Parallelization of a Multi-Block Navier-Stokes Solver

Patrik Rautaheimo, Esa Salminen and **Timo Siikonen** [1]

Abstract. A parallelization of a Navier-Stokes solver is presented. The flow solver is based on a multi-block structured grid. The parallelization is performed over the blocks, and the data on the block boundaries is exchanged using the MPI Standard. The parallelized code can also be run in a single processor mode or on a shared memory machine. In order to facilitate the pre- and post-processing a separate program for a domain decomposition has been written. The first tests indicate a good scaleability of the parallelization approach.

1 INTRODUCTION

For over a decade parallelization has been used to enhance the efficiency of flow solvers. The simplest method of parallelization, which can be used with shared memory machines, takes place on the DO-loop level. DO-loop level parallelization is ineffective for a large number of processors. Better performance from a large number of processors can be obtained by dividing the space into smaller sub-domains. With a shared memory machine like the Cray C94, the parallelization over the sub-domains is a trivial task, but with a massively parallel system like the Cray T3D things get more complicated. A common approach, applied e.g in [1] and [5], is to divide the computational domain into equally sized blocks and to apply message passing between the blocks.

In this paper the parallelization of a multi-block Navier-Stokes software is described. The parallelization is based on the Message Passing Interface (MPI) Standard [2]. The computational domain is divided into blocks and the block boundaries are updated using MPI. In order to get a balance between the processes, the blocks should be of equal size. However, the code can handle several (smaller) blocks in one process. This property can be utilized especially with small cases and with a small number of processors, when a good load balance is not so critical.

In addition to the changes in the flow solver, a separate preprocessor has been written to make the domain decomposition. This is because during the pre- and post-processing the grid and the results can be more easily handled in a few larger blocks. In the domain decomposition the most difficult task is to handle the definition of boundary conditions automatically from the original boundary file.

In the following, the flow solver and the changes required for the parallelization are briefly described. Next, the principles of the domain decomposition are given. Test runs have been performed with the T3D machine and with a cluster of SGI Indigo^2 workstations.

[1] Laboratory of Applied Thermodynamics, Helsinki University of Technology, 02150 Espoo, Finland

Computational Fluid Dynamics '96
Published in 1996 by John Wiley & Sons, Ltd.

2 BASIC FEATURES OF THE FLOW SOLVER

2.1 Numerical method

The flow simulation is based on the solution of the Reynolds averaged Navier–Stokes equations:

$$\frac{\partial U}{\partial t} + \frac{\partial (F - F_v)}{\partial x} + \frac{\partial (G - G_v)}{\partial y} + \frac{\partial (H - H_v)}{\partial z} = Q \qquad (1)$$

where U is the vector of dependent variables, F,G,H and F_v,G_v,H_v represent the inviscid and viscous parts of the fluxes, and Q is a possible source term. The flow solver utilizes a structured multiblock grid. For the solution Eq. (1) is written in a finite-volume form

$$V_i \frac{dU_i}{dt} = \sum_{\text{faces}} -S(\hat{F} - \hat{F}_v) + V_i Q_i \qquad (2)$$

where V_i is a cell volume, $\hat{F}$ and $\hat{F}_v$ are the inviscid and viscous parts of the flux on the cell surface. The sum is taken over the faces of the computational cell.

The solution proceeds blockwise after explicitly defined boundary conditions. The boundary conditions between the blocks are defined only on the highest grid level. In each block an implicit LU-factored solution with a multigrid acceleration of convergence is performed [6]. The underlying solution method is based either on the flux-difference [4] or flux-vector [9] splitting. The flux calculation utilizes a MUSCL-type differencing with a second- or third-order accuracy. The code has been applied for external [7] and internal [8] flows. Turbulence is modeled either by an algebraic model or a two-equation model. A Reynolds-stress model is under development and has been applied for simple test cases [3].

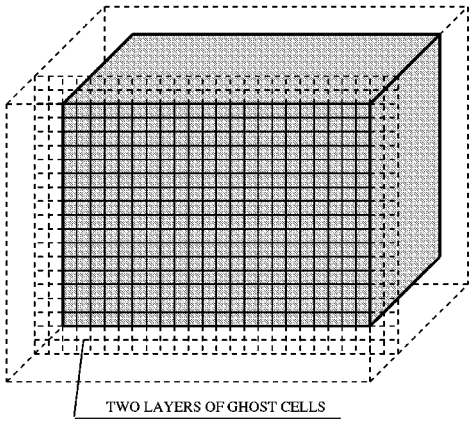

Figure 1. The blocks are surrounded by two layers of ghost cells.

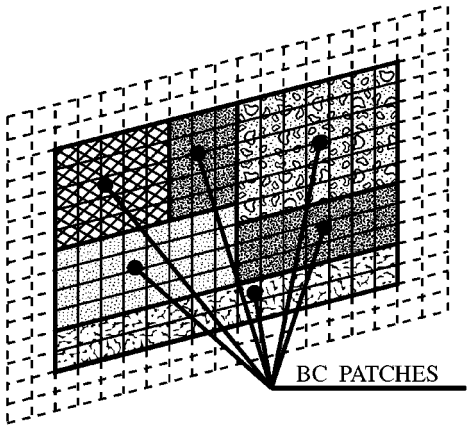

Figure 2. Different boundary conditions can be applied on the patches of the block surfaces.

2.2 Treatment of boundary conditions

The boundary conditions are handled using two layers of ghost cells on the block boundaries to preserve second-order accuracy, as seen in Fig. 1. The list of boundary conditions is given in Table 1. For connectivity and cyclic boundary conditions information is exchanged between the blocks. Since the blocks can be connected in an arbitrary way, the block boundaries are divided into 'patches'. A patch is formed by the common boundary shared by two adjacent blocks. Thus the block surface can contain several patches with different boundary conditions, as shown in Fig. 2. The definition of the patches and the corresponding boundary condition types are given as input data in a specific boundary condition file.

With a shared memory machine the patch data is written on a boundary array after an iteration cycle. The receiving block reads the information from this array at a given position. This procedure is performed block by block: firstly every block writes the values of the dependent variables on all its patches into the array and then the data is substituted from the array to the ghost cells of the appropriate blocks. Only the central memory of the machine is utilized in this approach.

Table 1. Boundary condition types.

Boundary condition	Exchange of data
Connectivity	yes
External	no
Inlet	no
Mirror	no
Outlet	no
Cyclic	yes
Singularity	no
Solid	no
Rotating solid	no
Moving solid	no

2.3 Computation and data arrangement

The computer code is programmed using standard Fortran77. The variables are stored in one-dimensional arrays. Starting addresses are utilized in calling routines to separate data for different blocks and for different multigrid levels. Computation is performed using three kinds of DO-loops. For some variables, e.g. for the calculation of the equation of state, there is a single loop over the entire block including the ghost cells. In order to exclude the ghost cells a three-level loop over the i-, j- and k- directions is utilized. Most of the calculation, including the evaluation of the fluxes and the performation of the implicit sweeps, is done slabwise as shown in Fig. 3. In this approach the ghost cells at the beginning and at the end of the row are included in the calculation, which increases the amount of calculation typically by a few per cent depending on the grid size. This treatment was originally designed for a vector computer, where the increased amount of computation is more than compensated by the enhanced efficiency owing to a longer vector. The treatment has been

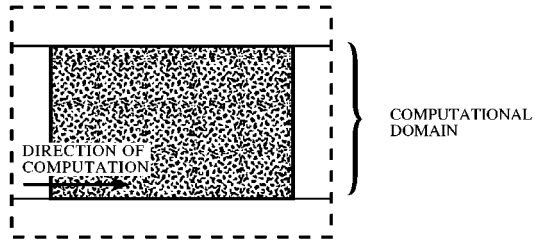

Figure 3. Computation proceeds slabwise including two parts of the ghost cell layers.

retained in the parallelization in order to maintain the original structure of the code and to facilitate portability between the vector- and RISC-machines.

3 PARALLELIZATION

3.1 Main principles

The code structure forms an ideal base for the parallelization. All the essential procedures are treated block by block including the updating of boundary conditions. By using block sizes like 32^3,40^3 and 48^3 several multigrid levels can be used in each block. If all the blocks are of an equal size, the work between the processors is balanced. With the current RISC processors and the block sizes given above, the calculation takes of the order of 10 seconds per iteration cycle. Since the majority of the calculation takes place slabwise, it is impractical to use very small block sizes owing to the useless computation of ghost cells. Even more important is to obtain a suitable balance between the times spent on the computation and the communication, which with current fast processors requires that blocks have a sufficient size.

There are some general requirements for the parallelization. Firstly, we wish that the same software can be used in a single processor mode or with a shared memory multiprocessor machine like the C94. In practice it is difficult to always use the specified block sizes. Because of this the possibility to compute several blocks per processor has to be retained. This property is important especially in small cases, where a good load balance is not so critical, and which can be calculated using a few processors. In large cases with complicated geometries the division into equally sized blocks may also be difficult or impractical. Then small blocks can be situated in the same processor or, if the number of small blocks is small in comparison with the standard equally-sized blocks, the idle time of the processors occupied by the small blocks does not decrease the overall efficiency significantly.

3.2 Parallelization using MPI

The parallelization is based on the Message Passing Interface (MPI) Standard [2]. MPI routines are implemented so that the program runs also in an environment where MPI is not implemented. The updating of boundaries between different processes is done using the basic MPI_SEND and MPI_RECV commands instead of using the array for boundary data, as in the case of a shared memory calculation. Also MPI_BCAST and MPI_GATHER are used to give input parameters to the processes, and to gather convergence histories. Due to the development history of the code, the communication between the boundaries is performed for each variable to be solved separately, instead of using a single pair of the MPI_SEND and MPI_RECV commands. These commands are performed in each block simultaneously using the standard communication mode of MPI.

The computational cycle is described in Fig. 4. One processing element (PE0) is the master and others are slaves.

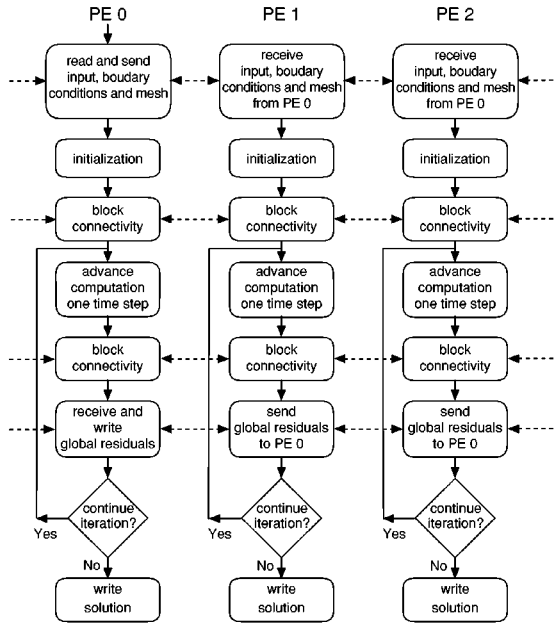

Figure 4. A flow diagram of the parallelized code. Communication between the processors is depicted by horizontal dashed lines.

Parallelization is done so that only the master process reads in the input parameters, but all the processes write the output files. The master process reads input parameters, including files where boundary conditions are specified and the grid is defined, and sends the desired input parameters and the appropriate parts of the grid to the slaves. After every iteration cycle slave processes send to the master convergence parameters (global residuals etc.), which are printed on a screen and stored in a convergence monitoring file. This is accomplished using the MPI_GATHER command. Because processes are highly independent of each other, the memory requirement per process comes from the size of the block(s) that a process simulates. Since the possibility to calculate a different number of differently sized blocks has been retained, a dynamic memory allocation is performed in each process separately.

Inside the process the communication can be done by using the central memory of the machine (default) or MPI subroutines can be utilized for a possible debugging.

4 AUTOMATIC GRID BLOCK SPLITTING

4.1 Main principles

In order to utilize the computing power of massively parallel system a program utilizing a simple domain decomposition algorithm has been developed for dividing large grid blocks into smaller ones. In addition to the grid splitting, the program

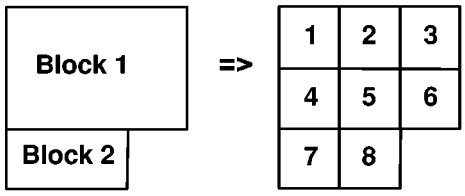

Figure 5. An example of the division of the original grid blocks into equally-sized sub-blocks.

also rewrites the boundary condition file and the computation control file. A good balance between processes is desirable. This can be achieved by dividing the space into equally sized sub-domains. However, from the point of view of grid generation, the above requirement increases the amount of manual work. This can be avoided by generating the original grid without considering too much the requirements of the parallel processing. With a separate tool the grid can be divided into sub-blocks suitable for parallel computing.

The goal is that the user does not need to work at all with the small blocks during the pre- and post-processing stages. In order to simplify the splitting algorithm it is assumed that the original blocks are directly divided into sub-blocks, i.e. the sub-blocks can occupy only a single original block, as shown in Fig. 5. Because of this the efficiency requires that the desired sub-block size, e.g. $40 \times 40 \times 40$, is taken into account in the grid generation. However, since the number of computational blocks per processor is arbitrary, the user can also deviate from the requirement of equal sub-blocks.

The grid splitting software keeps a record of the grid division so that after the simulation the sub-blocks can be merged into the original form. This is done for the grid itself and for the solution files in order to facilitate post-processing.

4.2 Grid splitting

The splitting program can be run in two different modes. In the first mode, the user explicitly defines the grid sub-block boundaries. Then the only task of the program is to rewrite the boundary condition file. In the second mode, the program automatically splits the blocks. The only information required from the user in the latter case is the desired block edge dimension.

In the automatic mode, the splitting strategy is as follows: The block is always fully cut. If the block edge dimension is smaller than the desired one, no cutting will take place. If the block edge dimension is larger than the desired one, but smaller than twice the desired size, a cutting line from the middle of the block face will be chosen as shown in Fig. 6a. In the future this could be improved by trying to leave a larger block on the possible solid wall side. This will make things more complicated, but improves the behaviour of the turbulence models, which require wall correction. If the block edge dimension is larger than twice the desired size, but cannot be equally distributed, the smaller block will be cut from the middle of the original block, i.e. the block next to the possible solid wall is always as large as possible. The resulting division is shown in Fig. 6b.

4.3 Redefinition of the boundary conditions

The boundary condition patch splitting is a complex task in comparison with the grid block splitting described above. This is especially true in cases where two connected blocks will not be cut at the same location. That is why the algorithm divides the boundary patches using the cutting lines from both blocks. First the limits of the new boundary condition patches are computed. As a separate step, the connectivity information is updated. The program does not assume anything about the grid topology. That is, the connections in C-type or O-type blocks do not need any special treatment.

The most challenging task in the BC patch splitting process is the determination of the additional cutting lines from connected patches. If the original blocks 1 and 2 are divided as shown in Fig. 7, the boundary patches in the neighbouring blocks will be cut correspondingly. When we calculate these lines, we must consider which block faces are connected and what is the orientation between the blocks. An additional difficulty comes from the relative position of the blocks. Since there are six faces on both blocks and four possible orienta-

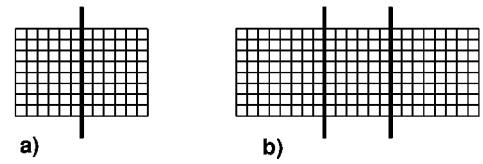

Figure 6. An example of the division in two different cases.

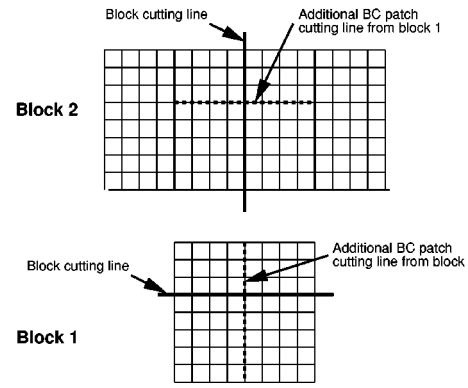

Figure 7. Additional BC patch cuttings caused by the connection between the blocks.

tions, we have $6 \times 6 \times 4$=144 combinations. The right case can be found by computing a magic number

$$MAGIC=IORI+(JFACE-1)*4+(IFACE-1)*24+1 \quad (3)$$

where the face numbers *IFACE* and *JFACE* can have values from one to six and the orientation *IORI* can have values from zero to three.

4.4 Sample case

This example illustrates the division of the boundary patches in complex connections between the blocks. This two-block grid is purely fictive and does not present any reasonable CFD geometry. The original grid and the split grid are shown in Fig. 8.

Note that in order to increase the complexity of the splitting task, the larger block is rotated about the y-axis so that its K-direction coincides with the smaller block's I-direction.

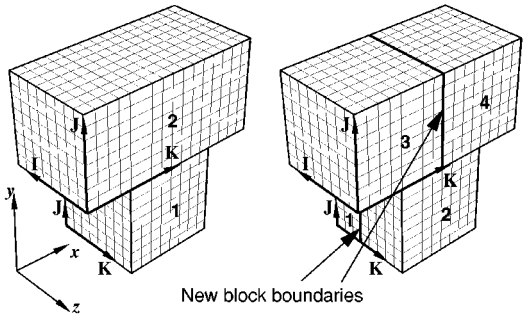

Figure 8. Original two-block grid and the split four-block grid.

Both blocks are divided in the K-direction at one predetermined location. In the larger block this location is the 9th node and in the smaller block the 3rd node. However, due to the rotation of the larger block, these cuttings are not parallel but perpendicularly crossing similar to the case shown in Fig. 7. Further, the 9th node in the larger block's K-direction coincides with the 5th node in the smaller block's I-direction, and the smaller block's 3rd node in the K-direction coincides with the larger block's 5th node in the I-direction. This is why the connective patch on both blocks has to be divided into four pieces.

5 TEST RUNS

In order to test the performance of the MPI-calls, the flow past a cropped delta wing was calculated at Ma_∞=0.85, Re_∞=4.5 $\times$ 10^6 and α=0^o or α=10.76^o. This case has also

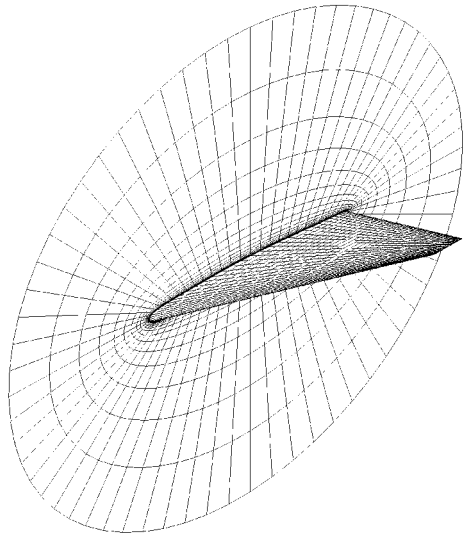

Figure 9. Delta wing. The symmetry plane of the grid is shown only partly.

been calculated in [7]. Grids were generated so that all the blocks had a size of $32 \times 32 \times 32$. The computational domain was split into 1 - 64 different blocks and each block was calculated in a different process. Thus the coarsest grid has 32,768 and the densest grid 2,097,152 grid points. A coarse level of the grid is shown in Fig. 9, and a calculated pressure distribution on a wing surface in Fig. 10.

The program was run on the Cray T3D machine and also on a cluster of SGI Indigo^2 (MIPS R4400SC) workstations. In the latter case the communication between the workstations was made through a standard, low-speed Ethernet. Since it was not possible to obtain the T3D results in an empty machine, the results varied over time. Also the calculations with the workstation cluster were performed when there was other communication in the Ethernet. The computation time was measured over 20 iteration cycles. The time spent for the initialization and termination of the run was not taken into account. For a single processor the initialization takes about 38 seconds and for 64 processors 97 seconds. Each processor counts its own computation time. The minimum of these is shown in Table 2.

The efficiency η is obtained directly from the CPU times. The communication time t_{com} is the total time spent in the subroutine that handles the block connectivity. During the calculations it was found that the collection of global residuals increases the CPU time as the number of processors is increased. Because of this the case with 64 processors was run without this stage. Speed up can be seen in Fig. 11 with the theoretical maximum speed-up together with the results obtained with the SGIs cluster of workstations. Also the esti-

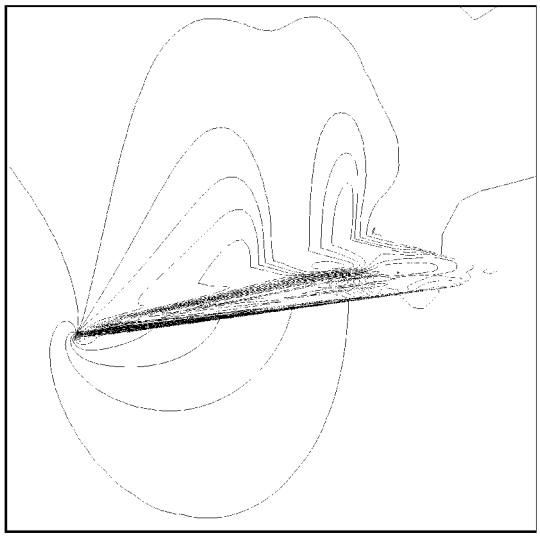

Figure 10. Pressure distribution on the delta wing.

Table 2. Performance of the parallelization.

NPS	CPU	N. of cells	t_{com}	η T3D	η SGI
1	538.02	32 768	0.54	1.000	1.000
2	542.17	65 536	2.17	0.992	0.950
4	549.76	131 072	2.20	0.978	0.850
8	549.96	262 144	8.25	0.978	0.904
16	546.43	524 288	8.80	0.985	0.859
32	558.41	1 048 576	8.38	0.963	N/A
64	589.34	2 097 152	21.17	0.913	N/A

mated speed up for a larger number of processes is estimated from Amdahl's law

$$\text{Speed up}=\frac{N}{1+c'N} \tag{4}$$

where N is number of processes and c' the time spent in communication per time spent in a single processor. As can be seen, the system still performs well with up to 64 processes. With 64 processes, the computational power is 58.43 times higher than in a single process mode.

Another way of testing parallelization is to divide a big problem into small ones. Here the case is the same Delta wing with a grid size of 128 × 64 × 64. The problem size was limited by the memory of the Cray T3D. Due to memory limitation only the cases with 16, 32 and 64 processors could be calculated. The corresponding block sizes were 32,768, 16,384 and 8,192.

As can been seen from Table 3 the scaling is not so good in this case. This is due to the fact that unnecessary ghost cells are calculated in each block. The smaller the block is, the relatively larger is the number of ghost cells. Also the

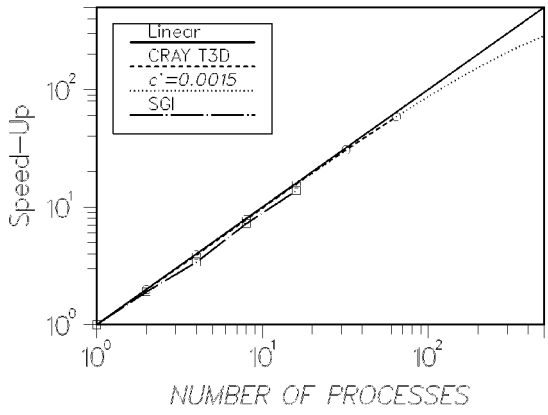

Figure 11. Performance of the parallelization.

Table 3. Performance of the parallelization.

NPS	CPU	N. of cells/block	η T3D
16	547.04	32 768	1.000
32	290.34	16 384	0.942
64	213.49	8 192	0.641

faces of the block are not of equal size. In the figures of Table 3 the collection of global residuals is included, which may significantly affect the efficiency with 64 processors.

Since the boundary conditions are treated explicitly, splitting of the computational domain into smaller parts will also reduce the performance of the implicit stage. This was tested by dividing the original grid sized 128 × 64 × 64 into pieces. Iteration histories of L_2-norm of x-momentum can be seen in Fig. 12 as calculated with different block sizes. It can be seen that the convergence is not much affected by the grid size. However, it should be noted that in this case the smallest block size is still relatively large, 32 × 32 × 16.

6 CONCLUSIONS

The parallelization of a multi-block flow solver has been presented. The parallelization takes place over the blocks and the communication between the blocks utilizes the MPI Standard. The computer code is portable with different types of machines.

The first test runs were made on a cluster of SGI work stations. The performance curve obtained is almost linear up to 16 processes. This is inspite of the fact that the workstations were connected with a standard, low-speed Ethernet.

With the Cray T3D machine, test runs have been made with up to 64 processors. The performance of the code is satisfactory, but not excellent. With a constant block size (32^3) the efficiency with 64 processors is about 91%. It should be noted that the figures obtained so far are approximative,

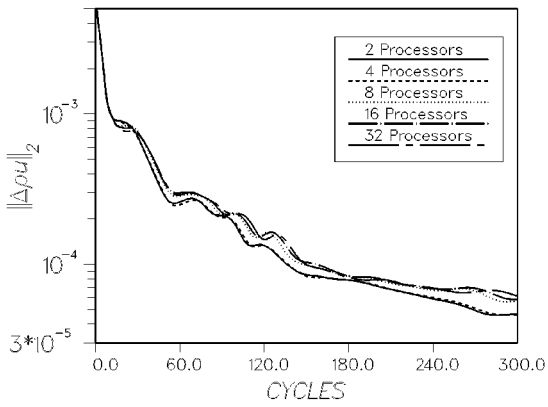

Figure 12. L_2-norm of x-momentum residual.

since the speed of the machine varied during the test runs. However, the obtained speed indicates that the machine can be effectively utilized with 100-200 processors.

If the size of the case is kept as constant and the parallelization is performed by dividing the grid into smaller and smaller blocks, the efficiency of the code decreases rapidly as the number of processors is increased. This is caused by the extra time spent in the calculation of the ghost cells. Because of this property, and the requirement of the multigrid, the block size cannot be smaller than (24^3) or (32^3). This limits the effective use of the parallelization on the T3D to cases where the number of grid points is of the order of one million or more. In practice this is no limitation if one has access to a vector computer with a sufficient memory size. The vector computer will be faster than the T3D or even the T3E in medium-sized jobs with 1-2 million grid points. The benefits of a massively parallel computation are obtained only with massively large cases.

In the future the bottlenecks of the present implementation will be studied carefully. A possible cause for the evident inefficiency with a large number of processors may lie in the communication for each variable separately. Also the use of the synchronous communication mode could improve the efficiency.

REFERENCES

[1] G. Bärwolff, K. Ketelsen, and F. Thiele, 'Parallelization of a finite-volume Navier–Stokes solver on a T3D massively parallel system', in *Sixth International Symposium on Computational Fluid Dynamics*, Lake Tahoe, Nevada, (Sept. 1995).

[2] Message Passing Interface Forum, 'MPI: A message-passing interface standard.', Technical report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville,TN, (1994).

[3] P.P. Rautaheimo and T. Siikonen, 'Implementation of the Reynolds-stress turbulence model', in *Proceedings of the ECCOMAS Congress*, Paris, (Sept. 1996).

[4] P.L. Roe, 'Approximate Riemann solvers, parameter vectors, and difference schemes', *Journal of Computational Physics*, **43**, 357–372, (1981).

[5] M.L. Sawley and J.K. Tegner, 'A comparison of parallel programming models for multiblock flow computations', *Journal of Computational Physics*, **122**, 280–290, (1995).

[6] T. Siikonen, J. Hoffren, and S. Laine, 'A multigrid LU factorization scheme for the thin-layer Navier–Stokes equations', in *Proceedings of the 17th ICAS Congress*, pp. 2023–2034, Stockholm, (Sept. 1990). ICAS Paper 90-6.10.3.

[7] T. Siikonen, P. Kaurinkoski, and S. Laine, 'Transonic flow over a delta wing using a $k-\epsilon$ turbulence model', in *Proceedings of the 19th ICAS Congress*, pp. 700–710, Anaheim, (Sept. 1994). ICAS Paper 94-2.3.2.

[8] T. Siikonen and H. Pan, 'Application of Roe's method for the simulation of viscous flow in turbomachinery', in *Proceedings of the first European Computational Fluid Dynamics Conference*, ed., Ch. Hirsch et al., pp. 635–641, Brussels, Belgium, (Sept. 1992). Elsevier Science Publishers B.V.

[9] B. Van Leer, 'Flux-vector splitting for the Euler equations', in *Proceeding 8th International Conference on Numerical Mehods in Fluid Dynamics*, Aachen, (1992). (also Lecture Notes in Physics, Vol. 170, 1982).