steps and mflops

Table 3 shows the elapsed time per time step obtained for each case. The average mflops, for several mesh sizes and Reynolds numbers, obtained in the *System Solution* stage of the program are 42 mflops with 1 processor, 83 mflops with 2, 165 mflops with 4 and 282 mflops with 7 processors.

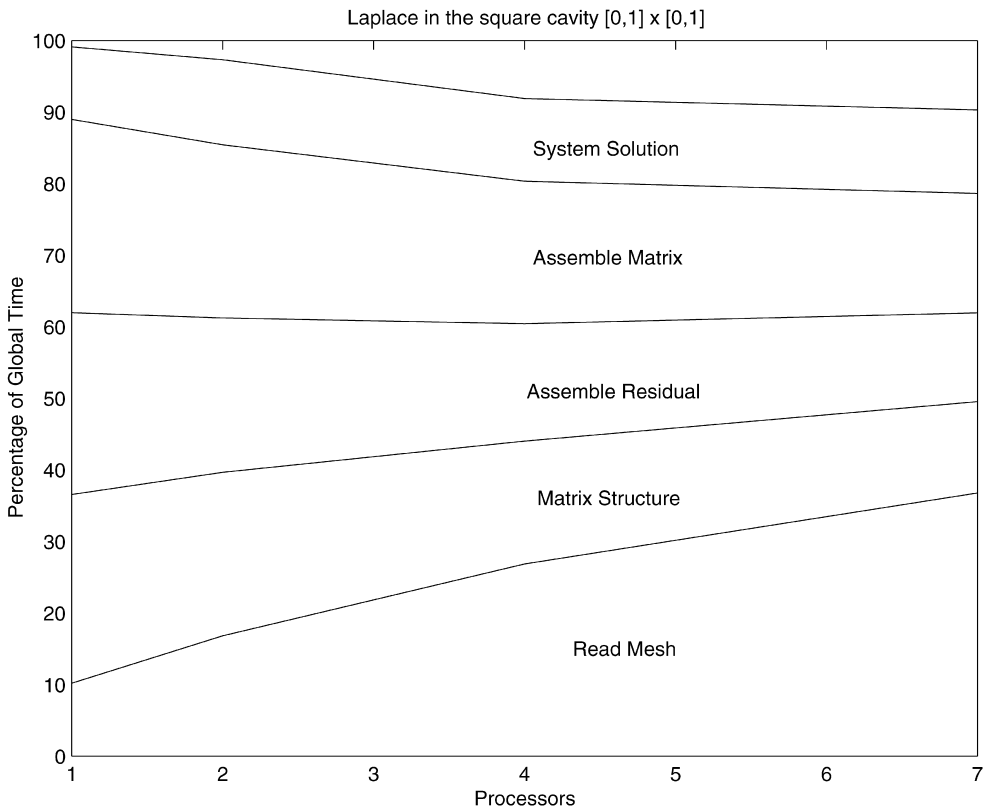

Fig. 5. Laplace problem: stages of the program.

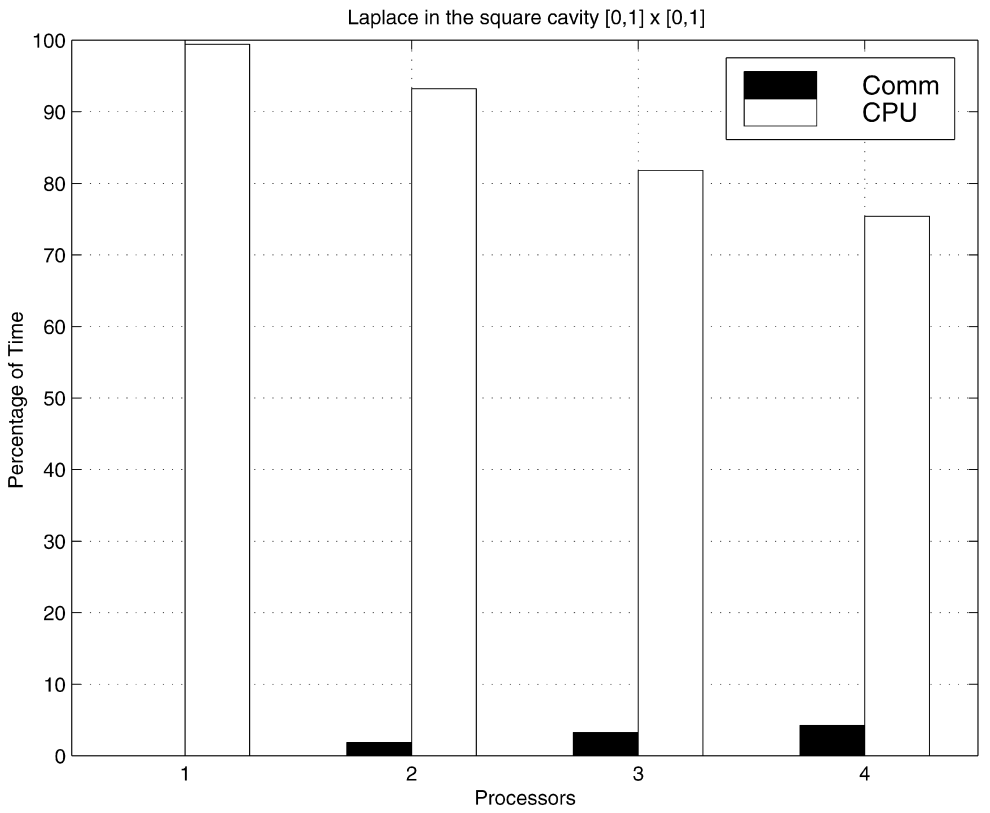

Fig. 6. Laplace problem: CPU and communication time.

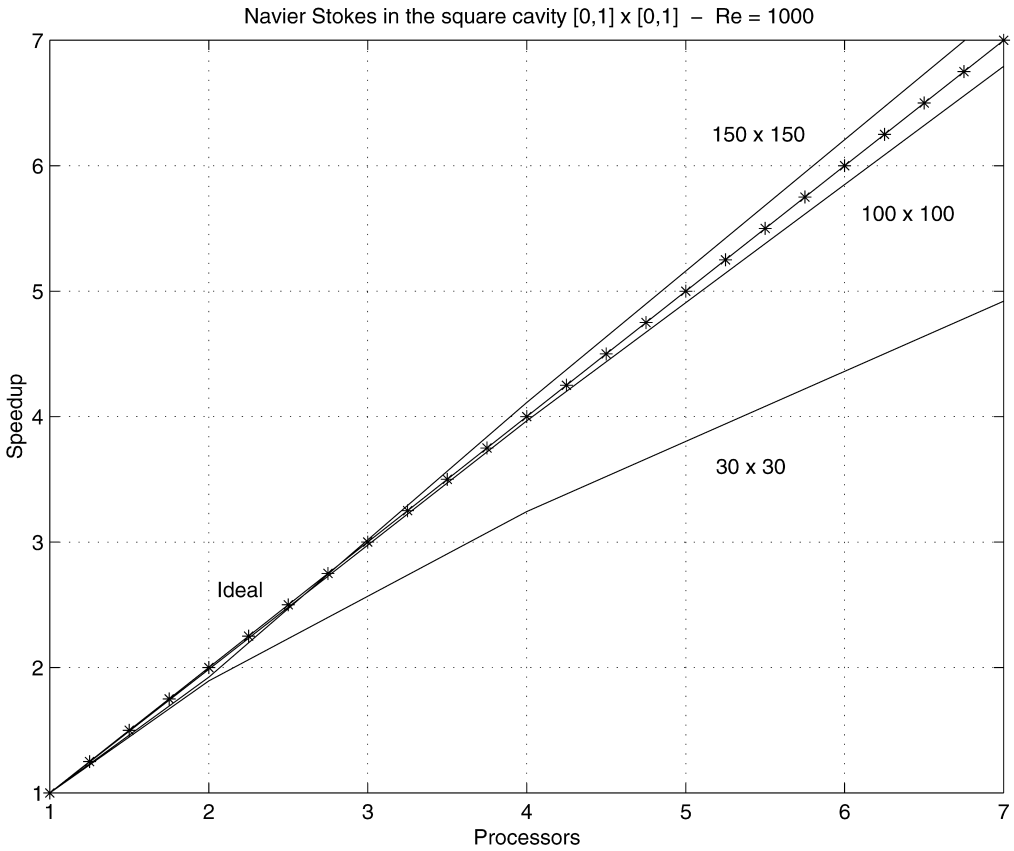

Fig. 7. Navier–Stokes problem: speedup on a heterogeneous network.

7.3. Times of execution and parallel aspects

The percentage of time spent on three stages of the program is shown in Fig. 9. The stages *Read Mesh* and *Matrix Structure* were excluded because they represent less than 1% of the global time.

The *Assemble Matrix* stage represents, on the average, 26% when using one processor and 21% with seven. The *Assemble Residual* stage decreases from 37% of the global time with one processor to 29.5% using 7 processors. On the other hand, the *System Solution* stage takes on the average 35.5% using one processor, and this percentage grows with the number of processors, being 47.5% with 7 processors. The last behavior is due to that the communication increases when using more nodes.

CPU and communication average times are given in the bar diagram of Fig. 10. The communication time was estimated with Eq. (47). One can see that the communication time grows slowly with the number of processors, while the CPU time decreases. The remaining time, that we have called 'overhead', could be due to tasks synchronization, coordination, or delays.

Taking $Re = 100$, the CPU time is between 3 and 7% below from those obtained with $Re = 1000$. The overhead time absorbs these loses.

A load balance of the program was carried out, taking the rate of the maximum to the minimum values in reduction operations, length and the number of messages. As it is shown in Fig. 11, the program is well balanced with two processors. Some differences in the length and the number of messages that each processor must send are observed using 4 processors; and this difference is bigger with 7 of them. This justifies the increase in the overhead time as the number of processors grows, being reflected in the loss in efficiency. In Navier–Stokes, the number of messages sent depends on the number of time steps carried out.

As an example, the case $N = 150$ (66,307 dof) with $Re = 1000$ shows that the average length of the messages grows with the size of the problem, but it decreases as the number of processors increases (Table 4). With 2 processors, the communication time will be dominated by the bandwidth, since the average length of the messages is superior to $n_{1/2} = 3042$ bytes. On the other hand, when using 4 or 7

Table 4
Number and length of messages—Navier–Stokes problem 66,307 unknowns

Processor	Number of messages		Length of messages	
	Average	Total	Average (bytes)	Total (MB)
2	16,110	32,210	5731	176
4	40,260	161,100	2116	325
7	55,100	385,700	1436	528

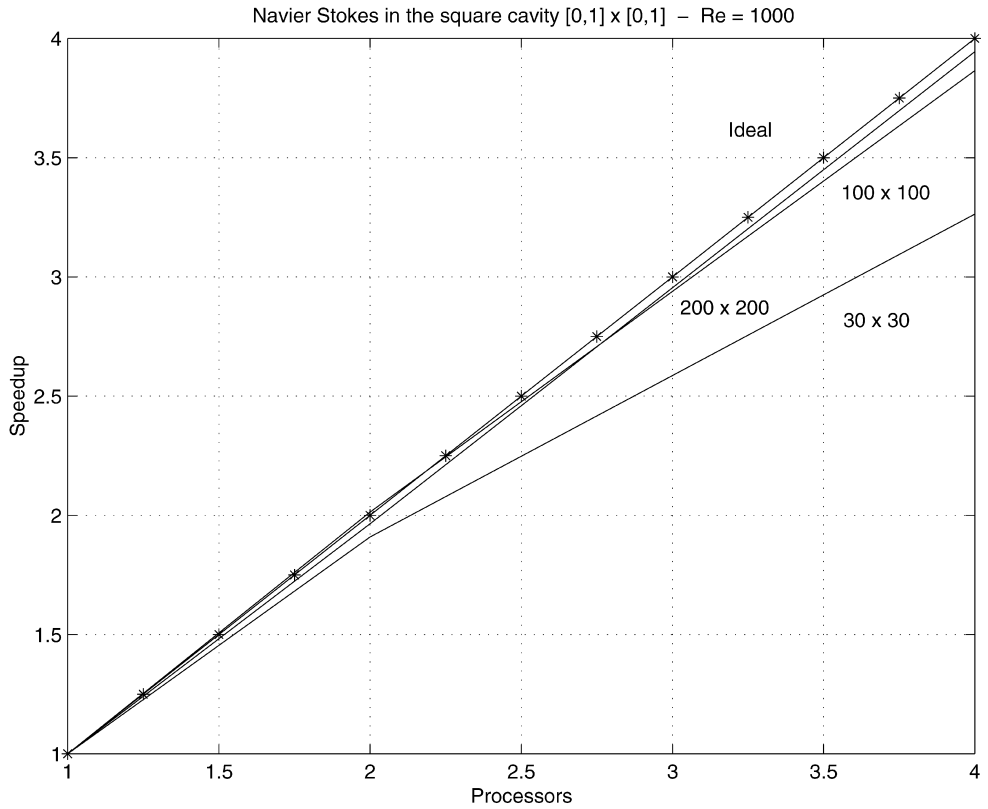

Fig. 8. Navier–Stokes problem: speedup on a homogeneous network.

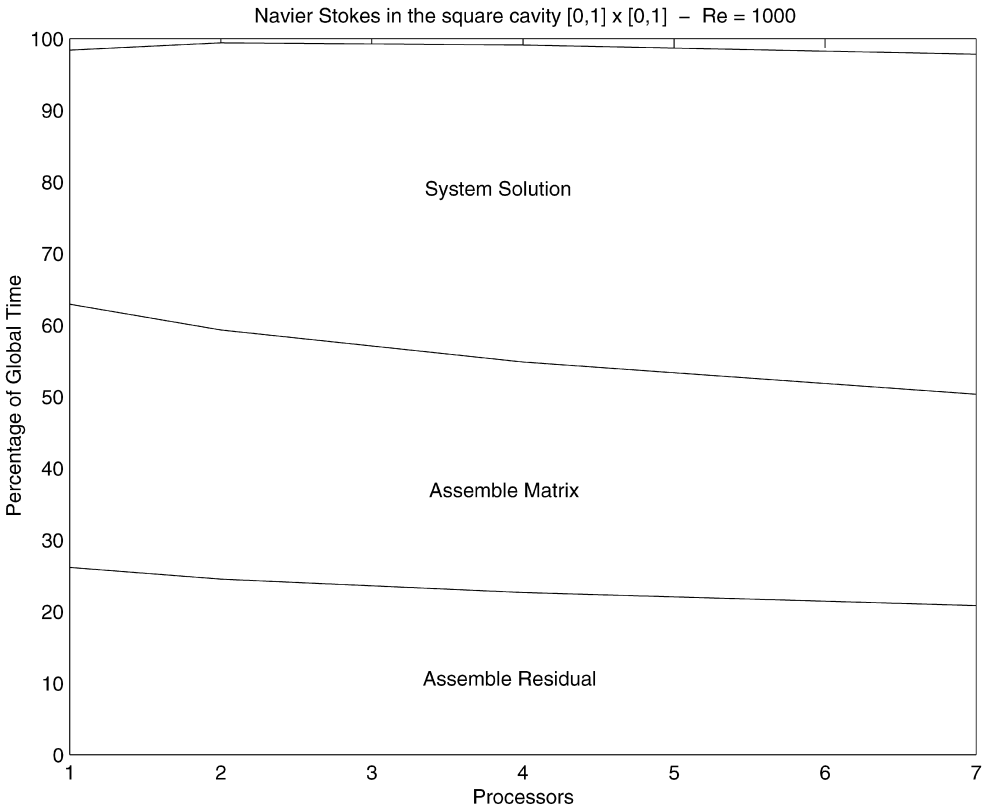

Fig. 9. Navier–Stokes problem: stages of the program.

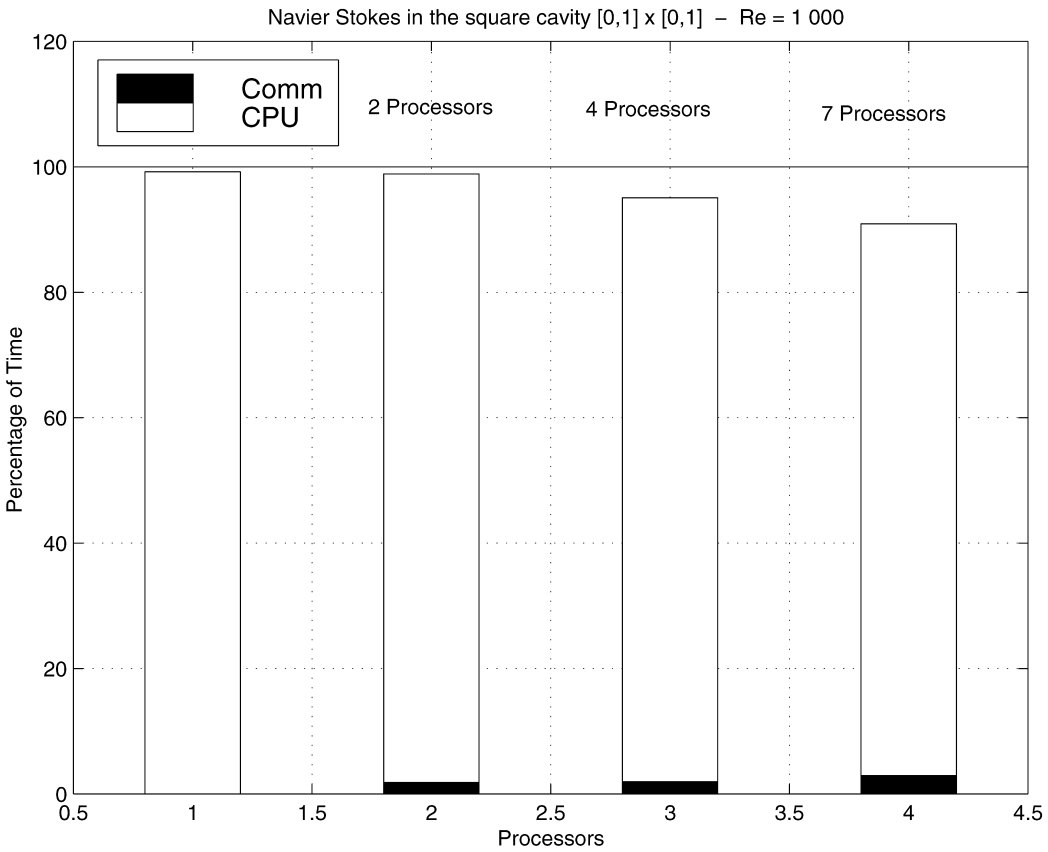

Fig. 10. Navier–Stokes problem: CPU and communication times.

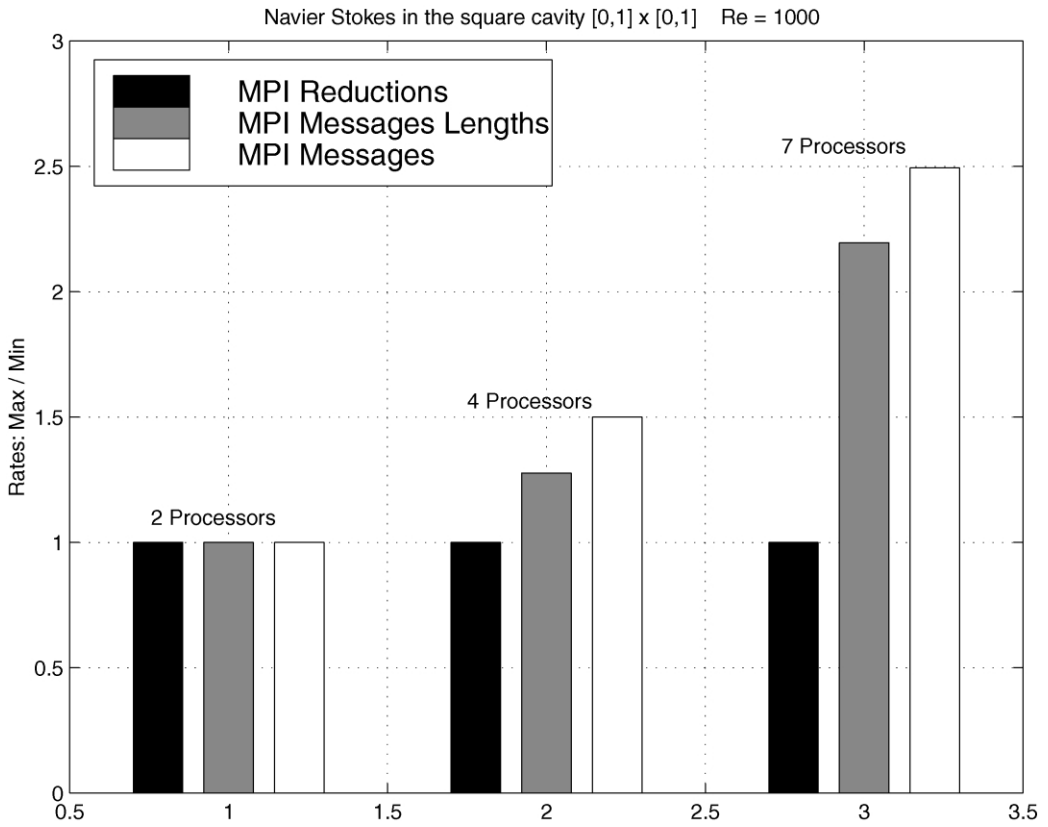

Fig. 11. Navier–Stokes problem: parallel aspects.

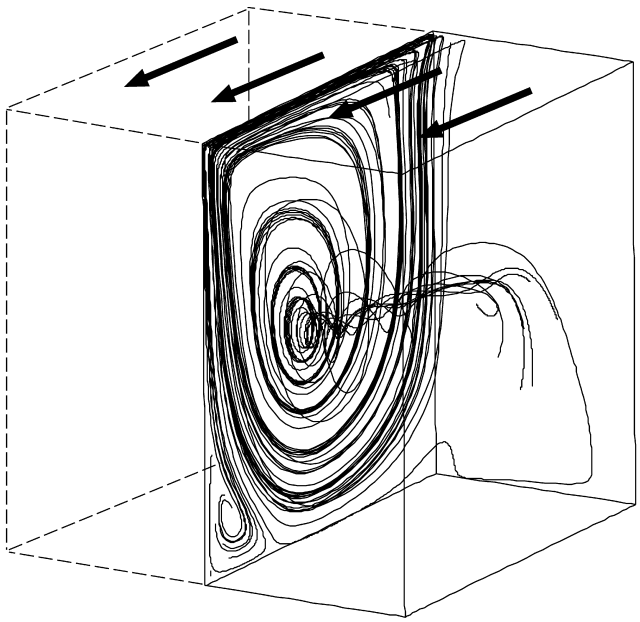

Fig. 12. Cubic cavity. Streamlines.

processors, the communication time will be dominated by the latency. In this particular case, 100 iterations of the time step have been done, showing that each processor sent, on the average, 161, 403 or 551 messages, either in 2, 4 or 7 processors, respectively.

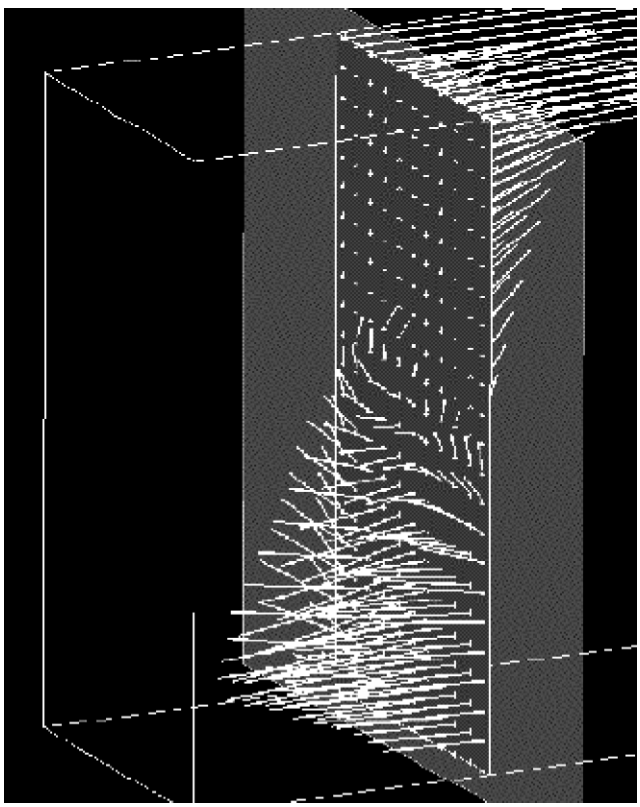

Fig. 13. Cubic cavity. Tufts at $x = 0.5$ plane.

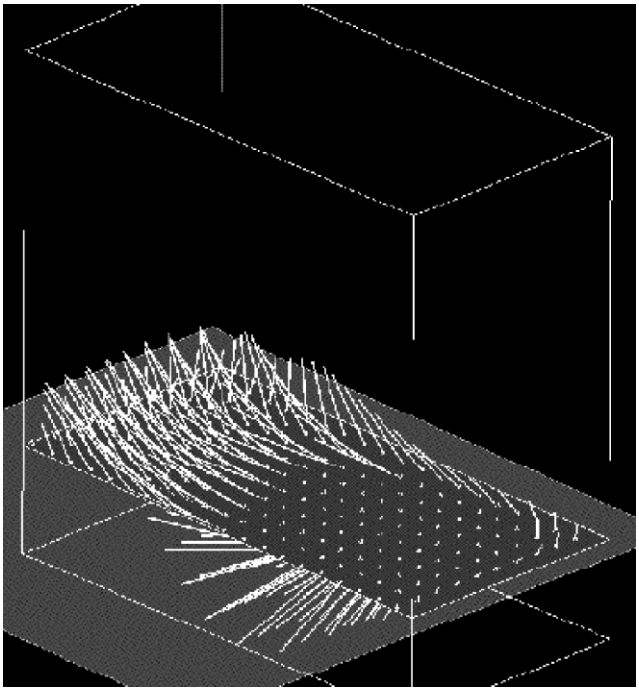

Fig. 14. Cubic cavity. Tufts at $y = 0.25$ plane.

8. Some applications of the Navier–Stokes code

8.1. Cubic cavity

In Figs. 12–19 we show numerical results obtained for the cubic cavity benchmark problem. The cavity occupies the region $0 \leq x, y, z, \leq 1$, and the boundary conditions are homogeneous Dirichlet boundary conditions (solid wall) on all sides of the cavity, except for the top side ($y = 1$) where $u = 1$, v, $w = 0$. Results are shown for Reynolds number

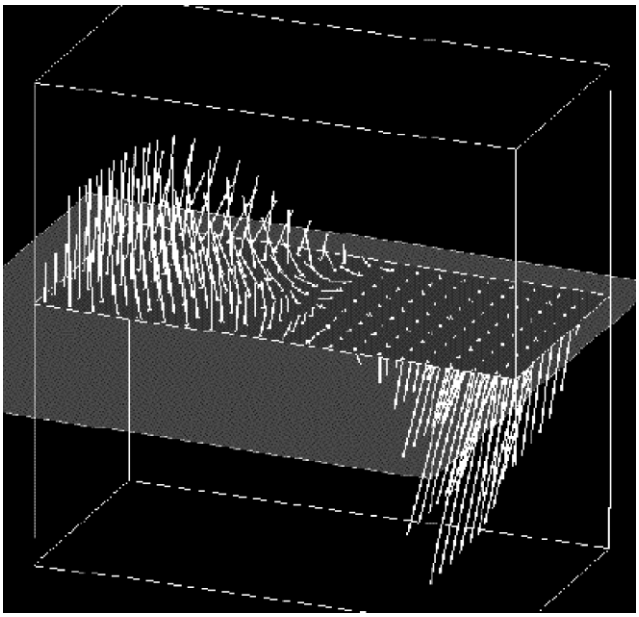

Fig. 15. Cubic cavity. Tufts at $y = 0.5$ plane.

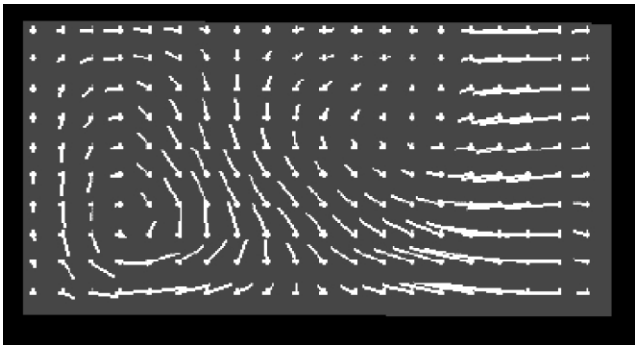

Fig. 16. Cubic cavity. Tufts $y = 0.5$. Projection on the $y = 0.5$ plane.

$Re = 1000$ with a mesh of 35,301 nodes covering one-half $0 \leq z \leq 1/2$ of the cavity. Considering one-half of the cavity is appropriate, provided that a unique steady (and then, symmetric) solution is obtained. We show streamlines and also 'tufts' (i.e. velocity vectors) at several constant x, y, z-planes. The streamlines have been traced upstream and downstream from starting points close to the symmetry plane ($z = 1/2$), and we can see that there is a streamline pattern on this plane similar to the well known 2D case. Of course, if the starting points were located exactly on the symmetry plane, then the streamlines would be contained in that plane since the velocity field is symmetric. Outside the symmetry plane there is a relatively small secondary flow (i.e. in the z-direction) oriented from the symmetry plane to the outer walls ($z = 0$) in the region of the rotating core and in the opposite side near the borders (x, $y = 0$, 1). This explains the behavior of some streamlines in Fig. 12 that start from the center of the outer wall ($z = 0$) and spiral around the axis of the rotating core towards the center of the symmetry plane.

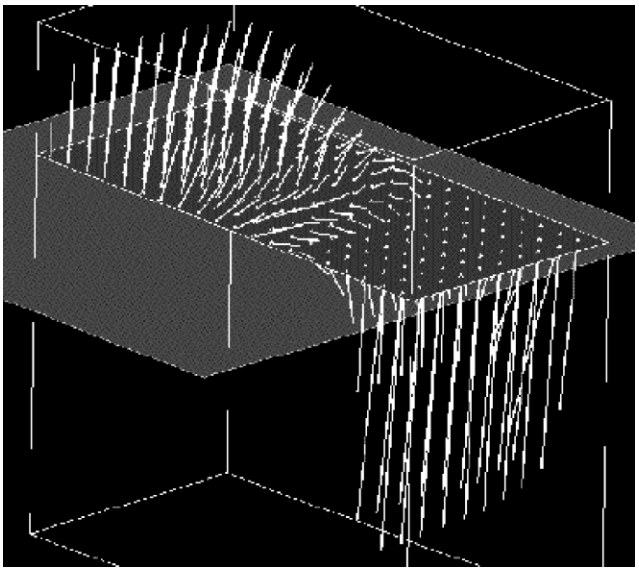

Fig. 17. Cubic cavity. Tufts on $y = 0.75$ plane.

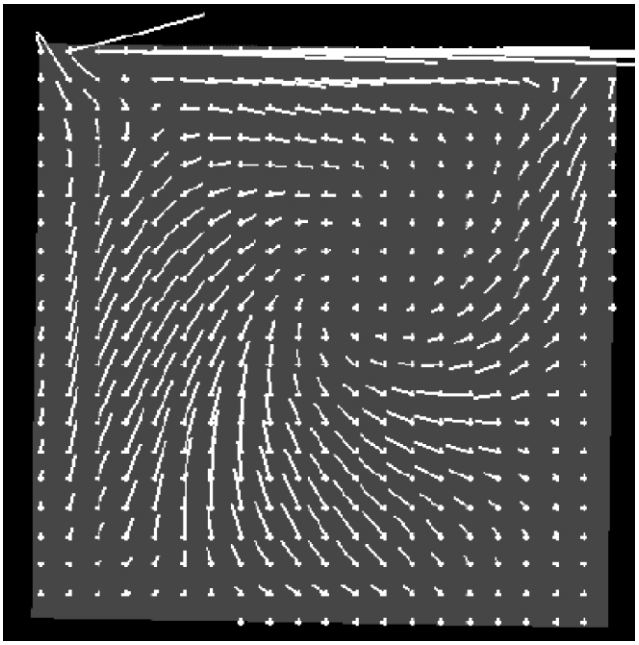

Fig. 18. Cubic cavity. Tufts at plane $z = 0$.

8.2. Unsteady 2D cylinder

This is a well-known problem of an homogeneous flow impinging transversally to a cylinder. We solved this problem at $Re = 100$ (diameter based) with a mesh composed of 28,000 elements, 28,390 nodes. As it is well known, for Reynolds numbers higher than some critical value near 40, there is no stable steady solution and the unsteady solution has a typical 'von Kàrman's vortex street'. The computed non-dimensional 'Strouhal number' ($d\omega/u_\infty$) was $St = 0.16$ which agrees very well with the

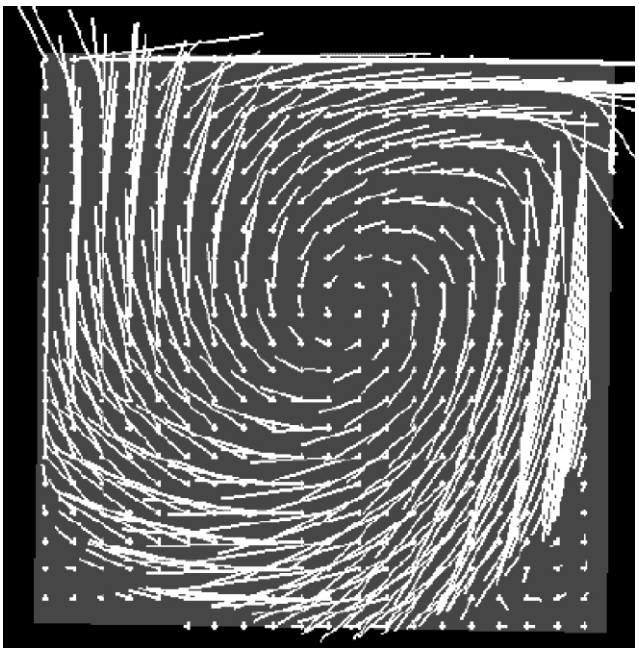

Fig. 19. Cubic cavity. Tufts at plane $z = 0.25$.

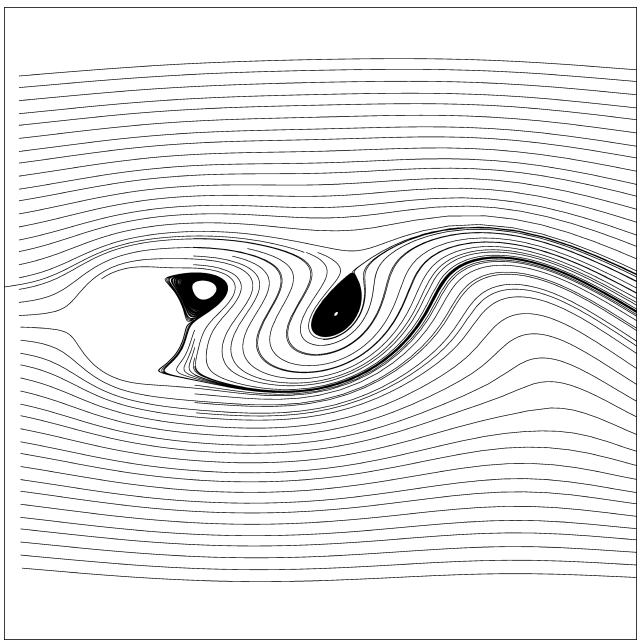

Fig. 20. Circular cylinder at $Re = 100$. Streamlines.

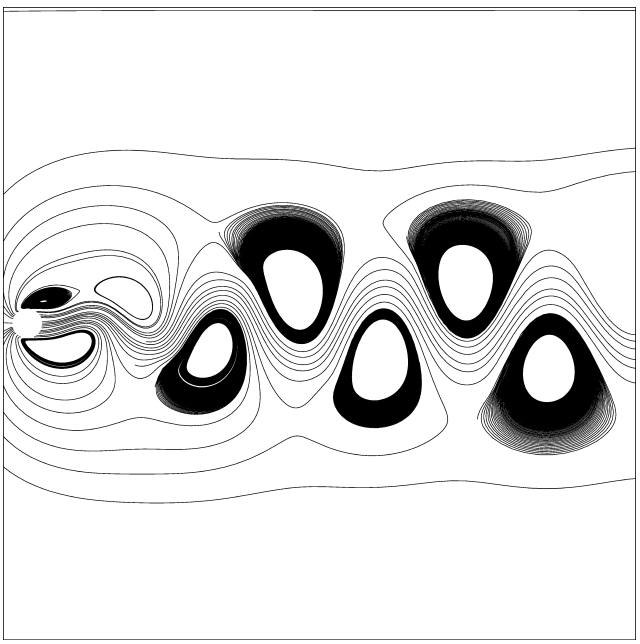

Fig. 21. Circular cylinder at $Re = 100$. Stationary streamlines.

value reported in the literature. In Fig. 20 we see the streamlines for the instantaneous velocity field, i.e. lines that are tangent everywhere to the local instantaneous velocity. Of course, as the problem is unsteady, these are not particle paths. However, if we assume that far from the cylinder the flow is steady with respect to a Galilean system moving with the unperturbed velocity of the fluid, then we can trace the streamlines for that flow, which in this case are particle paths. However, these are only approximately true, due to viscous dissipation of the vortices. A detail of these 'stationary streamlines' is shown in Fig. 21. They are computed from the instantaneous velocity field as lines tangent everywhere to the $\mathbf{u} - \mathbf{u}_\infty$ vector.

Acknowledgements

This work has received financial support from Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET, Argentina), Banco Interamericano de Desarrollo (BID) and Universidad Nacional del Litoral through grants: CONICET PIP 198/98, PIP 2552/2000; ANPCyT PICT51, PID99/74 and 6973; and UNL CAI + D 2000/43. We made extensive use of freely distributed software such as Linux OS, MPI, PETSc, Newmat, Visual3 and many others.

References

[1] The Beowulf project, http://www.beowulf.org.
[2] The Linux documentation project, http://sunsite.unc.edu/mdw/linux.html.
[3] The PVM project, http://www.epm.ornl.gov/pvm/.
[4] MPI forum, http://www.mpi-forum.org/docs/docs.html.
[5] Tezduyar T, Mittal S, Ray S, Shih R. Incompressible flow computations with stabilized bilinear and linear equal order interpolation velocity–pressure elements. Comput Meth Appl Mech Engng 1992;95.
[6] Sterling TL, Salmon J, Becker DJ, Savarese DF. How to build a Beowulf. Cambridge, MA: MIT Press; 1999.
[7] Storti M, Nigro N. PETSc-FEM: a general purpose, parallel, multi-physics FEM program, http://minerva.ceride.gov.ar/petscfem.
[8] Golub GH, Van Loan C. Matrix computation, 2nd ed. Baltimore, MD: The John Hopkins University Press; 1993.
[9] Succi C, Papetti F. An introduction to parallel computational fluid dynamics. New York: Nova Science Publishers; 1996.
[10] Dongarra JJ, Dunigam T. Message-passing performance of various computers. University of Tennessee and Oak Ridge National Laboratory, Report; 1997, www.netlib.org/utk/papers/commperf.ps.
[11] Dongarra JJ, Duff IS, Sorensen DC, Van der Vorst HA. Solving linear systems on vector and shared memory computers. Philadelphia, PA: SIAM; 1991.