INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 19, 707–721 (1994)

A DATA-PARALLEL APPROACH TO MULTIBLOCK FLOW COMPUTATIONS

M. L. SAWLEY AND J. K. TEGNÉR

Institut de Machines Hydrauliques et de Mécanique des Fluides, Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland

SUMMARY

Multiblock methods are often employed to compute flows in complex geometries. While such methods lend themselves in a natural way to coarse-grain parallel processing by the distribution of different blocks to different processors, in some situations a fine-grain data-parallel implementation may be more appropriate. A study is presented of the resolution of the Euler equations for compressible flow on a block-structured mesh, illustrating the advantages of the data-parallel approach. Particular emphasis is placed on a dynamic block management strategy that allows computations to be undertaken only for blocks where useful work is to be performed. In addition, appropriate choices of initial and boundary conditions that enhance solution convergence are presented. Finally, code portability between five different massively parallel computer systems is examined and an analysis of the performance results obtained on different parallel systems is presented.

KEY WORDS Data-parallel Block-structured mesh Compressible flow Load balancing Boundary conditions Code portability

1. INTRODUCTION

Computational fluid dynamics (CFD) methods are generally based on the resolution of a set of partial differential equations—such as the Euler equations for inviscid flow or the Navier–Stokes equations for viscous flow—that describe the continuum behaviour of the fluid. These differential equations are discretized using e.g. a finite difference, finite element or finite volume method and the resulting set of algebraic equations is solved on a computational mesh that covers the physical flow domain. For the computation of flows in complex geometries the *multiblock method* is widely employed. This method is based on the partitioning of the flow domain into a number of subdomains (blocks). Block-structured meshes are generally unstructured at the block level but retain a structured nature within each block. Multiblock methods provide an alternative to the use of a completely unstructured mesh and, depending on the problem at hand, may offer certain advantages, such as the use of simpler and more efficient algorithms to resolve the required flow equations.

Parallel techniques for the above-mentioned CFD methods are based on the computation of the flow values at different mesh points on different processors. Block-structured meshes exhibit both a coarse-grain parallelism at the block level and a fine-grain parallelism at the mesh point level. For efficient parallel computation the granularity of the problem needs to be matched to that of the parallel computer employed. Coarse-grain parallelism has been exploited in CFD calculations by a number of researchers using either shared-memory multiprocessor computers[1–3] or distributed-memory massively parallel MIMD (multiple-instruction, multiple-data)

CCC 0271–2091/94/200707–15

Received January 1994
Revised May 1994

computers.[1,4–6] These implementations have employed a *control-parallel* programming model whereby different blocks are computed in an independent manner on different processors, with information being transferred between blocks using either the (virtual) shared memory or message passing.

Control-parallel multiblock methods can be efficiently employed if the mesh is comprised of blocks having equal computational work, the computer has processors of equal computational power and the number of blocks is equal to or a multiple of the number of processors. Under these conditions *load balancing* between processors ensures that there is minimal processor idle time. For many flow computations the mesh is specially designed such that these requirements are met. Since generating an appropriate block-structured mesh for a complex geometry may take longer (in 'real' time) than the resolution of the flow equations on the mesh, it is desirable that parallelization constraints are not imposed on the mesh generation procedure in addition to the existing geometrical constraints. Generally, except for very complex 3D geometries, the number of blocks that is consistent with the geometrical constraints is of the order of ten(s). Therefore, without further subdivision, computations can only be performed with such block-structured meshes using computers with a limited number of processors. A further complication for the efficient implementation of control-parallel multiblock methods arises from the fact that the computational work of a block is not necessarily determined solely by the number of mesh points in the block. Depending on the nature of the flow to be computed, some blocks may require a larger number of iterations to obtain the desired level of convergence. If the final converged solution has been obtained in a block, further iteration will not entail useful work even though the floating point operation count will increase. To take account of such behaviour, various dynamic load-balancing algorithms have been proposed, such as the 'demand decomposition' technique described in Reference 5.

The fine-grain parallelism exhibited at the mesh point level of a structured mesh is well suited to the *data-parallel* programming model whereby different processors undertake the necessary computations of different mesh points in a synchronous manner. Data-parallel techniques have been employed by a number of authors in recent years for CFD applications on SIMD (single-instruction, multiple-data) computers, both for single-block-structured meshes[7–9] and unstructured meshes.[10,11] Data-parallel implementations on present-generation parallel computers generally make use of a Fortran-90-based language to render implicit the distribution of work to the different processors. This greatly simplifies programme coding and enhances portability. Such implementations have been found to be very efficient for local approximation techniques such as employed in the above-mentioned CFD methods.

While the application of the data-parallel programming model to date appears to have been limited to computations based on single-block-structured meshes and unstructured meshes, it can also be employed for computations using block-structured meshes. A study has therefore been undertaken to investigate a *serial data-parallel multiblock method* whereby individual blocks are treated in a sequential manner, the solution in each block being computed using a data-parallel approach. The present paper provides an illustration of the advantages that this method affords. In Section 2 the flow equations and the test case considered are presented, while the numerical implementation employed is described in Section 3. Since the different blocks are treated in a sequential manner, a relatively simple dynamic block management procedure has been employed as described in Section 4. The serial data-parallel multiblock method has been implemented on five different massively parallel processing (MPP) systems; a brief overview of these systems is given in Section 5. Code portability between the different systems is discussed in Section 6. Finally, Section 7 presents an analysis of the performance results obtained on different MPP systems.

2. FLOW EQUATIONS AND APPLICATION

The Euler equations governing inviscid compressible flow can be written in a conservative law form in a two-dimensional (x, y) Cartesian co-ordinate system as

$$\frac{\partial}{\partial t}\mathbf{w} + \frac{\partial}{\partial x}\mathbf{F}(\mathbf{w}) + \frac{\partial}{\partial y}\mathbf{G}(\mathbf{w}) = 0,$$

where

$$\mathbf{w} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix}, \qquad \mathbf{F}(\mathbf{w}) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(\rho E + p) \end{pmatrix}, \qquad \mathbf{G}(\mathbf{w}) = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(\rho E + p) \end{pmatrix}.$$

Here ρ is the mass density, p is the pressure, E is the total energy and (u, v) are the (x, y) components of the flow velocity. The above system of equations is closed via the equation of state

$$p = \rho(\gamma - 1)[E - \tfrac{1}{2}(u^2 + v^2)],$$

where γ is the ratio of specific heats ($\gamma = 1{\cdot}4$).

As an application we consider here the flow in a supersonic air intake. A freestream Mach number of 1·865 and a back pressure (static pressure at diffuser exit/freestream total pressure) equal to 0·83 have been imposed. The resulting flow, as shown in Figure 1, is characterized by an oblique shock that impinges on the cowl lip, followed by a series of reflected shocks in the diffuser section. After termination by a normal shock the flow becomes subsonic. A study of this simplified two-dimensional case can yield valuable information on the optimization of air flow at the diffuser exit. This flow case has been studied by the French aircraft engine manufacturer Snecma in the development of a power plant for the replacement of the Concorde supersonic aircraft.[1,2]

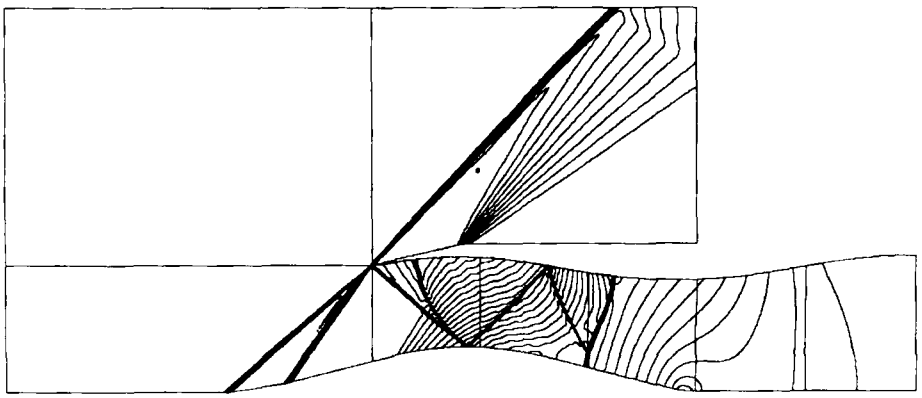

Figure 1. Pressure contours for inviscid flow in a supersonic air intake. The straight lines represent the block boundaries

3. NUMERICAL IMPLEMENTATION

The serial data-parallel multiblock method has been implemented in a relatively simple code[13] that solves the time dependent Euler equations for two-dimensional, inviscid, compressible flow. Results obtained using a data-parallel single-block version of this code have been reported previously.[8]

3.1. Discretization

The Euler equations are discretized in space using a finite volume formulation with a central difference scheme. Numerically induced oscillations near discontinuities (shock waves) are damped by the addition of a second-order artificial dissipation term. A fourth-order dissipation term is included to damp odd/even oscillations permitted by the numerical scheme. The resulting set of discretized equations is integrated in time using an explicit five-stage Runge–Kutta scheme. To enhance convergence to the required steady state, a local time-stepping technique is employed.

3.2. Computational mesh

To compute the flow field in the air intake, a block-structured mesh comprised of eight blocks has been employed. The construction of this mesh has been based on geometric considerations, with smaller blocks allocated to regions of greater interest (see Figure 1). Each block contains the same number of mesh cells; this choice is not related to load-balancing concerns but to the algorithm employed in the code, which is restricted to directly connected blocks. The number of mesh cells in each block (124×124), however, has been chosen to maximize performance on the MPP systems employed (see Section 7). While a control-parallel method could be efficiently used for such a block-structured mesh on a computer system comprised of eight processors of equal performance, additional artificial subdivision of the mesh would be required for larger parallel systems.

3.3. Initial and boundary conditions

An extensive study of the air intake problem has shown that the convergence of the solution procedure is particularly sensitive to the initial conditions and the boundary condition imposed at the diffuser outflow. Therefore, to maximize code performance—measured with respect to the computer time required to obtain the required flow solution—appropriate optimization of the initial and boundary conditions is crucial.

Indeed, by simply imposing freestream conditions throughout the flow domain and the back pressure fixed at the required value of 0·83, it has been observed that the normal shock in the diffuser section is not stabilized and thus convergence is not obtained. This problem can be avoided by setting the initial flow solution to include a normal shock in the diffuser section. For convenience the initial position of the normal shock has been chosen to correspond to a block boundary. Downstream of the normal shock the initial flow is subsonic and determined from the standard normal shock relations. Supersonic flow was initially set in four of the eight blocks and subsonic flow in the four blocks in the downstream section of the diffuser.

This choice of initial condition enables the outflow to be always subsonic. A *non-reflecting outflow boundary condition* for subsonic flow based on the following differential equation[14] has been employed:

$$\frac{\partial p}{\partial t} - \rho c \frac{\partial u}{\partial t} + \alpha(p - p_o) = 0,$$

where $p_o = 0{\cdot}83p_{T\infty}$ is the required outflow pressure and α is a parameter chosen to optimize the convergence rate ($\alpha = 1{\cdot}25$ for the present computations).

The above-described non-reflecting boundary condition has been found to be necessary in the initial stages of the solution procedure. Once the outflow pressure has stabilized close to the required value p_o, this value is directly imposed. In practice the non-reflecting boundary condition is imposed only during the iteration to the first convergence level (see Section 4), allowing a significant saving in computation time.

3.4. Code structure

The code employed in the present study stores in global arrays the values of the flow quantities throughout the flow domain. For the serial data-parallel multiblock method the global values for one block are copied to corresponding local arrays which contain an additional exterior layer of 'ghost cells' to provide data locality and facilitate the application of boundary conditions. The computation is performed using the local arrays in a manner similar to that for a single-block mesh.[8] The updated local array values are then returned to the global arrays. Data are transferred between different blocks (block connectivity) at each time step via the global arrays in an implicit fashion using the globally addressable memory. After the global array elements for one block are updated, the other blocks are treated in a sequential manner.

The code is written entirely in Fortran 90, using array assignments and intrinsic functions. Data distribution directives are used to map the arrays to the processors in such a way as to minimize communication (see Section 7). Extensive use is made of intrinsic functions, particularly the CSHIFT function that causes a circular shift of the array values by one element, either row-wise or column-wise.

4. DYNAMIC BLOCK MANAGEMENT

Since the blocks are treated in a sequential manner, a rather simple *dynamic block management* strategy has been employed. Every 20 iterations the values of the RMS (root mean square) residuals in each block (based on the local density) are calculated. The residual values are used to determine which of the blocks are to be considered for the next 20 iterations. The criterion for a block to be 'on' is that the maximum residual in the block or at the appropriate ghost cells in neighbouring blocks be greater than a predetermined convergence level. In this manner the computation is performed only in blocks where there is useful work to be done. Not only are blocks switched off to avoid unnecessary computations, but they are also switched back on should the development of the solution in a neighbouring block produce an influence. Initially the convergence level is set to $2{\cdot}5 \times 10^{-3}$. When all blocks are converged, this value is decreased by a factor of 10, all blocks are switched back on and further iterations performed. This procedure is continued until the RMS residual over the entire flow domain is less than 2×10^{-7}, corresponding to a total decrease in the residual of five orders of magnitude.

This dynamic block management strategy has been chosen to minimize the number of iterations required to obtain the converged flow solution and therefore to maximize the useful work performed. Such a strategy maintains the level of convergence in all blocks within certain bounds and thus avoids accumulation effects associated with large differences in the treatment of neighbouring blocks.

Figure 2(a) shows the residual of each of the eight blocks employed as a function of the number of iterations. Figure 2(b) presents the residual over the entire flow domain as a function of work units (one iteration of one block being defined as one-eighth of a work unit). The two

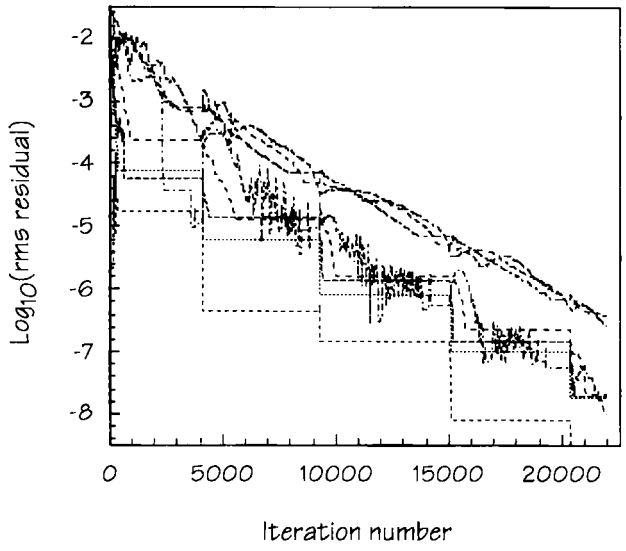

Figure 2(a). Convergence history for each of the eight blocks

Figure 2(b). Maximum residual in the flow domain as a function of work units, both with (——) and without (- - - -) dynamic block management

curves of Figure 2(b) correspond to cases with and without the application of dynamic block management.

From Figure 2(a) it can be observed that after a relatively small number of iterations (approximately 400) two blocks are switched off; a third block is switched off after 600 iterations and a fourth after 1000 iterations. The first two blocks correspond to the upstream section of the flow domain, the third to the region above the cowl lip of the air intake and the fourth to

the first block in the diffuser section. It should be noted that the flow in each of these regions is supersonic. Since the Euler equations are hyperbolic for supersonic flow, there is no upstream influence. (The same is valid, to a good approximation, for the discretized equations if the added artificial dissipation is sufficiently small.) Therefore, unless a subsonic region of the transient solution enters upstream, once the predefined convergence level is attained, no more useful work can be performed. Approximately 4000 iterations are required before the first convergence level is attained, with an additional 5000 iterations required to reach each of the subsequent convergence levels.

For the present problem Figure 2(b) demonstrates that employing the above-described dynamic block management leads to a reduction by a factor greater than two in the amount of work required to obtain the converged solution in the entire flow domain. In fact, for the majority of iterations it has been necessary to compute at most four of the eight blocks. The overhead required for the dynamic block management, both in coding effort and in computation time, is minimal.

5. PARALLEL COMPUTERS EMPLOYED

Computation of the air intake flow problem has been undertaken using the following MPP systems.

MP-1 and MP-2—MasPar Computer Corp.

The MP-1 and MP-2 are SIMD computers that employ the same architecture based on a two-dimensional torus grid.[15] Each processing element (PE) consists of a simplified RISC processor with 64 kbyte of local memory. The X-Net communication network has a bandwidth of 1·25 Mbyte s^{-1} per PE between neighbouring PEs, while a global router provides communication between arbitrary PEs with a bandwidth of 80 kbyte s^{-1} per PE. A fully configured system has 16k PEs, giving a total of 1 Gbyte of distributed memory and a peak 64-bit floating point performance of 0·55 and 2·4 Gflop s^{-1} for the MP-1 and MP-2 systems respectively.

CM-200—Thinking Machines Corp.

The CM-200 is an SIMD computer that uses a hypercube architecture.[16] Each node consists of 32 single-bit PEs, 4 Mbyte of local memory and a floating point accelerator. Communication between nearest neighbours on a Cartesian mesh uses the NEWS network with a bandwidth of 40 Mbyte s^{-1} per node; the global router provides more general communication. The largest CM-200 employed in the present study has 1k nodes with 4 Gbyte of distributed memory and a peak 64-bit floating point performance of 10 Gflop s^{-1}.

CM-5—Thinking Machines Corp.

The CM-5 is an MIMD system with each user partition consisting of both a control node and a number of processing nodes.[17] Each processing node has a standard RISC Sparc microprocessor with four vector pipes (each having eight 64-bit registers or 16 32-bit registers using a vector length of 8) to a total of 32 Mbyte of local memory, providing a peak 64-bit floating point performance of 128 Mflop s^{-1}. Data are transferred between nodes via a network that uses a 4-ary fat-tree with a maximum bandwidth of 20 Mbyte s^{-1} per node between neighbouring nodes.

T3D emulator—Cray Research Inc.

The Cray T3D is an MIMD system with each PE containing a DEC Alpha RISC microprocessor having a peak 64-bit floating point performance of 150 Mflop s^{-1}.[18] The computational nodes, each containing two PEs, are connected via a 3D torus network (300 Mbyte s^{-1} per node between neighbouring nodes). The system contains either 16 or 64 Mbyte of memory per PE. Although a Cray T3D system has not been available for the present work, an emulator that runs on a Cray Y-MP has been employed to study code implementation.

Each of the above MPP systems has a non-uniform memory access (NUMA) distributed memory whereby a PE (or node) can access its local memory faster than it can access the memory of other PEs. Such an architectural consideration has an important impact on the programming techniques required to achieve acceptable performance (see Section 7).

All the above systems offer Fortran compilers that can make use of the data-parallel programming model. The syntax employed in these compilers is based on the Fortran 90 standard, with a number of additions/modifications necessary for parallelization. In addition, the CM-5 and T3D systems provide other programming models (message passing, work sharing), although these have not been employed in the present study.

6. CODE PORTABILITY

The subject of code portability is of major importance, particulary for application software developed for parallel computer systems. Code portability is concerned with two principal aspects: compilation on different computer platforms and the ability to achieve an acceptable performance level.

6.1. Compilation

The use of standard Fortran 90 style has made relatively simple the task of compiling the code on each of the five above-mentioned MPP systems. In addition, the code has been compiled and run on both serial and vector/parallel computer systems. Particular deficiencies of the MPP systems that were found to require code modification to enable compilation include

(1) the lack of standard data distribution directives (the systems employed in the present study used the CMPF MAP, CMF$ LAYOUT, and CDIR$ SHARED directives to achieve suitable array mappings)
(2) the need for internal compiler directives on the Cray T3D (e.g. the CDIR$ MASTER and CDIR$ ENDMASTER directives to specify serial and parallel execution regions)
(3) the lack of standard timing routines
(4) features currently unavailable on the Cray T3D emulator (e.g. certain restrictions for do-loops and intrinsic function calls).

6.2. Performance

While adhering to the Fortran 90 standard has enabled the code to be run on serial, vector/parallel and MPP systems, it does not guarantee that the performance obtained on each of these platforms will be acceptable. Optimal code performance has been found to be inhibited by the following factors.

(1) Different MPP systems have different 'canonical' data distributions; non-canonical distributions may cause performance degradation (e.g. the serial axes of the global arrays are placed first on the CM-200 and CM-5 but last on the MP-1, MP-2 and T3D).
(2) While different means may be available in Fortran 90 for expressing the same operation, these are not always interpreted by the compiler in the same manner (e.g. on the CM-200 direct array assignment may use the global router, whereas the equivalent CSHIFT function call uses the NEWS network).
(3) The use of software library routines may greatly improve performance at the expense of portability (such routines, however, have not been employed to obtain the results presented in Section 7).
(4) Optimized data-parallel code for MPP systems may have many redundant operations (e.g. for the treatment of ghost cells to avoid the need for masking and hence decrease overheads), resulting in poor performance on serial and vector computers.
(5) The use of array syntax and calls to intrinsic functions can limit the length of vector loops to single programme statements, causing serious performance degradation on vector computers.

Some of the above portability restrictions are due to the present immaturity of compilers; it is thus to be expected that portability problems may become less important in the future. However, the performance of the serial data-parallel multiblock method on MPP systems is strongly dependent on the ability of the compiler to interpret code in an efficient manner for data-parallel computations. This will in turn be dependent on both hardware and software considerations.

7. CODE PERFORMANCE

The study and comparison of code performance on different computer systems—and the interpretation of such comparisons—are fraught with danger. [19,20] The goal of the present paper is not to present a benchmarking study for the present code, but rather to provide an indication of the programming effort required to obtain acceptable performance on the MPP systems considered.

While five different MPP systems have been considered in the present study, performance figures are presented only for the MP-1, MP-2, CM-200 and CM-5 systems, since the data-parallel Fortran compiler for the T3D hardware has not yet been released. The results obtained on the CM-5 are based on a beta version of the system software and consequently are not necessarily representative of the performance of the full version of this software.

The performance of the multiblock flow solver is determined by the time required for the following tasks:

(1) copying flow values between the global and local arrays (before and after the computation of each individual block)
(2) performing the computational kernel (computing the convective and numerical fluxes, time stepping)
(3) implementing the boundary conditions
(4) undertaking the block connectivity
(5) input/output (e.g. reading mesh and initial flow values, writing solution).

Each of the above tasks involves the transfer of data, which for the NUMA distributed-memory MPP systems used in the present study is a major concern. In order to maximize performance, the most rapid means of data transfer must therefore be employed (see Figure 3). (For serial

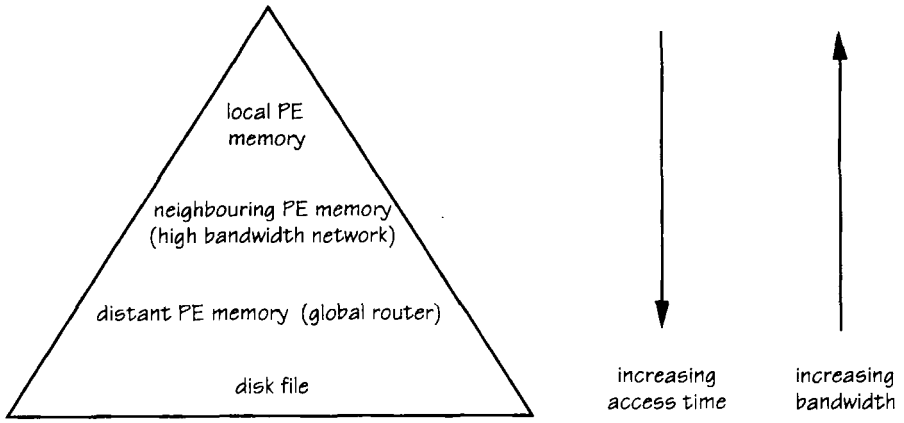

Figure 3. Communication hierarchy for an NUMA distributed memory parallel computer

and vector computers data transfer between arrays stored in memory is not a consideration; thus performance is generally determined by task (2).)

For optimal performance of a control-parallel method the elements of a local array (corresponding to one block) are stored in the same local PE memory. This ensures that the computational kernel for each block can be performed by the same PE without communication with other PEs. In contrast, in the present data-parallel method the elements of both global and local arrays are distributed across different PEs. However, the distribution is chosen in such a way that all flow values for one mesh cell are stored in the same PE memory; copying flow values between the global and local arrays thus involves only local memory references. Task (1) can therefore be performed very efficiently.

For a local approximation method, such as the finite volume method used in the present study, the computational kernel involves the addition and subtraction of neighbouring elements according to a stencil determined by the spatial discretization employed. The associated data transfer can be performed using the CSHIFT intrinsic function. If the number of array elements is less than or equal to the number of PEs, the CSHIFT operation will direct all PEs to obtain an array value from a different PE; the CSHIFT operation will therefore involve all remote memory references. For much larger arrays, however, the majority of the CSHIFT operations will involve local memory references. For NUMA distributed-memory MPP systems the efficiency of such an operation will therefore increase with the problem size.[8] This indicates that for optimal performance on a given MPP system the number of mesh cells in each block needs to be as large as possible; thus for a given size computational mesh the number of blocks should be minimized. This requirement will generally not be in conflict with the need to simplify the mesh generation procedure. The efficiency with which task (2) can be performed is therefore dependent on the efficiency of the CSHIFT operation, the number of PEs employed and the problem size (number of blocks and mesh cells). High-bandwidth networks are provided on the MPP systems to perform communication between neighbouring PEs.

The implementation of the boundary conditions (task (3)) and block connectivity (task (4)) generally requires more complex patterns of data transfer. It is important that, if possible, the above-mentioned high-bandwidth networks are used rather than the slower global communication network. For the present code implementation this consideration has necessitated the

replacement of array section assignment statements with the equivalent CSHIFT intrinsic functions to enforce the use of the fast network on the CM-200 and CM-5.

The relative importance of input/output (task (5)) depends on the initialization procedure, the amount of data being read and written and where these data are stored (e.g. front-end disk, parallel disk array). For the present computation of the converged flow solution this contribution was of secondary importance and therefore not considered in the following performance figures.

In Figure 4 is presented, for MP-1 and MP-2 systems with 16k PEs and CM-200 and CM-5 systems with 512 nodes, the time required to perform the block connectivity (task (4)), the boundary conditions (task (3)) and the computational kernel (task (2)). The performance obtained using MPP systems of different sizes is shown in Figure 5. Since the same problem has been resolved for each system size, the performance is inversely proportional to the time required to obtain the converged flow solution.

Figure 4 indicates that for the computation of the flow in the air intake the majority of time is spent performing the computational kernel. Nevertheless a significant fraction is also required for the implementation of the boundary conditions. The overhead required for block connectivity is acceptably low (about 7% for the MP-1, MP-2 and CM-5 systems, though substantially higher for the CM-200). These fractions could be reduced by the simultaneous application of boundary conditions and block connectivity on all four sides of each block, at the cost of increased programming complexity and reduced generality. Measurements made for different size MPP sytems show that the block connectivity overhead remains approximately constant for the MP-1 and MP-2 systems, while the percentage of the total elapsed time required for the

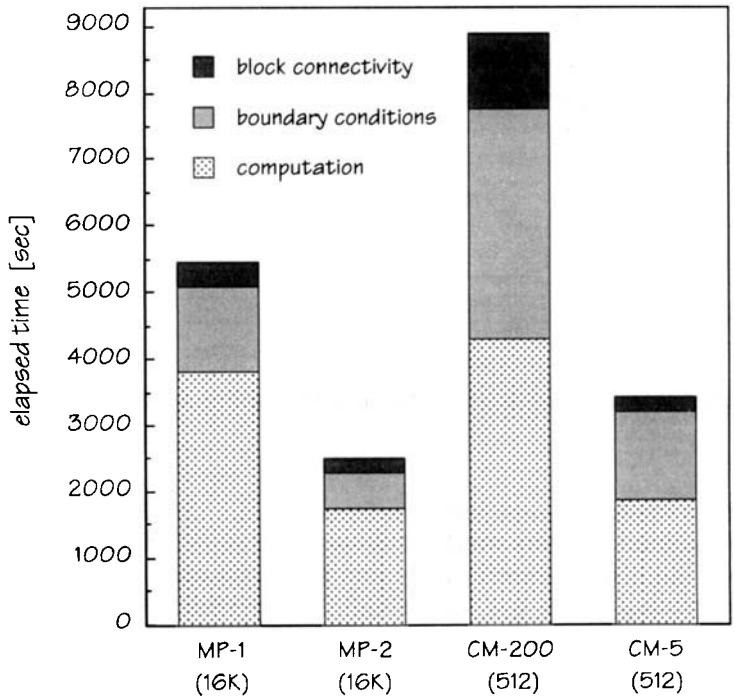

Figure 4. Elapsed time required for different tasks for four different MPP systems

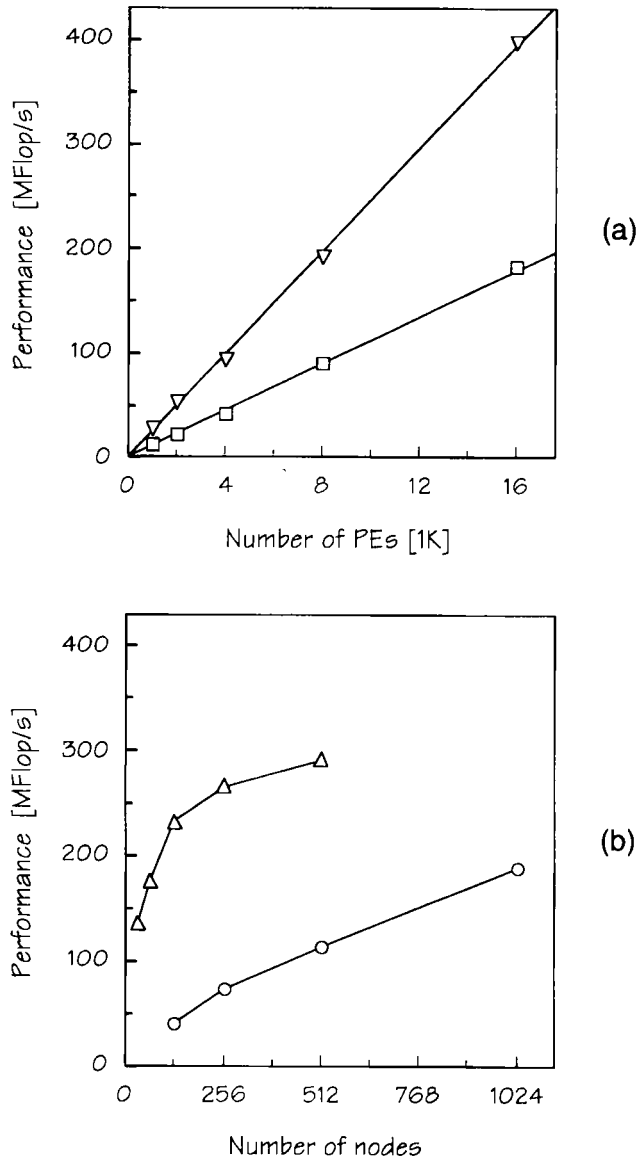

Figure 5. Performance (64-bit) as a function of the size of the MPP system: □, MP-1; ▽, MP-2; ○, CM-200; △, CM-5

boundary condition implementation increases from 10% for 1k PEs to 23% for 16k PEs. For the CM-200 the block connectivity overhead decreases from 18% for 128 nodes to 7·5% for 1k nodes, while both the computational kernel and the application of the boundary conditions increases with increasing system size. In contrast, for the CM-5 the relative time required for the different tasks has been found to be independent of the system size in the range from 32 to 512 nodes.

Figure 5(a) shows that on both the MP-1 and MP-2 the performance scales linearly with the number of PEs employed. Such a scaling is to be expected, since the number of mesh cells in

each block is always greater than or equal to the number of PEs. Since each local array has a total of 128 × 128 elements, for a system with 16k PEs each PE performs the work for a single mesh cell. For systems having a smaller number of PEs, block (hierarchical) virtualization has been employed.[8] On the other hand, if a block-structured mesh with fewer cells per block was used, the performance on a system with 16k PEs would be degraded, since not all PEs would be active. The performance on the CM-200 and CM-5, presented in Figure 5(b), does not exhibit the same linear scaling with system size. Indeed, for the CM-5 the performance per node decreases significantly with increasing system size. Profiling information has shown that as the size of the CM-5 system increases, the ratio of time spent on communicating between nodes to that needed for floating point operations increases significantly.

It is interesting to note that while the peak floating performance per PE of the MP-2 is over four times that of the MP-1, the performance obtained using the CFD code employed in the present study is only a factor of two larger. Thus, while a performance of approximately 33% of the peak value has been measured for the MP-1 system, only 17% of the peak value for the MP-2 is obtained. For both the CM-200 and CM-5 systems the measured performance for the present CFD code is only a small percentage of the peak performance of these systems. These observations reinforce the fact that the peak performance of an MPP system is not the sole relevant measure of its performance ability. The numerical method employed for the present flow calculations requires a high level of communication between PEs (or nodes); an efficient computation therefore necessitates a balanced MPP system with a sufficiently high communication bandwidth.

Finally, it should be remarked that the present study is concerned with the computation of a two-dimensional flow problem having a fixed number of mesh cells per block. A previous study[8] has shown that the performance obtained using a data-parallel approach depends strongly on the size of the computational problem. Higher performance levels are to be expected for high-resolution three-dimensional flows (e.g. for the direct numerical simulation of turbulent flows) and for more complex numerical schemes involving a greater number of floating point operations per iteration.

8. CONCLUSIONS

The present study has shown that the serial data-parallel multiblock method provides the following advantages.

(1) It retains the simplicity of the data-parallel approach, each block being treated individually in the same manner as for a single-block computation.
(2) It does not impose any parallelization constraints on the mesh generation procedure; in principle, any number of blocks of unequal size can be employed (although power-of-two size arrays are preferable for performance reasons).
(3) The transfer of data between blocks (block connectivity) is performed in a transparent manner via globally addressable memory; this contrasts with the explicit data transfer required by message-passing implementations.
(4) Since individual blocks are treated sequentially, a simple dynamic block management algorithm can be applied to avoid performing unnecessary operations.
(5) The use of standard Fortran 90 facilitates code portability between different platforms.
(6) Acceptable performance can be obtained on current MPP systems supporting the data-parallel programming method.

The present study has shown that appropriate choices of initial and boundary conditions and solution procedure should be employed together with the use of parallel computation methods to minimize the time required to obtain the flow solution.

Finally, the serial data-parallel multiblock method has been applied in the present study to the computation of a CFD problem using a block-structured mesh. However, such a method could also be applied to computationally intensive problems in other application areas for which the data-parallel programming model has been shown to be well suited.

ACKNOWLEDGEMENTS

The authors wish to thank Magnus Bergman (KTH, Stockholm), Tom MacDonald (Cray Research), Roch Bourbonnais (Thinking Machines) and Björn Malmberg (DEC) for their assistance in the present study. Acknowledgement is gratefully made for access to the MP-1 (KTH; MasPar Computer Corp., Sunnyvale), MP-2 (University of Bergen), CM-200 (KTH; AHPCRC, University of Minnesota) and CM-5 (IPG, Paris; AHPCRC) systems. The T3D emulator installed on the Cray Y-MP 4E of the EPFL was also employed. The study was supported by the Fonds National Suisse and by a contract between the U.S. Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center.

REFERENCES

1. C. Mensink and H. Deconinck, 'A 2D parallel multiblock Navier–Stokes solver with applications on shared- and distributed-memory machines', in Ch. Hirsch, J. Périaux and W. Kordulla (eds), *Computational Fluid Dynamics '92*, Elsevier, Amsterdam, 1992, pp. 913–920.
2. Y. Yadlin and D. A. Caughey, 'Block implicit multigrid solution of the Euler equations', in H. D. Simon (ed.), *Parallel Computational Fluid Dynamics: Implementations and Results*, MIT Press, Cambridge, MA, 1992, pp. 127–145.
3. M. L. Sawley, 'Control- and data-parallel methodologies for flow calculations', *Proc. Supercomputing Europe '93*, Utrecht, February 1993, pp. 169–187.
4. J. Häuser and R. Williams, 'Strategies for parallelizing a Navier–Stokes code on the Intel Touchstone machines', *Int. j. numer. methods fluids*, **14**, 51–58 (1992).
5. D. M. Smith and S. P. Fiddes, 'Efficient parallelisation of implicit and explicit solvers on a MIMD computer', in R. B. Pelz, A. Ecer and J. Häuser (eds), *Parallel Computational Fluid Dynamics '92*, North-Holland, Amsterdam, 1993, pp. 383–394.
6. E. Schreck and M. Peric, 'Computation of fluid flow with a parallel multigrid solver', *Int. j. numer. methods fluids*, **16**, 303–327 (1993).
7. P. Olsson and S. L. Johnsson, 'A dataparallel implementation of an explicit method for the three-dimensional compressible Navier–Stokes equations, *Parallel Comp.*, **14**, 1–30 (1990).
8. M. L. Sawley and C. M. Bergman, 'A comparative study of the data-parallel approach for compressible flow calculations', *Parallel Comput.*, **20**, 363–373 (1994).
9. F. P. Brueckner, D. W. Pepper, T. H. Sobota and R. H. Chu, 'The calculation of three-dimensional compressible flow through a rectangular nozzle using a data parallel finite element model', in R. B. Pelz, A. Ecer and J. Häuser (eds), *Parallel Computational Fluid Dynamics '92*, North-Holland, Amsterdam, 1993, pp. 51–62.
10. S. W. Hammond and T. J. Barth, 'Efficient massively parallel Euler solver for two-dimensional unstructured grids', *AIAA J.*, **30**, 947–952 (1992).
11. Z. Johan, T. J. R. Hughes, K. K. Mathur and S. L. Johnsson, 'Data parallel finite element techniques for computational fluid dynamics on the Connection Machine systems', in R. B. Pelz, A. Ecer and J. Häuser (eds), *Parallel Computational Fluid Dynamics '92*, North-Holland, Amsterdam, 1993, pp. 215–229.
12. G. Freskos and O. Penanhoat, 'Numerical simulation of the flow field around supersonic air-intakes', *ASME Paper 92-GT-206*, 1992.
13. C. M. Bergman, 'Development of numerical techniques for inviscid hypersonic flows around re-entry vehicles,' *Ph.D. Thesis 341*, Institut National Polytechnique de Toulouse, 1990.
14. D. H. Rudy and J. C. Strikwerda, 'A nonreflecting outflow boundary condition for subsonic Navier–Stokes calculations', *J. Comput. Phys.*, **36**, 55–70 (1980).
15. J. R. Nickolls, 'The design of the MasPar MP-1: a cost effective massively parallel computer', *Proc. IEEE Compcon*, Spring 1990, IEEE, New York, 1990, pp. 25–28.

16. *The Connection Machine CM-200 Series Technical Summary*, Thinking Machines Corp., Cambridge, MA, 1991.
17. *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corp., Cambridge, MA, 1991.
18. W. Oed, *The Cray Research Massively Parallel Processor System CRAY T3D*, Cray Research GmbH, Munich, 1993.
19. D. H. Bailey, 'Twelve ways to fool the masses when giving performance results on parallel computers', *Supercomputer*, **45**, 4–7 (1991).
20. R. Hockney, 'A framework for benchmark performance analysis,' *Supercomputer*, **48**, 9–22 (1992).