Transactions on Information and Communications Technologies vol 9, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

A parallel multi-block Navier-Stokes solver on distributed memory machines

G. Manzini, L. Stolcis
CRS4, via N. Sauro 10, 09123 Cagliari, Italy

Abstract

This work describes the development of a parallel communication strategy on distributed memory platforms and its application to a Navier-Stokes solver for compressible fluid flows. In the present method, distributed message passing libraries are used in order to allow inter-processor communications. In particular, the implementation on the IBM-9076 SP1 available at CRS4 has been performed using the native IBM message-passing library [3] in order to enhance the usage of the inter-processor switch.

1 Introduction

Computational Fluid Dynamics (CFD) has reached a certain degree of maturity and numerical methods based on the solution of the Reynolds-averaged Navier-Stokes equations are widely used for the simulation of turbulent compressible flows. However, such simulations are computationally intensive and require the use of modern supercomputers in order to be effectively used within an industrial environment. Accurate and realistic simulations of engineering problems, require considerable computer memory and CPU time. Between these two requirements there is generally a trade-off. In fact, a code designed to minimize the memory allocation will have prohibitive CPU-times. Vice-versa, if one tries to keep the computational time at a minimum level, there will be severe restrictions concerning the maximum mesh size that can be used, and, hence, the spatial accuracy of the solutions to be computed.

The introduction of parallel computers, together with new programming methodologies based upon distributed memory environments, has given a

relevant impulse to the development of high-performance computational methods. The main objective of the present work is to show that an explicit multi-block structured code can be easily and efficiently extended to distributed memory environments, [4]. The starting point of the present work has been an existing software for the solution of the Navier-Stokes equations for compressible fluid flows developed at CRS4 and named *Tharros* [5]. The program solves a mixed parabolic-hyperbolic system of PDEs which express the basic physical principles of conservation of mass, momentum and energy, which are completed with an additional algebraic equation of state for the pressure, and the Fourier's law, to relate heat flux to temperature gradients, see [2]. For turbulent flows, an eddy-viscosity approach has been employed, and the total viscosity is evaluated as the sum of a laminar- and a turbulent viscosity. Two different turbulence models has been implemented: an algebraic model (Baldwin-Lomax), and a two equation k-ϵ. The spatial discretization is performed with a structured finite volume 2^{nd}-order scheme (both upwind and symmetric TVD schemes have been implemented), and an explicit multistage Runge-Kutta method is used to advance the solution in time. For steady state-solutions, acceleration techniques such as local time-stepping are implemented to provide numerical efficiency and to increase the convergence rate. The turbulent transport equations employed for the $k-\epsilon$ model are solved in an implicit way in order to enhance stability and convergence [1]. A sequential multi-block approach has been originally chosen for the implementation in order to allow the treatment of complex geometry.

2 Parallel strategy

In the present work we have adopted a coarse-grain approach for the parallelization of our CFD solver. Different approach are also possible, for example tuning the code at a do-loop level (fine-grain parallelization). On massively parallel SIMD platform, such as the CM-200 or CM-5, this is managed in a transparent way using sophisticated programming languages like Fortran 90 or HPF. On MIMD machines like shared memory system (Convex, Cray) and virtual shared memory systems (BBN, KSR) the parallel tuning of do-loops is provided by some compiler directives that must be specified by the user or by some "ad hoc" extension of the standard Fortran. It is still an open question which is the best strategy, the answer depending on too many factors. It is undoubtedly true that this latter type of parallelism could demand a substantial rewriting of parts of the code. On the contrary, a sequential multi-block code can be easily parallelized with minor modifications, by simply adding a set of high-level routines to manage communications in a multiprocessor environment. The basic idea relies on the fact that in an explicit multi-block solver, the global domain is

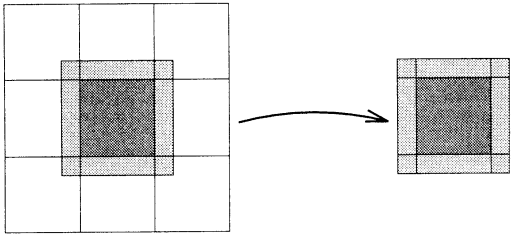


Figure 1: A block with its dummy cells

partitioned into blocks whose update is independent. This intrinsic parallelism can be directly exploited by defining a correspondence between blocks and processors. The stencil required to update flow variables inside a given cell has a limited local spatial extension, because information propagates with a finite velocity. For the cells close to an internal boundary, that is a boundary shared by two adjacent blocks, a part of the computational stencil falls outside the block and inside a neighbouring one. Hence, the update of each block requires the knowledge of the values taken by the flow variables inside a region larger than a single block but much smaller than the global domain of computation, as shown in Figure 1. If this latter block is treated by a different processor, it is necessary to establish a communication through message passing send-receive primitives. Finally, the global solution is obtained by "putting together" the local solutions of all the blocks. The original code has been designed to advance the solution in time by sequentially running one block at a time and the exchange of informations is performed via a simple copy procedure in a dummy cell data structure. The parallel implementation differs from the sequential one for a set of routines which manage via send-receive primitives this exchange of information.

3 Parallel Implementation Details

3.1 Load Balancing Consideration

The parallel efficiency is strongly affected by the mapping of blocks on processors. Unfortunately, it is very difficult to develop an algorithm which automatically maps blocks on processors in an optimal way. Hence, the user is required to supply such a mapping as an input data file. The information to be given by the user are limited to the number of blocks treated by each process in a global numbering system and a corresponding local number on the processor to which the block has been assigned. A one-to-one mapping is usually considered in parallel solvers, but we generalized this approach in order to allow more than one block on any processor. Two main reasons justify this strategy: the mapping block-to-processor is inde-

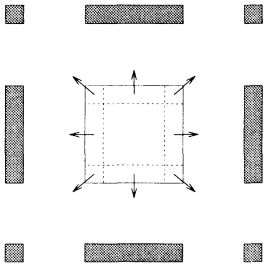


Figure 2: Send operation

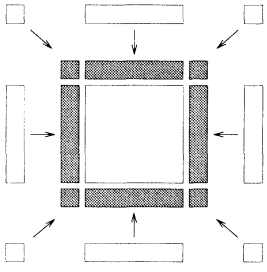


Figure 3: Receive operation

pendent from the number of processors available, and allows an easy control of load-balancing on the processors, by suitable re-distribution of blocks on the processors available. Such a generalized block-processor mapping allows the runs to be independent from the number of processors effectively available on the machine. If more blocks than processors are present, some processes will run in multi-block mode treating more than one block. Furthermore, this approach also allows a restart with a number of processors different from the previous run. Moreover, this feature provides an easy way to control the load balancing. When several blocks with different sizes are present due to a complex geometry, these blocks can be grouped into clusters of approximately the same size. In this way a rough balancing of the computational effort can always be ensured. Finally, we want to stress that the file defining the distribution of blocks on processors is the only supplementary input data with respect to the input files for sequential run required to the user.

3.2 Boundary conditions, connectivities and communications

The solver sets the boundary conditions at the beginning of each stage of the Runge-Kutta cycle by a standard ghost-case technique. Two layers of dummy cells are used to set the values used by the multi-block solver. The single-processor version of the code fills these dummy cells by a simple copy procedure. The parallel version works in the same way when the dummy cells overlap a block which is assigned to the same process (*internal* connectivity). When the dummy cells contain values belonging to a block mapped on a different process, an inter-processor communication is necessary (*external* connectivity).

Communications are directly managed by the parallel solver through a communication list, which is based on the distribution of blocks on processors and block connectivities. This communication list allows an easy-to-use

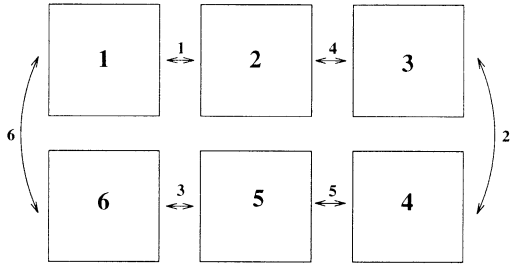


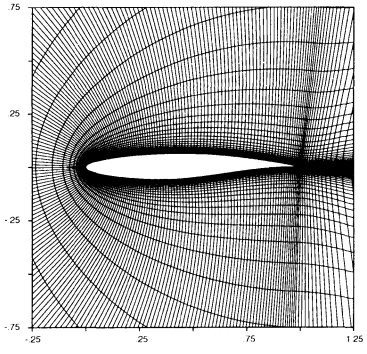
Figure 4: Communication among six blocks

way of synchronize the processes during the communication phase. The native communication library EUI-PE of the SP1/SP2 machine requires such a synchronization, because the communication buffer is only 128 bytes long. When the send process starts its send operation, it fills the communication buffer, and the receiver process must be in receiving mode to empty it. The send-receive operation stops if the buffer is full. Hence, if the send/receive operation is not synchronized, the communication falls in deadlock. During the communication step, every process scans the communication list, and when a communication in which it is engaged is found, the communication is started.

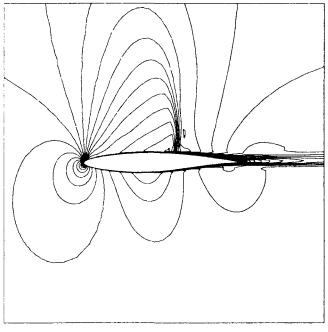
The performances are strongly affected by the order of the communications executed by the code. Therefore, it can be very difficult to develop a general algorithm able to optimize the communication list. For this reason, the standard communication list is always built in a sequential way, and the possibility of reordering the communications is left to the user as a final tuning of the parallelization. Figure 3.2 shows an example of a possible reordered communication sequence among six processors. Communications $(1, 2, 3)$ and (resp.) $(4, 5, 6)$ take place simultaneously, optimizing the synchronization delays among processors.

3.3 The I/O operations

All the operations involving I/O on disk are performed by a "master" process identified with $id = 0$. Hence, all the initialization operations, which requires to read data from disk, are performed by this process. The master process initializes itself, by reading all the input data files from disk, with boundary conditions, mesh, and starting solution, and then it broadcasts all the starting values and the parameters of the run to the other processes. Some diagnostics (for example the monitoring of convergence) that are to be executed during the run requires global computations. Processes perform partial diagnostics on their local data, which are sent to the master process to perform the final global diagnostics.



Computational grid



Iso-Mach contours

Figure 5: Rae 2822 Airfoil ($M_\infty = 0.734$, $\alpha = 2.79^o$, Re=6500000)

4 Performance measurements

As benchmark for performance measurements, we considered the standard RAE 2282 aerofoil configuration. The original "C" mesh has been radially divided in 20 blocks, 16 of size 53×65 and 4 of size 49×65, with a total amount of 67870 points. Figure 5 presents the computational mesh and the final iso-mach contours. Three different calculations of industrial interest have been taken into account: inviscid flows, viscous laminar flows, and viscous turbulent flows. Table 1 presents the elapsed time required by the sequential solver to perform the first 200 iterations in the time-stepping loop without the start-up time of communications in the initialization phase.

The performances of the 2D multiprocessor solver are reported in Tables 2-4, in term of speed-up and efficiency defined by:

$$Sp. = \frac{\text{elapsed time on } one \text{ processor}}{\text{elapsed time on } p \text{ processors}}, \quad Eff. = \frac{S_p}{p}. \tag{1}$$

In the threes cases, times were taken in runs with 1, 2, 5, 10, and 20 processors by assigning respectively 20, 10, 4, 2 and 1 blocks to each node. The ratio between computation and communication time per iteration become less favorable as the number of processors grows up. Nevertheless, the measured speed-ups shows a good scalability of the schemes implemented in the code. As a general trend, increasing the complexity of the physical model – inviscid $\rightarrow$ viscous laminar $\rightarrow$ viscous turbulent – gives better performances, due to the increased amount of calculations required per time-step with respect to the communications.

5 Final Remarks

Our results show that distributed parallel computing is useful for engineering CFD to reduce the elapsed time for large-scale calculations because of a good scalability of the schemes generally adopted in computational fluid dynamics. We have also shown that an explicit multi-block code can easily be parallelized on a distributed memory machine, exploiting in a direct way the intrinsic parallelism provided by the partition of the computational domains in independent blocks. It is clear that the present parallel code is only a starting point. In the future we want to investigate the possibility of using in a distributed parallel environment more sophisticated schemes for time-stepping, such as multigrid and implicit time-marching. These latter ones will allow us to realize more efficient calculations both from a numerical- as well as from a computational point of view.

References

[1] Davidson, L. Implementation of a $k - \epsilon$ model and a Reynolds stress model into a multi-block code, Appmath Report 93-21, CRS4, Cagliari, Italy;

[2] Hirsch, C. *Numerical computation of internal and external flows* , England, 1990;

[3] IBM AIX Parallel Environment, Release 1.0, International Businnes Machines Corporation, 1993;

[4] Manzini, G. and Stolcis, L. Development of a parallel block-structured Navier-Stokes solver on distributed memory platforms, Appmath Report 94-16, CRS4, Cagliari, Italy;

[5] Mulas, M. (private communication, 1994), CRS4, Cagliari, Italy.

Acknowledgement

The present work has been carried out with the financial support of the Sardinia Regional Government. Author's names are listed in alphabetic order.

Table 1: Performances of the 2D solver: time in seconds for 200 iterations on one SP1 node for an inviscid, a laminar , and a turbulent calculation

#Nodes	#Block per node	Euler	Laminar NS	k-ϵ NS
1	20	2987.4 secs	4159.5 secs	4427.5 secs

Table 2: Speed-ups and efficiency for Euler solver (inviscid flows)

#Nodes	#Block per node	Speed-up	Efficiency
2	10	1.96	0.98
5	4	4.56	0.91
10	2	8.69	0.87
20	1	15.55	0.78

Table 3: Speed-ups and efficiency for Navier-Stokes solver (laminar flows)

#Nodes	#Block per node	Speed-up	Efficiency
2	10	1.96	0.98
5	4	4.72	0.94
10	2	8.91	0.89
20	1	15.27	0.76

Table 4: Speed-ups and efficiency for Navier-Stokes solver (turbulent flows)

#Nodes	#Block per node	Speed-up	Efficiency
2	10	1.95	0.97
5	4	4.68	0.93
10	2	8.95	0.89
20	1	15.58	0.78