JOURNAL OF COMPUTATIONAL PHYSICS 122, 280–290 (1995)

A Comparison of Parallel Programming Models for 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

Received November 22, 1994; revised April 18, 1995

A study is presented of the implementation of four different parallel programming models in a code that solves the fluid flow equations on block structured meshes. Performance results obtained on a number of distributed-memory parallel computer systems are given, in particular, for a 1024 processor Cray T3D system. Using the appropriate programming model, it is shown that excellent performance scaling can be obtained even for small problem sizes. The relative merits of each programming model in terms of ease of use, functionality, and performance are assessed.

1. INTRODUCTION

Computational fluid dynamics (CFD) is becoming increasingly employed for the numerical simulation of a wide range of flow problems of both academic and industrial interest. In combination with analytical methods and wind tunnel experiments, CFD is potentially an extremely powerful tool for flow applications in, for example, the aeronautical, aerospace, automotive, and chemical industries. The numerical simulation of complex three-dimensional flows requires, however, efficient numerical methods and large computational resources (both in processing speed and memory). Since the solution of many complex flows is presently not accessible via traditional vector supercomputers, there has recently been a strong interest within the CFD community in massively parallel processing (MPP).

The majority of CFD methods are 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. A large number of different studies have investigated the parallel computation of these flow equations (see, e.g., the studies described in [1, 2]). These studies have not only employed a wide range of different parallel computer systems, but also a variety of different parallel programming models.

Generally, the choice of programming model has been governed by the computer system used, for example, the data parallel model on SIMD (single instruction, multiple data) computers, or message passing on distributed-memory MIMD (multiple instruction, multiple data) systems. Thus previous authors have almost exclusively presented results obtained using one parallel programming model. However, faced with the task of developing a large-scale scientific application—such as a complex 3D flow simulation—flexibility in the choice of the parallel programming model is essential [3]. It is therefore of interest to investigate different programming models presently available in order to assess and compare their current capabilities.

The goal of the present study is to examine the use of four different parallel programming models for flow simulations based on a block-structured mesh. Particular reference is made to the Cray T3D system, which provides each of these programming models. Four different code implementations have been considered, with performance results obtained on the Cray T3D and other distributed-memory MIMD and SIMD computer systems. This has enabled an assessment of the relative merits of the different models for numerical flow simulations.

2. PARALLEL PROGRAMMING CONCEPTS

Distributed-memory systems, such as those considered in the present study, are characterized by non-uniform memory access (NUMA). To obtain high performance on such systems using a SPMD (single program, multiple data) programming style, the following three considerations are critical.

Data distribution. It is necessary on distributed-memory systems to distinguish between data objects that are shared among all PEs (processing elements) and those that are private to a PE. Only one copy of *shared data* objects exists that is accessible to all PEs and, if the object is an array, may be distributed across multiple PEs. In contrast, *private data* objects are not distributed across processors, but instead each PE has a personal copy of a private object. For some parallel systems all data objects are implicitly shared (e.g., SIMD computers, such as the Thinking Machines CM-200 and MasPar MP-1/2) or implicitly private (e.g., many MIMD computers, such as the Intel Paragon). For the Cray T3D, compiler directives are used to distinguish shared and private data objects. The distribution of shared data across PEs is generally determined by compiler

0021-9991/95 $12.00

directives (e.g., SHARED for Cray, LAYOUT for Thinking Machines, MAP for MasPar systems).

Work distribution. In addition to distributing the data amongst PEs, it is also necessary to specify which of the PEs will perform the different computational work. For performance reasons, it is desirable to employ an "owner computes" execution, with operations performed by the PE whose local memory contains the data required by the operation. Work distribution for shared arrays may either be implicitly imposed by the use of Fortran 90 style array syntax or, for the Cray T3D, enforced explicitly using the DOSHARED compiler directive.

Data exchange. In all except "embarrassingly parallel" applications, it is necessary to exchange data between PEs at various times during the computation. For the programming models considered in the present study, this is undertaken in one of the following three ways:

- global addressing, involving implicit communication of shared data by simply addressing the required data,
- message passing, for the explicit communication of private data using the PVM (parallel virtual machine) interface,
- explicit shared memory, involving low-level explicit communication based on the system's shared memory (available on the Cray T3D, using the SHMEM library).

In order to perform parallel computations, a choice must be made for each of the above three considerations. An appropriate designation of data objects as private or shared, and a suitable distribution across PEs, together with a corresponding work distribution, is essential to maximize the number of operational PEs and to minimize the transfer of data between PEs. In addition, for communication-dominated applications an efficient means for data exchange is essential.

The choice of the parallel programming model is, however, often governed not solely by performance considerations, but also by functionality and ease of implementation. Often the choice involves a trade-off between the programming convenience of implicit styles and the performance advantage of explicit styles. Figure 1 presents an overview of different programming models considered in the present study, namely:

- *data parallel* (using array syntax)
- *message passing* (using the PVM library)
- *work sharing* (using the DOSHARED compiler directive)
- *explicit shared memory* (using the SHMEM library).

Code portability can also be an important factor in the choice of parallel programming model. The data parallel model has been traditionally employed on SIMD computers, such as the Thinking Machines CM-200 and MasPar MP-1/2 systems. Message passing is employed on a wide range of distributed-memory MIMD systems, as well as on networked parallel systems such as workstation clusters. Work sharing is compatible with certain restrictions with shared-memory multiprocessors such as the Cray Y-MP and C90 systems, but also with a limited number of MPP systems. The explicit shared memory model is specific to the Cray T3D and is not yet available on other parallel computer systems.

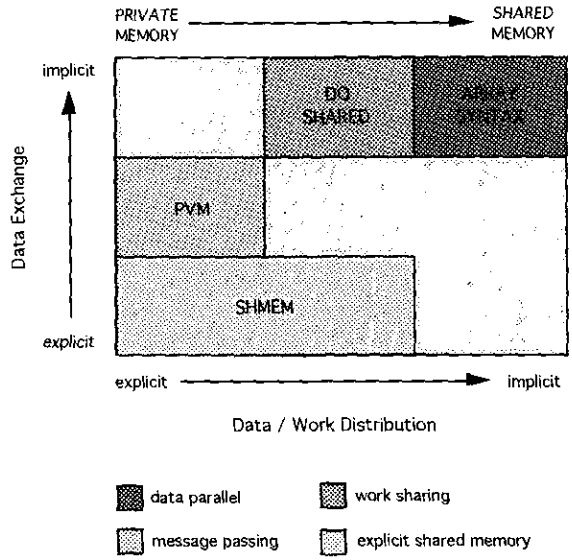

FIG. 1. *Overview of the programming models employed in the present study.*

3. PARALLEL MULTIBLOCK METHODS

Multiblock methods are based on a subdivision of the flow domain into a number of subdomains, with each subdomain being covered by a block consisting of a structured submesh. Subdomains can be interconnected in an unstructured manner, thus *block-structured* meshes provide a measure of flexibility for complex flow geometries. However, since each submesh has a structured nature, multiblock methods retain many of the advantages of methods based on structured meshes. Multiblock methods can be viewed as intermediate between methods based on fully structured or fully unstructured meshes.

Parallel techniques for the CFD methods considered in the present study are based on the computation of the flow values at different mesh points on different processors. Two levels of parallelism can be naturally exploited for block structured meshes:

- *mesh point level,* with the computations for each mesh point (or cell) being performed by a different (virtual) processor. Such a *fine-grain parallelism* is well suited to the data parallel programming model, for which different processors undertake the necessary computations of different mesh points in a synchronous manner. Data parallel techniques have been employed in recent years for CFD applications by a number of authors on SIMD computers (see, e.g., [4–6]).
- *subdomain level,* with each subdomain being assigned to

a different processor. This *coarse-grain parallelism* has been exploited in CFD calculations using either shared-memory multiprocessors [5, 7] or distributed-memory MIMD computers [7, 8].

The implementation of the four different programming models has been studied using different versions of a CFD code that employs block structured meshes. This code solves the time-dependent Euler equations for inviscid, compressible flow in two-dimensional geometries. The Euler equations can be written in a conservative law form in an (x, y) cartesian coordinate 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

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

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 ($\gamma = 1.4$).

The Euler equations are discretized in space using a cell-centered, finite volume formulation with central differences. For the flow problems considered in the present study, the required steady state is obtained as a converged solution of the time-dependent equations. An explicit five-stage Runge–Kutta scheme is employed for the time integration, using local (spatially dependent) time stepping to accelerate convergence.

Different code versions have been developed that employ the four different parallel programming models (Fig. 2). All code versions have the same basic subroutine structure (Fig. 3). Each time iteration of the discretized flow equations consists of:

- *communication* phase, in which the flow values at the subdomain interfaces are exchanged (*block connectivity*),
- *boundary condition* phase, during which the flow values at the edges of the computational domain are imposed,
- *computation* phase, during which the flow solution in each subdomain is updated.

For the present cell-centered numerical method, it has been found adequate to perform the block connectivity only once each time iteration, while the boundary conditions are imposed during the first two stages of the Runge–Kutta scheme.

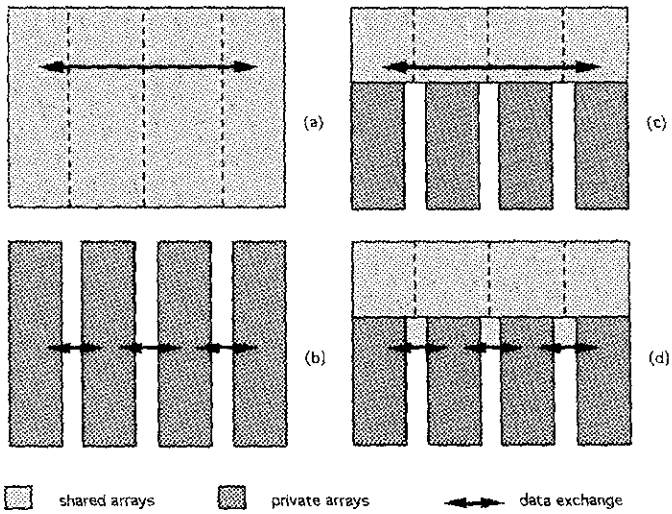

FIG. 2. Schematic diagrams illustrating for four processors the different parallel multiblock implementations considered: (a) data parallel; (b) message passing; (c) work sharing; and (b), (d) explicit shared memory. The horizontal direction indicates the different processors, while the vertical direction represents subroutine layers.

Each 50 iterations, the following global quantities are computed:

- *global time step*, equal to the minimum of the local time steps in each subdomain (this value is actually not required if local time stepping is employed),
- *maximum and rms residuals*, to determine if the solution has converged to the required accuracy.

4. IMPLEMENTATIONS AND PERFORMANCE RESULTS

The code versions described in this paper have been developed on different parallel computer systems. Unless otherwise stated, minor specific modifications have been made in order to obtain equivalent levels of optimization on the different systems. All code versions are written entirely in Fortran without the use of low-level languages (except for interprocessor communication). The computations on all computer systems presented here were performed using 64-bit arithmetic, with performance results given for the entire code, excluding input/output and initialization.

4.1. Data Parallel

4.1.1. *Implementation*

For this implementation, a serial data-parallel multiblock method [9] is employed; each of the blocks are treated in a sequential manner, the solution in each block being computed using fine-grain mesh-point level parallelism.

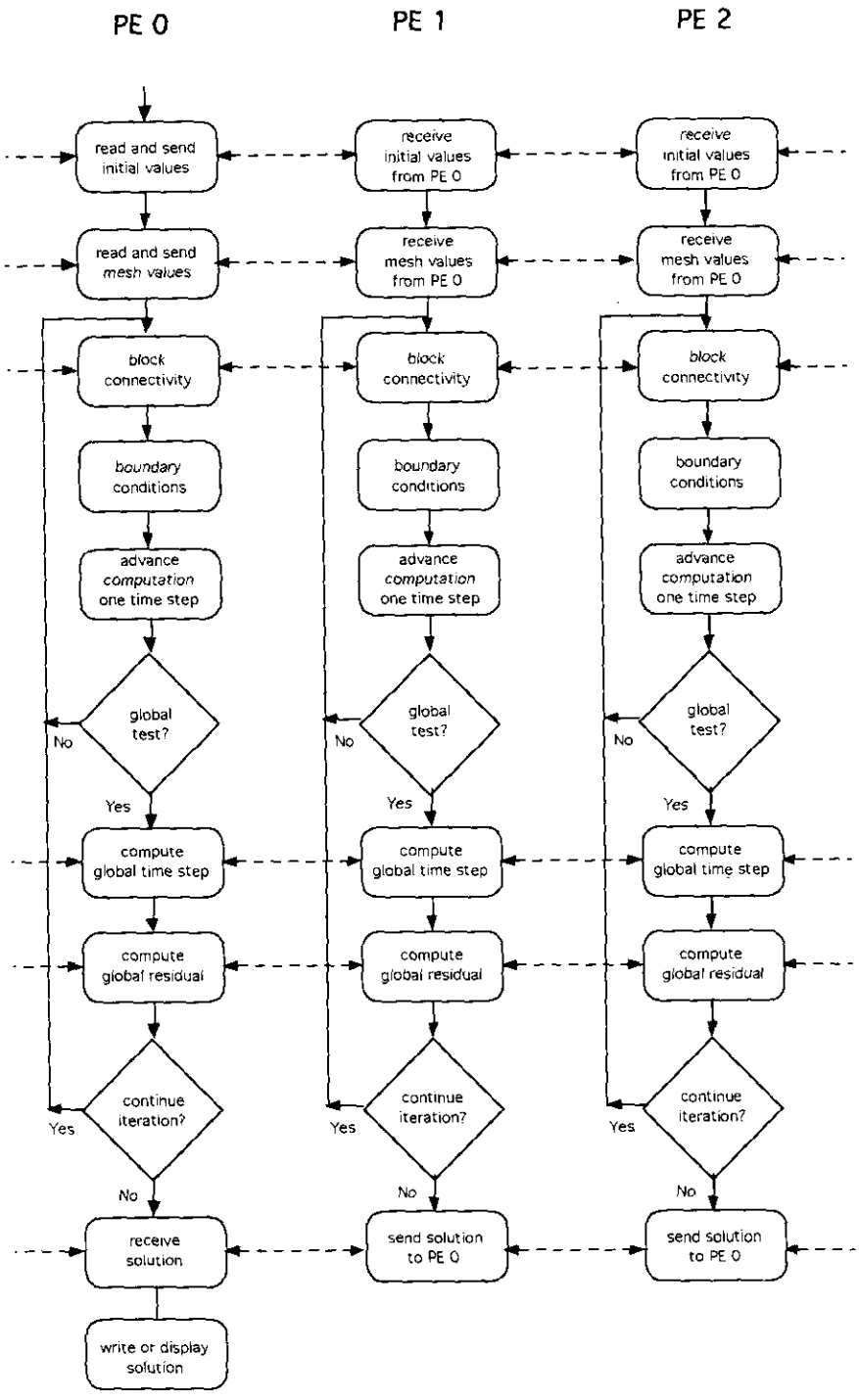

FIG. 3. Flow diagram for the parallel multiblock code. Interprocessor communication, denoted by dashed lines, is undertaken either explicitly or implicitly depending on the programming model considered.

This code version contains three types of data objects:

- *global arrays* containing the mesh coordinates and flow values for the entire flow region. These arrays are four-dimensional: block number, coordinate or value number, and i and j values of the mesh.
- *local arrays* containing the mesh coordinates, flow values, and various work quantities. These arrays are two-dimensional: the i and j values of the mesh. The local arrays contain an additional exterior layer of "ghost cells" to provide data locality and to facilitate the application of boundary conditions.
- *scalar* quantities, such as the freestream pressure.

For the serial data-parallel multiblock method, the variable values for one block are copied from the global arrays to the corresponding local arrays, and the computation is performed using the local arrays in a manner similar to that for a single block mesh [5, 6]. The updated local array values are then returned to the global arrays. Block connectivity is achieved using the global arrays. After the global array elements for one block are updated, the other blocks are treated in a sequential manner.

All scalar quantities are treated as private variables. Both global and local arrays are treated as shared and are distributed so that the values of all quantities at the same mesh cell are stored in the same PE memory (that is, BLOCK distribution for Cray, NEWS for Thinking Machines, XBITS/YBITS for MasPar systems for the i and j values of the mesh or flow quantities, degenerate distribution for any other dimensions). This distribution optimizes the locality of the data on the available PEs and, hence, minimizes data transfer. It should be noted that both the computation of the flow in each subdomain and the block connectivity involves the exchange of data between different PE memory, which is undertaken in an implicit fashion using the globally addressable memory (Fig. 2(a)).

The data parallel code version is written entirely in Fortran 90 style. For a local approximation method such as the finite volume method, the computationally intensive sections of the code involve the addition and subtraction of neighbouring elements according to a stencil determined by the spatial discretization employed. This can be conveniently undertaken using the standard CSHIFT intrinsic function. Since intrinsic functions for shared arrays are currently unavailable for computations on the Cray T3D, these were replaced by the equivalent array assignment statements.

Finally, it is noted that the data parallel model can be implemented on the Cray T3D using FORTRAN 77 with DOSHARED directives preceding the do-loops for explicit work distribution [10]. While this method has not been employed in the present study, it is anticipated that it would lead to a performance similar to that obtained using array syntax.

4.1.2. *Performance Results*

As an application, we consider Mach 1.865 supersonic flow through an air intake (Fig. 4). This flow case has been studied by the French aircraft engine manufacturer Snecma in the development of a powerplant for the replacement of the Concorde supersonic aircraft [11]. A computational mesh consisting of eight blocks each with 15,376 mesh cells was employed.

Performance results obtained using the data parallel code version on different parallel computer systems are given in Fig. 5. These results show that for the SIMD systems (MP-1/2 and CM-200) the performance scales approximately linearly with the number of processors; this is to be expected since for the flow problem considered, the computational arrays have always at least as many elements as there are processors. The performance of the Cray T3D system is also linear up to about 128 PEs; however, further doubling of the number of PEs results in significantly less than a twofold increase in performance. The performance scaling of the CM-5 system departs considerably from linear behaviour, even for a relatively small number of processors. In addition, while the largest configurations of the two MIMD systems have peak performances far superior to those of the SIMD systems, the maximum performance measured for each of the systems differs by only a factor of 2.

There are several potential explanations for the relatively poor performance of the MIMD systems compared to the SIMD systems. First, data parallel computations are communication intensive and require a high nearest-neighbour communication bandwidth. Since the SIMD systems can only be programmed using the data parallel model, they have specially designed hardware for this function. Second, no additional optimization has been performed for the MIMD systems. In particular, on the Cray T3D system the compiler constrains the global array dimensions (except the last) to be a power-of-two; no array padding has been implemented to avoid potential cache thrashing. Third, the computational mesh employed to solve this 2D flow problem, while adequate to resolve the physical flow features, is relatively small. Since data parallel code performance has been shown to increase significantly with problem size [6], it is to be expected that the MIMD systems would provide substantially higher performance levels for large-scale 3D viscous flow computations. Finally, it is important to note that the results obtained on the Cray T3D and CM-5 systems are based on preliminary compiler versions, and are consequently not necessarily representative of the performance available using future compiler releases. In particular, the Cray T3D compiler presently treats many array element references as remote, even though the data is stored in local memory; this leads to a significant degradation in performance, which should be overcome in a subsequent compiler release.

4.2. Message Passing

4.2.1. *Implementation*

The message-passing code version employs a coarse-grain subdomain level parallelism. The computations for different blocks are performed on different processors; for the present implementation only one block is assigned to each processor.

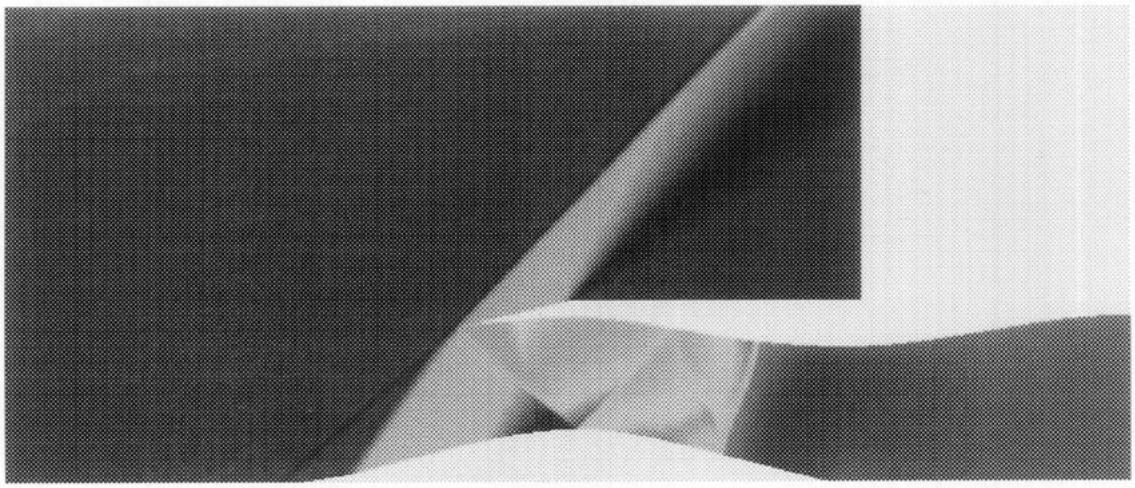

FIG. 4. Pressure contours for supersonic flow through an air intake.

This code version contains only two types of data objects: local arrays (containing the mesh coordinates and flow values) and scalar quantities. All the data are treated as private, with data contained in a different PE memory accessible only via message passing (Fig. 2(b)). The message-passing code version is written entirely in FORTRAN 77.

Block connectivity is undertaken using standard PVM library routines for necessary tasks such as data sending (`pvmfsend`) and receiving (`pvmfrecv`). The use of blocking receives ensures that the update of the flow solution in a subdomain does not proceed before the appropriate values at the edge of the subdomain have been imposed; this imposes block-to-block synchronization between the different subdomains. No other synchronization (either explicit or implicit) is employed at each time step. However, the updating of the global time step and residuals each 50 time steps imposes a global synchronization.

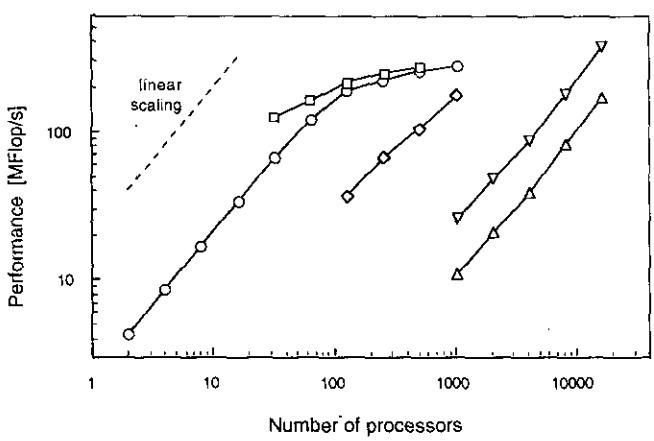

FIG. 5. Performance of the data parallel code version as a function of the number of processors on different MIMD: Cray T3D (○), Thinking Machines CM-5 (□); and SIMD: Thinking Machines CM-200 (◇), MasPar MP-1 (△), and MP-2 (▽) parallel computer systems.

This means that between these global updates all PEs are not necessarily involved in the same code phase, providing the possibility for some overlap between computation and communication. This is particularly important for imposing the boundary conditions at physical domain edges in parallel with block connectivity at subdomain interfaces.

The use of standard PVM provides a high level of portability. The same code, with only minor modifications, has been run on a Cray T3D, an Intel Paragon, two different homogeneous workstation clusters, and a heterogeneous networked computer system consisting of a Cray Y-MP and different workstations [12]. Nevertheless, due to the poor performance of the PVM code on the Intel Paragon, the manufacturer-specific NX message-passing library has been employed to obtain the performance results presented for this computer system.

4.2.2. *Performance Results*

Computations have been undertaken for different parallel computer systems for Mach 2 supersonic flow through a duct (Fig. 6). A computational mesh comprised of 43,008 mesh cells has been employed (chosen to fit into a local memory of 16 MB). This mesh has been subdivided into a number of equal-sized submeshes, corresponding to the number of processors used in the parallel computation.

The performance results obtained using the message-passing code version on the different parallel systems is presented in Fig. 7. For each of the systems considered, the performance can be seen to scale approximately linearly with the number of processors. This indicates that despite the relatively small mesh size—and hence, relatively large ratio of required communication to computation—the processor floating-point performance, and not the communication overhead, is the limiting factor for these computations. (Indeed, for each of the parallel systems considered in Fig. 7, the measured single PE performance is significantly less than peak processor performance.

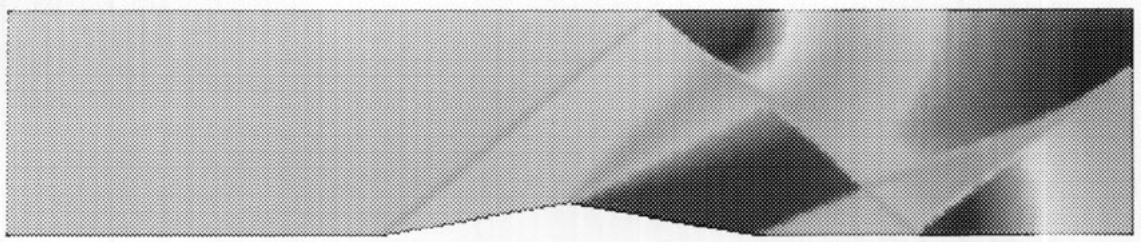

FIG. 6. Pressure contours for supersonic flow through a duct.

This is associated with the low level of data cache reuse by the present code for these cache-based RISC processors.) It should be noted, however, that the maximum number of processors available in the workstation clusters was significantly less than for the integrated MPP systems. For more than 16 workstations, communication overhead is expected to become increasingly dominant, due to the inherent serial (or interleaved) nature of the data traffic on the interconnect network of these workstation clusters.

A more detailed assessment of the relative importance of computation and communication time for the Cray T3D is provided by Fig. 8. This plot shows the time spent performing the computation and communication sections of the code. As the number of PEs increases, the computation time decreases linearly since the same problem size is computed in parallel. The block connectivity overhead also decreases slightly, due to the reduction in the size of the blocks and hence the amount of data exchanged between subdomains. The calculation of the global time step and residuals uses the same hypercube algorithm. However, the time required to compute the global time step is much higher, since it includes the necessary synchronization time between PEs (as discussed above). This synchronization overhead becomes increasingly important as the number of PEs increases and, for a calculation using 1024 PEs, is greater than the time required for actual data exchange.

It is important to note that the computational mesh employed for these computations is relatively small. Indeed, for a 1024 PE system (with local memory of 64 MB/PE) it is estimated that only 0.02% of the total distributed memory is employed! To examine the performance of the Cray T3D for larger problem sizes, the same flow problem was considered but with a computational mesh that increases linearly in size with the number of PEs employed (that is, with 43,008 mesh points per PE). The performance measured for this scaled problem is given in Fig. 9. As expected, the performance scales linearly with the number of PEs over the entire range considered. Figure 10 shows that for the scaled problem, the computation time remains approximately constant (since the computational work per PE remains constant) and is always over two orders of magnitude greater than the communication time.

4.3. Work Sharing

4.3.1. *Implementation*

The work-sharing code version, which also employs a coarse-grain subdomain level parallelism, was implemented on the

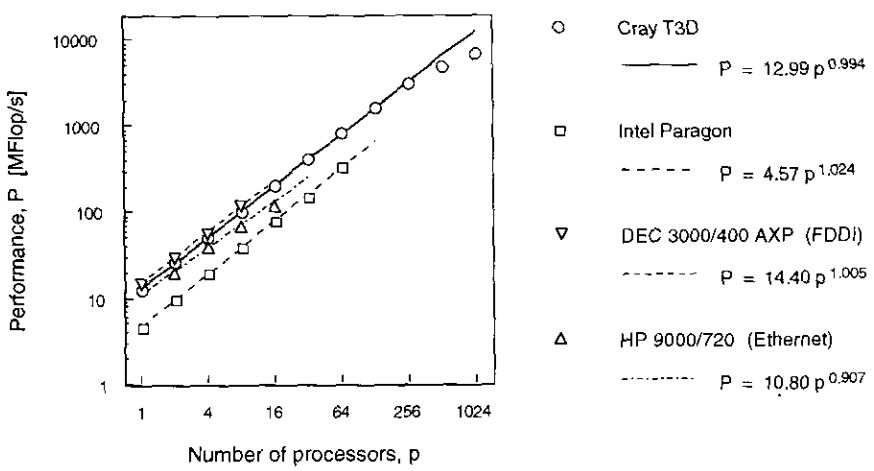

FIG. 7. Performance of the message-passing code version as a function of the number of processors on different MIMD parallel computer systems and workstation clusters.

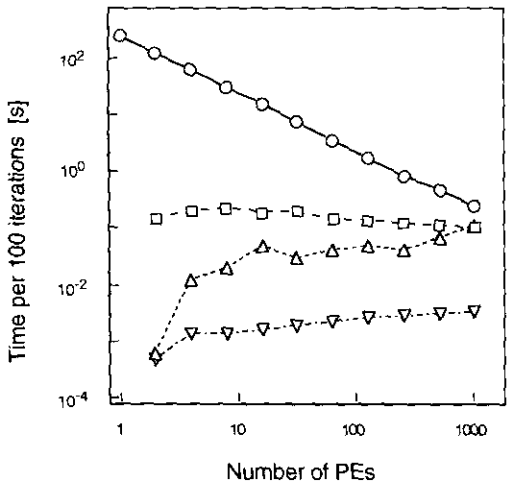

FIG. 8. Time required on the Cray T3D for 100 iterations of the message-passing code version using a constant mesh size (○ computation; □ block connectivity; △ global time step; ▽ global residual).

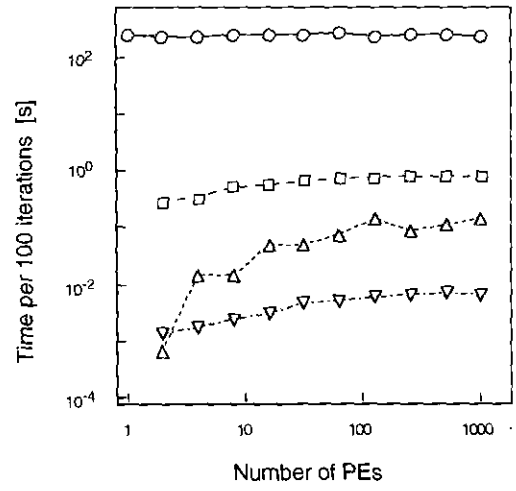

FIG. 10. Time required on the Cray T3D for 100 iterations of the message-passing code version using a scaled mesh size (○ computation; □ block connectivity; △ global time step; ▽ global residual).

Cray T3D. This version contains global arrays, local arrays, and scalar quantities; however, in contrast to the data parallel version the local arrays are declared private. In fact, the work-sharing version can be seen to be intermediate between the data parallel and message-passing versions, comprised of the upper subroutine layers of the first and the lower layers of the second (Fig. 2(c)). The necessary communication between PEs is achieved implicitly via the shared global arrays.

The global arrays are distributed using BLOCK distribution for the block number dimension and degenerate distribution for the other dimensions. The transition from the shared global arrays to the private local arrays is achieved using shared-to-private coercion [10]. The computationally intensive section of the code uses the private arrays to enhance performance. Furthermore, additional private arrays are declared having independent dimensioning to allow array padding to avoid potential cache thrashing. This does not entail the need for additional storage compared to the above-mentioned code versions since in all versions at each time step the flow values at the previous time step must be stored.

4.3.2. *Performance Results*

The computationally intensive section of the work-sharing version is performed using the same subroutines as for the message-passing code version. However, the use of the global

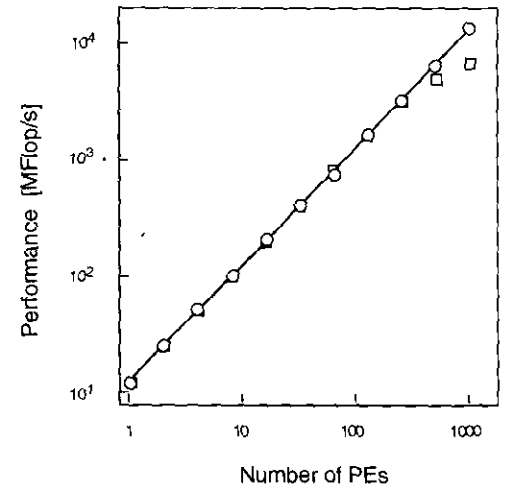

FIG. 9. Performance on the Cray T3D of the message-passing code version for constant (□) and scaled (○) computational mesh sizes.

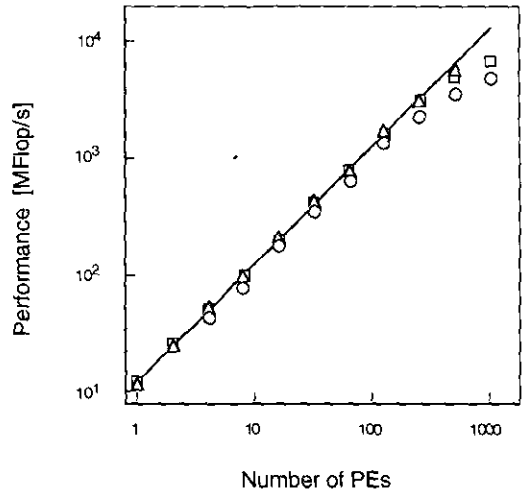

FIG. 11. Performance on the Cray T3D of the work-sharing (○), explicit shared memory (△), and message-passing (□) code versions for a constant mesh size.

arrays for communication adds a global synchronization each time step (at the end of the DOSHARED loop). The consequence of this synchronization can be seen in Fig. 11, which compares the measured performance of the work-sharing version with that of the message-passing version for the constant computational mesh size (43,008 mesh cells). From this figure it can be observed that the performance obtained using the work-sharing version is approximately 20% lower than with the message-passing version.

4.4. Explicit Shared Memory

4.4.1. *Implementation*

The shared memory access (SHMEM) library [13] consists of manufacturer-specific routines that allow low-level explicit data communication via the shared memory of the Cray T3D. In particular, the `shmem_put` routine can be used for the fast transfer of data from a local address to a remote address on a different PE. For point-to-point transfers, `shmem_put` has a latency over 30 times smaller and a bandwidth over 3 times larger than corresponding PVM routines [14].

The SHMEM library can be used in a straightforward manner for a multiblock code based on coarse-grain subdomain level parallelism. This can be achieved either by replacing the PVM routine calls in the message-passing version (Fig. 2(b)), or by replacing the global array addressing in the work-sharing version (Fig. 2(d)).

Since the `shmem_put` routine writes directly into the memory of another PE, it is necessary before the transferred data is used by the remote PE to ensure:

- *synchronization*, which can be achieved via the use of an explicit global barrier,
- *cache coherency*, guaranteed by explicitly flushing the data cache.

4.4.2. *Performance Results*

A preliminary study has been undertaken by implementing `shmem_put` routine calls into the message-passing version. The use of a barrier for global synchronization when performing the block connectivity was found to degrade performance. This can be attributed to the fact that the increase in synchronization overhead due to the addition of the barrier outweighs the advantage of a shorter communication time. This problem was overcome using PE-to-PE synchronization, which is available in the SHMEM library (`shmem_wait`). This resulted in a significant reduction in the time required for block connectivity compared to the message-passing code version, especially when a large number of PEs were employed (Fig. 12). However, for the computational mesh with 43,008 mesh cells the global synchronization (imposed in the calculation of the global time step) exceeded the communication time required for block connectivity when more than 100 PEs were used. Thus only a slight increase in the overall computational performance was measured (Fig. 11).

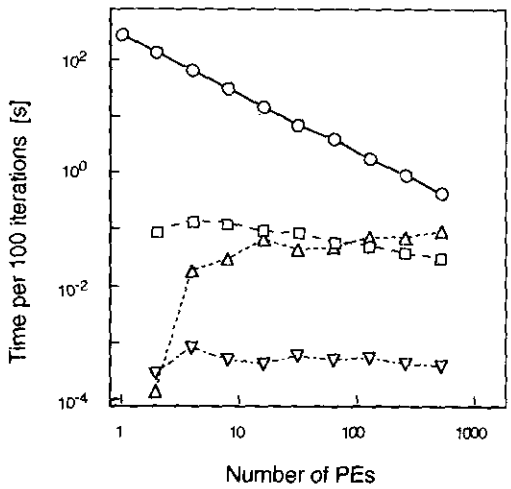

FIG. 12. Time required on the Cray T3D for 100 iterations of the explicit shared memory version using a constant mesh size (○ computation; □ block connectivity; △ global time step; ▽ global residual).

5. COMPARISON OF PROGRAMMING MODELS

The *data parallel* model is the most implicit programming model employed in the present study. This model has the advantage that much of the burden of parallel programming is relieved from the programmer. In particular, no additional partitioning of the data is required to distribute them amongst the processors. This simplifies the preprocessing task of mesh generation, which for complex geometries is already sufficiently complex and time-consuming. In addition, since during each time step the flow in different subdomains is computed in a sequential manner, load balancing between processors can be achieved in a straightforward manner [9]. The performance results obtained in the present study confirm that the data parallel model can be used efficiently on SIMD systems for multiblock flow computations. However, this model places a large burden on the compiler, especially for distributed-memory MIMD computer systems. This is reflected in the performance of the Cray T3D system, which is substantially inferior to that obtained using the other programming models. In addition, the data parallel model can be overly restrictive, for example, in the application of boundary conditions. Nevertheless, the availability of the data parallel model on MIMD systems enables the straightforward porting of codes developed on SIMD computer systems.

Message passing provides a flexible model that places a large amount of control—and responsibility—in the hands of the programmer. This enables high performance levels to be obtained. However, such flexibility is not always required for the SPMD programming style. Indeed, the present message-passing implementation of the multiblock code uses only a small fraction of the functionality available in the PVM library. For large-

scale application codes, the message-passing model is often considered to be too explicit and requires careful attention to avoid programming errors. In addition, the explicit distribution of data to the PEs often necessitates further artificial subdivision of subdomains and can pose difficulty in load balancing. Indeed, it is not always possible before runtime to determine an appropriate work distribution since the "useful work" to be performed is not necessarily proportional to the number of mesh cells. (For example, the computation of the flow cases considered in the present study entails for the upstream blocks merely a recalculation each time step of the freestream flow conditions.) Nevertheless, the message-passing programming model—and, in particular, PVM—is used on a large number of integrated and networked parallel computers, allowing the direct porting of codes between these systems.

The *work-sharing* programming model appears to be a suitable compromise between the above two models. It retains the ease of use of the globally addressable memory, while avoiding the excessive flexibility of a more explicit model. While the present study has shown that there is a performance price to pay on the Cray T3D, this appears to be minimal and presumably warranted for a number of applications. However, only very few distributed-memory parallel computer systems currently offer a programming model based on a globally addressable memory, restricting portability amongst MPP systems. In addition, the work-sharing model considered for the present multiblock code uses the same subdomain level parallelism as the message-passing model and thus necessitates the same availability of suitable meshes for optimal performance.

Finally, a low-level communication library can be employed on the Cray T3D for the *explicit shared memory* model in order to obtain maximum performance. However, the present study indicates that due to its low-level nature it must be used in an appropriate manner. In view of the modest performance gains obtained, the use of this model appears unwarranted for flow simulations based on the numerical method employed in the present study. Nevertheless, the implementation of other numerical methods that are more communication intensive may benefit significantly from this programming model.

6. CONCLUSION

The present study has illustrated different implementations of a multiblock code using four different programming models. These models differ in the distribution of data and work to the PEs, the exchange of data between PEs, and the synchronization between PEs. It has been seen that these issues are important for high performance on a distributed-memory parallel system.

The availability of each of these different programming models on the Cray T3D results in an increase in programming flexibility over that available on other MPP systems. This enhanced flexibility provides the application programmer with the choice of the appropriate parallel programming methodology for the computational problem at hand and simplifies the porting of existing codes from other parallel computer systems. In fact, a suitable combination of the different programming models may provide the most appropriate means to solve a given computational problem.

An assessment of the performance of each model has been presented for two-dimensional inviscid flow computations. The flow cases considered have generally necessitated only a relatively small number of mesh cells to resolve the flow features. As the complexity of the flow problem increases, the size of the computational mesh also increases, leading to a decrease in the communication-to-computation ratio required in obtaining the flow solution on a given number of processors. The present results indicate that for CFD computations based on block-structured meshes the performance bottleneck is generally not interprocessor communication, but single PE performance (that is, a memory-to-CPU communication bottleneck).

Finally, while the present study has considered CFD applications, the appropriate choice of a parallel programming model is of general concern for computational physics and engineering. It is expected that the results of this programming model evaluation, in terms of ease of use, functionality, and performance, are therefore relevant to a large number of application areas.

ACKNOWLEDGMENTS

The authors thank Magnus Bergman (KTH Stockholm), Tom MacDonald, Michel Roche (Cray Research), Roch Bourbonnais (TMC) for their aid in the present study. Zdenek Sekera and Peter Corbett are acknowledged for their assistance in running the codes on the 1024 PE Cray T3D and workstation clusters. Access to parallel computer systems at the AHPCRC University of Minnesota, Cray Research Inc., ETH Zurich, IPG Paris, KTH Stockholm, MasPar Computer Corp., and the University of Bergen is also gratefully acknowledged. This study was supported by the Fonds National Suisse, by the Cray Research–EPFL Parallel Application Technology Program, 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. H. D. Simon (Ed.), *Parallel Computational Fluid Dynamics: Implementations and Results* (MIT Press, Cambridge, MA, 1992).
2. R. B. Pelz, A. Ecer, and J. Häuser (Eds.), *Parallel Computational Fluid Dynamics '92* (North-Holland, Amsterdam, 1993).
3. K. J. M. Moriarty, T. Trappenberg, and C. Rebbi, *Comput. Phys. Commun.* **81,** 153 (1994).
4. P. Olsson and S. L. Johnsson, *Parallel Comput.* **14,** 1 (1990).
5. M. L. Sawley, F. Perrel, C. M. Bergman, and I. Persson, *EPFL Supercomput. Rev.* **4,** 2 (1992).
6. M. L. Sawley and C. M. Bergman, *Parallel Comput.* **20,** 363 (1994).
7. C. Mensink and H. Deconinck, "A 2D Parallel Multiblock Navier–Stokes Solver with Applications on Shared- and Distributed-Memory Machines,"

in *Computational Fluid Dynamics '92*, edited by Ch. Hirsch, J. Périaux, and W. Kordulla, Vol. 2, p. 913, (Elsevier, Amsterdam, 1992).

8. J. Häuser and R. Williams, *Int. J. Numer. Methods Fluids* **14,** 51 (1992).
9. M. L. Sawley and J. K. Tegnér, *Int. J. Numer. Methods Fluids* **19,** 707 (1994).
10. D. M. Pase, T. MacDonald, and A. Meltzer, MPP Fortran Progamming Model (Cray Research Inc., 1994).
11. G. Freskos and O. Penanhoat, ASME Paper 92-GT-206, 1992 (unpublished).
12. C. M. Bergman and M. L. Sawley, *EPFL Supercomput. Rev.* **5,** 10 (1993).
13. R. Barriuso and A. Knies, *SHMEM User's Guide for Fortran* (Cray Research Inc., 1994).
14. R. Numrich, P. L. Springer, and J. C. Peterson, "Measurement of Communication Rates on the Cray T3D Interprocessor Network," in *Proceedings, HPCN Europe '94, Munich, April, 1994.*