Internal report Nr 97/11

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Thermo and Fluid Dynamics

A parallel multiblock extension to the CALC-BFC code using PVM

by

Håkan Nilsson

Göteborg, March 1997

Preface

I believe that it is important to follow the fast evolution of computers, which will make it possible to solve larger CFD-problems faster and more exact. With a parallel multiblock solver, several goals can be reached: Complex geometries that are solved using a multiblock method can be solved in parallel. The computational speed can be increased. Larger problems can be solved since the memory requirements is divided between the processors. More exact solutions can be obtained because of the extra memory available. Parallel supercomputers may be utilized.
The results from such calculations will help us understand a little bit more about the complexity of turbulent flow.

Acknowledgments

I would like to record here my gratitude to those who made this work feasible. Professor Lars Davidson, who introduced me to Computational Fluid Dynamics and became my supervisor during this work, has been extremely helpful and encouraging throughout the work. Anders Ålund at ITM (Swedish Institute for Applied Mathematics), who came to rescue me when I was struggling with computers that sometimes refused to cooperate with each other (or me). Without his frequent advice I would probably not have achieved these results.

Contents

1 Introduction

A multiblock extension to the CALC-BFC code, where the blocks are solved in parallel using PVM (Parallel Virtual Machine), is implemented to increase the computational speed. The parallel version of the code is verified in two-dimensional, laminar lid-driven cavity and backward facing step flow, comparing the results from the parallel computations with single processor and benchmark calculations. The parallel multiblock solver produces good results more than 15% faster (four blocks) than the standard code for these cases. For some tests, the computational speed has been doubled. A follow-up on this work should focus on increasing the computational speed further by implementing a more effective Poisson-equation solver.
The reader is assumed to have some experience of CFD, so the methods are only briefly described in this text. For a more thorough description, the reader is advised to read the papers referred to in the reference list.

1.1 Parallel computations

To make the calculations as fast as possible the problem is divided into a number of sub-problems (tasks) which are solved at the same time, in parallel. Here, the computational domain is divided into a number of smaller domains for which the equations can be solved faster than for the whole domain. Solving these tasks at the same time would, if they were independent of each other, increase the speed of computation by up to 'number of tasks' times the original speed of computation.
However, when the domain is divided into sub-domains, they will adjoin each other on two-dimensional planes (inner boundaries). These inner boundaries are no boundaries in a physical meaning, but virtual boundaries with unknown inhomogeneous Dirichlet boundary conditions, stemming from the neighboring sub-domains.
Since the task boundary conditions are inadequate, solving in parallel requires an iterative process; solving for each task and exchanging inner boundary values between the tasks. Unfortunately, requiring extra computational effort, these iterations reduces the increased speed of computation. Since the velocity field is easily solved, given a good pressure-distribution, this iterative process need only be applied solving the Poisson equation (for the pressure). The parallelization has been accomplished using PVM (Parallel Virtual Machine, see section B), a number of message-passing subroutines, simulating a multi processor computer. PVM can also be run on real multi processor computers. The reader is advised to read through sections B.1-B.8 to get an idea of how the parallel extension to the program works and to get introduced to the terminology used throughout this text.

2 The model

To model the flow, a finite volume method [2, 5, 3, 7] (see section A.1) with a collocated grid arrangement [4] is used. The computations are based upon the solution of the partial differential Navier Stokes and continuity equations, governing the flow. The equations are written in a non-orthogonal coordinate system. The finite volume method is applied to transform the partial differential equations to algebraic relations which link the values of the dependent variables at the nodes of the computational grid. The momentum equations are solved with an implicit method using central differencing, which is second order accurate, for all terms. The pressure is obtained from a Poisson equation, which is solved with a conjugate gradient method, with an incomplete Cholesky factorization as a preconditioner. The solver is a part of the **SLAP** package (Sparse Linear Algebra Package), implemented by Renard and Gresser [13], available on **netlib**. The incomplete Cholesky factorization is fairly expensive in terms of CPU, but fortunately the coefficients in the matrix stemming from the discretized Poisson equation are constants, which means that we need to apply the preconditioner only at the first time step.

3 The Numerical Method

To solve the Navier Stokes- and continuity-equations an implicit two-step time-advancement method is used. It can be summarized as follows.

When the filtered (see section A.2) Navier-Stokes equation for $\bar{u}_i$

$$\frac{\partial \bar{u}_i}{\partial t} + \frac{\partial}{\partial x_j}\left(\bar{u}_i \bar{u}_j\right) = -\frac{1}{\rho}\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j} \tag{3.1}$$

is discretized, using

$$\frac{\partial \bar{u}_i}{\partial t} = \frac{\bar{u}_i^{n+1} - \bar{u}_i^n}{\Delta t},$$

where n denotes the time step, it can be written

$$\bar{u}_i^{n+1} = \bar{u}_i^n + \Delta t H\left(\bar{u}_i^n, \bar{u}_i^{n+1}\right) - \alpha_p \frac{\Delta t}{\rho}\frac{\partial \bar{p}^{n+1}}{\partial x_i} - (1-\alpha_p)\frac{\Delta t}{\rho}\frac{\partial \bar{p}^n}{\partial x_i} \tag{3.2}$$

where

$$\begin{aligned} H\left(\bar{u}_i^n, \bar{u}_i^{n+1}\right) &= \left(\alpha_{vel}\nu \frac{\partial^2 \bar{u}_i^{n+1}}{\partial x_j \partial x_j} + (1-\alpha_{vel})\nu \frac{\partial^2 \bar{u}_i^n}{\partial x_j \partial x_j}\right) \\ &- \left(\alpha_{vel}\frac{\partial}{\partial x_j}\left(\bar{u}_i^{n+1}\bar{u}_j^{n+1}\right) + (1-\alpha_{vel})\frac{\partial}{\partial x_j}\left(\bar{u}_i^n \bar{u}_j^n\right)\right) \\ &- \left(\alpha_{vel}\frac{\partial \tau_{ij}^{n+1}}{\partial x_j} + (1-\alpha_{vel})\frac{\partial \tau_{ij}^n}{\partial x_j}\right) \end{aligned}$$

The pressure and the velocities are thus implicitly time-advanced with different time-advancement constants (α_p and α_{vel}, respectively). This makes it possible to treat the time-advancement for the pressure and the velocities differently.

An intermediate velocity field $\bar{u}_i^*$ is defined as

$$\bar{u}_i^* \equiv \bar{u}_i^n + \Delta t H\left(\bar{u}_i^n, \bar{u}_i^{n+1}\right) - (1-\alpha_p)\frac{\Delta t}{\rho}\frac{\partial \bar{p}^n}{\partial x_i} \tag{3.3}$$

Equation 3.2 can thus be written as

$$\bar{u}_i^{n+1} = \bar{u}_i^* - \alpha_p \frac{\Delta t}{\rho}\frac{\partial \bar{p}^{n+1}}{\partial x_i} \tag{3.4}$$

Taking the divergence of Eq. 3.4 requiring that continuity should be satisfied on level $n+1$, i.e.

$$\frac{\partial \bar{u}_i^{n+1}}{\partial x_i} = 0, \tag{3.5}$$

a Poisson equation

$$\frac{\partial^2 \bar{p}^{n+1}}{\partial x_i \partial x_i} = \frac{\rho}{\Delta t \alpha_p}\frac{\partial \bar{u}_i^*}{\partial x_i} \tag{3.6}$$

for the pressure is obtained.
The discretized, filtered Navier Stokes equations (for $\bar{u}$, $\bar{v}$, $\bar{w}$, i.e. eq. 3.4) and the discretized Poisson equation (for the pressure, i.e. eq. 3.6) are solved during a numerical procedure.

3.1 Numerical procedure

The numerical procedure, solving for $\bar{u}$, $\bar{v}$, $\bar{w}$ and $\bar{p}$, at each time step can be summarized as follows.

I Calculate $\bar{u}^{n+1}$, $\bar{v}^{n+1}$, $\bar{w}^{n+1}$ and $\bar{p}^{n+1}$ (i.e. the next time step) by iterating through pts. $\mathrm{I.I}$ - $\mathrm{I.VIII}$ until a global convergence criteria (see section A.3) is fulfilled.

$\mathrm{I.I}$ Solve the discretized filtered Navier-Stokes equation for $\bar{u}$, $\bar{v}$ and $\bar{w}$ for each task. The equation for $\bar{u}$, for example, reads [12, 4] (see section A.1 and cf. eq. 3.2)

$$a_P \bar{u}_P^{n+1} = \sum_{NB} a_{NB}\bar{u}_{NB}^{n+1} + b - \alpha_p \frac{\partial \bar{p}^{n+1}}{\partial x}\delta V \tag{3.7}$$

where the most recent values for $\bar{u}_{NB}^{n+1}$ and $\bar{p}^{n+1}$ are used and b is a source term, containing contributions from the previous time step. Until the convergence criteria is fulfilled, this solution for $\bar{u}_i^{n+1}$ is a 'temporary' solution, used in the iteration process.

$\mathrm{I.II}$ Create an intermediate velocity field $\bar{u}^*$, $\bar{v}^*$ and $\bar{w}^*$. The $\bar{u}^*$ velocity, for example, is computed as (cf. eq. 3.4)

$$\bar{u}^* = \bar{u}^{n+1} + \alpha_p \frac{\partial \bar{p}^{n+1}}{\partial x}\frac{\delta V}{a_P}$$

I.III Exchange inner boundary values for the intermediate velocity field.

I.IV Compute the intermediate face velocities $\bar{u}_e^*$, $\bar{u}_w^*$, $\bar{v}_n^*$, $\bar{v}_s^*$, $\bar{w}_h^*$ and $\bar{w}_l^*$ using linear interpolation.

I.V Compute

$$\begin{aligned} \delta \dot{m}^* &= (\bar{u}_e^* - \bar{u}_w^*)\,\Delta_y \Delta_z \\ &+ (\bar{v}_n^* - \bar{v}_s^*)\,\Delta_x \Delta_z \\ &+ (\bar{w}_h^* - \bar{w}_l^*)\,\Delta_x \Delta_y \end{aligned}$$

used in the source term when solving the Poisson equation.

I.VI Solve the Poisson equation (eq. 3.6), to obtain the pressure, by iterating through pts. I.VI.i - I.VI.ii until a global pressure convergence criteria[1] (`res`) is fulfilled.

I.VI.i Solve the Poisson equation (eq. 3.6) for each task until a local pressure convergence criteria (`tol`) is fulfilled.

I.VI.ii Exchange inner boundary values, between the tasks, for the pressure.

I.VII Exchange inner boundary values, between the tasks, for all the variables involved.

I.VIII Correct the face velocities.

4 Boundary conditions

For the pressure, Neumann boundary conditions are used at all boundaries, i.e.

$$\frac{\partial \bar{p}}{\partial n} = 0 \tag{4.1}$$

where n is the coordinate direction normal to the boundary.
This is accomplished by extrapolating the values inside the boundaries onto the boundaries and setting the numerical coefficients, adjacent to the boundary, to zero.

4.1 Walls

No-slip boundary conditions are used for the velocities at walls. The Neumann condition in eq. 4.1 requires no-slip conditions at the walls also for the intermediate velocity field $\bar{u}_i^*$ in order to satisfy global conservation in the Poisson equation (eq. 3.6) for the pressure [1, 10].

[1] The pressure convergence criteria is fulfilled when the pressure has the same global residual as in a singletask solution

4.2 Inlet/outlet conditions

At an inlet, all flow properties are prescribed to an approximate velocity profile. They can be interpolated from experimental data or from a fully developed profile, for instance; a parabolic profile for laminar flow or a 1/7-profile for a turbulent flow.
At a wide outlet, sufficiently far downstream and without area change, the flow may be assumed as fully developed, which implies negligible streamwise gradients of all velocities, i.e.

$$\frac{\partial u_i}{\partial n} = 0$$

where n is the coordinate direction normal to the outlet.

To increase convergence rate, a velocity increment

$$u_{incr} = \frac{\dot{m}_{in} - \dot{m}_{out}^{comp}}{(\rho A)_{out}},$$

where $\dot{m}_{in}$ is the convection into the domain at the inlet, $\dot{m}_{out}^{comp}$ is the computed convection out of the domain at the outlet and A is the outlet area, is added to the computed velocity at the outlet, i.e.

$$u_{out} = u_{out}^{comp} + u_{incr} \tag{4.2}$$

This ensures that global continuity is fulfilled.

4.3 Symmetric boundaries

At symmetry planes, there is no flux of any kind normal to the boundary, either convective or diffusive. Thus, the normal velocity component, as well as the normal gradients of the remaining dependent variables, are set to zero.

5 Validation

To validate the program, the multitask solutions of two-dimensional, unsteady, laminar lid-driven cavity and backward facing step flow have been compared with singletask and benchmark calculations. The pressure is implicitly time advanced ($\alpha_p = 1$) and for the velocities, a Crank Nicolson time advancement scheme ($\alpha_{vel} = 0.6$) is used. The results are shown in the following sections.

5.1 Lid-driven cavity

The program has been validated in two-dimensional, unsteady, laminar lid-driven cavity flow, using a clustered 64×64×6 - grid (see fig. 5.1). The domain is described as a square in the z-plane (with a few nodes in the z-direction for computational reasons) with height $H = 1m$ and length $L = 1m$. The Reynolds number is $Re = u_{wall}L/\nu = 400$, based on

$u_{wall} = 1m/s$ and $L = 1m$. The global residual has been reduced with a factor 10^{-4} each time step. With a time step of $\Delta t = 2.6 \cdot 10^{-3} s$ during these calculations, the CFL-number[2] ($u\Delta t/\Delta x$) has a maximum of 1.36 at the beginning, but quickly goes to a steady value of 1.17, which gives good, time accurate calculations. For the lid-driven cavity flow, the multitask solutions are represented by a four task solution (two in the x-direction and two in the y-direction). The steady state solution of the lid-driven cavity flow is shown in figure 5.2. The lower and the vertical boundaries are walls (see section 4.1). The top boundary has velocity $u = 1m/s$. In the z-direction, symmetric boundaries (see section 4.3) are applied. The vertical and horizontal lines, going through the domain, indicates from which task the solution is taken. This is thus a four task solution.

In figure 5.3 the multitask solution is compared with the corresponding singletask solution. These are the u-velocities at $x = L/2$ after 200 time steps. Since the solutions are exactly equal, the parallel extension to the program seems to work.

In figure 5.4 the multitask steady-state solution is compared with the benchmark calculation of Ghia *et al.* [9]. These are the u-velocities at $x = L/2$ after 16 652 time steps. Since the solutions are approximately equal, the numerical method and the parallel extension to the program seems to produce good results.

To show that the steady-state solution has been reached, the u-velocity at two locations, ($x = L/2$, $y = 0.80H$) and ($x = L/2$, $y = 0.29H$), are shown in figure 5.5. Since the velocities at the control points have reached approximately steady values after 16 652 time steps, the flow is assumed to have reached steady state.

[2]The CFL-number is a measurement of how far a fluid particle moves during one time step, in terms of control volume length, and should be kept below 2

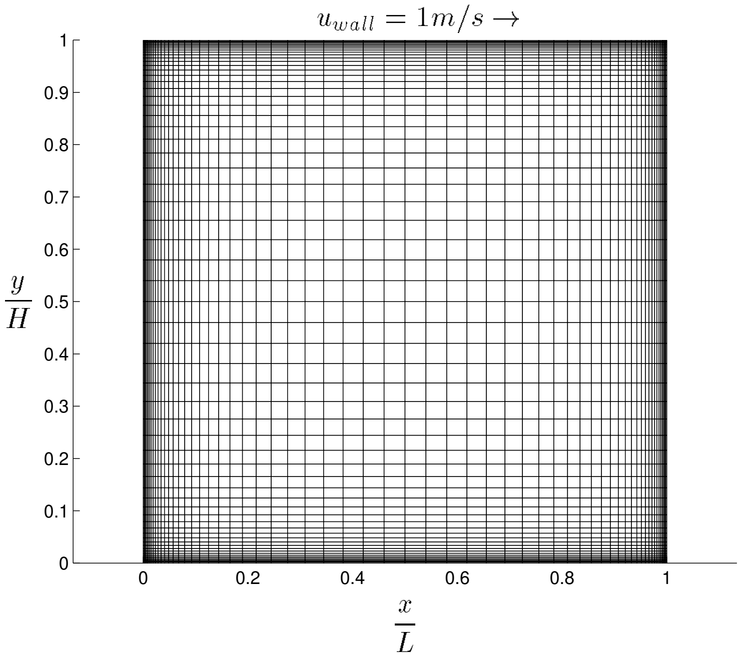

Figure 5.1: Two-dimensional 64×64 clustered lid-driven cavity grid.

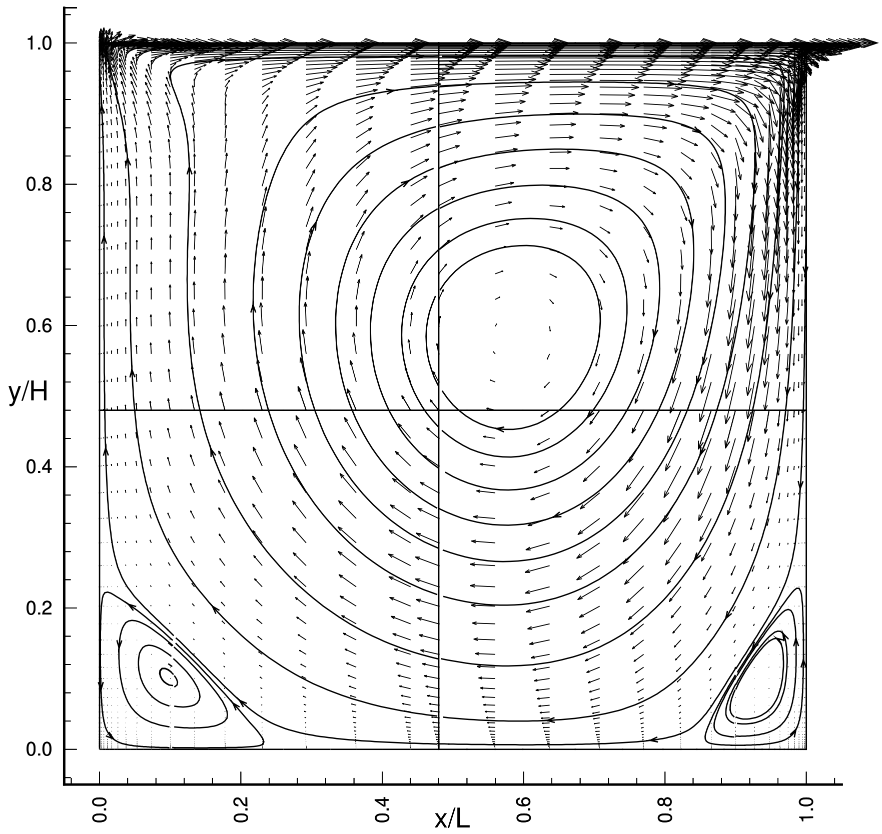

Figure 5.2: Two-dimensional lid-driven cavity. Steady-state velocity vectors and streamlines. $u = 1m/s$ at top boundary and the other boundaries are walls. In the lower corners, two secondary circulations can be observed.

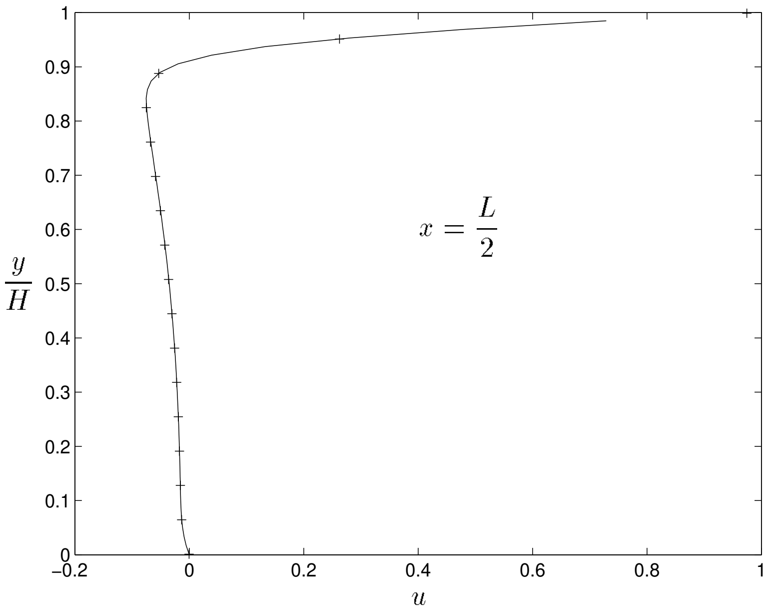

Figure 5.3: Two-dimensional lid-driven cavity. u-velocity at $x = L/2$ after 200 time steps. Multitask vs. singletask. Solid line: Multitask; $+$: Singletask

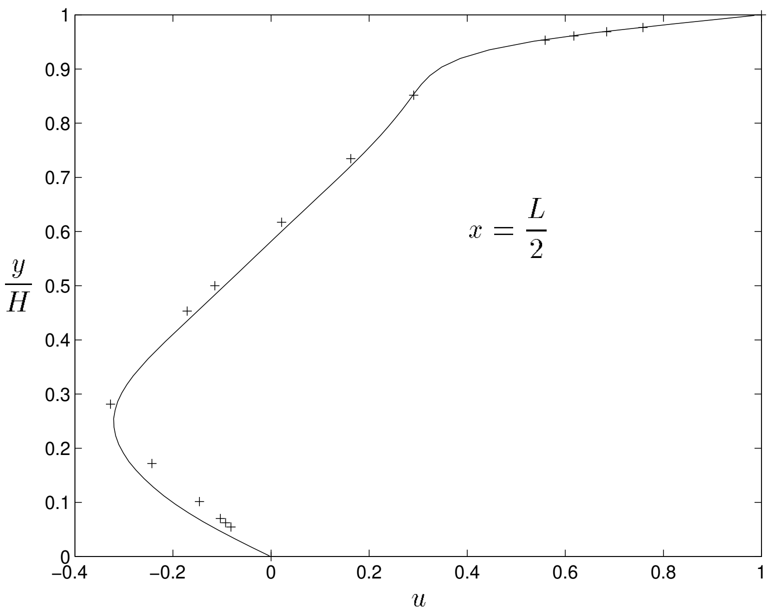

Figure 5.4: Two-dimensional lid-driven cavity. u-velocity at $x = L/2$ after 16 652 time steps. Multitask vs. benchmark calculations of Ghia *et al.* [9]. Solid line: Multitask; $+$: Benchmark.

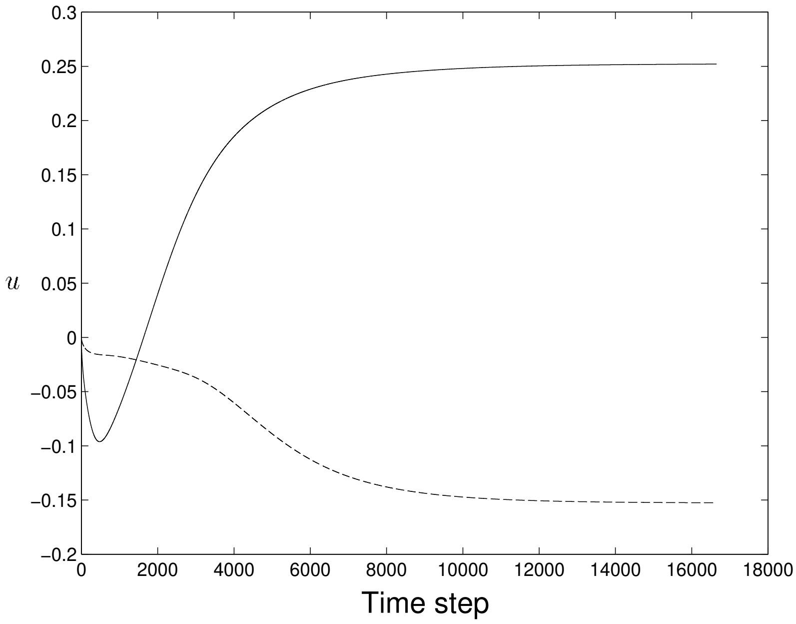

Figure 5.5: Convergence to steady state. Multitask two-dimensional lid-driven cavity. u-velocity at $x = L/2$, $y = 0.8H$ and $y = 0.29H$. Solid line: $y = 0.80H$; Dashed line: $y = 0.29H$

5.2 Backward-facing step.

The program has been validated in two-dimensional, unsteady, laminar backward-facing step flow, using an equidistant 160×40×6 - grid (see fig. 5.6, where the inlet-part of the domain is shown). The domain is described as a rectangle in the z-plane (with a few nodes in the z-direction for computational reasons) with height $H = 1m$ and length $L = 16m$. The Reynolds number is $Re = u_b H/\nu = 800$, based on $u_b = 1m/s$ and $H = 1m$ (u_b denotes bulk velocity). A parabolic inflow velocity profile is specified for u and v is set to zero at the inlet. Outflow boundary velocities (see section 4.2) are obtained by first order extrapolation ($\partial/\partial x = 0$ at the outlet) and global continuity has been forced to be fulfilled to increase convergence rate. The global residual has been reduced with a factor 10^{-3} each time step. With a time step of $\Delta t = 0.1s$ during these calculations, the CFL-number ($u\Delta t/\Delta x$) has a maximum of 2.89 after 82 time steps, but then it goes to a steady value of 1.48, which gives good, time accurate calculations. In the backward facing step calculations, a five task (five tasks in the x-direction) configuration is chosen to represent the multitask solutions.

The steady-state solution of the backward-facing step flow is shown in figure 5.7. Because of the geometry of the domain the figure is divided into three parts. The flow is from left to right, with inlet at the left boundary in the uppermost part at $0.5 \leq y \leq 1$ (the rest of the left boundary is a wall) and outlet at the right boundary in the lowermost part, at $x = 16$. The upper and lower boundaries are walls (see section 4.1). In the z-direction, symmetric boundaries (see section 4.3) are applied.

In figure 5.8 the multitask solution is compared with the corresponding singletask solution. These are the u-velocities at $x = 7$ after 800 time steps.

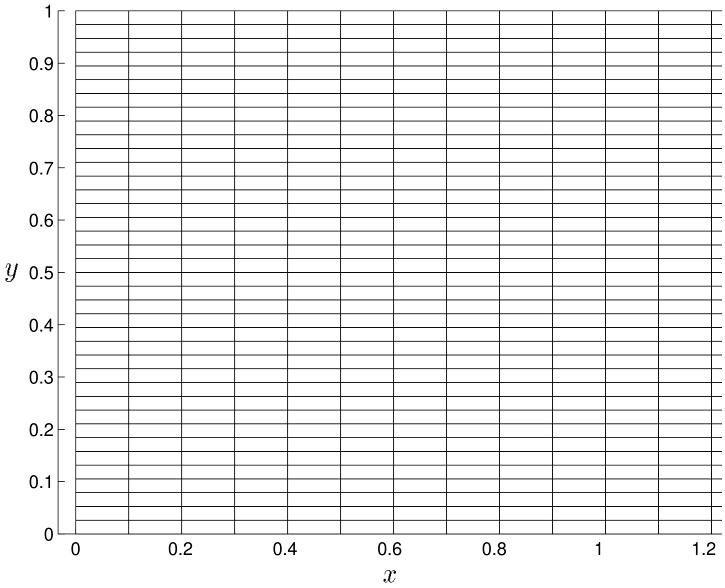

Figure 5.6: Two-dimensional equidistant 160×40 backward-facing step grid (only the inlet part of it). Upper and lower boundaries are walls. Inlet at the left boundary at $0.5 \leq y \leq 1$. The rest of the left boundary is a wall. Outlet far downstream, at the right boundary at $x = 16$.

The small discrepancy between the multitask and singletask solutions is probably due to an inadequately developed steady state solution. Since the two results are approximately equal, the parallel extension to the program seems to work.

In figure 5.9 the multitask solution is compared with the benchmark calculation of Gartling [8]. These are the u-velocities at $x = 7$ after 800 time steps. The calculations are assumed to have reached steady state, since the number of iterations at each time step has been equal to one for a large number of time steps. Since the solutions are approximately equal, the numerical method and the parallel extension to the program seems to produce good results.

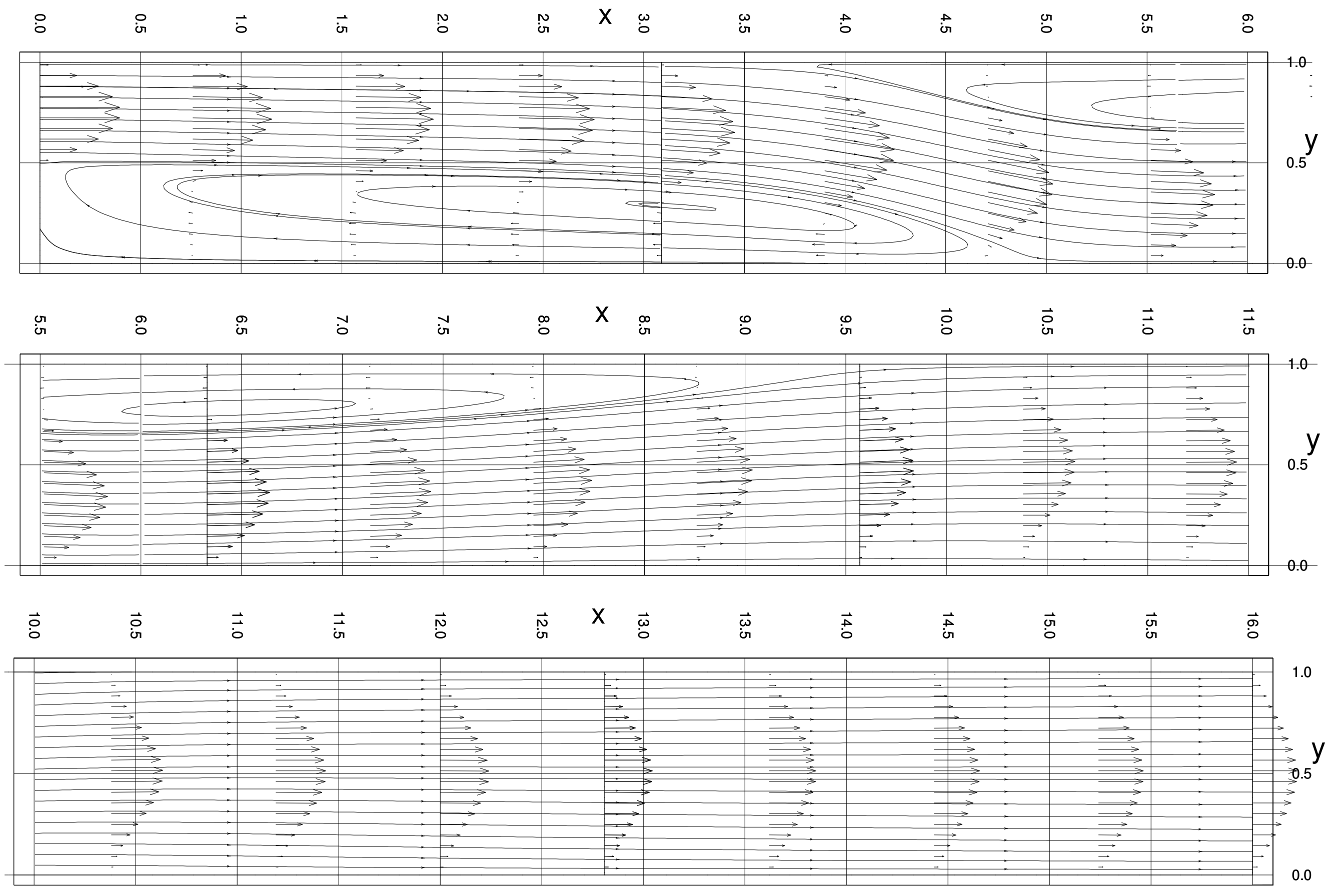

Figure 5.7: Multitask two-dimensional backward-facing step. Steady-state velocity vectors and streamlines. Every eighth vector, in x-direction, is displayed. Upper ($y = 1$) and lower ($y = 0$) boundaries are walls. Inlet at the left boundary in the uppermost figure, at $0.5 \leq y \leq 1$. Outlet to the right in the lowermost figure, at $x = 16$. Two secondary circulations can be observed, at $0 \leq x \leq 4.75$ and $3.75 \leq x \leq 9$.

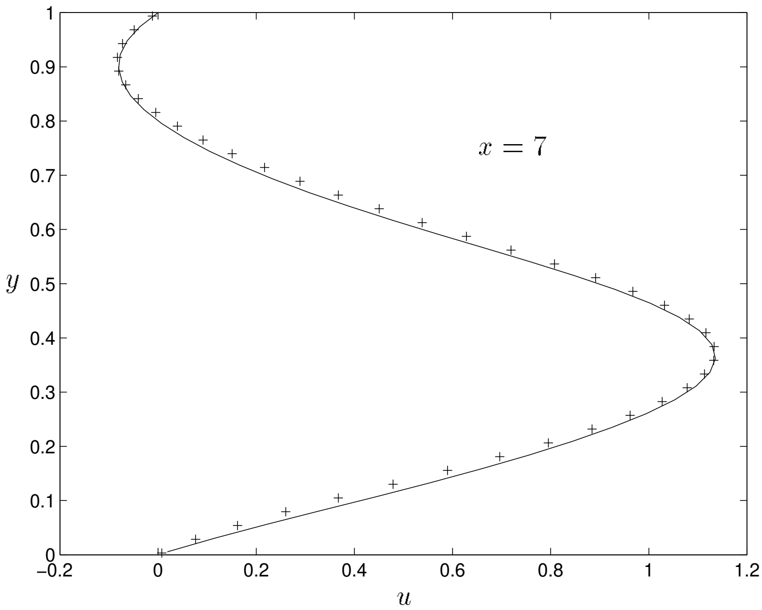

Figure 5.8: Two-dimensional backward-facing step. u-velocity at $x = 7$ after 800 time steps. Multitask vs. singletask. Solid line: Multitask; $+$: Singletask.

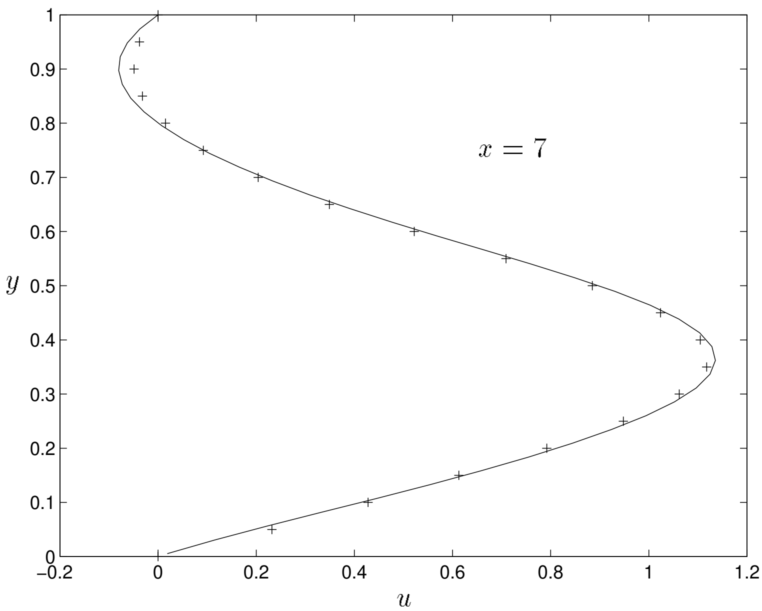

Figure 5.9: Two-dimensional backward-facing step. u-velocity at $x = 7$ after 800 time steps. Multitask vs. benchmark calculations by Gartling [8]. Solid line: Multitask; $+$: Benchmark.

6 Speed tests

The two-dimensional laminar lid-driven cavity and backward-facing step configurations has been run on an 8×supersparc 50Mhz SPARC 1000. The execution times of the multitask and the corresponding singletask computations has been compared to test the speed of execution.

6.1 Execution times

In table 6.1, the execution times are displayed. For the testing of the program, the lid-driven cavity configuration is used. The global pressure convergence criteria is, for multitask calculations, `res`=0.1 and, for singletask calculations, `tol`=0.1. This gives equal multitask and singletask global pressure residuals when leaving the **SLAP**-solver. The times in the table are in minutes required for solving 50 time steps of the flow. The multitask calculations are represented by a four task configuration, where the domain is divided into (`lnodes`, `mnodes`, `nnodes`) = (2, 2, 1) parts. The 64×64×6 grid has been used. The execution times are reduced to 0.85 times the execution time of the corresponding singletask calculation for `tol`=0.45. Tests regarding two speed increasing changes has been made;

- Introduction of overrelaxation on inner boundaries has not been advantageous and has therefore been abandoned.
- Extended overlapping of the subdomains results in a less parallel program, but since the inner boundaries converge much faster, the speed of execution is raised to the double of the corresponding singletask case. The calculations give steady global convergence similar to the singletask calculation, starting at 7 iterations at the first time step. Because of the increased convergence rate for the inner boundaries, the number of times through the **SLAP**-solver each time step has a mean of less than ten. The execution times are displayed in table 6.1. This is only a first approach to inreasing the computational speed by overlapping the subdomains more, so the times are only displayed as examples of the possibilities.

From the information in the table, it is quite obvious that it is possible to get execution times that are much lower than for the singletask case, using this parallel multiblock solver. This is only a first approach to reducing the computational times and should be followed up with more tests.

tol	2×2×1 *multitask with different* `tol`*-values*					
	0.3	0.4	0.45	0.5	0.6	0.7
Multitask	120min	95min	93min	93min	142min	141min
Multitask % of singletask	110%	87%	85%	85%	130%	129%
overlap=2			63min			
overlap=2 m-task % of singletask			58%			
overlap=3			54min			
overlap=3 m-task % of singletask			50%			
overlap=4			66min			
overlap=4 m-task % of singletask			61%			
overlap=5			59min			
overlap=5 m-task % of singletask			54%			
overlap=10			88min			
overlap=10 m-task % of singletask			81%			

Table 6.1: Execution times for solving 50 time steps of the lid-driven cavity configuration. Different values of the local pressure convergence criteria (`tol`) for the multitask case is applied and the execution times are compared with the corresponding singletask case (109min, 100%). Execution times for different amount of overlapping is also dispayed.

7 Discussion and future work

A parallel multiblock extension to CALC-BFC [4, 2, 5, 3, 7] has been developed in this work.
According to verifications in two-dimensional, laminar lid-driven cavity and backward-facing step flow, the numerical model as well as the parallel multiblock extension seem to produce good results.
The reason for the parallel implementation was to increase the computational speed of the program. This has been achieved but the loop describing the Poisson-equation solver in pt.I.VI, section 3.1, seem to induce a quite slow convergence rate for the inner boundaries and thus for the global solution. Using overrelaxation on inner boundaries has not been successful. By overlapping the subdomains more than necessary, the program becomes less parallel, but the convergence rate for the inner boundaries is increased. Calculating the inner boundary values using a multigrid method could also increase the computational speed. It should be noted that in the present algorithm (an implicit two-step time-advancement method), the global convergence criteria for the pressure must be relatively strong (`res`$\approx$0.1). This means that the overall computational time is strongly dependent on the pressure solver. Using a SIMPLE method, the global convergence criteria for the Poisson equation for the pressure correction equation is much less strong (`res`$\approx$0.95). The parallel multiblock solver developed in this work is thus expected to be much more efficient in connection with SIMPLE. At this moment, a parallel multiblock code using the SIMPLE method is implemented.
When solving for the pressure in a multitask case, the number of global iterations (pt.I, section 3.1) is the same as for the corresponding singletask case. This indicates that the Poisson-equation solver produces good results and that the rest of the program is efficient, given a good pressure distribution. The software library *BlockSolve95*, available on **netlib** is considered as 'reasonably efficient for problems that have only one degree of freedom associated with each node, such as the three-dimensional Poisson problem'. It sounds like it is worth an investigation.
The suggestions of the speed increasing changes described above is summarized in the following list.

Suggestions of speed increasing changes

- Overrelaxate the pressure on inner boundaries
- Calculate inner boundary pressure values using a multigrid method
- Overlap the subdomains more
- Use a SIMPLE method with less strong pressure convergence criteria
  (`res = 0.95` instead of `res = 0.1`)
- Use a parallel Poisson equation solver
  (*BlockSolve95*)

References

[1] BLOSCH, E., SHYY, W., AND SMITH, R. The role of mass conservation in pressure-based algorithms. *Numer. Heat Transfer. Part B 24* (1993), 415–429.

[2] DAVIDSON, L. Implementation of a large eddy simulation method applied to recirculating flow in a ventilated room. Report, ISSN 1395-7953 R9611, Dep. of Building Technology and Structural Engineering, Aalborg University, 1996.

[3] DAVIDSON, L. Large eddy simulation: A dynamic one-equation subgrid model for three-dimensional recirculating flow. In *Eleventh Symp. on Turbulent Shear Flows (to be presented)* (Grenoble, 1997).

[4] DAVIDSON, L., AND FARHANIEH, B. CALC-BFC: A finite-volume code employing collocated variable arrangement and cartesian velocity components for computation of fluid flow and heat transfer in complex three-dimensional geometries. Rept. 92/4, Thermo and Fluid Dynamics, Chalmers University of Technology, Gothenburg, 1992.

[5] DAVIDSON, L., AND NIELSEN, P. Large eddy simulations of the flow in a three-dimensional ventilated room. In *5th Int. Conf. on Air Distributions in Rooms, ROOMVENT'96* (Yokohama, Japan, 1996), S. Murakami, Ed., vol. 2, pp. 161–68.

[6] DEARDORFF, J. A numerical study of three-dimensional turbulent channel flow at large Reynold numbers. *J. Fluid. Mech. 41* (1970), 453–480.

[7] EMVIN, P., AND DAVIDSON, L. Development and implementation of a fast large eddy simulations method. *submitted for journal publication* (1997).

[8] GARTLING, D. A test flow for outflow boundary conditions - flow over a backward-facing step. *Int. J. Num. Meth in Fluids 11* (1990), 953–967.

[9] GHIA, U., GHIA, K., AND SHIN, T. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Comp. Phys. 48* (1982), 387–411.

[10] LE, H., AND MOIN, P. Direct numerical simulation of turbulent flow over a backward facing step. Report no. TF-58, Stanford University, Dept. Mech. Eng., 1994.

[11] LEONARD, A. Energy cascade in Large-Eddy Simulations of turbulent fluid flows. *Advances in Geophysics 18* (1974), 237–248.

[12] PATANKAR, S. *Numerical Heat Transfer and Fluid Flow*. McGraw-Hill, New York, 1980.

[13] RENARD, J., AND GRESSER, D. Computational modelling of 2d hill flows. Diploma thesis. rept. 95/6, Thermo and Fluid Dynamics, Chalmers University of Technology, Gothenburg, 1995.

A Theory

This section briefly describes the methods used in this work. For further information, the reader is advised to read the papers referred to in the text.

A.1 Solution methodology

When using a finite volume method [4, 12], the computational domain is divided into a finite number of control volumes (see fig. A.1). In order to extend the capabilities of the finite difference method to deal with complex geometries, a boundary fitted coordinate method is used.
The basic idea in this method is to map the complex flow domain in the physical space to a simple rectangular domain in the computational space by using a curvilinear coordinate transformation. In other words, the Cartesian coordinate system x_i in the physical domain is replaced by a general non-orthogonal system ξ_i.
The momentum equations are solved for the velocity components u, v and w in the fixed Cartesian directions on a non-staggered grid. This means that all the variables are computed and stored at the center of the control volume.
The transport equation for a general dependent variable Φ in the Cartesian coordinates can be written as

$$\frac{\partial}{\partial t}(\rho\Phi) + \frac{\partial}{\partial x_i}(\rho u_i \Phi) = \frac{\partial}{\partial x_i}\left(\Gamma_\Phi \frac{\partial \Phi}{\partial x_i}\right) + S_\Phi \tag{A.1}$$

where Γ_Φ is the exchange coefficient and is equal to the dynamic viscosity in the momentum equations. The dependent variable Φ can be equal to u, v, w etc. and the source term S_Φ can contain, for instance, contributions from the pressure distribution.

The total flux, convective and diffusive fluxes, is defined as

$$I_i = \rho u_i \Phi - \Gamma_\Phi \frac{\partial \Phi}{\partial x_i} \tag{A.2}$$

Equation A.1 can thus be rewritten as

$$\frac{\partial}{\partial t}(\rho\Phi) + \frac{\partial I_i}{\partial x_i} = S_\Phi \tag{A.3}$$

or, in vector notation

$$\frac{\partial}{\partial t}(\rho\Phi) + \nabla \cdot \vec{I} = S_\Phi \tag{A.4}$$

Integration of eq. A.4 over a control volume in the physical space, using Gauss' law gives

$$\int_A \vec{I} \cdot d\vec{A} = \int_V \left(S_\Phi - \frac{\partial}{\partial t}(\rho\Phi)\right) dV$$

or

$$\int_A \vec{I} \cdot d\vec{A} = \int_V S dV \tag{A.5}$$

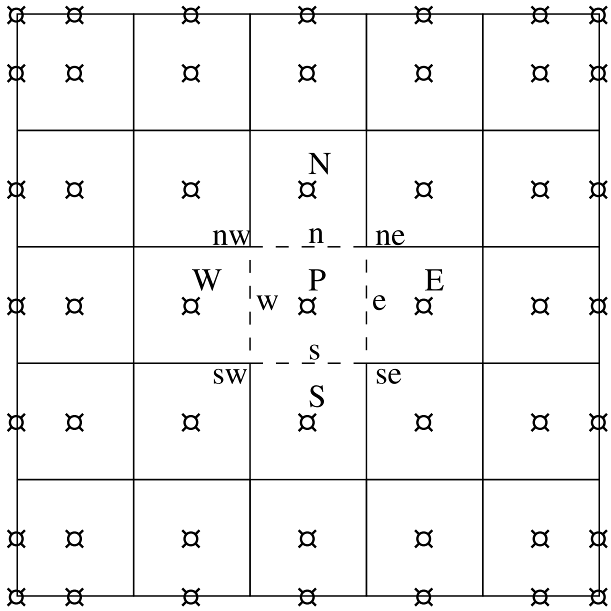

Figure A.1: The division of the domain into a finite number of control volumes. Two-dimensional example. The nodes are placed in the center of the control volumes except at the boundaries, where they are placed at the boundary. At the center control volume (dashed line), the nomenclatures for control volumes (nodes (P, E, W, N, S), faces (e, w, n, s) and corners (ne, nw, se, sw)) are introduced.

Equations A.4 and A.5 are used for performing the transformation to the computational space coordinates (general non-orthogonal coordinates) ξ_i. The scalar advection-diffusion equation A.5 is discretized. The integration of this gives (assuming that S is constant over the control volume)

$$\left(\vec{I}\cdot\vec{A}\right)_e + \left(\vec{I}\cdot\vec{A}\right)_w + \left(\vec{I}\cdot\vec{A}\right)_n + \left(\vec{I}\cdot\vec{A}\right)_s + \left(\vec{I}\cdot\vec{A}\right)_h + \left(\vec{I}\cdot\vec{A}\right)_l = S\delta V \tag{A.6}$$

where e, w, n, s, h and l refer to the faces of the control volume. The discretized equation is rearranged [12], using a differencing scheme for $\vec{I}$, to the standard form

$$a_P\Phi_P = \sum_{NB} a_{NB}\Phi_{NB} + S_C \tag{A.7}$$

where the source term has been linearized as $S = S_C + S_P\Phi_P$ and

$$a_P = \sum_{NB} a_{NB} - S_P$$

$$(NB = '\mathrm{neighbouring\ nodes}')$$

The coefficients a_{NB} contains the contribution due to convection and diffusion and the source terms S_P and S_C contains the remaining terms.
Solving this equation system (eq. A.7), using known or computed source terms, gives an approximate solution to the transport equation.

A.2 Filtering

The code is intended to be used for Large Eddy Simulations (LES) where the variables are filtered in space. The filtering is presented below, although it is not used for the laminar flow in the test cases.
A *filtered* variable $\bar{\Phi}$ is obtained by filtering the small-scaled variations of the variable Φ away [11], i.e.

$$\bar{\Phi}(\vec{r},t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} G\left(\vec{r}-\vec{\xi};\Delta\right)\Phi\left(\vec{\xi},t\right)d^3\vec{\xi}$$

where $\Delta = \Delta_x\Delta_y\Delta_z$.[3]
The filter function, G, is normalized by requiring that

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} G(\vec{r}-\vec{\xi};\Delta)d^3\vec{\xi} = 1$$

Here, the box filter [6]

$$G\left(\vec{r}-\vec{\xi};\Delta\right) = \begin{cases} 1/\Delta^3, & \text{if } |x_i-\xi_i| < \Delta_{x_i}/2 \\ 0, & \text{otherwise} \end{cases}$$

$$\Rightarrow \bar{\Phi}_i(\vec{r},t) = \frac{1}{\Delta^3}\int_{x-\Delta_x/2}^{x+\Delta_x/2}\int_{y-\Delta_y/2}^{y+\Delta_y/2}\int_{z-\Delta_z/2}^{z+\Delta_z/2}\Phi_i(\vec{\xi},t)d\xi d\eta d\gamma$$

i.e. **average over a small volume**, is applied.

A.3 Convergence criteria

The iterations in pt.I, section 3.1, terminates when the convergence criteria described below is fulfilled.

For the velocities, the residuals ($\mathrm{resor}(u_i)$) are calculated as (cf. eq. 3.7)

$$\mathrm{resor}(u) = \sum_{\text{all tasks}}\sum_{\text{all C.V.}}\left| a_P\bar{u}_P^{n+1} - \left(\sum_{NB} a_{NB}\bar{u}_{NB}^{n+1} + b - \alpha\frac{\partial\bar{p}^{n+1}}{\partial x}\delta V\right)\right|$$

For the continuity, the residual ($\mathrm{resor}(cont)$) is calculated as

$$\mathrm{resor}(cont) = \sum_{\text{all tasks}}\sum_{\text{all C.V.}}|\delta\dot{m}|$$

where

$$\begin{aligned}\delta\dot{m} &= (\bar{u}_e-\bar{u}_w)\,\Delta_y\Delta_z \\ &+ (\bar{v}_n-\bar{v}_s)\,\Delta_x\Delta_z \\ &+ (\bar{w}_h-\bar{w}_l)\,\Delta_x\Delta_y\end{aligned}$$

(i.e. the continuity error).
Velocity reference residuals ($\mathrm{reref}(vel)$) are chosen as the largest residual, amongst all the velocities, at the first iteration at the first time step[4] i.e.

$$\mathrm{reref}(vel) = \max(\mathrm{resor}_o(u), \mathrm{resor}_o(v), \mathrm{resor}_o(w)).$$

[3] Δ_x is the grid size in x-direction for the computational cell
[4] $\mathrm{resor}_o(u)$ denotes the residual for u at the first iteration at the first time step

A continuity reference residual ($\text{reref}(cont)$) is chosen as the largest residual, for the continuity, at the first iteration at the first time step i.e.

$$\text{reref}(cont) = \text{resor}_o(cont).$$

The largest relative residual at each iteration is then calculated as

$$\text{resmax} = \max\left(\frac{\text{resor}(u)}{\text{reref}(vel)}, \frac{\text{resor}(v)}{\text{reref}(vel)}, \frac{\text{resor}(w)}{\text{reref}(vel)}, \frac{\text{resor}(cont)}{\text{reref}(cont)}\right)$$

where 'relative' indicates that

$$0 \leq \text{resmax} \leq 1$$

The convergence criteria is fulfilled when

$$\text{resmax} \leq sormax$$

i.e. when the largest residual has been reduced by a factor $sormax$ eqs. 3.2 and 3.5 are considered solved and fulfilled, respectively.

B Parallelization

B.1 Introduction to PVM

To parallelize the program, PVM (Parallel Virtual Machine), available on **netlib**[5], is used. All the information needed is available on **Internet**, but the information relevant to this work is summarized here.
PVM is a message passing system that enables a network of UNIX (serial, parallel and vector) computers to be used as a single distributed memory parallel computer. This network is referred to as the virtual machine and a member of the virtual machine is referred to as a host. PVM is a very flexible message passing system. It supports everything from NOW (Network Of Workstations), with inhomogeneous architecture, to MPP (massively parallel systems). By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel. PVM provides routines for packing and sending messages between tasks. The model assumes that any task can send a message to any other PVM task, and that there is no limit to the size or number of such messages. Message buffers are allocated dynamically. So the maximum message size that can be sent or received is limited only by the amount of available memory on a given host.
Several avenues exist for getting help with using PVM. A bulletin board exist on **Internet** for users to exchange ideas, tricks, successes and problems. The news group name is `comp.parallel.pvm`. The PVM developers also answer mail as time permits. PVM problems or questions can be sent to `pvm@msr.epm.ornl.gov` for a quick and friendly reply.

B.2 PVM basics

Every program using PVM should include the PVM header file. This contains needed information about the PVM programming interface. This is done by putting
`include 'fpvm3.h'`
at the beginning of a Fortran program.
The first PVM function called by a program, usually `pvm_mytid()`, enrolls the process in PVM. All processes that enroll PVM are represented by an integer task identifier *tid*. The *tid* is the primary and most efficient method of identifying processes in PVM. When a PVM program is done it should call `pvm_exit`.

B.3 Compiling and running a PVM program

In order to run a PVM-program, it must be compiled. Some special flags are needed for the compilation. A normal compilation can look like
`f77 -c -r8 -o program program.f -L$(LDIR) -lfpvm3 -lgpvm3 -lpvm3 -lnsl -lsocket -lthread`
where `$(LDIR)` is the architecture-specific directory of the PVM libraries.
The program of this work is compiled using `make` in the appropriate directory.

[5]`http://www.netlib.org/`. See also `http://www.epm.ornl.gov/pvm/pvm_home.html`

After compiling, the executable must be moved to
`~/pvm3/bin/$PVM_ARCH`[6],
the architecture-specific directory of the executables. This is done automatically when using `make`. To run the program, PVM has to be started. This is done by typing `pvm` on any UNIX command line. Once started, the console prints the prompt:

```
pvm>
```

and accepts commands from standard input. If you get the message 'Cant Start pvmd', consult the online help on **Internet**. Now hosts can be added (`add`) and deleted (`delete`) to/from the configuration. To watch the configuration, type `conf`. To watch which tasks are executing, type `ps -a`. Type `help` on the PVM-prompt or consult the online help on **Internet** for more information. Leaving PVM, using `quit`, leaves the PVM-daemon running (using `halt` kills the deamon). The executable may then be run from a UNIX command line on any host in the virtual machine, like any other program.

B.4 Relevant PVM subroutines

To use PVM in a Fortran program, a number of subroutines has to be added to the code. On **Internet**, detailed descriptions of all the subroutines in PVM are available. In this section, PVM-routines relevant to this work are briefly described.

B.4.1 `pvmfbarrier`

Syntax: `pvmfbarrier(` *group*, *count*, *info*)
Blocks until *count* members of the *group*[7] have called pvmfbarrier. In general, *count* should be the total number of members of the group. If `pvmfbarrier` is successful, *info* will be 0. If some error occurs then *info* will be < 0.

B.4.2 `pvmfbcast`

Syntax: `pvmfbcast(` *group*, *msgtag*, *info*)
Broadcasts the data in the active messagebuffer to all the members of the *group*. The *msgtag* can be any integer, as an extra message to the *group*. If `pvmfbcast` is successful, *info* will be 0. If some error occurs then *info* will be < 0.

B.4.3 `pvmfcatchout`

Syntax: `pvmfcatchout(` *onoff*, *info*)
This routine causes the calling task (the parent) to catch output from tasks spawned after the call to `pvmfcatchout` when *onoff*=1. Characters printed on *stdout* and *stderr* in children and grandchildren (spawned by children)

[6] $PVM_ARCH is the computer architecture the executable is supposed to run on
[7] The tasks are usually members of one or more *group* of tasks

tasks are collected by the pvmds and sent in control messages to the parent task, which tags each line and appends it to the specified file. If `pvmfexit` is called while output collection is in effect, it will block until all tasks sending it output have exited, in order to print all their output. If `pvmfcatchout` is successful, *info* will be 0. If some error occurs then *info* will be < 0.

B.4.4 `pvmfexit`

Syntax: `pvmfexit(` *info*`)`
Tells the pvm daemon that the task is leaving PVM. If `pvmfexit` is successful, *info* will be 0. If some error occurs then *info* will be $<$0.

B.4.5 `pvmfgettid`

Syntax: `pvmfgettid(` *group*, *tid*, *inum*`)`
Returns the integer instance number (*inum*) of the task identified by a *group* name and a task id number(*tid*). If `pvmfgettid` is successful, *inum* will be a positive integer. If some error occurs then *inum* will be < 0.

B.4.6 `pvmfinitsend`

Syntax: `pvmfinitsend(` *encoding*, *bufid*`)`
Clears default send buffer and specifies message encoding. The encoding scheme used for packing the send buffer is specified by *encoding* (see manual). If `pvmfinitsend` is successful, then *bufid* will contain the message buffer identifier. If some error occurs then *bufid* will be < 0.

B.4.7 `pvmfjoingroup`

Syntax: `pvmfjoingroup(` *group*, *tid*`)`
Enrolls the calling task in the *group* and returns the task id number *tid* of this task in this *group*. Task id numbers start at 0 and count up, uniquely identifying the tasks. If some error occurs then *tid* < 0.

B.4.8 `pvmflvgroup`

Syntax: `pvmflvgroup(` *group*, *info*`)`
Unenrolls the calling task from the *group*. If there is an error when using `pvmflvgroup`, *info* will be < 0.

B.4.9 `pvmfmytid`

Syntax: `pvmfmytid(` *inum*`)`
Returns the integer instance number *inum* of the task. If there is an error, *inum* will be < 0. This routine is often used to enroll a task into PVM, thus generating a unique integer instance number *inum* ≥ 0. The routine can be called multiple times in an application.

B.4.10 `pvmfpack`

Syntax: `pvmfpack(` *datatype*, *var*, *icoun*, *stride*, *info*)
Packs the active message buffer with arrays of specified data type. The type of data being packed is specified by *datatype*. The beginning of a block of bytes is pointed at by *var*. The total number of items to be packed is specified by *icoun*. The stride to be used when packing the items is specified by *stride*. If the packing is successful, *info* will be 0. If some error occurs then info will be < 0.

B.4.11 `pvmfrecv`

Syntax: `pvmfrecv(` *inum*, *msgtag*, *bufid*)
Receives a message with the message tag *msgtag* from the task with integer instance number *inum*. The value of the new active receive buffer identifier is *bufid*. If there is an error then *bufid* < 0. The routine blocks until the message has arrived.

B.4.12 `pvmfreduce`

Syntax: `pvmfreduce(` *redop*, *var*, *icoun*, *datatype*, *msgtag*, *group*, *root*, *info*)
Performs a reduce operation (*redop*) on the variable *var* with size *icoun* and datatype *datatype* over the members of the *group*. An integer message tag (*msgtag*) is also applied. The result of the reduction operation appears on the user specified *root* task. The reduce operation need to be declared at the beginning of the program/subroutine, with

```
external PvmSum, PvmMax
```

for example. It is also possible to create custom-made reduce operations. Consult the manuals on how to do this. If the reduce operation is successful, *info* will be 0. If an error occurs *info* will be < 0.

B.4.13 `pvmfsend`

Syntax: `pvmfsend(` *inum*, *msgtag*, *info*)
Sends the data in the active message buffer, together with a message tag (*msgtag*), to the task specified by *inum*. If `pvmfsend` is successful, *info* will be 0. If some error occurs then info will be < 0.

B.4.14 `pvmfspawn`

Syntax: `pvmfspawn(` *task*, *flag*, *where*, *ntask*, *tids*, *numt*)
Starts up *ntask* copies of the exectutable named *task* on terminal *where* using a spawn option *flag*. The integer array *tids* of length at least *ntask* gets the task id:s of the PVM tasks started by this `pvmfspawn` call. If there is an error starting a given task, then that location in the array will contain the associate error code. The actual number of tasks started is returned by *numt*. Values less than zero indicate a system error. A positive value less than *ntask* indicates a partial failure. In this case the user should check the *tids* array for the error code(s).

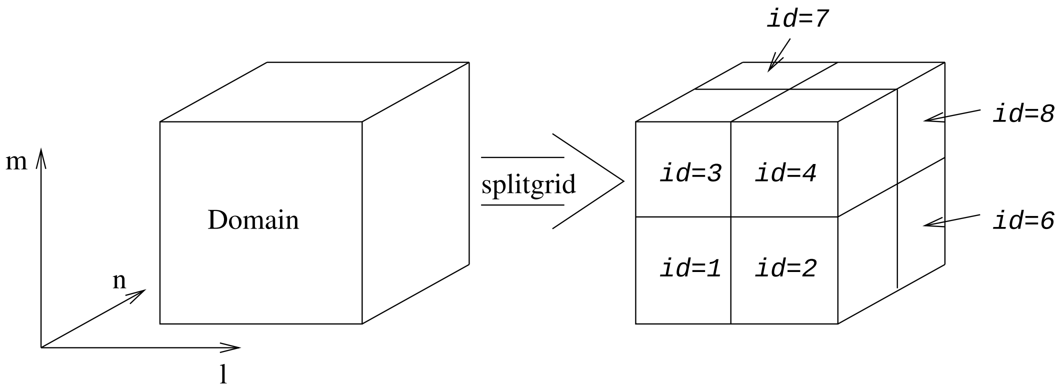

Figure B.1: Division of the domain, in `splitgrid`

B.4.15 `pvmfunpack`

Syntax: `pvmfunpack(` *datatype*, *var*, *icoun*, *stride*, *info*)
See pvmfpack. The messages should be unpacked exactly like they were packed to insure data integrity.

B.5 Parallelizing subroutines

This section contains a brief description of the subroutines developed in this work. They are a part of the parallelization of the code.

B.5.1 `splitgrid`

Syntax: `splitgrid`
In this subroutine, the grid is read from a *group*-specific file (see section B.9.2). The computational domain is divided into `lnodes`×`mnodes`× `nnodes` equally sized sub-domains (see figure B.1), using the task-specific id-number `id`, automatically specified in setup.f. The variables `lnodes`, `mnodes` and `nnodes` are specified by the user, in setup.f, and tells the program how many tasks the domain should be divided into in the `l`-, `m`- and `n`-directions respectively. By making the sub-domains equally sized, the computational times for the sub-domains should be approximately equal. This is however not always true, since the flow can be more or less complex in different parts of the domain. In this work it is assumed that all nodes require the same amount of computational effort. The sub-grids produced are extended to overlap four control volumes (one boundary node and one dummy-node from each task) at inner boundaries (see figure B.2). This ensures that every node is calculated and that second order accurate calculations can be performed at inner boundaries. A set of task identifiers (`idl`, `idm` and `idn`) are introduced. For instance, 1 ≤ `idl` ≤ `lnodes`, thus telling the program where the task is positioned in `l`-direction[8]. This makes it easy to locate inner/outer[9] boundaries and neighbouring tasks.

[8] l denotes the same direction as i in fig. B.2, but it is used when identifying tasks instead of identifying nodes

[9] The boundaries defining the domain are referred to as outer boundaries

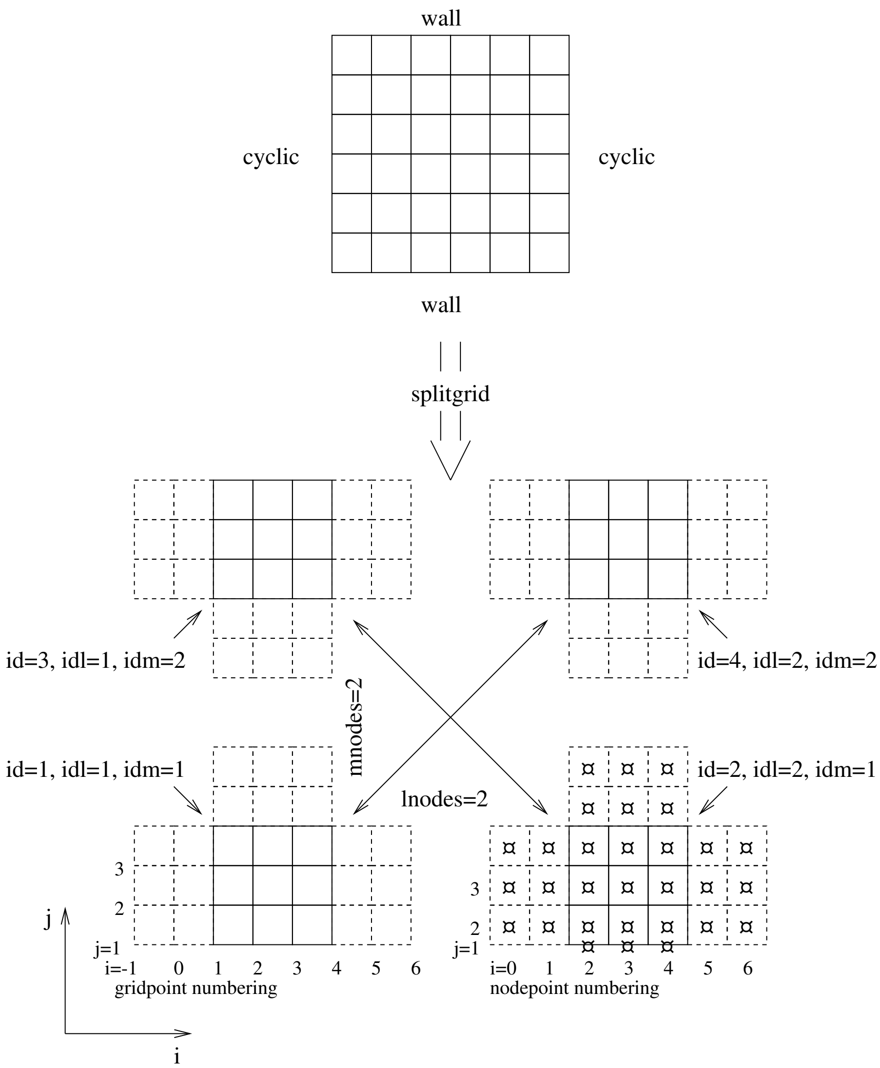

Figure B.2: Division of the grid, in `splitgrid`

Correct values of `ni`, `nj` and `nk`[10] are calculated. The number of grid points in `i`/`j`/`k`-direction for each task is then given by `nim1`/`njm1`/`nkm1`[11]. Only the grid-values corresponding to the specific task are stored, in order to save memory.

B.5.2 `pvmbound`

Syntax: `pvmbound`
This subroutine looks for inner boundaries, connect the tasks at the inner boundaries and send inner boundary values between the tasks. It sends the node values for all variables in two inner-boundary node-planes (node i=2..3, j=2..`njm1` in fig. B.2) to the neighbouring tasks. It then receives the values from the neighbouring tasks and applies them as new boundary values and a dummy-node plane outside the inner boundary (node i=0..1, j=2..`njm1` in fig. B.2). The dummy node plane is used when second order accurate calculations are needed. See section B.9.4 for an example of how the information is sent and received.

B.5.3 `pvmbphi`

Syntax: `pvmbphi( variable)`
This subroutine does the same thing as `pvmbound`, but only for one variable, specified when calling the subroutine. This enables packing and sending as little information as possible.

B.5.4 `pvmbphic`

Syntax: `pvmbphic`
This subroutine does the same thing as `pvmbound`, but only for `phic` (the intermediate velocity field). This enables packing and sending as little information as possible.

B.5.5 `residual`

Syntax: `residual`
This subroutine calculates the global[12] residual `res`, for the pressure, used as a convergence criteria in the Poisson equation solver (see section B.9.6).

B.5.6 `stopprog`

Syntax: `stopprog`
This subroutine enables the tasks to terminate the program at the same time, thus letting all output from children tasks be received and printed by the parent task (see section B.9.7).

[10] `ni`, `nj` and `nk` are the number of computational nodes in each direction for each task. In fig. B.2; `ni=nj=5`

[11] `nim1=ni-1`, `njm1=nj-1`, `nkm1=nk-1`

[12] The term **global** is used when something is computed over the whole domain, i.e. over all tasks

B.6 Modifications to standard routines

This section describes the parallel multiblock extensions to the standard subroutines in CALC-BFC. For a detailed description of the rest in the subroutines, consult CALC-BFC [4].
Inner and cyclic boundaries are as far as possible treated as if there were no boundaries at all, i.e. all calculations have been extended to include boundary and dummy nodes. Below, some of these changes are described.
The exchange of inner boundary values are accomplished by calling an appropriate communication subroutine (see section B.5) at an appropriate location (see section 3.1).

B.6.1 `setup`

Syntax: `setup`
The parallel extension is initialized. Tasks are spawned. *Group* and *task* specifics are created. See section B.9.3.

B.6.2 `init`

Syntax: `init`
The loops, calculating the geometrical quantities (areas, volumes and weight-functions), are extended to include inner and cyclic boundaries, thus treating these boundary control volumes as any other inner control volume.

B.6.3 `coeff`

Syntax: `coeff`
The loops, calculating the coefficients (used when solving the momentum equations), are extended to include inner and cyclic boundaries, thus treating these boundary control volumes as any other inner control volume.

B.6.4 `calcpe`

Syntax: `calcpe`
The loops, calculating the coefficients (used when solving the Poisson equation), are extended to include inner and cyclic boundaries, thus treating these boundary control volumes as any other inner control volume.

B.6.5 `main`

Syntax: `main`
The global residual is calculated and broadcast to all tasks (see section B.9.5). This ensures that the solution is the same as for the whole domain, calculated in one task, and that the tasks terminates the timesteps at the same time (which is necessary).

B.6.6 `slapsolver`

Syntax: `slapsolver( variable)`
A pressure reference level (chosen from node (i, j, k) = (2, 2, 2) in task `id` = 1) is subtracted from all nodes in all subdomains. The global residual `res`, for the pressure, is calculated using the subroutine `residual` (see section B.9.6) and broadcast to all tasks. This ensures that the solution of the Poisson equation is the same as for the whole domain, calculated in one task, and that the tasks terminates the solving of the Poisson equation at the same time (which is necessary).

B.6.7 `save1`

Syntax: `save1( filename)`
Saves to task- and group-specific files, not to interfer with other tasks or groups.

B.7 Running a new problem

When running a new problem, the following steps has to be done.

`COMMON`

Change the parameters `it`, `jt` and `kt` to values a little bit greater than the task values of `ni`, `nj` and `nk`, say; `it` = `ni` + 3,`jt` = `nj` + 3 and `kt` = `nk` + 3.

`mod.f`

Set up the boundary and initial conditions in `mod.f`. It is very important to make sure that the appropriate boundaries (not inner boundaries) are addressed, when applying boundary conditions. This is not always as easy as it seems, so convince yourself that you got it right.

`setup.f`

Set the constants and logical parameters in `setup.f`. Especially, set `cycli`, `cyclj`, `cyclk`, `lnodes`, `mnodes`, `nnodes`, `group`, `ntstep` and `dt(i)`. The `cycl_`-parameters denotes cyclic boundaries, see section B.5. The `_nodes`-constants describes the division of the domain, see section B.5. The string `group` is used by PVM, the executable name and the output and input file-names. To be able to run several programs at once, the `group` name must be program-specific. `ntstep` is the number of timesteps of length `dt(i)` to be calculated. In order to get a CFL-number ($u_i \Delta t/\Delta x$) less than 2, it is important to set `dt(i)` to a value small enough.

`slapsolver.f`

Change the constants `tol` and `res` to get a good convergence rate. `tol` is the pressure residual reduction, for each task, each time through the slapsolver. If a singletask[13] problem is calculated, `tol` is also the global pressure convergence criteria. `res` is the global pressure convergence-criteria for multitask[14] calculations. `res` for a multitask case should be the same as `tol` for the equivalent singletask case, to reach the same pressure convergence-criteria.

Compile

Compile the program, using `make` in the directory of the Fortran files. The executable will automatically be moved to the directory of the executables; `~/pvm3/bin/$PVM_ARCH`. The program has to be compiled on every architecture occuring in the virtual machine configuration.

Executable name

Change the name of the executable. The name depends on the `group` name, specified in `setup.f`, as `calc_`*group(1:3)*[15]

Grid

Create a grid and save it as described in section B.9.2.

Run

After starting PVM and adding a few more hosts than needed[16], the program can be run from the UNIX prompt in the directory of the executables, writing
`calc_`*group(1:3)* > `out`*group(1:3)* `&`
PVM then chooses the most appropriate hosts to run the program on. To get the best performance of the program, check that no host has multiple tasks. This is done by typing `ps -a` on a PVM-command line. If a host has multiple tasks type `reset` on a PVM-command line and rerun the program. It does not seem to be possible to avoid this problem automatically in Fortran. When running in a 'que'-system on a parallel computer a script, see section B.9.1, has to be used. The script is supposed to start PVM and `quit`, run the program and `halt`. On parallel computers no hosts are supposed to be added. PVM distributes the tasks amongst the processors by itself.

Results

When the program has terminated, the results can be found in
`~/pvm3/bin/$PVM_ARCH/output/`*group(1:3)*`id`*id*`.dat`.
To view the results in **TECPLOT**, the output files has to be connected as

[13] The term singletask is used when a program is using only one task

[14] The term multitask is used when a program is using more than one task

[15] *group(1:3)* denotes the first three letters in `group`

[16] Adding a few more hosts than needed makes it possible for PVM to choose hosts that are not heavily loaded

`cat lidid1.dat lidid2.dat lidid3.dat lidid4.dat > lid.dat`[17],
and processed with
`preplot lid.dat`,
generating a `lid.plt` - file that can be viewed in **TECPLOT**.

B.8 Problems and solutions

This section contains a list of some of the problems and solutions encountered during the implementation of the code. If any other problem occurs, read the manual, consult the information on **Internet** or send a question to the news group (`comp.parallel.pvm`).

- When a couple of `pvmfreduce` calls are placed directly after each other, there is a risk of losing information. Temporarily, this has been solved by calling `pvmfbarrier` between the calls to `pvmfreduce`.
- When starting PVM, a PVM-daemon file is temporarily placed in `/tmp/pvmd.<uid>`. If PVM has been abnormally stopped, this file continues to exist and prevents PVM from being restarted on that particular host. Remove the file and PVM will be able to start again.
- If the program has been abnormally stopped, there might be a number of hosts still running. When trying to run the program again, with the same `group`-name, the tasks will get inappropriate task id numbers. The program can still be run, but the execution will halt at the first communication since PVM can not find the appropriate hosts. By typing `reset` on a PVM command line, the remaining active hosts will disappear and enable the program to be run again.
- When spawning a task, it is always started in the HOME-directory. To make it possible for child tasks to find necessary files, some UNIX environment variables has to be exported. By writing `setenv PVM_EXPORT PVM_ROOT:PVM_DPATH`[18] on a UNIX command line (or in `.cshrc`), the variables `PVM_ROOT` and `PVM_DPATH` will be exported when spawning. `PVM_ROOT` is the directory of the executables and `PVM_DPATH` is the path of the daemon.
- When reading/writing from/to files, the whole path has to be declared since the child tasks are started in the HOME-directory.
- To be able to run several PVM programs at the same time, on the same hosts, the `group`-name must be program-specific. The reasons for this is that: **1)** the groupname is included in the program-name, **2)** the task id numbers becomes wrong otherwise and **3)** the output is written to `group`-specific files (not to interfere with other programs).

[17] Here *group(1:3)*=`lid` and *id*=`1, 2, 3` and `4`
[18] PVM_ROOT = ~/pvm3, PVM_DPATH = $PVM_ROOT/lib/pvmd

- For a PVM-program to work, the executable must have the same name as the *task*-string used when spawning the child tasks with pvmfspawn.

B.9 Technical descriptions

This section contains and describes the computer codes used in this work. Vital parts of the parallel multiblock extension are displayed in order to give the reader an idea of how the PVM subroutines are used in the code.

B.9.1 Shell script

To be able to run and time the program in a 'que', a shell script had to be written. Here, the script used for the speed tests (see section 6) is displayed.

```
#!/bin/ksh

PROG=calc_un1

HOME=/users/tfd/gu92hani
export PVM_ROOT=$HOME/pvm3
export PVM_ARCH=`$HOME/pvm3/lib/pvmgetarch`
export PVM_DPATH=$PVM_ROOT/lib/pvmd
export PVM_EXPORT=PVM_ROOT:PVM_DPATH

echo quit | $PVM_ROOT/lib/pvm

cd $PVM_ROOT/bin/$PVM_ARCH
time $PROG > $PROG.log
echo halt | $PVM_ROOT/lib/pvm
```

B.9.2 The grid

The domain is divided into a finite number of control volumes. The locations of the corners of the control volumes forms a grid, which describes the control volumes. Since the problem is three-dimensional, we need three indexes to describe the grid. The indexes i, j and k are used here. They number the gridpoints (the corners of the control volumes), starting with (1,1,1) in one corner of the domain and ending with (nim1,njm1,nkm1) in the opposite corner of the domain. The number of gridpoints in the i-, j- and k-directions are thus nim1, njm1 and nkm1 respectively.
To describe one point in space we need to give its position in three directions; x, y and z. xc(i,j,k), yc(i,j,k) and zc(i,j,k) are three-dimensional matrices that contains x-, y- and z-position, respectively, for gridpoint (i,j,k).
The grid is generated in a separate program, as described above, and saved in a grid-file arranged as follows:

```
write(unit,*)nim1+1,njm1+1,nkm1+1
do k=1,nkm1
```

```
do j=1,njm1
do i=1,nim1
     write(unit,*)xc(i,j,k), yc(i,j,k), zc(i,j,k)
end do
end do
end do
```

The program automatically reads the appropriate grid from
`~/pvm3/bin/$PVM_ARCH/grids/`*group(1:3)*`grid.dat`.
It is up to the user to make sure that this file exists and contains the correct grid before the program is run. The smallest number of grid points in a symmetric direction (see section 4.3) is two, to make room for an inner node-plane.

B.9.3 Initializing the parallel computations

In `setup.f`, the parallel computations are initialized. The procedure below spawns tasks, creates *group* and *task* specifics and calls `splitgrid`, to get the appropriate part of the grid for each task.

```
c- section 12 ----grid specification ----------------------------------
c
c Prepared for PVM-programming
c lnodes is number of computational tasks in i-direction
c mnodes is number of computational tasks in j-direction
c nnodes is number of computational tasks in k-direction
c id tells the program which task to compute

      if (pvm) then
         lnodes=2
         mnodes=1
         nnodes=4
c        ----------
c        Enroll pvm
c        ----------
         call pvmfmytid( mytid )

         if( mytid .lt.  0 ) then
            write(6,*)'Error in pvmfmytid error=',mytid
            stop
         endif

c        ------------------------------------------
c        Join a group and if I am the first instance
c        i.e.  id=1 spawn more copies of myself
c        ------------------------------------------
         group='fur' !  Allways 3 char.  in filenames.  Abs.  max 15 char.
         call pvmfjoingroup( group, idt )
         id=idt+1
```

```
         ntasks=lnodes*mnodes*nnodes
         call pvmfcatchout(1,info)
         if (idt.eq.0.and.ntasks.gt.1) then
            call pvmfspawn('calc_'//group(1:3),PVMDEFAULT,'*',
                  ntasks-1,tids(1),info)
         endif
         if (idt.eq.0) then
            write(6,*)'Actual number of tasks :  ',ntasks
            write(6,*)'Number of tasks started:  ',info+1
         endif
c        ---------------------------------------------
c        Wait for everyone to startup before proceeding
c        ---------------------------------------------
         call pvmfbarrier( group,ntasks,info)
         write(6,*)'My task id is ',mytid,' id= ',id
c-------------------------------------------------------------------
      else
c        never change these values
         lnodes=1
         mnodes=1
         nnodes=1
         id=1
      end if

c-----Open outid1.dat etc.  specific for each id.
      if (echo) open(unit=20,file=
          ' /pvm3/bin/SUN4SOL2/out'//group(1:3)//'id'
          //char(id+48)//'.dat')

      call splitgrid
```

B.9.4 Common send/receive procedure

When sending and receiving information (*in subroutines* `cyclic`, `cycphi`, `cycphic`, `dummynodes`, `pvmbound`, `pvmbphi` and `pvmbphic`), the procedure looks as follows (example from `pvmbound`);

```
subroutine pvmbound
include '../COMMON'
include '~/pvm3/include/fpvm3.h'
parameter (nphitt = nphit+1)
dimension phiew(2,jt,kt,nphitt)
integer src, dest, info

if (idl.ne.1) then
    do 10 j=1,nj
    do 10 k=1,nk
    do 10 l=1,nphitt
       if (l.ne.nphitt) phiew(1,j,k,l)=phi(2,j,k,l)
       if (l.ne.nphitt) phiew(2,j,k,l)=phi(3,j,k,l)
```

```
      if (l.eq.nphitt) phiew(1,j,k,l)=vis(2,j,k)
      if (l.eq.nphitt) phiew(2,j,k,l)=vis(3,j,k)
10  continue
    call pvmfinitsend( PVMDEFAULT, info)
    icoun=2*jt*kt*nphitt
    call pvmfpack( REAL8, phiew, icoun, 1, info)
    call pvmfgettid(group, idt-1, dest)
    call pvmfsend(dest, 1, info)
end if

if (idl.ne.1) then
    call pvmfgettid(group, idt-1, src)
    call pvmfrecv( src, 1, info)
    icoun=2*jt*kt*nphitt
    call pvmfunpack( REAL8, phiew, icoun, 1, info)
    do 30 j=1,nj
    do 30 k=1,nk
    do 30 l=1,nphitt
      if (l.ne.nphitt) phi(0,j,k,l)=phiew(1,j,k,l)
      if (l.ne.nphitt) phi(1,j,k,l)=phiew(2,j,k,l)
      if (l.eq.nphitt) vis(0,j,k)=phiew(1,j,k,l)
      if (l.eq.nphitt) vis(1,j,k)=phiew(2,j,k,l)
30  continue
end if
return
```

This procedure sends (west) and receives (from west) nphitt = 5 variables (u = phi(i,j,k,1), v = phi(i,j,k,2), w = phi(i,j,k,3), p = phi(i,j,k,4) and ν = vis(i,j,k)) in two inner boundary node planes, using the temporary variable phiew. nj and nk are the number of computational nodes in j- and k-direction for each task. jt and kt are parameters set in COMMON, defining the maximum size of the computational domain (including dummynodes) for each task. They must be at least be it = ni + 2. To locate the inner boundary, idl (see fig. B.2) is used.

B.9.5 Calculating the global residual

In main.f, the global residuals for all variables are calculated using the procedure below.

```
c--------calculate and broadcast the residuals-------------------------
c-----Calculate resor for u,v,w and p in id=1
      ntasks=lnodes*mnodes*nnodes
      if (pvm) then
         call pvmfbarrier(group,ntasks,info)
         call pvmfreduce(PvmSum,resor,4,REAL8,1,group,0,info)
      end if

c-----Calculate reref of u,v,w and p in id=1
      if (id.eq.1) then
         do 150 nphi=1,3
```

```
            if (itstep.eq.1.and.iter.eq.1) reref(nphi)=0.
            reref(nphi)=max(reref(nphi),resor(1),resor(2),resor(3))
150       continue
          if (itstep.eq.1.and.iter.eq.1) reref(p)=0.
          reref(p)=max(reref(p),resor(p))
        end if

c-----Calculate resmax over all tasks in id=1
        resmax=0.
        if (id.eq.1) then
          do 120 nphi=1,nphmax
            res=resor(nphi)/reref(nphi)
            if (nphi.eq.te.or.nphi.eq.ed) res=0.
            resmax=max(resmax,res)
120       continue
        end if

c-----Broadcast resmax from id=1 to all tasks
        if (pvm) then
          if (id.eq.1) then
            call pvmfinitsend(PVMDEFAULT,info)
            call pvmfpack(REAL8,resmax,1,1,info)
            call pvmfbcast(group,1,info)
          else
            call pvmfrecv(-1,-1,info)
            call pvmfunpack(REAL8,resmax,1,1,info)
          end if
        end if
c--------calculate and broadcast the residuals--------------------------
```

B.9.6 The pressure residual

For the Poisson equation solver to know when to terminate the iterations solving for the pressure, a global residual for the pressure is calculated using `residual.f`, displayed below.

```
      subroutine residual(res)

      include 'COMMON'
      include '~/pvm3/include/fpvm3.h'
c --- External declaration of PVM reduce function
      external PvmSum

      resorr=0.
      bb=0.
      nphi=p
      resm=0.

      do 102 k= 2,nkm1
      do 102 j= 2,njm1
```

```
      do 102 i= 2,nim1
c---------residual
         res= an(i,j,k)*phi(i,j+1,k,nphi)
            +as(i,j,k)*phi(i,j-1,k,nphi)+ae(i,j,k)*phi(i+1,j,k,nphi)
            +aw(i,j,k)*phi(i-1,j,k,nphi)+ah(i,j,k)*phi(i,j,k+1,nphi)
            +al(i,j,k)*phi(i,j,k-1,nphi)+su(i,j,k)
            -ap(i,j,k)*phi(i,j,k,nphi)
         resorr= resorr+res**2
         resm=resm+abs(res)
         bb=bb+su(i,j,k)**2
102   continue

c-----Calculate total residual
      tasks=lnodes*mnodes*nnodes
      if (pvm) then
         call pvmfbarrier(group,ntasks,info)
         call pvmfreduce(PvmSum,resorr,1,REAL8,1,group,0,info)
         call pvmfbarrier(group,ntasks,info)
         call pvmfreduce(PvmSum,resm,1,REAL8,1,group,0,info)
         call pvmfbarrier(group,ntasks,info)
         call pvmfreduce(PvmSum,bb,1,REAL8,1,group,0,info)
      end if

      res=sqrt(resorr/bb)

c-----Broadcast res from id=1 to all tasks
      if (pvm) then
         if (id.eq.1) then
            call pvmfinitsend(PVMDEFAULT,info)
            call pvmfpack(REAL8,res,1,1,info)
            call pvmfbcast(group,1,info)
         else
            call pvmfrecv(-1,-1,info)
            call pvmfunpack(REAL8,res,1,1,info)
         end if
      end if

      return
      end
```

B.9.7 Stopping the program

In order to stop the program without losing output from child tasks, this stopping-subroutine had to be written. It waits for all the group members before terminating the program.

```
      subroutine stopprog
      include '~/pvm3/include/fpvm3.h'
      include '../COMMON'
```

```
ntasks=lnodes*mnodes*nnodes
call pvmfbarrier(group,ntasks,info)
call pvmflvgroup( group, info )
call pvmfexit(info)
write(6,*)'Program reached normal stop.'
stop
end
```