# Domain Decomposition for the Simulation of Transient Problems in CFD

R. LÖHNER* AND K. MORGAN†

## 1. Introduction

The two key factors that have always led to progress in Computational Fluid Dynamics (CFD) are better algorithms and faster computers. Advances in both fields have made it possible to simulate routinely many fluid dynamic problems which were thought untractable only a decade ago.

Among the many reasons that may lead us to consider domain decomposition methods (DDMs) for the solution of transient problems in CFD, the following two are the most important:

a) use DDMs to exploit optimally parallel machines,

b) develop DDMs to save CPU-times on any machine.

## 2. DDMs that exploit optimally parallel machines

The only way to circumvent the inherent limitation given by the finite speed of light in a single CPU-machine is parallelism in hardware. During the last decade the easiest form of parallelism was exploited with the advent of vector machines. An arithmetic operation which had to be performed many times was partitioned into sub-operations. Each of these sub-operations was carried out concurrently on different elements of a string of numbers. The achievable gain in speed of these vector-machines (CRAYs and CYBER-205s) was a factor of 10-20, depending on the arithmetic operation and computer architecture. Most important of all, it was limited by the number of subop-erations into which the original arithmetic operation could be subdivided.

* Berkeley Research Associates, Springfield, VA 22150, and Laboratory for Computational Physics and Fluid Dynamics, Naval Research Laboratory, CODE 4410, Washington, D.C. 20375.

† Institute for Numerical Methods in Engineering, University College of Wales, Swansea SA2 8PP, Wales.

However impressive this speed-up may seem, many practical problems (such as the simulation of turbulence and flow past complete configurations) will require machines that are about 10-100 times faster (and larger) than today's fastest computer. The only way to achieve these speeds is by spreading the computational workload to many processors. In the near, foreseeable future several vector-processors will be aligned together, whereas in the more distant future we expect to see machines with thousands of individual processors. According to which machine is available, the implementation of algorithms in codes will vary. Let us therefore take a closer look at both types of machines.

2.1 Machines consisting of few vector processors

In order to exploit the architecture of these mildly parallel machines algorithms must be constructed that:

a) allow the individual processors to operate independently,

b) exploit the 'vector-architecture' of the processors,

c) minimize the information transfer to and from and between processors.

The typical implementation of algorithms on this type of machine is as follows:

i) take a scheme that lends itself to vectorisation,

ii) subdivide the computational domain $\Omega$ into (a few) subdomains $\Omega_i, i = 1, ...m$,

iii) minimize the necessary information transfer between the subdomains $\Omega_i, i = 1, ...m$,

iv) operate in vector mode on each individual CPU/$\Omega_i$.

Remark: if the tranfer of information from memory to CPU is fast one could perform the subdivision of operations at the DO-loop level. Each processor would then operate on a portion of the original DO-loop. Practical experience indicates that this mode of operation, called 'microtasking' in CRAY-jargon, offers a much better 'return of investment' if the original DO-loop is long enough. In fact, even debugging the multiprocessor-code becomes much easier. This would indicate that this type of domain decomposition must only be implemented for machines that have slow memory to CPU transfer rates. However, it is not yet clear if machines of this type will see widespread dissemination in the future.

2.2 Machines consisting of thousands of scalar processors

In order to exploit the architecture of these massively parallel machines algorithms must be constructed that:

a) maximize the parallelism of the original scheme,

b) minimize the information transfer beyond 'nearest neighbors'.

The typical implementation of algorithms on this class of machine would be as follows:

i) take some explicit (iterative) scheme with a high degree of inherent parallelism,

ii) subdivide the computational domain $\Omega$ into as many subdomains $\Omega_i, i = 1, ...m$ as possible,

iii) minimize the necessary information transfer between the subdomains $\Omega_i, i = 1, ...m$,

iv) operate in scalar mode on each individual CPU/$\Omega_i$.

Remark: In order to satisfy step ii), for typical Finite Difference or Finite Element schemes each individual cell or element will be allocated to a processor. This means that the coding would look very similar to that written for a scalar, one-processor machine of the sixties. The aim would be to perform as many operations as possible at the cell or element level, then interchange information with the nearest neighbors.

The rather disappointing result is that if one seeks to develop DDMs in order to exploit optimally parallel machines, the outcome does not look very different from algorithms currently in use. The only type of machine that requires extensive use of DDMs in order to run efficiently is the mildly parallel machine with slow memory to CPU transfer rates.

## 3. DDMs to save CPU-times on any machine

In many applications, the region of maximum physical or geometrical complexity is confined to a small subregion of the whole domain under consideration. Therefore, one can seek to reduce CPU-costs by combining different levels of physical modelling and/or discretization.

On different subdomains one could model or solve for:

1) different PDEs (e.g. potential, Euler and Navier Stokes eqns.),

2) different algorithms for a given PDE (e.g. Runge-Kutta, Lax-Wendroff, TVD for the Euler eqns.),

3) different grids (structured, unstructured [2,3]),

4) different spatial approximations (FDM, FVM, FEM, Spectral,... [2,3]),

5) different order of spatial approximation (2nd, 4th order [4], p,h refinement [5],...),

6) different time-stepping schemes (explicit, implicit, semi-implicit [6]),

7) different order of time-stepping schemes (2nd, 3rd order, ...),

8) different timestep-sizes ( $\Delta t, 2\Delta t$, ... [1,6]).

Practical experience with this class of DDMs indicates that the savings in CPU-times that can be achieved with them is at most a factor of 10, and usually lies between 2-4. The main reason for this disappointing gain is that the region of highest physical and geometrical complexity is usually that which contains the highest number of gridpoints/degrees of freedom. As a typical example, consider the simulation of flow past an airfoil using as the most complex physical model the Reynolds-averaged Navier-Stokes equations: most of the gridpoints will be located in the boundary layer, where the most complex physical model is employed, whereas only a relatively small percentage of gridpoints lies in those regions where simpler physical models (e.g., the Euler equations or the Laplace equation) can be used. Moreover, if an algorithm of

this class is implemented on a mildly parallel machine (e.g. a CRAY-XMP), balancing the workload on all processors may prove to be difficult.

### 3.1 Domain Decomposition according to multiples of $\Delta t_{min}$

A typical grid used for the simulation of compressible flows will exhibit a large variation in mesh-size, as the analyst tries to cluster gridpoints only in those regions where they are needed. If we try to advance the solution time-accurately with an explicit (conditionally stable) solver, the allowable timestep, given by that of the 'smallest' element, will have to be employed throughout the grid. This implies

a) a waste of CPU-time, as bigger timesteps could be taken for the larger elements, and

b) a possible degradation of accuracy for those zones where the Courant- number is very small.

Therefore, an algorithm is needed that advances the solution in a time-accurate manner with a Courant-number that is similar throughout the grid . Several authors have proposed algorithms that meet this design criterion [1,6]. Our own algorithm was described in [1]. We therefore only give a brief description of it here.

Algorithmically, the domain-splitting routine proceeds in the following order:

i) Compute the allowable time-step $\Delta t_e$ of each element.

ii) Grade the allowable time-steps of the elements according to the smallest allowable time-step $\Delta t_{min}$, with $\Delta t_{min} = min_e \Delta t_e$. This produces regions of elements, where

region 1 contains elements with $\Delta t_{min} \leq \Delta t_e < 2\Delta t_{min}$ ,

region 2 contains elements with $2\Delta t_{min} \leq \Delta t_e < 4\Delta t_{min}$ ,

region 3 contains elements with $4\Delta t_{min} \leq \Delta t_e < 8\Delta t_{min}$ , etc.

iii) For each region $n$: a) find the boundary nodes of this region, b) include into region $n$ all those elements of the regions $m > n$ that have all their nodes on the boundary of region $n$. This smoothes the shape of the regions, avoiding saw-tooth-type boundaries.

iv) Add two layers of elements from regions $m > n$ to each region $n$, in order to overlap, storing the boundary nodes. This overlapping of regions is necessary in order to achieve a time-accurate algorithm. The overlap-region could be several elements thick, but it was found that an overlap-region of two elements was sufficient [1].

The solution is then advanced accordingly in time, that is to say:

2 steps in region $I \rightarrow 2\Delta t_m$

1 step in region $II \rightarrow 2\Delta t_m$

2 steps in region $I \rightarrow 4\Delta t_m$

1 step in region $II \rightarrow 4\Delta t_m$

1 step in region $III \rightarrow 4\Delta t_m$

....

A major concern that arises when trying to exploit optimally parallel machines is the achievement of similar workloads for all processors. One could envision cases for which one region contains many more elements than all other regions. In order to be able to avoid this problem, the DDM must also be applicable to splittings of equal timesteps. It was shown in [1] that the described DDM satisfies this requirement.

## 4. Conclusions

We have described the scope and use of domain decomposition methods (DDMs) for the simulation of transient problems in CFD. It was found that if the design criterion for a DDM is to exploit optimally parallel machines, then the only type of machine for which DDMs are well suited is one that has only few processors with slow memory transfer rate and high CPU speed. If memory access is fast, or if the machine has many thousands of processors, then microtasking vector-code or distribution of each cell/element to a processor will outperform typical DDMs.

If the design criterion is to reduce CPU requirements on any machine, then many possibilities are open. We have described a tentative list in section 3. However, the achievable gain in speed is at most a factor of ten, and usually lies between two and four. The reason is that typically physical complexity (CPU per degree of freedom) and the number of degrees of freedom go hand in hand. This produces a 'quadratic' increase of work, that precludes higher saving factors. As an example, consider the transonic flow past an airfoil. Suppose that we decide to decompose the domain according to the simplest partial differential equation that still reproduces the correct physics. We would then have a region of potential flow (Laplace eqn.), a region of inviscid, rotational flow (Euler eqns.), a region of laminar, viscous flow (Navier-Stokes eqns.), and a region of turbulent, viscous flow (Reynolds-averaged Navier-Stokes eqns.). Practical experience shows that we need more gridpoints in the small regions of viscous flow than in the larger region of inviscid, rotational flow. In turn, we need more points in the region of inviscid, rotational flow than in the large region of inviscid flow. Moreover, we need more floating point operations per gridpoint in the 'Navier-Stokes regions' than in the 'Euler regions', and still less in the 'Laplace region'. Thus, the gain in speed achieved by DDMs remains bounded.

## References:

[1] R. Löhner, K. Morgan and O.C. Zienkiewicz - The Use of Domain Splitting with an Explicit Hyperbolic Solver; *Comp. Meth. Appl. Mech. Eng.* 45, 313-329 (1984).

[2] K. Harumi, T. Kano, H. Okada anf A. Ootsuki - New Method Combining Finite Element Method with Finite Difference Method for Tidal Flow Computation; pp. 1011-1018 in *Proc. Fourth Int. Symp. Finite Element Methods in Flow Problems* (T. Kawai ed), University of Tokio Press (1982).

[3] K. Nakahashi - FDM-FEM Zonal Approach for Computations of Compressible Viscous Flows; *Proc. 10th Int. Conf. Num. Meth. Fluid Dynamics*, Beijing, July 1986. Springer Lecture Notes in Physics (1987).

[4] W. Schönauer, K. Raith and K. Glotz - The Principle of Difference Quotients as a Key to the Self-Adaptive Solution of Nonlinear Partial Differential Equations; *Comp. Meth. Appl. Mech. Eng.* 28, 327-359 (1981)

[5] O.C. Zienkiewicz, J.P. de S.R. Gago and D.W. Kelly - The Hierarchical Concept in Finite Element Analysis; *Comp. Struct.* 16, 53-65 (1983).

[6] T. Belytschko, h.-I. Yen and R. Mullen - Mixed Methods for Time Integration; *Comp. Meth. Appl. Mech. Eng.* 17/18, 259-275 (1979).