Finite Difference Simulations of the Navier-Stokes Equations using Parallel Distributed Computing

João Paulo De Angeli[1], Andrea M. P. Valli[1], Neyval C. Reis Jr.[2], Alberto F. De Souza[1]
[1]Departamento de Informática, Centro Tecnológico – UFES
Av. Fernando Ferrari, s/n, 29.060-970 – Vitória – ES
{jpda@inf.ufes.br, avalli@inf.ufes.br, alberto@inf.ufes.br}
[2]Departamento de Engenharia Ambiental, Centro Tecnológico – UFES
Av. Fernando Ferrari, s/n, 29.060-970 – Vitória – ES
{neyval@inf.ufes.br}

Abstract—
This paper discusses the implementation of a numerical algorithm for simulating incompressible fluid flows based on the finite difference method and designed for parallel computing platforms with distributed-memory, particularly for clusters of workstations. The solution algorithm for the Navier-Stokes equations utilizes an explicit scheme for pressure and an implicit scheme for velocities, i. e., the velocity field at a new time step can be computed once the corresponding pressure is known. The parallel implementation is based on domain decomposition, where the original calculation domain is decomposed into several blocks, each of which given to a separate processing node. All nodes then execute computations in parallel, each node on its associated sub-domain. The parallel computations include initialization, coefficient generation, linear solution on the sub-domain, and inter-node communication. The exchange of information across the sub-domains, or processors, is achieved using the message passing interface standard, MPI. The use of MPI ensures portability across different computing platforms ranging from massively parallel machines to clusters of workstations. The execution time and speed-up are evaluated through comparing the performance of different numbers of processors. The results indicate that the parallel code can significantly improve prediction capability and efficiency for large-scale simulations.

Keywords— **Parallel Processing, Finite Difference Method, Navier-Stokes, MPI.**

I. INTRODUCTION

In this paper, we study the parallel implementation of the numerical discretization of the Navier-Stokes equations based on the finite difference method suggested by Griebel, Dornseifer and Neunhoeffer [1]. This method was strongly influenced by the *marker-and-cell* (MAC) technique of Harlow et al. from the Los Alamos National Laboratory [2]. It consists of an implicit scheme for pressure, which uses successive overrelaxation (SOR) iterations, and an explicit scheme for velocities with a first-order time discretization. Despite its simplicity, this method is surprisingly flexible and relatively efficient, and may be applied to a variety of transient problems with fixed and free boundary domains [3, 4]. In addition, improvements to this algorithm can be attained by using multigrid methods to solve the pressure equations, by treating the momentum equations (semi) implicitly, or by employing higher order methods. Such techniques applied before parallelization increases efficiency. However, in this work we focus on the parallelization of the finite difference scheme. We are interested on improving prediction capability and efficiency for large-scale simulations using parallel computations. Our goal is to reduce the total computing time by dividing the computational work between several processors, which perform their calculations concurrently. To avoid the cost limitations imposed by supercomputers, our code was mainly developed for clusters of workstations.

Rapid increase of microprocessor and network performance has enabled the implementation of clusters of workstations with high levels of computing power for a small fraction of the price of supercomputers. However, the use of clusters of workstation requires different programming paradigms since the system architecture is based on a distributed-memory model, which differs considerably from the shared-memory mode widely used in the last decades. We have used the domain decomposition coordinate bisection technique [5] for implementing our parallel algorithm for cluster of workstations. In this technique, the number of points is equally divided between processors, but no attempt is made to obtain a domain division that minimizes communications between processors. The parallel computations include initialization, coefficient generation, linear solution on the sub-domain, and inter-node communication. The exchange of information across sub-domains, or processors, is achieved using the message passing interface standard, MPI. The use of MPI ensures portability across different computing platforms, ranging from massively parallel machines to clusters of workstations.

The frequency at which the communication between processors occurs within each SOR iteration was also investigated. Griebel et al. [1] proposed that a communication step should be performed after each SOR iteration, so that the values at the boundaries are updated at every SOR iteration. This procedure introduces a significant amount of communication, which, according to our experiments, slows down the computation. An alternative procedure that we have used is to perform a few SOR iterations prior to communication. This reduces the amount of communication without slowing down the convergence rate. In addition, the communication steps were performed in three different ways: all-to-all, master-slave and binary-tree. In order to assess the parallel performance of the algorithm and to evaluate all different parallelization strategies we have used, simulations with 1, 2, 4, 8, 16, 32, 48 and 64 processors were performed and the execution time, speed-up and parallel efficiency were measured. The results indicated that, for more than 32 processors, the binary-tree approach outperformed the others by a large margin.

The outline of the paper is as follows. First, we present the class of governing equations under investigation and briefly the finite difference formulation. Then, we discuss the parallelization strategy adopted for the solution algorithm. In Section 4 we present the main conclusions.

II. GOVERNING EQUATIONS AND THE FINITE DIFFERENCE FORMULATION

We consider the stationary and transient flow of a viscous incompressible fluid as described by the two-dimensional Navier-Stokes equations

$$\frac{\partial u}{\partial t}+\frac{\partial p}{\partial x}=\frac{1}{\sqrt{\mathrm{Re}}}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)-\frac{\partial(u^2)}{\partial x}-\frac{\partial(uv)}{\partial y}+g_x \quad (1)$$

$$\frac{\partial v}{\partial t}+\frac{\partial p}{\partial y}=\frac{1}{\sqrt{\mathrm{Re}}}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)-\frac{\partial(uv)}{\partial x}-\frac{\partial(v^2)}{\partial y}+g_y \quad (2)$$

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0 \quad (3)$$

where u and v are the horizontal and vertical components of the velocity, p is the pressure, g_x and g_y are body forces, $Re=(\rho_\infty u_\infty L)/\mu$ is the dimensionless Reynolds number, ρ_∞ u_∞ and L are given scalar constants (namely, fluid density, characteristic velocity and characteristic length, respectively) and μ is the dynamic viscosity. To complete the mathematical statement of the problem, we need initial and boundary conditions. We consider velocities or flux boundary conditions.

The numerical treatment of the Navier-Stokes equations is based on the finite difference scheme suggested by Griebel, Dornseifer and Neunhoeffer in [1]. In the usual way, the flow domain is discretized into i_{max} cells of equal sizes in the x-direction and j_{max} cells in the y-direction. The region is discretized using a staggered grid, in which the pressure p is located in the cell centers, the horizontal velocity u in the midpoints of the vertical cell edges, and the vertical velocity v in the midpoints of the horizontal cell edges. This staggered arrangement of the unknowns prevents possible pressure oscillations, which could occur had we evaluated all three unknown values u, v and p at the same grid points.

The discretization of the spatial derivatives requires a mixture of central differences and donor-cell discretization to maintain stability for strongly convective problems. Because the convective terms $\partial(u^2)/\partial x$, $\partial(uv)/\partial y$, $\partial(uv)/\partial x$ and $\partial(v^2)/\partial y$ in the momentum equations become dominant at high Reynolds numbers or high velocities, stability problems may occur when the grid spacing is chosen too coarse. To avoid stability problems, these convective terms are treated using a weighted average of central difference and donor-cell scheme as suggested by (Hirt et al., 1975). The first order spatial derivatives $\partial u/\partial x$, $\partial v/\partial y$ and the second order derivatives $\partial^2 u/\partial x^2$, $\partial^2 u/\partial y^2$, $\partial^2 v/\partial x^2$ and $\partial^2 v/\partial y^2$, forming the so-called diffusive terms, may be replaced by central differences using half the mesh width. Details of the spatial discretization can be found in (Griebel et al., 1998). To obtain the time discretization of the momentum equations (1) and (2), we discretize the time derivatives $\partial u/\partial t$ and $\partial v/\partial t$ using the Euler's method. Introducing the functions

$$F^{(n)}=u^{(n)}+\delta t\left[\frac{1}{\sqrt{\mathrm{Re}}}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)-\frac{\partial(u^2)}{\partial x}-\frac{\partial(uv)}{\partial y}+g_x\right]^{(n)} \quad (4)$$

$$G^{(n)}=v^{(n)}+\delta t\left[\frac{1}{\sqrt{\mathrm{Re}}}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)-\frac{\partial(uv)}{\partial x}-\frac{\partial(v^2)}{\partial y}+g_y\right]^{(n)} \quad (5)$$

where the superscript n denotes the time level, we have the fully discrete momentum equations

$$u_{i,j}^{(n+1)}=F_{i,j}^{(n)}-\frac{\delta t}{\delta x}\left(p_{i+1,j}^{(n+1)}-p_{i,j}^{(n+1)}\right)$$
$$i=1,\ldots,i_{\max}-1 \qquad j=1,\ldots,j_{\max} \quad (6)$$

$$v_{i,j}^{(n+1)}=G_{i,j}^{(n)}-\frac{\delta t}{\delta y}\left(p_{i,j+1}^{(n+1)}-p_{i,j}^{(n+1)}\right)$$
$$i=1,\ldots,i_{\max} \qquad j=1,\ldots,j_{\max}-1 \quad (7)$$

which may be characterized as being explicit in the velocities and implicit in the pressure; i.e., the velocity field at time step t_{n+1} can be computed once the corresponding pressure is known. Substituting the equations (6) and (7) for the velocity field into the continuity equation (3), we obtain the Poisson equation for the pressure $p^{(n+1)}$ at time t_{n+1}

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} = \frac{1}{\delta t}\left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y}\right) \qquad i = 1,\ldots,i_{\max} \quad j = 1,\ldots,j_{\max} - 1 \tag{8}$$

which requires boundary values for the pressure. We assume $p_{0,j} = p_{1,j}$, $p_{imax+1,j} = p_{imax,j}$, $p_{i,0} = p_{i,1}$, $p_{i,jmax+1} = p_{i,jmax}$, with $i = 1,\ldots, i_{max}$ and $j = 1,\ldots, j_{max}$. In addition, we need values of F and G at the boundary to compute the right-hand side of (8). We set $F_{0,j} = u_{0,j}$, $F_{imax,j} = u_{imax,j}$, $G_{i,0} = v_{i,0}$ and $G_{i,jmax} = v_{i,jmax}$, with $i = 1,\ldots, i_{max}$ and $j = 1,\ldots, j_{max}$.

As a result, we have to solve a linear system of equations (8) containing $i_{max}\, j_{max}$ unknowns $p_{i,j}$, $i = 1,\ldots, i_{max}$ and $j = 1,\ldots, j_{max}$. In this work, we obtain approximate solutions for the pressure using the SOR method. To avoid generating oscillations, an adaptive stepsize control based on the famous Courant-Friedrichs-Lewy (CFL) conditions is used in order to ensure stability of the numerical algorithm [6]. The new time-step size is given by

$$\delta t = \tau \min\left[\frac{\mathrm{Re}}{2}\left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{u_{\max}}, \frac{\delta y}{v_{\max}}\right] \tag{9}$$

where the factor $\tau \in (0, 1]$ is a safety factor. In summary, the entire procedure consists of the steps outlined in Fig. 1.

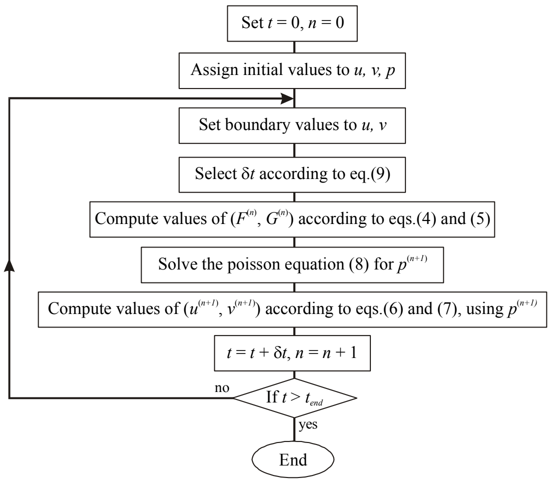

Set $t = 0$, $n = 0$
Assign initial values to u, v, p
Set boundary values to u, v
Select δt according to eq.(9)
Compute values of ($F^{(n)}$, $G^{(n)}$) according to eqs.(4) and (5)
Solve the poisson equation (8) for $p^{(n+1)}$
Compute values of ($u^{(n+1)}$, $v^{(n+1)}$) according to eqs.(6) and (7), using $p^{(n+1)}$
$t = t + \delta t$, $n = n + 1$
If $t > t_{end}$
no
yes
End

Fig. 1 – Algorithm for the Navier-Stokes equations.

III. Parallelization Strategy

While for shared-memory parallelism is mainly directed to execute an identical set of operations in parallel on the same data structure (do-loop parallelization), parallelism in distributed-memory systems is mainly directed to sub-divide the data structures into sub-domains and assign each sub-domain to one processor. In this case, the same code runs on all processors with its own set of data.

Fig. 2a shows a schematic representation of a mesh, for the simulation of the backward facing step problem, containing 2400 nodal points. By dividing the computational domain into four sub-domains (Fig. 2b), it is possible to spread the workload between four different processors. However, it is important to note that, in order to compute the variables for each nodal point, the variables at its neighboring points are required. Thus, in order to calculate the variables at the points close to the interface between sub-domains, one processor will require information stored in the memory of a neighboring processor. This requires communication at regular intervals, which may slow down the computation.

In general, the computation procedure involves three steps (1) partitioning of the solution domain; (2) performing computations on each processor to update its own data set; (3) communicating data between processors. This technique is called domain decomposition. The key for an efficient computation is to maintain the communications between processors to a minimum level, as well as, to divide the workload equally between processors.

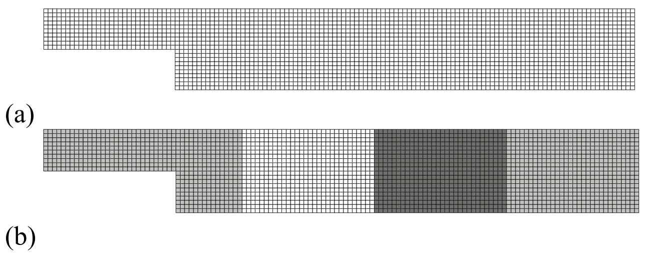


Fig. 2 - (a) Schematic representation of a mesh for the simulation of the backward facing step problem and (b) its decomposition in 4 sub-domains.

In this work, domain decomposition coordinate bisection is used [5]. This method divides the number of points equally between processors, but makes no attempt to obtain a domain division that minimizes the communication between processors, i.e., a division with the smallest number of nodal points in boundaries between sub-domains. Therefore, coordinate bisection may produce sub-domains with long interfaces that will lead to a large amount of communication. This can be partly overcome by recursive application of alternate x, y (and in 3D, z) bisections. The grid is first divided into 2 grids using bisection of the x-length of the calculation domain. Then to each of the resulting domains, y-bisection is applied, resulting in four blocks (or sub-domains). The procedure can be continued to obtain eight blocks, sixteen blocks, thirty two blocks, etc.

Once a multi-block domain has been established, calculations on each block can begin in parallel if the boundary conditions of these blocks are known. This may be either a physical boundary condition or an internal boundary condition generated as a consequence of the domain decomposition. The physical boundary data of each block, if any, are provided by the user, while the internal boundary data must be received from neighboring blocks, which may reside on different processors. Internal boundary data are hold by buffers on the boundary of each block as shown in Fig. 3, which illustrates a calculation sub-domain and the buffer cells used to store the overlap data. Once the buffer data has been received from all sides of a block, the computation of this block can start, using the sequential algorithm. On completion of the solution for the block, the data at its boundaries is sent to the neighboring blocks. Calculation in this block then waits for the buffer update provided by this block's neighbors, after which the next computation cycle can start.

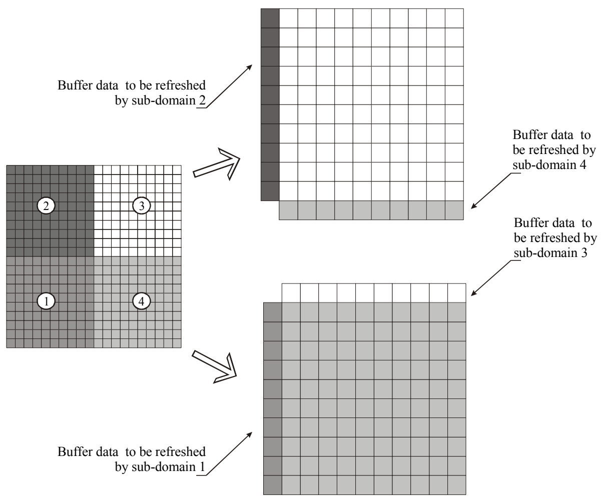

Buffer data to be refreshed by sub-domain 2
Buffer data to be refreshed by sub-domain 4
Buffer data to be refreshed by sub-domain 3
Buffer data to be refreshed by sub-domain 1
2
3
1
4

Fig. 3 – Schematic representation of a calculation sub-domain divided into four sub-domains, indicating the buffer cells used to store the internal boundaries data

Fig. 4 shows the sequence of operations involved in the computation. Each processor performs the computation on its own sub-domain, which includes the initialization of the values of velocities and pressure, and time-step size calculations. Since each processor calculates a local value for the time-step size based on its own local data (eq.9), some communication is required to choose a time-step to be used in all calculation sub-domains. This is required because all processors need to advance to the next time-step using the same values of δt in the present solution algorithm. The time-step chosen has to attend the CFL criterion (eq. 9) for all sub-domains, as such the smallest time-step size is selected. After that, the values of F and G are determined, and the coefficients of the linear set of equations for pressure are calculated. Then, each block individually deals with the solution of the pressure equation for the nodal points at its sub-domain, solving the linear system of equations using SOR iterations. Once pressure values are calculated for each block, the pressure values at each block boundary are communicated to its neighbors. This procedure is repeated until convergence is reached, which is checked comparing the values in the buffers prior and after communication. We assume convergence if the perceptual difference is less than 0.1%. Velocity components are then calculated on each sub-domain, and their values at each block boundary communicated to its neighbors. This iterative process is continued until the whole process reaches convergence.

The information exchange across sub-domains is performed using the message passing interface standard, MPI. The use of MPI ensures portability across different computing platforms, ranging from massively parallel machines to clusters of workstations.

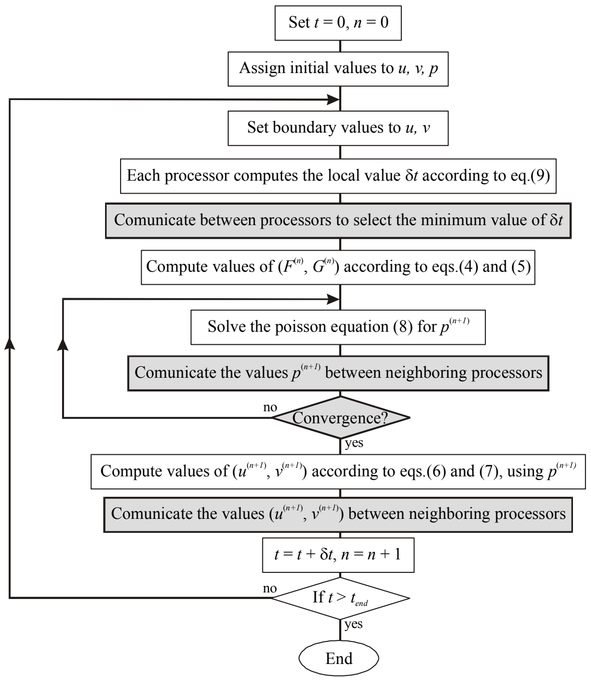

Set $t = 0$, $n = 0$
Assign initial values to u, v, p
Set boundary values to u, v
Each processor computes the local value δt according to eq.(9)
Comunicate between processors to select the minimum value of δt
Compute values of ($F^{(n)}$, $G^{(n)}$) according to eqs.(4) and (5)
Solve the poisson equation (8) for $p^{(n+1)}$
Comunicate the values $p^{(n+1)}$ between neighboring processors
no
Convergence?
yes
Compute values of ($u^{(n+1)}$, $v^{(n+1)}$) according to eqs.(6) and (7), using $p^{(n+1)}$
Comunicate the values ($u^{(n+1)}$, $v^{(n+1)}$) between neighboring processors
$t = t + \delta t$, $n = n + 1$
no
If $t > t_{end}$
yes
End

Fig. 4 – Parallel algorithm for the Navier-Stokes

It is important to note that there are 4 different communication steps during the calculation procedure: (1) one required by the selection of the minimum δt; (2) one involving the communication of the pressure for the neighboring blocks, (3) one required for convergence checking and (4) another one involving the communication of the velocities for the neighboring blocks. The communication steps (2) and (3) are the most critical to the efficiency of the computation, since they occur several times during the calculations, while the steps (1) and (4) occur only once per time-step. The communication step (2) only requires communication between neighboring processors. On the other hand, step (3) requires communications between all processors to check if all sub-domains have reached convergence. This can be performed using two different approaches: (a) each processor communicates its error value to all other processors and each processor check if the convergence was reached or not; or (b) each processor communicates its error value to a leader processor, which checks if convergence has been reached or not, and then communicates it to all other processors. The number of messages involved in the approach (b) is far less than that of approach (a), especially when the number of processors grows. However, (b) requires two communication cycles (all send to one and one send to all), while (a) requires only one cycle (all send to all). This is very important for clusters of workstations, since most of the network technology used in clusters of workstations suffers from high latency. Thus, the time required to perform two communication cycles can be more important than the total number of messages. Here, both approaches are implemented and the performance of each one measured, the results are discussed in the next section.

IV. RESULTS AND DISCUSSION

In order to assess the parallel performance of the algorithm and to evaluate different parallelization strategies, simulations with 1, 2, 4, 8, 16, 32, 48 and 64 processors were performed. The experiments were run on the cluster of the *Laboratório de Computação de Alto Desempenho* of the *Departamento de Informática – UFES* (www.inf.ufes.br/~lcad). This cluster has 64 processors (Athlon 1800+, 128k cache L1, 512k cache L2, 256 Mb RAM) and uses Fast-Ethernet network technology (100MB/s).

A typical CFD application was simulated – "The lid driven cavity problem" – that consists of a square container (whose sides length are equal to 1.0 m) filled with a fluid. The lid of the container moves at a given constant velocity, thereby setting the fluid in motion. No-slip conditions are imposed on all four segments of the boundary except for the upper boundary, along which the velocity u in the x-direction is equal to the given lid velocity u_0, in order to simulate the moving lid.

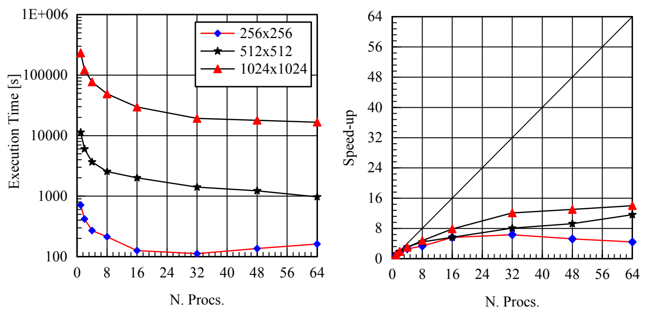
It is important to emphasize that the simulation performed with 1 processor is, in fact, slightly different from those performed with two or more processors, since a truly sequential code was utilized in the 1 processor case. In this way, it was possible to evaluate the real performance gain of using parallel instead of sequential computing.

The algorithm analyses are divided into three different sections. The first section analyses the speed-up obtained with the use of additional processors and the impact of the problem size on the performance. The second and third sections deal with the evaluation of the impact of the inter-processor communication strategies used. As discussed previously, there are two stages of the computational procedure: one that requires communication between all processors, the convergence check after a SOR iteration; and the determination of the value of δt for the next time iteration. The communication required for convergence check can be performed using at least three different mechanisms: (*i*) an *all-to-all* communication, i.e., a broadcast operation between all processors; (*ii*) a *master-slave* communication, where every processor sends a value to a single processor (the master), which, after some computation, sends the data back to all processors; and (*iii*) a binary-tree, where communications are combined pair-wise to yield a single value, which is sent back to all processor using the same binary tree. In the second section is studied the impact of the different communication strategies used to communicate the data necessary to the convergence check. In the third section, the frequency at which the communication between processors should occur is evaluated. Griebel et al. [1] proposed that a communication step should be performed after each SOR iteration, so that the values at the boundaries get an update at every SOR iteration. This procedure may introduce a significant amount of communication, which may slow down the computation. An alternative procedure is to perform a few SOR iterations prior to communication. The results presented below show that this can considerably reduce the amount of communication without slowing down the convergence rate.

A.1 Speed-up results

The performance evaluation was based on the execution time and the speed-up provided by the increase of the number of processors used in the computations. Fig. 5a presents the execution time for simulations with 3 different meshes: 256×256, 512×512 and 1024×1024, which represents approximately 65×10^3, 260×10^3 and 1×10^6 nodal points, respectively. It is possible to note a considerable reduction in the execution time for all 3 meshes as the number of processors increases (the execution time axis is in a logarithmic scale).

Fig. 5b compares the speed-up results with the ideal speed-up, which represents a linear reduction of the computation time as the number of processors increases. It is possible to note that the speed-up obtained for the 256×256 mesh is far from the ideal speed-up. When two or four processors are used, there is a significant gain in performance, but not a linear gain. As the number of processors increases this difference between achieved and ideal speed-ups becomes more evident. This happens because, when the number of processors increases, the block size decreases. Although the number of nodal points is divided equally among processors, when the block size is small, the nodal points of each block that require communication represent a large proportion of the total number of nodal points in each block. Since the network throughput is limited, each processor spends a significant amount of time waiting for information coming from others in this case. The smaller the number of nodal points on a sub-domain, the faster a processor is able to perform the do-loops operations assigned to it, so that the amount of time spent on communication becomes more significant. This fact is even more noticeable when the number of processors is increased and the proportion of nodal points of each computation block requiring communication becomes larger. As shown in Fig. 5a, the computing time of the 256x256 mesh with 48 processors is slightly larger than that with 32 processors – this tendency is more pronounced for smaller meshes. Thus, larger meshes tend to present larger speed-ups and a more efficient use of the additional processors in the system.

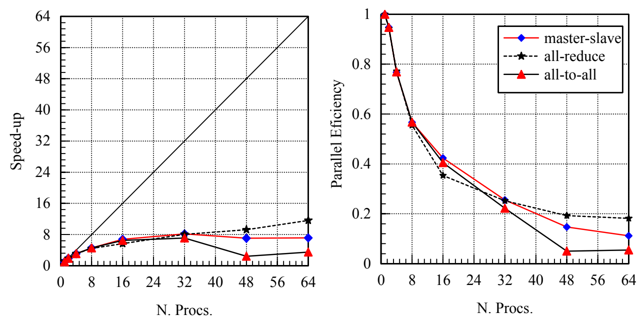


(a) (b)

Fig. 5 – (a) Execution times and (b) speed-up's obtained for meshes of 256×256, 512×512 and 1024×1024 nodal points, with simulations running on 1, 2, 4, 8, 16, 32, 48 and 64 processors.

A.2 Communication strategy

As described earlier, there are two stages in the computational procedure that require communication between all processors. For instance, in the computation of the time step size for the next time iteration, every processor calculates it own maximum allowed local time step size (eq. 9); however, the time step value used for the next time iteration needs to be the smallest value among all processors. As such, this operation can be performed at least in 3 different manners. In the first, every processor communicates its time step size to any other processor in the system and, then, each processor calculates the smallest value and selects it as the global time step size for the next time iteration. This approach requires *all-to-all* or broadcast communication, which might cause network contention, since $n(n-1)$ messages should be sent (where n is the number of processors involved in the computation). In the second, every processor sends its time step size to a master processor, which selects the smallest value and broadcast it back to all processors. This procedure involves a considerable reduction on the number of messages sent ($2n$), but has to be performed in two stages, which may cause some slow down due to the latency of the network. This procedure will be referred to as *master-slave*. In the third approach, the MPI "*all-reduce*" function is used. This performs all communications required using a binary tree, i.e., communications are combined pair-wise to yield a single corresponding result of the operation in the binary tree root processor receive buffer, which is then forwarded back to all processors using the same binary tree. The use of the *all-reduce* function slightly decreases the number of messages to $2(n-1)$; however, avoiding network contention, and distributing the computation of the smallest value among all processors. Nevertheless, since a binary tree communication algorithm is used, the number of communication stages is equal $log_2 n$, i.e., a 32 processor *all-reduce* operation will be carried out using 5 communication steps.



(a) (b)

Fig. 6 – (a) Speed-up and (b) parallel efficiency obtained for communication: *master-slave, all-to-all* and *all-reduce*, with simulations running on 1, 2, 4, 8, 16, 32, 48 and 64 processors.

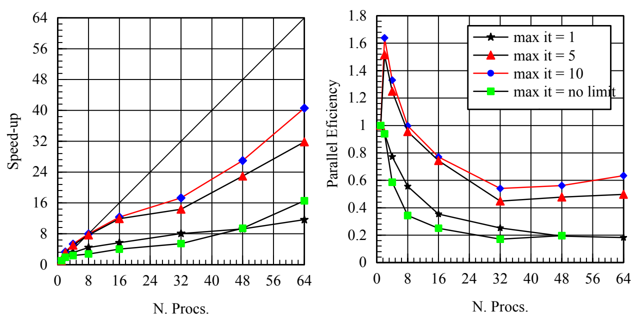
Thus it is important to verify which procedure is more efficient on the computations of this class of problem. Fig. 6 shows the speed-up and parallel efficiency obtained for *master-slave*, *all-reduce and all-to-all* with simulations running on 1, 2, 4, 8, 16, 32, 48 and 64 processors for a 512×512 mesh. The results indicate that there is no significant difference between implementations for simulations running on less than 32 processors. In fact, the results for the *all-reduce* implementation displays a marginally lower efficiency for 16 processors, which may be related to the increased number of communication steps required to perform the communication. However, as the number of processors reaches 32, the advantage of an *all-reduce* implementation is quite noticeable. As the number of sub-domains in the computation increases, the number of messages becomes an important limiting factor and tends to slow down the computation. It can be noted that an *all-to-all* exhibits the worst marks of parallel efficiency for large number of processors, while the *master-slave* implementation obtains middling results.

A.3 Frequency of communication

As stated previously, Griebel et al. [1] proposed that, after each SOR iteration, a communication step should be performed, so that the values at the boundary get an update at every SOR iteration. Although this procedure ensures a fast convergence rate, due to a strong coupling between sub-domains, this may also introduce a significant amount of communication, which may slow down the computation.

Alternatively, one could perform a few SOR iterations prior to each communication, reducing the amount of communication during computation. However, the reduction of the frequency in communication reduces the coupling between sub-domains, which may reduce the convergence rate yielding to a larger number of global iterations. Thus, the choice of the communication frequency is a compromise between the coupling of the solution of the sub-domains and the network contention due to the high volume of messages. In fact, one wishes to communicate as often as possible to ensure a fast convergence but without causing network overload.

In order to evaluate the frequency at which the communication between processors should occur, the parallel performance was measured as the number of SOR iterations prior to communication was varied. Fig. 7 shows the speed-up and parallel efficiency obtained by various communication frequencies, with simulations running on 1, 2, 4, 8, 16, 32, 48 and 64 processors for a 512×512 mesh. The number of SOR iterations between communications was varied from 1 to the extreme case where complete convergence of the SOR procedure was reached on a sub-domain prior to communication. It is important to note that, on this test, the communication step is performed if a maximum iteration count is reached or if local convergence on a sub-domain is reached.



(a) (b)

Fig. 7 – (a) Speed-up and (b) parallel efficiency obtained for various communication frequencies, setting the number of SOR iterations between communications to 1, 5, 10 SOR iterations or until local convergence is reached, i.e., no iteration count limit.

The results indicate a significant reduction of the execution time by reducing the communication frequency, especially for larger numbers of processors. For instance, by using 64 processors a speed-up of approximately 40 was obtained by performing 10 SOR iterations prior to communication, while, for the simulation where a communication step was performed for each SOR iteration, the speed-up obtained was close to 12. For the simulation where no limit for the number of SOR iterations prior to communication was set, it possible to note that the speed-up obtained is only marginally better than the speed-up for the simulation where a communication step was performed for each SOR iteration.

One interesting feature of the speed-up and parallel efficiency curves obtained is that the speed-up for the simulation performing 5 and 10 SOR iterations prior to communication for 2 and 4 processors was slightly superior to the ideal speed-up, yielding to a parallel efficiency larger than one. This is probably related to a more efficient use of the processor cache. Although there is a tendency for a slow down of the computation due to communication, the memory size required for the computation of each block was considerably smaller. Thus, dividing the 512×512 mesh into 2 or 4 blocks, enables a more efficient use of the processor L2 cache, reducing the number of cache misses. Since the access time of the cache memory is 5 to 10 times faster that that of the conventional memory, each processor performs the computation on its own sub-domain slightly faster, reducing the total computing time. However, as the domain is further divided, the increase of performance provided by the more efficient use of the processor's memory hierarchy is not sufficient to avoid the performance degradation due to communication.

The larger the number of SOR iterations prior to communication, the larger is the cache hit ratio during the computation, and thus, larger is the MFLOPS produced by the machine's processors. However, it is important to note that the relationship between execution time and MFLOPS achieved is not straightforward. An increase in the number of SOR iterations prior to communication will result in an increase in the MFLOPS achieved, but also in a reduction of the coupling between sub-domains. For instance, a simulation performed with 10 SOR iterations prior to communication has reached a sustained performance of 2.4 GFLOPS on 32 processors, while a simulation performed with 100 SOR iterations prior to communication has reached a sustained performance of 3.4 GFLOPS on 32 processors. In spite of the smaller number of GFLOPS, the simulation with 10 SOR iterations prior to communication obtained a smaller execution time (656s for 10 SOR iterations vs. 2915s for 100 SOR iterations), since the number of global iterations per time step was smaller.

V. Conclusion

A numerical algorithm for simulating incompressible fluid flows was presented. The algorithm was based on the finite difference method and was designed for parallel computing platforms with distributed-memory, particularly for clusters of workstations.

In order to assess the parallel performance of the algorithm and to evaluate different parallelization strategies, simulations with 1, 2, 4, 8, 16, 32, 48 and 64 processors were performed. Where communication between all processors was required, three communication strategies were evaluated to deal with the communication stages: (*i*) an *all-to-all* communication, i.e., a broadcast operation between all processors, (*ii*) a *master-slave* communication, where every processor sends a value to a single processor (the master), which, after some computation, sends the data back to all processors, and (*iii*) a binary-tree, where communications are combined pair-wise to yield a single value, which is the sent back to all processor using the same binary tree. The results indicated that there is no significant difference between implementations for simulations running on less than 32 processors. Nevertheless, the binary-tree approach, implemented via the MPI function "*all-reduce*" outperformed the other methodologies by a large margin as the processor number increases.

The frequency at which the communication between processors should occur was also investigated. In the simulations performed, the best results where obtained for a limit of 10 SOR iterations prior to communication. However, it is the authors' opinion that the optimum number is a function of the nature of the problem studied, and further investigation is required on this subject to automatically identify optimum parameters for each class of problem.

References

[1] Griebel, M., Dornseifer, T. & Neunhoeffer, T., 1998, Numerical Simulation in Fluid Dynamics – A Pratical Introduction, SIAM, Philadelphia.

[2] Harlow, F. & Welch, J., 1965, Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface, Phys. Fluids, vol. 8, pp. 2182-2189.

[3] Reis Jr., N. C., Griffiths, R. F., Roberts, E. P. L. , 1998, Finite Volume Method to Solve Free-Surface Fluid Flow Problems, Numerical Methods on Computational Fluid Dynamics VI, Oxford, p.475 – 483.

[4] Hirt, C., Nicolas, B., & Romero, N., 1975, SOLA – A Numerical Solution Algorithm for Transient Fluid Flows, Technical report LA-5852, Los Alamos, NM: Los Alamos National Lab.

[5] Streng, M., 1996, Load Balancing for Computational Fluid Dynamics Calculations, in High Performance Computing Fluid Dynamics, ed. P. Wesseling, Kluwer Academic Publishers.

[6] Anderson Jr., J. D., Computational Fluid Dynamics – The Basics with Applications, McGraw Hill Inc., 1995.