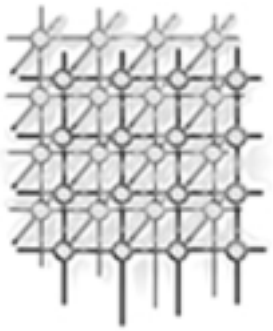
CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **0**:1–20 *Prepared using* cpeauth.cls *[Version: 2000/05/12 v2.0]*

A parallelization technique applied to the computation of rotating flows in cavities



Rudnei Dias da Cunha*,† and Álvaro Luiz de Bortoli‡

Postgraduate Programme on Applied Mathematics, Institute of Mathematics, Federal University of Rio Grande do Sul, BRAZIL

SUMMARY

In this paper, we investigate the parallel solution of rotating internal flow problems, using the Navier-Stokes equations as proposed by Speziale and Thangam (1983) and Speziale (1985). A Runge-Kutta time-stepping scheme was applied to the equations and both sequential and message-passing implementations were developed, the latter using `MPI` , and were tested on a 4-processor SGI Origin200 distributed, global shared memory parallel computer and on a 32-processor IBM 9076 SP/2 distributed memory parallel computer. The results show that our approach to parallelize the sequential implementation requires little effort whilst providing good results even for medium-sized problems.

KEY WORDS: Parallel computing; Message-passing; Computational Fluid Dynamics

Introduction

Speziale and Thangam [19] and Speziale [18] have developed a formulation for the problem of rotating internal flows, i.e. determining the behaviour of "pressure-driven laminar flows in straight ducts, subjected to a steady spanwise rotation" (see [18]). The governing equations are the Navier-Stokes equations and the continuity equation in a rotating framework, which can be written as follows

$$\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} = -\frac{1}{\rho} \nabla P + \nu \nabla^2 \vec{v} - 2\Omega \times \vec{v} \tag{1}$$

*Correspondence to: Postgraduate Programme on Applied Mathematics, Institute of Mathematics, Federal University of Rio Grande do Sul, Av. Bento Gonçalves 9500 - Agronomia - Prédio 43111-A - 91501-970 Porto Alegre - RS, BRAZIL
†E-mail: rudnei@mat.ufrgs.br
‡E-mail: dbortoli@mat.ufrgs.br

Received March 2000
Revised September 2000
Accepted October 2000

$$\nabla \cdot \vec{v} = 0 \tag{2}$$

where $\vec{v}$ is the velocity vector, P is the modified pressure which includes both the gravitational and centrifugal force potentials, Ω is the steady spanwise rotation, ρ is the density of the fluid and ν is the kinematic viscosity of the fluid. The axial pressure gradient $\partial P/\partial z = -G$ is constant.

For a nonzero rotation rate, the velocity vector is of the form $\vec{v} = u(x,y)\vec{i}+v(x,y)\vec{j}+w(x,y)\vec{k}$, w being the axial velocity and u and v representing the secondary flow. As the rotation is around $\vec{j}$, it is of the form $\Omega = \Omega\vec{j}$; since the flow properties are independent of z, equations (1) and (2) may be written in component form as

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + \nu\nabla^2 u - 2\Omega w \tag{3}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial P}{\partial y} + \nu\nabla^2 v \tag{4}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} = \frac{G}{\rho} + \nu\nabla^2 w + 2\Omega u \tag{5}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{6}$$

Due to the simplified form of the mass conservation equation (6), it follows that a secondary flow stream function ψ exists such that the velocity components are

$$u = -\frac{\partial \psi}{\partial y}, \quad v = \frac{\partial \psi}{\partial x}. \tag{7}$$

The function ψ is the solution to Poisson's equation $\nabla^2\psi = \partial v/\partial x - \partial u/\partial y = \zeta$, where ζ is the axial component of the velocity vector and is expressed by

$$\frac{\partial \zeta}{\partial t} + u\frac{\partial \zeta}{\partial x} + v\frac{\partial \zeta}{\partial y} = \nu\nabla^2\zeta + 2\Omega\frac{\partial w}{\partial y} \tag{8}$$

Introducing the velocity and length scales W_0 and D, the equations to be solved numerically are written as follows:

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} = C + \frac{1}{Re}\nabla^2 w + 2Ro\,u \tag{9}$$

$$\frac{\partial \zeta}{\partial t} + u\frac{\partial \zeta}{\partial x} + v\frac{\partial \zeta}{\partial y} = \frac{1}{Re}\nabla^2\zeta + 2Ro\frac{\partial w}{\partial y} \tag{10}$$

$$\nabla^2\psi = \zeta \tag{11}$$

$$u = -\frac{\partial \psi}{\partial y}, \quad v = \frac{\partial \psi}{\partial x} \tag{12}$$

where Re and Ro are the Reynolds and Rossbi numbers, C is the dimensionless pressure gradient, and the following relationships hold:

$$Re = W_0 D/\nu, \quad Ro = \Omega D/W_0, \quad C = GD/\rho W_0^2. \tag{13}$$

The initial condition for w, i.e. a non-rotating flow, satisfies the equation

$$\nabla^2 w = -\frac{G}{\rho \nu} \tag{14}$$

The reader is referred to [18] for more details.

The set of equations (9)-(12) are to be computed numerically and a rectangular grid of $M \times N$ points, $M > N$ are placed over a duct with length D and height H ($H > D$), with spacings $h_x = D/(M+1)$ and $h_y = H/(N+1)$; therefore we may refer to the variables of interest in those equations for an specific time-step t as discrete points on the grid with indices i, j along the vertical and horizontal directions respectively.

Boundary conditions are $u = 0, v = 0, w = 0$ and $\psi = 0$ on the walls of the duct; also, by taking a Taylor's expansion of (11), boundary conditions on the axial vorticity at time-step t are obtained and written as

$$\zeta_{i,0}^t = \frac{8\psi_{i,1}^t - \psi_{i,2}^t}{2h_x^2}, \quad \zeta_{i,M+1}^t = \frac{8\psi_{i,M+1}^t - \psi_{i,M}^t}{2h_x^2} \tag{15}$$

$$\zeta_{0,j}^t = \frac{8\psi_{1,j}^t - \psi_{2,j}^t}{2h_y^2}, \quad \zeta_{N+1,j}^t = \frac{8\psi_{N+1,j}^t - \psi_{N,j}^t}{2h_y^2} \tag{16}$$

In this work, we have used a modified explicit Runge-Kutta time-stepping integration scheme, approximating time and spatial derivatives by forward and central finite-differences respectively. Other authors (see [15, 17, 13]) have used different approaches for fluid dynamics simulations, mainly finite-element discretizations and implicit time-stepping integration schemes, and the parallelization is expressed via domain-decomposition.

Description of the explicit method

The computation is divided in two main parts. Initially, we solve Equation (14) for w with an iterative method, writing the Laplacian in central finite-differences and obtaining the value of w at the (i, j) cell from

$$w_{i,j}^{k+1} = \delta \left(h_x^2 h_y^2 \frac{G}{\rho \nu} + h_y^2 (w_{i,j-1}^k + w_{i,j+1}^k) + h_x^2 (w_{i-1,j}^k + w_{i+1,j}^k) \right) \tag{17}$$

where $\delta = (2h_x^2 + 2h_y^2)^{-1}$, $i = 1, 2, \ldots, N$, $j = 1, 2, \ldots, M$, $k = 0, 1, \ldots, k_{max}$, and we proceed with the iterations until

$$|| \, w^{k+1} - w^k \, ||_\infty < \epsilon_w \tag{18}$$

where ϵ_w is sufficiently small.

The second part is the solution of equations (9)-(12) which is made using a modified explicit Runge-Kutta time-stepping scheme [11, 6, 5]. We proceed from time-step t to $t+1$ as follows:

$$w_{i,j}^0 = w_{i,j}^t, \zeta_{i,j}^0 = \zeta_{i,j}^t, u_{i,j}^0 = u_{i,j}^t, v_{i,j}^0 = v_{i,j}^t \tag{19}$$

$$w_{i,j}^k = w_{i,j}^0 + \alpha_k h_t \left(C + \frac{1}{Re}\nabla^2 w_{i,j}^{k-1} + 2Ro\, u_{i,j}^{k-1} - u_{i,j}^{k-1}\frac{\partial w_{i,j}^{k-1}}{\partial x} - v_{i,j}^{k-1}\frac{\partial w_{i,j}^{k-1}}{\partial y} \right) \tag{20}$$

$$\zeta_{i,j}^k = \zeta_{i,j}^0 + \alpha_k h_t \left(\frac{1}{Re}\nabla^2 \zeta_{i,j}^{k-1} + 2Ro\frac{\partial w_{i,j}^k}{\partial y} - u_{i,j}^{k-1}\frac{\partial \zeta_{i,j}^{k-1}}{\partial x} - v_{i,j}^{k-1}\frac{\partial \zeta_{i,j}^{k-1}}{\partial y} \right) \tag{21}$$

$$\nabla^2 \psi^k = \zeta^k \tag{22}$$

$$u_{i,j}^k = -\frac{\partial \psi_{i,j}^k}{\partial y}, \quad v_{i,j}^k = \frac{\partial \psi_{i,j}^k}{\partial x} \tag{23}$$

$$w_{i,j}^{t+1} = w_{i,j}^K, \zeta_{i,j}^{t+1} = \zeta_{i,j}^K, u_{i,j}^{t+1} = u_{i,j}^K, v_{i,j}^{t+1} = v_{i,j}^K \tag{24}$$

where h_t is the time-step length, $k = 1, 2, \ldots, K$ (K is the order of the Runge-Kutta scheme) and $\alpha^{(K)} = \{\alpha_k\}_{k=1}^K$ is the set of weights for the integration, with $\alpha^{(3)} = \{1/2, 1/2, 1\}$, $\alpha^{(4)} = \{1/4, 1/3, 1/2, 1\}$ and $\alpha^{(5)} = \{1/4, 1/6, 3/8, 1/2, 1\}$ (see [6]). All derivatives appearing in equations (20)-(23) are replaced by central finite-differences.

To stabilize the computation, the time-step h_t is chosen such that at each iteration it satisfies the condition

$$h_t \leq (2\nu(h_x^{-2} + h_y^{-2}) + ||\, u\, ||_\infty / h_x + ||\, v\, ||_\infty / h_y)^{-1} \tag{25}$$

Now the iterations in t proceed until

$$||\Delta(w)||_\infty + ||\Delta(\zeta)||_\infty + ||\Delta(\psi)||_\infty + ||\Delta(u)||_\infty + ||\Delta(v)||_\infty < \epsilon \tag{26}$$

where $\Delta(f)$ denotes $f^{t+1} - f^t$ and ϵ is sufficiently small. In case this tolerance has not been achieved, the boundary conditions on the axial vorticity ζ are updated, using (15) and (16), and another iteration is performed. The algorithm used is outlined below:

Algorithm 1. *Rotating flow algorithm*

1. initialize constants, boundary conditions, etc.
   for $k = 1, 2, \ldots, kmax$
2. compute (17)
3. **if** $||\, w^{k+1} - w^k\, ||_\infty < \epsilon_w$
   then break;
   endfor

for $t = 0, 1, \ldots, t_{max}$
  for $k = 1, 2, \ldots, K$
4. compute (20)-(23)
  endfor
5. **if** $||\Delta(w)||_\infty + ||\Delta(\zeta)||_\infty + ||\Delta(\psi)||_\infty + ||\Delta(u)||_\infty + ||\Delta(v)||_\infty < \epsilon$
    then break;
6. update boundary conditions on ζ using equations (15)-(16)
7. correct h_t using equation (25)

endfor
8. output the results.

A review of related work on parallelizing Navier-Stokes computations

As the solution of the Navier-Stokes equations is a computing-intensive task, requiring large amounts of memory and presenting very long run times, several authors have worked on its parallelization. In what follows we will attempt to give a brief overview of past related work.

Emerson [7] reports on the parallelization of a 2-D Navier-Stokes solver, using a finite-volume discretization. The code used an early message-passing library, `FORTNET`, and domain decomposition was used to partition the data, the subdomains being then assigned to the processors. The best results were obtained on an Intel iPSC/860 hypercube, with a speed-up of 2.5 on 4 processors.

Roose and Van Driessche [16] report on several approaches to used distributed-memory parallel computers on CFD computations. For explicit time integration, similar to ours, they mention that one can reduce the communication overhead of a Runge-Kutta solver if the update of the overlapping regions (i.e., those data residing on neighbouring processors that are needed for the local computations) are updated only after a complete time-integration step is performed, rather than at each Runge-Kutta k-step. This may lead to a worse convergence and perhaps divergence, depending on the problem. We have not followed this approach, as will be seen in the sequel, as we have found that a larger number of iterations would be required, increasing the overall run time.

Satofuka et al. [17] give results for a parallel Navier-Stokes solver on transputer arrays and a workstation network. Their solver is based on the method of lines and the parallel algorithm is implemented using `PVM`. Speed-ups of up to 3 over 4 processors were attained.

Vatsa and Faulkner [20] have recently ported the parallel version of the TLNS3D solver, which uses `MPI` , to an Ethernet-connected cluster of 36 dual-processor Pentium Pro personal computers. They have used both synchronous and asynchronous communication and report that the latter is "preferred as the number of compute nodes is increased". The results obtained are good, with a speed-up of 17 over 25 nodes for a $289 \times 65 \times 49$ grid; though this is lower than that obtained on a SGI Origin2000 (23 over 25 processors), they argue that the considerably lower cost of the cluster of PCs far outweigh the gains in run time.

Lundin [14] solves the time-dependent flow of a rotating incompressible fluid using the velocity-vorticity formulation of the Navier-Stokes equations in cylindrical coordinates,

obtaining an overdetermined set of linear equations at each time-integration step. The corresponding least-squares problem is solved with a preconditioned conjugate-gradients method. The parallel implementation uses `MPI` and data is exchanged in "ready-mode" with persistent communications, to reduce the latency. This approach is equivalent to using asynchronous communication, though one has to assure that the process will be awaiting the data when the transfer is issued. Speed-ups of approximately 9 on a 16-processor IBM SP/2 (67MHz "thin node" processors) and 26 on a 32-processor SGI Origin2000, for a $63 \times 66 \times 129$ grid, are obtained.

Parallelization of the method

The parallelization of an explicit method such as that described previously requires a careful analysis of the equations in use in order to ascertain the relationships between the variables involved, since this will determine the flow of data in the code and between the many processors collaborating in the parallel computation.

The parallel algorithm developed is based on the *single-program, multiple-data, SPMD* paradigm and we consider that the number of processors available, p, is less than the number of computational cells ($M \times N$). No assumption is made with regard to as how the processors are interconnected, though both *point-to-point* communication and *reduction* operators are supposed to be available.

To partition the grid among the processors, we consider that since the domain is regular, the only major requirement to attend is that we must partition it across the largest dimension, thereby increasing the computation-communication ratio and leading to a potential good parallel performance. In our case, we divide the domain across M (as $M > N$), obtaining $m = M/p$ panels of N cells, and assign to each one of p processors $m \times N$ contiguous cells; if M is not an integer multiple of p, then one extra row of N cells is assigned to some r processors, where r is the modulus of M/p (i.e. these processors will store panels of size $(m+1) \times N$). This partitioning leads to a *logical* interconnection of the processors as if they were on a linear array (a topology which can be easily embedded on other physical interconnections available in parallel computers, like hybercubes, 2D/3D grids and others).

While other strategies could be followed to achieve load-balance between the processors, this one makes the communication pattern regular, as each processor has to exchange at most two rows of N cells with its two neighbours (or a single row if it is at one of the ends of the linear array).

It should be stressed that though the partitioning by panels is very simple, it can be used on a variety of other problems, including those involving complex geometries, if the problem is recast using generalized coordinates and type *C*- and *O-grids* (see [9, V.2,Ch.12] and [8]).

Analysing the flow of data between equations (20)-(23), we note that there is a feedback mechanism in the overall Runge-Kutta scheme, as once a variable in the k-th step is produced, it is used in the computation of the next variable in sequence. This mechanism implies the need of data exchange between the processors inside the Runge-Kutta scheme, in order to compute the finite-differences approximations to the derivatives.

Thus, once every processor has computed w^k in their assigned portion of cells using (20), they swap their left- and right-most columns of cells of w^k (say, $w_{:,1}$ and $w_{:,m}$, where the colon indicates a whole column) and also of ζ^{k-1} with their left and right neighbours (this is done in a single message of length $2N$ instead of two messages of length N to reduce the effect of message-passing latencies in the performance of the algorithm). Every processor is then able to compute ζ^k in their cells; afterwards, they exchange ζ^k in the same way in order to solve Poisson's equation for ψ^k and once this is completed, ψ^k is exchanged in order to compute the velocities u^k and v^k. Therefore in every step of the Runge-Kutta scheme there are three data-exchanges between neighbouring processors.

Also, note that the initial condition for w is the solution of Equation (14); as it is solved in the form (17), the computation can be organized such that one data exchange is suppressed, since for the first iterations (i.e. for $k = 0$ at $t = 0$), a processor will have already received columns $w_{:,m}$ and $w_{:,1}$ from its left and right neighbouring processors, which has been done in the last iteration prior to convergence using (17).

The boundary conditions on u, v, w, ψ and ζ can be computed without any communication due to their simple form. For the update of the boundary conditions on ζ – equations (15) and (16) – we use the same approach as explained in the previous paragraph, since every processor will have stored $\zeta_{:,m}^{t+1}$ and $\zeta_{:,1}^{t+1}$ of its left and right neighbours, from the last Runge-Kutta iteration.

The whole algorithm is organized by dividing the computation of equations (20)-(23) into two parts: one that refers to data stored locally in a single processor, and another which depends on the local availability of data stored in its neighbouring processors. If now we make use of *asynchronous* point-to-point communications (as present in `MPI` [10]), then we can compute any one of the variables involved using the following algorithm:

Algorithm 2. *Parallel computation of a variable f*

1. asynchronously send variable f to its left and right neighbours
2. compute variable f with its local data, i.e. from columns 2 to $m-1$
3. request the m-th column of f from processor $p-1$ (left) and store locally into column 0 of f
4. request the 1-st column of f from processor $p+1$ (right) and store locally into column $m+1$ of f
5. compute columns 1 and m of variable f

It is then possible to almost completely hide the time spent communicating between two processors, provided the amount of time spent in step 2 of the above algorithm is greater than the time needed for the two point-to-point communications between a processor and its two neighbours. A sufficiently large grid will allow this to happen; in our specific case, we are interested when the ratio M/N is large, since that will *maximize* the amount of local computation for a given p while keeping small (relative to the local computation) the communication time. This approach has been successfully applied in other parallel applications (see [2], [3], [4] and [1]).

However, there are two penalties brought about by the parallel computation of equations (20)-(23):

1. The amount of time needed to set-up the asynchronous sends and the retrieval of data from the local communications buffer into the appropriate memory locations of the user's program;
2. The computation of *reductions*, needed to obtain the norms used in the stopping criteria of the iterations.

We can not hide these times within the local computation time and therefore they are the main causes for being unable to achieve the optimal speed-up; but we may expect that by dividing the storage of the variables among p processors, the use of processors with cache memories will provide some interesting phenomenon for large size problems and small number of processors.

The parallel algorithm can now be described as follows. Each processor stores its m columns of N cells, for each variable (w, ζ, ψ, u and v) in arrays of size `0:(N+1),0:(m+1)`, where the two extra rows and columns serve to hold the boundary conditions values. Due to the simple form of some of the boundary conditions specified, one could argue that it is not needed to store them; however this would lead to a specific piece of code be written to compute the equations in the cells where the boundary conditions are involved.

We wrote the code to compute each of the equations (20)-(23) as a pair of loops scanning the columns and rows of the array holding the variable values at the cells. As an example, we will show how a sequential code to compute $\psi^k_{i,j}$ in Equation (20) was transformed into a parallel code according to Algorithm 2. The sequential code is as follows

```
 DO J = 1,M
    DO I = 1,N
       DXPSI = PSI(I,J-1) + PSI(I,J+1)
       DYPSI = PSI(I-1,J) + PSI(I+1,J)
       PSINEW(I,J) = APSI*ZETANEW(I,J) + BPSI*DXPSI + CPSI*DYPSI
    END DO
 END DO
```

where `APSI`, `BPSI` and `CPSI` are constants involving h_x and h_y, derived from the central finite-differences equations. Its equivalent parallel version, using `MPI` , is

```
* 1. Asynchronously send PSI to its left and
*    right neighbours
 CALL SNDRCV(MYID,P,PSI,NP1,MP1,900,1000,IDSND,IDRCV)

* 2. Compute PSI with its local data
 DO J = 2,MYM-1
    DO I = 1,N
       DXPSI = PSI(I,J-1) + PSI(I,J+1)
       DYPSI = PSI(I-1,J) + PSI(I+1,J)
       PSINEW(I,J) = APSI*ZETANEW(I,J) + BPSI*DXPSI + CPSI*DYPSI
    END DO
 END DO

* 3-4. Request columns from neighbouring processors
 CALL GETDATA(MYID,NPROCS,IDSND,IDRCV)

* 5. Compute columns 1 and m of PSI
```

```
DO J = 1,MYM,MYM-1
   DO I = 1,N
      DXPSI = PSI(I,J-1) + PSI(I,J+1)
      DYPSI = PSI(I-1,J) + PSI(I+1,J)
      PSINEW(I,J) = APSI*ZETANEW(I,J) + BPSI*DXPSI + CPSI*DYPSI
   END DO
END DO
```

where `MYM` is m and `SNDRCV` and `GETDATA` are subroutines which call the `MPI` routines `MPI_ISEND` and `MPI_IRECV`, and `MPI_WAIT` respectively. Note that by using the `MPI_ISEND` and `MPI_IRECV` routines we have an asynchronous parallel implementation which maximizes the use of the processors. A fully asynchronous implementation, on the other hand, is not possible, for the underlying numerical method can not cope with the nonlinear instabilities that may be generated by that kind of implementation. We must stress the fact that our parallel algorithm is synchronous, only the communication is done asynchronously.

With this approach, once a sequential version of the code has been tested and certified to be producing the desired results, it is easy to obtain its parallel version, since the second pair of `DO` loops is the same as the first, apart from the indices on `J`. It is less error-prone, since the loop body remains unchanged; in fact, if the first pair is encapsulated in a subroutine, having the indices on `J` as parameters, then if a modification in the body of the loops was required, just a single part of the code would need attention. As for the performance of such code, if one uses a compiler which is capable of inlining a subroutine, then it will not be affected by this approach.

Another possible way of writing the parallel code (which we have also done) would be to provide three different parts to handle the computation, depending on the position of each processor: the first, the last, and those in the middle of the linear array. It is easy to see that this would increase three-fold the size of the code, and make it even more difficult to maintain; one could make use of subroutines which would certainly make the code more readable but, for an efficient program execution, the subroutines should be inlined, thereby increasing the object code size accordingly. As an example, with the first approach, the ratio of source code sizes of the parallel to the sequential versions is $1.46 : 1$, whereas for this latter approach it was $3.28 : 1$.

The other modification required in the sequential code to produce the parallel version is in the computation of the norms. This requires a reduction operation over several values (i.e. the partial norms) stored in the processors. Due to the *SPMD* programming model used, the reduced value (i.e. the ∞-norm of a variable) is required to be present in every processor. Therefore, a reduction, followed by a *broadcast* of the reduced value to all processors is employed, this being implemented by the `MPI` routine `MPI_ALLREDUCE`. It is a costly operation; note that the reduction and broadcast require sending/receiving several messages between the cooperating processors (with the associated latencies to set-up the message transfers), albeit some of those may be done in parallel.

As such, we look at Algorithm 1 and notice that norms are required in steps 3, 5 and 7. Now we ask ourselves: can we combine the reductions in the last two steps into a single one, therefore reducing the latencies? If we consider that the computational cost of a reduction of r values followed by a broadcast is $2\lfloor \log_2 p \rfloor(\alpha + r\beta)$, where α and β are the latency and the

transfer rate between two processors (directly related to each other), then it is easy to see that if we combine the five reductions needed in step 5 with the two reductions in step 7, we will be saving one latency per reduction. For the overall computation, we will have

$$2t_{\max}(\alpha \lfloor \log_2 p \rfloor + 7\beta) < 2t_{\max}(2\alpha \lfloor \log_2 p \rfloor + 7\beta) \tag{27}$$

and the savings will be greater for large p, as we shall see in §.

With the above reasoning, if the sequential code corresponding to the computation of steps 5 and 7 is written as

```
* 5. Compute norms
 DO I = 1,5
    NORMS(I) = 0.0
 END DO
 DO J = 1,M
    DO I = 1,N
       NORMS(1) = MAX(NORMS(1),ABS(W(I,J)-WNEW(I,J)))
       NORMS(2) = MAX(NORMS(2),ABS(ZETA(I,J)-ZETANEW(I,J)))
       NORMS(3) = MAX(NORMS(3),ABS(PSI(I,J)-PSINEW(I,J)))
       NORMS(4) = MAX(NORMS(4),ABS(U(I,J)-UNEW(I,J)))
       NORMS(5) = MAX(NORMS(5),ABS(V(I,J)-VNEW(I,J)))
    END DO
 END DO

 NORM = NORMS(1) + NORMS(2) + NORMS(3) + NORMS(4) + NORMS(5)

* 6. Update boundary conditions on ZETA
 ...

* 7. Time-step stabilization test
 NORMS(1) = 0.0
 NORMS(2) = 0.0
 DO J = 1,M
    DO I = 1,N
       NORMS(1) = MAX(NORMS(1),ABS(UNEW(I,J)))
       NORMS(2) = MAX(NORMS(2),ABS(VNEW(I,J)))
    END DO
 END DO
 MAXHT = 2.0*NU*(INVHXSQ+INVHYSQ) + INVHX*NORMS(1) + INVHY*NORMS(2)
 HT = MIN(HT,1.0/MAXHT)
```

then an equivalent parallel code, including the computation of the $||\, u \,||_\infty$, $||\, v \,||_\infty$ needed in step 7, is

```
* 5. Compute norms
 DO I = 1,7
    NORMS(I) = 0.0
 END DO
 DO J = 1,MYM
    DO I = 1,N
       NORMS(1) = MAX(NORMS(1),ABS(W(I,J)-WNEW(I,J)))
       NORMS(2) = MAX(NORMS(2),ABS(ZETA(I,J)-ZETANEW(I,J)))
```

```
      NORMS(3) = MAX(NORMS(3),ABS(PSI(I,J)-PSINEW(I,J)))
      NORMS(4) = MAX(NORMS(4),ABS(U(I,J)-UNEW(I,J)))
      NORMS(5) = MAX(NORMS(5),ABS(V(I,J)-VNEW(I,J)))
      NORMS(6) = MAX(NORMS(6),ABS(UNEW(I,J)))
      NORMS(7) = MAX(NORMS(7),ABS(VNEW(I,J)))
   END DO
END DO

CALL MPI_ALLREDUCE(NORMS,REDUOUT,7,MPI_REAL,MPI_MAX,MPI_COMM_WORLD,IERR)

NORM = REDUOUT(1) + REDUOUT(2) + REDUOUT(3) + REDUOUT(4) + REDUOUT(5)

* 6. Update boundary conditions on ZETA
 ...

* 7. Time-step stabilization test
MAXHT = 2.0*NU*(INVHXSQ+INVHYSQ) + INVHX*REDUOUT(6) + INVHY*REDUOUT(7)
HT = MIN(HT,1.0/MAXHT)
```

where `REDUOUT` is the buffer holding the reduced `NORMS` values and which is present in all processors after the call to `MPI_ALLREDUCE`.

Output of the results

At the end of the overall computation, we save the values of the variables involved in files for later analysis. We consider that each processor has parallel access to the disk filesystem and each processor is thus able to open its own file, all p files being written as simultaneously as possible. In our experiments, even for the large problems, this proved to be efficient and accounted for less then 1% of the run-time. An in-house developed visualization program (see [12]) is later used, which opens the several files in sequence and exhibits the data in a variety of forms (eg. colour maps, particle traces and vector fields).

Theoretical models of computation

In this section, we will derive equations that express the computational cost for the sequential and parallel versions of the code. In the sequel, $C_\bullet$ is the computational cost of a $\bullet$ operation.

Analysing equations (15)-(26), it is possible to count the number of operations required. In our implementation, all constant values involved in those equations have been computed previously and stored in separate scalar variables, thus guaranteeing that no unnecessary floating-point operations will be done.

Sequential version

For the solution of (17), we have a cost of

$$C_w^0 = k_{\max}(3MNC_*) \tag{28}$$

where $k_{\max}$ is the number of iterations required until convergence is obtained.

For the solution of the Navier-Stokes equations, we have the following costs

$$\begin{aligned} C_w &= KMN(8C_+ + 12C_*) && (29)\\ C_\zeta &= KMN(8C_+ + 13C_*) && (30)\\ C_\psi &= KMN(4C_+ + 3C_*) && (31)\\ C_u + C_v &= 2KMN(C_+ + C_*) && (32)\\ C_{||.||} &= 5MNC_+ && (33)\\ C_{\text{B.C.on}\,\zeta} &= 2(M+N)C_+ + 4(M+N)C_* && (34)\\ C_{h_t\ \text{correction}} &= 2C_+ + 2C_* + 2C_\div + C_{1/x} && (35) \end{aligned}$$

where K is the order of the Runge-Kutta integration scheme. Adding the above equations we have

$$\begin{aligned} S_* &= (30KMN + 4M + 4N + 2)C_* && (36)\\ S_+ &= (22KMN + 5MN + 2M + 2N + 2)C_+ && (37)\\ C_{\text{N-S}} &= t_{\max}(S_* + S_+ + 2C_\div + C_{1/x}) && (38) \end{aligned}$$

where $t_{\max}$ is the number of iterations required for convergence.

Adding (28) to (38) and disregarding the terms involving divisions and inversions, the expression for the cost of the sequential version of the code is

$$C_S = t_{\max}(S_* + S_+) + k_{\max}(3MNC_*) \tag{39}$$

Parallel version

For the parallel solution of (17), we have a cost of

$$C_w^0 = k_{\max}\left(\max(3MN/pC_*, C_{\text{comm}}(n)) + C_{\text{red}}(1)\right) \tag{40}$$

where

$$C_{\text{comm}}(n) = \alpha + \beta n \tag{41}$$

is the cost of sending n words between two neighbouring processors, with latency α (in seconds) and rate of transmission β (in seconds/word), and

$$C_{\text{red}}(r) = 2\lfloor \log_2 p \rfloor\, C_{\text{comm}}(r) \tag{42}$$

is the cost of a reduction over p processors of r values, followed by a broadcast.

Equation (40) involves a maximum of two costs due to the organization of the Algorithm 1; for instance, if the workload in each processor is not enough to mask the communication time, then this last dominates the whole computation (degrading the performance).

For the parallel solution of the Navier-Stokes equations, we have the following costs

$$\begin{aligned} C_w &= KMN/p(8C_+ + 12C_*) && (43)\\ C_\zeta &= \max(KMN/p(8C_+ + 13C_*), C_{\text{comm}}(2N) && (44)\\ C_\psi &= \max(KMN/p(4C_+ + 3C_*), C_{\text{comm}}(N)) && (45)\\ C_u + C_v &= \max(2KMN/p(C_+ + C_*), C_{\text{comm}}(N)) && (46)\\ C_{||.||} &= 5MN/pC_+ + C_{\text{red}}(7) && (47)\\ C_{\text{B.C.on}\,\zeta} &= (2(M/p+N)C_+ + 4(M/p+N)C_*) && (48)\\ C_{h_t\ \text{correction}} &= 2C_+ + 2C_* + 2C_\div + C_{1/x} && (49) \end{aligned}$$

where the cost for $C_{\text{B.C.on}\zeta}$ was considered that of the first and last processors, since due to holding the first and last columns of the grid respectively, they have more work to do while computing this correction.

Now in the case when the grid is sufficiently large to offset the point-to-point communication between two neighbouring processors, we may disregard the C_{comm} terms above and adding the equations obtain

$$\begin{aligned} P_* &= (30KMN/p + 4M/p + 4N + 2)C_* && (50)\\ P_+ &= (22KMN/p + 5MN/p + 2M/p + 2N + 2)C_+ && (51)\\ C_{\text{N}-\text{S}} &= t_{\max}\left(P_* + P_+ + 2C_\div + C_{1/x} + C_{\text{red}}(7)\right) && (52) \end{aligned}$$

Again, we disregard divisions and inversions in the above equation and adding C_w^0 to $C_{\text{N}-\text{S}}$, the asymptotical behaviour of the parallel version is given by

$$C_P = t_{\max}\left(P_* + P_+ + C_{\text{red}}(7)\right) + k_{\max}\left(3MN/pC_* + C_{\text{red}}(1)\right) \quad (53)$$

Analysis of scalability

Considering the ratio C_S/C_P i.e. the parallel speed-up (not the optimal speed-up), the scalability of the parallel version with respect to p, per iteration (i.e. $t_{\max} = k_{\max} = 1$), is given by

$$S_P = \left(\frac{S_* + S_+ + 3MNC_*}{pP_* + pP_+ + pC_{\text{red}}(7) + pC_{\text{red}}(1)}\right)p \quad (54)$$

and since the constant multiplying p is less than unity, the optimal scalability of p cannot be achieved. The terms most responsible for this loss of parallel performance are those accounting for the reductions.

The same equation shows that for a fixed p and for large M and/or N, the terms involving MN will dominate the expression in parentheses and that its value tends to unity; therefore for a large grid, the parallel version will provide an acceleration of almost p over the sequential version of the code.

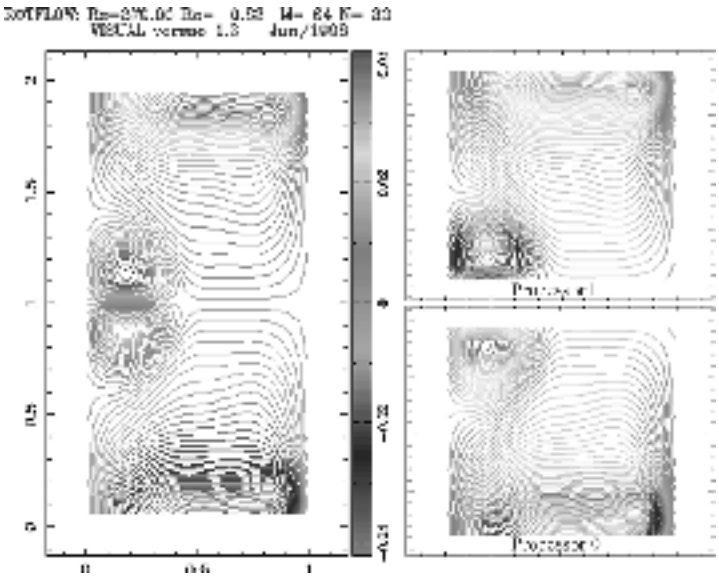


Figure 1. Typical results for a 64×32 mesh, $Re = 279$. From left to right: the contour streamlines for the complete domain and the contours generated by each processor.

Experiments

A number of experiments were carried out on a 4-processor Silicon Graphics Origin200 distributed, global access memory parallel computer located at the Brazilian National Supercomputing Centre (CESUP), and on a 32-processor IBM 9076 SP/2 distributed memory parallel computer located at the Brazilian National Scientific Computing Laboratory (LNCC). Our software is a `FORTRAN 77` code which implements Algorithm 1 with the parallelization expressed as in Algorithm 2. All computations were carried out in single-precision (32 bits) and with the computer in dedicated mode.

The SGI Origin200 used is a four-processor machine in a twin-tower configuration, each tower equipped with two MIPS R10000, 180MHz processors with 1MB cache memory each, and interconnected via a CrayLink$^{\text{TM}}$ cable. It has an aggregate RAM memory size of 256MBytes. The machine is a "scalable, shared-memory processor (S2MP)" and it has a hierarchical memory, with increasing memory access time for data requested from farther processors. It is interconnected like a hypercube, with the use of CrayLink cables and routers. For an Origin computer with 16 and 32 processors, XpressLink$^{\text{TM}}$ interconnects are added to the interconnection network, making use of the spare ports on the routers, minimizing latency and increasing the bandwidth. Nonetheless, the fact remains that this machine does not have a constant latency and transfer rate between any pair of processors. The `MPI` library used is

an SGI proprietary implementation, which makes use of remote shared-memory read/write functions to provide the send/receive message-passing operations.

The IBM 9076 SP/2 has 40 processors: eight are IBM RS6000/590 ("wide node") with 1GByte RAM and the remaining thirty-two are IBM RS6000/390 ("thin node") with 256MBytes RAM (these were the processors used in their tests). The processors are interconnected via an Omega-like network called the High-Performance Switch (HPS). The system was running AIX 4.1.5; the code was compiled with the IBM *mpxlf* compiler. The `MPI` implementation is `MPICH 1.0.12`, developed by Argonne National Laboratory and Mississippi State University.

Typical results

With regards to the flow problem itself, typical results that were obtained are shown in Figure 1, which shows the streamlines for ψ. In that experiment, taken from [18, p. 272], $Re = 279$, $Ro = 0.833$, $\Omega = 0.1\ rad/s$, $G = 6 \times 10^{-4}\ lb/ft^3$ and $h_t = 10^{-4}$ (throughout the iterations). The tolerance for convergence for the initial condition on w was 10^{-5} and it was achieved in $2,595$ iterations, taking $0.0008s$ on two processors. Convergence of the solution of the Navier-Stokes equations using the three-term Runge-Kutta scheme ($K = 3$) for a tolerance of 10^{-4}, took $406.9987s$ after $202,255$ iterations. The figures show a similar appearance to that presented in [18, p. 272].

Scalability on the SGI Origin200

The experimental results given in Table I show the run-time (in seconds/iteration) for several mesh sizes. It can be seen that as the mesh sizes increase, the scalability increases as well. Also noticeable is that in two cases (1024×64 and 1024×128) a substantial increase of the run-time occurs. Using the SGI `perfex` performance analyser, which reports among other data the number of loads and stores per floating point instruction, we see that for the 512×64 mesh, this value is 2.2582, whereas for the 1024×64 mesh it jumps to 83.3133. It appears that the increased data traffic to/from the memory is responsible for the larger run-time exhibited in the latter case.

Another important effect being shown is that of a speed-up larger than 2 for the larger problem sizes when using two processors. Theoretically, such *"superlinear"* effect is impossible to achieve; however, if we are using a parallel computer with a separate cache memory for each processor, what may happen if we double the number of processors in use is that we have at our disposal the double of cache memories while at the same time we are halving the amount of data being accessed locally (supposing that load balance is achieved which is our case). In this case, more data will fit in these extra cache memories when compared to using a single processor. This reasoning was confirmed by the *data cache hit rate* reported by `perfex`; in Table II we show this rate for a few problem sizes. Note that when $p = 2$ the data cache hit rate becomes 1 but for the 256×64 problem size, which is small compared to the others.

Table I. Run time (in seconds per iteration) and speed-ups on the SGI Origin200.

$M \times N$	$p = 1$	$p = 2$	$p = 3$	$p = 4$
16×16	0.0019	0.0018	0.0022	0.0024
		1.0649	0.8836	0.7982
32×16	0.0031	0.0026	0.0026	0.0025
		1.1795	1.1997	1.2103
64×16	0.0058	0.0038	0.0034	0.0031
		1.5190	1.7272	1.9086
128×16	0.0116	0.0068	0.0053	0.0045
		1.7152	2.1810	2.5667
256×16	0.0224	0.0127	0.0094	0.0081
		1.7643	2.3875	2.7770
32×32	0.0058	0.0039	0.0035	0.0033
		1.4865	1.6426	1.7815
64×32	0.0116	0.0068	0.0053	0.0047
		1.7114	2.1770	2.4609
128×32	0.0226	0.0124	0.0093	0.0079
		1.8245	2.4448	2.8567
256×32	0.0441	0.0236	0.0166	0.0133
		1.8704	2.6489	3.3117
512×32	0.0889	0.0466	0.0321	0.0265
		1.9090	2.7719	3.3486
64×64	0.0224	0.0123	0.0092	0.0088
		1.8199	2.4278	2.5415
128×64	0.0442	0.0235	0.0168	0.0132
		1.8844	2.6384	3.3605
256×64	0.0863	0.0458	0.0317	0.0246
		1.8836	2.7232	3.5041
512×64	0.1806	0.0900	0.0624	0.0469
		2.0055	2.8940	3.8479
1024×64	42.5168	21.7635	14.4899	10.8587
		1.9536	2.9342	3.9155
128×128	0.0869	0.0457	0.0317	0.0245
		1.9018	2.7394	3.5505
256×128	0.1816	0.0892	0.0615	0.0468
		2.0360	2.9547	3.8818
512×128	0.3829	0.1857	0.1232	0.0958
		2.0625	3.1093	3.9961
1024×128	85.0758	43.5478	28.9583	21.8141
		1.9536	2.9379	3.9000

Table II. Data cache hit rates for some problem sizes. A value closer to 1 represents a better cache memory usage.

	$p = 1$	$p = 2$	$p = 3$	$p = 4$
256×64	0.9696	0.9926	1.0	1.0
512×64	0.9789	1.0	1.0	1.0
1024×64	0.9635	1.0	1.0	1.0
256×128	0.9802	1.0	1.0	1.0
512×128	0.9753	1.0	1.0	1.0
1024×128	0.9635	1.0	1.0	1.0

Table III. Comparison between run times for (a) combined and (b) separate reductions.

	$M \times N$			
	64×32	128×32	256×64	512×64
$p = 1$	0.3170	0.6436	2.4457	5.1776
$p = 2$				
(a)	0.1852	0.3528	1.2984	2.5817
(b)	0.2244	0.3979	1.3739	2.7304
gain(%)	21.1594	12.7980	5.8145	5.7586
$p = 3$				
(a)	0.1456	0.2633	0.8981	1.7891
(b)	0.1843	0.3097	0.9773	1.9242
gain(%)	26.5539	17.6365	8.8164	7.5498
$p = 4$				
(a)	0.1288	0.2253	0.6980	1.3456
(b)	0.1694	0.2802	0.7862	1.4756
gain(%)	31.5413	24.3674	12.6447	9.6674

Reducing the latency in the computation of norms

As noted in the discussion of the parallel algorithm developed, we combined the reductions needed for the computation of the norms appearing in steps 5 and 7. Table III shows the time (in seconds) taken by two implementations of the parallel algorithm (one with combined reductions and the other with separate reductions, and the respective gains). As can be seen in that table, the combined reductions are a means of increasing the speed-up for small grids, whilst still providing a reduction in the run time with increasing p, even for a large grid.

Table IV. Run time (in seconds per iteration) and speed-ups on the IBM 9076 SP/2.

$M \times N$	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
128×32	0.0355	0.0209	0.0173	0.0155	0.0145	0.0125
		1.6957	2.0526	2.2941	2.4375	2.8352
256×32	0.0709	0.0400	0.0291	0.0182	0.0164	0.0110
		1.7727	2.4375	3.9000	4.3333	6.4274
512×32	0.1455	0.0745	0.0455	0.0273	0.0209	0.0126
		1.9512	3.2000	5.3334	6.9565	11.5011
1024×32	0.2827	0.1543	0.0826	0.0433	0.0397	0.0227
		1.8234	3.4231	6.5312	7.1245	12.4209
2048×32	0.7309	0.3950	0.2050	0.1100	0.0750	0.0378
		1.8504	3.5654	6.6446	9.7454	19.3184
128×64	0.0691	0.0400	0.0300	0.0218	0.0191	0.0141
		1.7273	2.3030	3.1667	3.6191	4.9002
256×64	0.1427	0.0745	0.0445	0.0291	0.0227	0.0142
		1.9146	3.2041	4.9062	6.2800	10.0529
512×64	0.2818	0.1464	0.0818	0.0473	0.0400	0.0227
		1.9255	3.4444	5.9615	7.0454	12.3971
1024×64	0.6555	0.3550	0.1800	0.1000	0.0700	0.0351
		1.8463	3.6414	6.5545	9.3636	18.6660
2048×64	1.4609	0.7850	0.4000	0.2100	0.1150	0.0595
		1.8610	3.6523	6.9567	12.7036	24.5403
128×128	0.1391	0.0745	0.0445	0.0300	0.0218	0.0138
		1.8659	3.1225	4.6364	6.3750	10.0492
256×128	0.2782	0.1455	0.0809	0.0473	0.0318	0.0181
		1.9125	3.4382	5.8846	8.7429	15.3358
512×128	0.5591	0.2827	0.1500	0.0836	0.0527	0.0269
		1.9775	3.7273	6.6848	10.6034	20.7763
1024×128	1.3164	0.7000	0.3200	0.2000	0.1000	0.0524
		1.8805	4.1136	6.5818	13.1636	25.1240
2048×128	2.9082	1.5600	0.7950	0.4000	0.2200	0.1123
		1.8642	3.6581	7.2705	13.2190	25.9025

Scalability on the IBM 9076 SP/2

The same experiments were carried out on an IBM 9076 SP/2 and the run times and respective speed-ups are given in Table IV.

It can be seen that for a moderate small grid, 256×128, we are using efficiently more than half the number of processors (16) used. For a grid twice as high ($M = 512$), a speed-up of approximately 20 is obtained on 32 processors. One can also notice that for a fixed value of M, doubling N does not degrade the performance - as it could be expected, since it leads to messages twice as long. This is explained by the fact that the amount of local data increases in the same rate, and the communication time is still hidden by the compute time.

Concluding remarks

We have presented a parallel algorithm for the solution of the rotating flow problem described by the Navier-Stokes equations, using an explicit Runge-Kutta time-stepping integration scheme.

We believe the results presented show that our approach to parallelize the computation is good and can be used for the solution of related problems. Moreover, it can be used as a framework for the parallelization of other techniques, as long as the possibility of breaking down the computation in two parts - depending on local and remote stored data - exists.

We intend to further develop and apply it to other fluid flow problems, including three-dimensional domains with complex geometries, using generalized coordinates which will allow us to use the same parallelizing technique presented here.

Acknowledgements

The authors wish to thank Mr. E. Meneghetti (Brazilian National Supercomputing Centre) for his invaluable support and FAPERGS (Research Support Agency of the State of Rio Grande do Sul) for partial financial support.

The authors also wish to thank the referees for their comments which have helped to improve this paper.

The experiments in this work were carried out at the Brazilian National Supercomputing Centre (CESUP) and at the Brazilian National Scientific Computing Laboratory (LNCC).

REFERENCES

1. R.D. da Cunha. A benchmark study based on the parallel computation of the vector outer-product $a = uv^T$ operation. *Concurrency: Practice & Experience*, 9(8):803–819, August 1997.
2. R.D. da Cunha and T.R. Hopkins. Parallel preconditioned Conjugate-Gradients methods on transputer networks. *Transputer Communications*, 1(2):111–125, 1993. Also as TR-5-93, Computing Laboratory, University of Kent at Canterbury, U.K.
3. R.D. da Cunha and T.R. Hopkins. A parallel implementation of the restarted GMRES iterative method for nonsymmetric systems of linear equations. *Advances in Computational Mathematics*, 2(3):261–277, April 1994. Also as TR-7-93, Computing Laboratory, University of Kent at Canterbury.
4. R.D. da Cunha and T.R. Hopkins. The Parallel Iterative Methods (PIM) package for the solution of systems of linear equations on parallel computers. *Applied Numerical Mathematics*, 19(1-2):33–50, November 1995.
5. A.L. de Bortoli. Solution of incompressible flows using a compressible flow solver. 129-94/18, DLR-IB, 1994.
6. E. Dick. *Introduction to Finite Volume Techniques in Computational Fluid Dynamics*, pages 270–297. J.F. Wendt (Ed.), Computational Fluid Dynamics - An Introduction (2nd Ed.). Springer-Verlag, Berlin, 1996.
7. D.R. Emerson. Porting a Navier-Stokes code to parallel systems. Parallel Computing in Computational Fluid Dynamics, Collaborative Computational Project 12 and CFD Community Club Joint Meeting, Daresbury, U.K., 22nd May, 1991.
8. J.H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer-Verlag, Berlin, 1996.
9. C.A.J. Fletcher. *Computational Techniques for Fluid Dynamics*. Spring-Verlag, Berlin, 2nd edition, 1991.
10. Message Passing Interface Forum. MPI: A message-passing interface standard. TR CS-93-214, University of Tennessee, November 1993.

11. A. Jameson, W. Schmidt, and E. Turkel. Numerical solution of the Euler equations by finite volume methods using Runge-Kutta time-stepping schemes. AIAA Paper 81-1259, 1981.
12. D.A.R. Justo. Visual. Manual do usuário, Laboratório Integrado de Computação Científica, Instituto de Matemática, Universidade Federal do Rio Grande do Sul, 1998.
13. V. Kalro and T. Tezduyar. Parallel 3D computation of unsteady flows around circular cylinders. *Parallel Computing*, 23:1235–1248, 1997.
14. L.K. Lundin. Computing the velocity of a rotating flow. *Parallel Computing*, 24:2021–2034, 1998.
15. L. Paglieri, D. Ambrosi, L. Formaggia, A. Quarteroni, and A.L. Scheinine. Parallel computation for shallow water flow: a domain decomposition approach. *Parallel Computing*, 23:1261–1277, 1997.
16. D. Roose and R. Van Driessche. Distributed memory parallel computers and computational fluid dynamics. Report tw186, Department of Computer Science, Katholiek Universiteit Leuven, March 1993.
17. N. Satofuka, M. Obata, and T. Suzuki. Parallel computation of super-/hypersonic flows on workstation network and Transputer arrays. *Parallel Computing*, 23:1293–1305, 1997.
18. C.G. Speziale. Numerical solution of rotating internal flows. *Lectures in Applied Mathematics*, 22:261–288, 1985.
19. C.G. Speziale and S. Thangam. Numerical study of secondary flows and roll-cell instabilities in rotating channel flow. *Journal of Fluid Mechanics*, 130:377–395, 1983.
20. V.N. Vatsa and T.R. Faulkner. Navier-Stokes computations on commodity computers. 25th National and First International Conference on Fluid Mechanics and Fluid Power, Delhi, India, December 15-17, 1998.