A SCALABLE STRATEGY FOR THE PARALLELIZATION OF MULTIPHYSICS UNSTRUCTURED MESH-ITERATIVE CODES ON DISTRIBUTED-MEMORY SYSTEMS

Kevin McManus
Mark Cross
Chris Walshaw
Steve Johnson
Peter Leggett

CENTRE FOR NUMERICAL MODELLING AND PROCESS ANALYSIS, UNIVERSITY OF GREENWICH, LONDON

Summary

Realizing scalable performance on high performance computing systems is not straightforward for single-phenomenon codes (such as computational fluid dynamics [CFD]). This task is magnified considerably when the target software involves the interactions of a range of phenomena that have distinctive solution procedures involving different discretization methods. The problems of addressing the key issues of retaining data integrity and the ordering of the calculation procedures are significant. A strategy for parallelizing this multiphysics family of codes is described for software exploiting finite-volume discretization methods on unstructured meshes using iterative solution procedures. A mesh partitioning-based SPMD approach is used. However, since different variables use distinct discretization schemes, this means that distinct partitions are required; techniques for addressing this issue are described using the mesh-partitioning tool, JOSTLE. In this contribution, the strategy is tested for a variety of test cases under a wide range of conditions (e.g., problem size, number of processors, asynchronous/synchronous communications, etc.) using a variety of strategies for mapping the mesh partition onto the processor topology.

Address reprint requests to Mark Cross, Centre for Numerical Modelling and Process Analysis, University of Greenwich, Maritime Greenwich University Campus, 30 Park Row, Greenwich, London SE10 9LS.

The International Journal of High Performance Computing Applications, Volume 14, No. 2, Summer 2000, pp. 137-174
© 2000 Sage Publications, Inc.

1 Introduction

Computational mechanics (CM) modeling and analysis software now underwrites almost all the design functions for almost every industry sector. Most CM analysis activities fall into three categories:

- solids-dominated problems, serviced by the extensive finite element analysis (FEA) community (Zienkiewicz and Taylor, 1991);
- flow-dominated systems, serviced by the more compact computational fluid dynamics (CFD) community (Patankar, 1980; Versteg and Malalasekera, 1995);
- electromagnetics problems, serviced by a developing computational electromagnetics (CEM) community (el Dabaghie et al., 1998).

The numerical techniques, particularly implementation strategies and software structures for the above themes, have developed through rather different routes and so have the appearance, at least, of a disparate family of analysis tools. As one example, most FEA tools are structured on the basis of direct solvers, while the vast majority of CFD codes exploit segregated iterative solvers, and this one distinction has a significant effect on the resulting software.

We now see CFD or FEA being applied to problems with a mesh of 10^5 to 10^6 elements/cells, for 10+ variables per cell/element and for 10^2 to 10^3 time steps. The demand to solve such problems in a reasonable time frame (i.e., hours, not days or weeks) means that parallel computer architectures with substantial raw processing power and large, though distributed, memory capability are being increasingly exploited. As such, for the past decade, a huge research effort has been under way to develop, implement, and demonstrate parallelization strategies for CM codes on a wide range of hardware.

There is now a well-established consensus that the most effective way to parallelize CM codes to achieve scalable efficient parallel performance involves the use of the single-program multiple-data (SPMD) paradigm whereby

- Each processor runs essentially the same program but operates on its own subset of the total data.
- The mesh that covers the solution domain is partitioned so that each processor stores only its own submesh plus some halo layers to hold data that are generated on another processor but required on the current one.

- Standard message-passing tools (e.g., PVM, MPI) to send data to or receive data from neighboring processors are used. These are required to emulate the solution process in scalar mode, or the equivalent data access in shared-memory systems using open MP, for example, is facilitated.

"The mesh-partitioning task is trivial for single-block structured meshes, typical of many CFD codes, and can be easily achieved at runtime with a simple algorithm that allocates sets of cell layers to a processor."

The mesh-partitioning task is trivial for single-block structured meshes, typical of many CFD codes, and can be easily achieved at runtime with a simple algorithm that allocates sets of cell layers to a processor. For block-structured and unstructured meshes, the mesh-partitioning problem is more complex, and techniques and tools have been developed for this purpose.

While the focus on phenomena-specific analysis tools exemplified by the CFD and FEA software communities have adequately served the needs of most engineering design functions, this constraint is not appropriate for many manufacturing processes, which may well involve interactions among the fluids, solids, and thermal and electromagnetic phenomena. To model such processes adequately requires a distinctive approach. For a variety of reasons, it is not practical to "hook together" phenomena-specific codes:

- The incompatibility of the meshes and variable data representation requires a good deal of numerical filtering that compromises analysis accuracy.
- If the phenomena (and, consequently, the codes) are anything other than very loosely coupled, the whole simulation process is dominated by data transfer between the software tools. If the simulation is to operate in parallel (which it may well need to be), this problem is exacerbated.
- Achieving reliably converged solutions of tightly coupled problems can be extremely difficult.

This means that if these "multiphysics" problems are to be addressed effectively in the next decade, a more coherent approach is required. One could argue (Cross, 1996) that solving closely coupled multiphysics problems requires a single software framework that has the following key features:

- a single mesh, so that as the problem requires, different physics (and their interactions) can be switched on or off dynamically throughout the whole domain;
- a single database, so that there is no data transfer between programs but only usage by separate modules;

- the solution procedures for each of the phenomena have a measure of compatibility;
- every opportunity to enable high numerical accuracy filtering and use of data generated in one module but used by another.

In addition, one could further delineate the structure of the solution of such problems via the fully assembled matrix approach or through the segregated solution of groups of equations representing each of the phenomena within some iterative loop. We have been involved in one such venture for most of the past decade. This venture has been based on a family of numerical procedures using finite-volume techniques but has been extended to an unstructured mesh. Procedures have been developed for heat conduction with solidification/melting, Navier-Stokes flow (including free surfaces), electromagnetically driven flow, and solid mechanics. The solution strategy has followed the segregated approach because procedures for each distinctive phenomenon are well established and so successful techniques for multiphysics computations could more readily be identified. Finite-volume techniques were employed because it was recognized that many multiphysics simulations involve complex Navier-Stokes flows (e.g., free surfaces, multiphase), and at the time this work was initiated (~1990), the former was demonstrably better able to address such phenomena than conventional finite-element methods. The initial product of the Greenwich effort was the UIFS code, which had the capability to address fluid flow (Chow, Cross, and Pericleous, 1995), heat transfer with melting/solidification (Chow and Cross, 1992), and nonlinear solid mechanics (Taylor, Bailey, and Cross, 1995). This code was particularly targeted at casting processes that typically involve the pouring of hot liquid metal into a (relatively) rigid solid mold, residual convection of the liquid metal, and cooling and solidification, followed by the development of stress in both the solidified metal and mold plus their relative deformation (Bailey et al., 1996). This work has been subsequently extended to three dimensions and now forms the basis of the multiphysics analysis software, PHYSICA (Cross, 1996). This tool is one of a number of multiphysics codes emerging in the engineering analysis community, including SPECTRUM and, most recently, the TELLURIDE code at Los Alamos.

As suspected at the outset, it very quickly became obvious that simulations involving interactions of the above phenomena would require significant processing power. Hence, a strong need for operating such codes in parallel was perceived. However, although it was obvious that a conventional SPMD strategy should be employed, its application in a multiphysics context gives rise to a range of unique problems that need to be addressed. Hence, we decided to investigate these issues in the context of the UIFS code because all the perceived challenges appeared to be no less difficult in 2-D than 3-D. Moreover, the resulting strategies have been subsequently directly incorporated into the 3-D code, PHYSICA (Cross, 1996). This paper describes the parallelization strategy pursued to deliver high-efficiency parallel operation for multiphysics codes, involving fluids interacting with solids and using iterative procedures over an unstructured mesh.

The layout of the paper is as follows: a brief overview of the target multiphysics code is given to place the work into a computational context. It is followed by a description of the parallelization strategy (including some key implementation details), an assessment of the parallelization strategy on a typical multiphysics test problem for a range of scenarios, and a conclusion.

2 Overview of the Target Multiphysics Code

2.1 KEY SET OF EQUATIONS

The UIFS code is two-dimensional (Bailey et al., 1996). The key sets of continuum equations are as follows.

Navier-Stokes Flow

$$\frac{\partial}{\partial t}(\rho u_i) + \nabla(\rho v u_i) = \nabla(\mu \nabla u_i) - \frac{\partial p}{\partial n_i} + S_{u_i} (i = x, y), \quad (1)$$

which expresses the conservation of momentum, and

$$\frac{\partial \rho}{\partial t} + \nabla(\rho v) = S_c, \quad (2)$$

which describes the conservation of mass—see the list of nomenclature. The momentum source is essentially a function of two factors—a buoyancy force and the flow resistance due to the presence of solidified material:

$$S_{u_i} = \rho(T)g_o - \frac{\gamma}{K}u_i. \tag{3}$$

Energy

$$\frac{\partial}{\partial t}(\rho h) + \nabla(\rho v h) = \nabla(k\nabla T) + S_h. \tag{4}$$

The enthalpy is related to the temperature via $h = \int_{T_{ref}}^{T} c dT$.

The source term is associated with the release or consumption of latent heat during a phase change and is given by

$$S_h = -\frac{\partial}{\partial t}(L\rho v f_\ell), \tag{5}$$

where the liquid fraction, f_ℓ, is a function of temperature.

Solid Mechanics

The general equilibrium equations governing the conservation of force on a static body are

$$\frac{\partial}{\partial_x}\Delta\sigma_{xx} + \frac{\partial}{\partial_y}\Delta\sigma_{xy} = \Delta_{fx},$$

$$\frac{\partial}{\partial_y}\Delta\sigma_{yy} + \frac{\partial}{\partial_x}\Delta\sigma_{xy} = \Delta_{fy}. \tag{6}$$

In matrix form, these equations may be transformed to

$$\Delta\sigma = D\Delta\varepsilon^{(e)}, \tag{7}$$

where

$$D = \frac{E}{(1-\mu^2)}\begin{bmatrix} 1 & \mu & 0 \\ \mu & 1 & 0 \\ 0 & 0 & \frac{1-\mu}{2} \end{bmatrix}.$$

The total and elastic strains are related by

$$\Delta\varepsilon^{(T)} = \Delta\varepsilon^{(e)} + \Delta\varepsilon^{(vp)} + \Delta\varepsilon^{(th)},$$

which is related to displacements by

$$\Delta\varepsilon^{(\mathrm{T})} = L\Delta d, \tag{8}$$

where

$$L = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix},$$

$$\Delta\varepsilon_{xx}^{(T)} = \frac{\partial}{\partial x}\Delta u, \text{ etc.,}$$

$$\Delta\varepsilon_{xx}^{th} = \alpha\Delta T, \text{ etc.,}$$

and the viscoplastic strains are given by the Perzyna model form:

$$\Delta\dot{\varepsilon}_{ij}^{(VP)} = \frac{d\Delta\varepsilon_{ij}^{(VP)}}{dt} = \gamma\left\langle \frac{\sigma^{(eff)}}{Y} - 1 \right\rangle^{N} \frac{3}{2\sigma^{(eff)}} S_{ij} \quad \sigma^{(eff)} > Y, \tag{9}$$

where

$$S_{ij} = \sigma_{ij} - \delta_{ij}\frac{1}{3}Tr(\sigma_{ij}), \tag{10}$$

and the other variables are explained in the list of nomenclature.

2.2 THE ISSUE OF COUPLING

Obviously, the fluid flow field affects the thermal energy field and vice versa. The relationship between the fluid flow/heat transfer and solid-mechanics phenomena is more subtle. The influence of the temperature from the energy calculations on the solid-mechanics calculations is straightforward enough. However, in most casting applications, the development of stress and relative deformation in the solidifying metal and the mold gives rise to a gap at the metal-mold interface, which causes an inhibition of heat transfer across it. The heat transfer across the metal mold is adequately governed by a lumped expression,

$$\frac{\partial T}{\partial n} = \pm h_{eff}\left(T_{metal} - T_{mold}\right), \tag{11}$$

where the heat transfer coefficient is defined by

$$h_{eff} = \frac{K_{gap}}{\Delta_{gap}}. \tag{12}$$

The parameter, Δ_{gap}, is the normal distance between the metal and the mold at the interface and is calculated from the solid-mechanics equations. As such, the flow, heat transfer, and solid-mechanics behavior are closely coupled and have to be solved in a manner that reflects the direct interactions among the phenomena.

The application of this computational modeling software to the analysis of shape-casting processes is described in detail elsewhere (Bailey et al., 1996).

2.3 DISCRETIZATION USING A FINITE-VOLUME UNSTRUCTURED MESH APPROACH

In the approach described, all the above equations are solved on a single mesh. However, the flow and heat transfer equations are solved using a cell-centered discretization, while the solid-mechanics equations are solved using a vertex-centered discretization; this distinction is illustrated in Figure 1.

Cell-Centered Scheme

The general form of the mass, momentum, and energy equations may be written as

$$\frac{\partial}{\partial t}(\rho\phi) + \nabla(\rho v\phi) = \nabla\left(\Gamma_\phi \nabla\phi\right) + S_\phi, \tag{13}$$

which may be transformed using the divergence theorem to give

$$\int_v \frac{\partial}{\partial t}(\rho\phi)dt + \int_s \rho v\phi ds = \int_s \Gamma_\phi \frac{\partial\phi}{\partial n} ds + \int_v S_\phi dv. \tag{14}$$

Each of the above terms may be discretized as (Chow, Cross, and Pericleous, 1995) the following:

$$\int_v \frac{\partial}{\partial t}(\rho\phi)dt = \frac{\left[(\rho\phi)_p - (\rho\phi)_p^{old}\right]V_p}{\Delta t},$$

$$\int_s \rho v\phi ds = \sum_{A=1}^{n}(\rho\phi(u\Delta y - v\Delta x)_A,$$

$$\int_s \Gamma_\phi \frac{\partial\phi}{\partial n} ds = \sum_{N=1}^{n}\left(\phi_N - \phi_p\right)\left(\Gamma_\phi\left(\frac{\Delta y}{\delta x} - \frac{\Delta x}{\delta y}\right)\hat{n}\right)_N, \tag{15}$$

$$\int_v S\phi dV = S\phi V_p$$

over the cell-centered control volume. The fluid flow and heat transfer equations are then solved by the standard SIMPLE procedure (Patankar, 1980; Versteg and Malalasekera, 1995), using iterative solvers for each of the key variables—$\underline{u}$, h, and the pressure correction term. As is current practice, because the pressure (correction) and velocity components are colocated at the cell center, the Rhie-Chow approximation is used to prevent checkerboarding of the pressure field (Rhie and Chow, 1982).

Cell Vertex Scheme

The equilibrium equations are integrated over the control volume illustrated in Figure 1, where P is the node vertex, to give

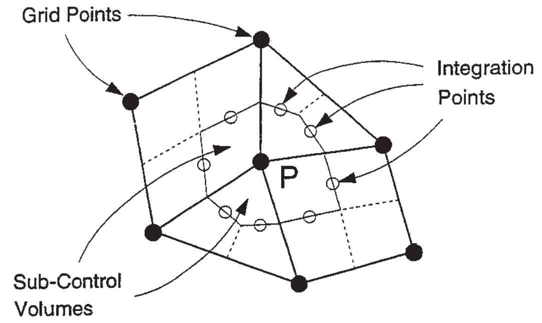

Fig. 1 Control volumes in unstructured meshes

$$\int_v \left(\frac{\partial \Delta\sigma_{xx}}{\partial x} + \frac{\partial \Delta\sigma_{xy}}{\partial y} \right) dv = \int_v \Delta F_x dv, \text{ etc.}, \tag{16}$$

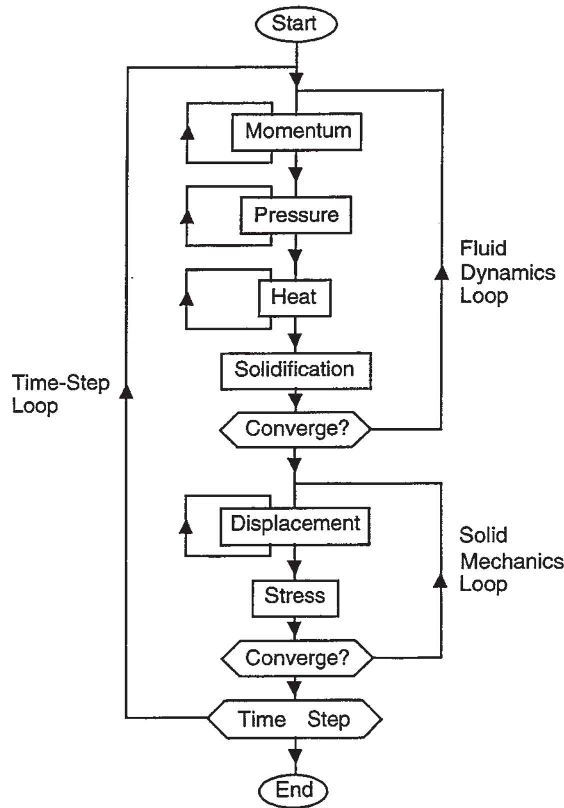

Fig. 2 Flowchart for UIFS

"Many factors have influenced the selection of discretization. For example, vertex-based boundary conditions are most convenient for vertex-based discretization procedures for solid mechanics. Conversely, for flow calculations, flux boundary conditions are straightforward to apply for a cell-centered discretization."

which converts to the following discretized form:

$$\sum_{A=1}^{n} \left(\Delta\sigma_{xx} N_x - \Delta\sigma_{xy} N_y \right) = \Delta Fx \Delta V,$$

$$\sum_{A=1}^{n} \left(\Delta\sigma_{xy} N_x + \Delta\sigma_{yy} N_y \right) = \Delta Fy \Delta V, \tag{17}$$

where the summation is over the number of faces that constitute the control volume. Substituting equation (17) into (16) gives a first-order equation for the solution of the displacements. In the x-direction, this yields

$$\begin{aligned} \sum_{A=1}^{n} \frac{\partial \Delta u}{\partial x} \left\{ \frac{N_x}{(1-\mu)} + \frac{N_y}{2} \right\} &= \frac{(1+\mu)}{E} \Delta F_x \Delta V \\ &- \sum_{A=1}^{n} \frac{\partial \Delta V}{\partial y} \left\{ \frac{N_x}{(1-\mu)} + \frac{N_y}{2} \right\}, \\ &+ \frac{1}{1-\mu} \sum_{A=1}^{n} \left\{ \begin{matrix} N_x \left[\left(\Delta\varepsilon_{xx}^{(VP)} + \mu \Delta\varepsilon_{yy}^{(VP)} \right) + (1+\mu)^2 \alpha \Delta T \right] \\ + N_{y.} \dfrac{(1-\mu)}{2} \Delta\varepsilon_{xy}^{(VP)} \end{matrix} \right\}, \end{aligned} \tag{18}$$

with a similar expression for the Δv.

Many factors have influenced the selection of discretization. For example, vertex-based boundary conditions are most convenient for vertex-based discretization procedures for solid mechanics. Conversely, for flow calculations, flux boundary conditions are straightforward to apply for a cell-centered discretization.

2.4 INTEGRATION IN THE UIFS CODE

The integration of the fluid, heat transfer, solidification, and solid-mechanics procedures are illustrated in Figure 2. The flow/heat transfer/solidification group of processes is solved as one implicit procedure using a derivative of the well-established SIMPLE procedures (Patankar, 1980; Versteg and Malalasekera, 1995). The SIMPLE strategy solves the momentum variables (u, v), the pressure (p), heat transfer (h, T), and phase change (f) in a segregated manner, although, of course, each variable is solved iteratively within the overall loop. When this has achieved convergence at a time step, it passes onto the solid-mechanics loop. Here, the whole problem for (Δu, Δv) is assembled and solved as a single matrix using a preconditioned conjugate-gradient (PCCG) solver. In principle, it is quite possible to make the whole procedure fully im-

plicit essentially, using the idea of false time stepping. In practice, this never showed any advantage for the problems addressed with the UIFS code. Each of the solvers may be turned on or off to suit the requirements of a given problem. As the fluid mechanics stage often requires more effort to obtain a satisfactory solution than solid mechanics, the solid-mechanics solver loop may be masked to only run every kth time step. Even with $k = 1$, the bulk of the computational effort is usuall
y expended in the fluid-mechanics loop. This is, of course, problem dependent; for a solidification-type problem, the initial time steps may be entirely fluid while the closing time steps are entirely solid.

The above strategy has enabled the evaluation of a range of levels of numerical coupling within the software framework. However, certainly in metals-processing applications, the straightforward coupling (i.e., not fully implicit) has proved more than adequate. The features that have made this possible are the numerical fidelity inherent in the single code, which has one database on a single mesh. This code has been used to evaluate the coupled strategy in a number of solidification-based processes—particularly related to shape casting (Bailey et al., 1996). In Figure 3, we show plots of the residual flow at an early phase, the temperature profile, and the effective stress and deformation at a later stage of the process. The code, though only two-dimensional, adequately demonstrates the potential for solving genuine multiphysics problems (Bailey et al., 1996).

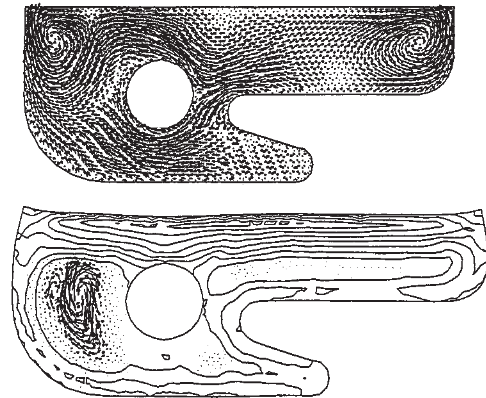

Fig. 3 Flow vectors for the fluid dynamics test case and residual stress contours with flow vectors for solidification test case

3 Strategy for Parallelization

As stated earlier, the parallelization strategy is based on the now standard SPMD paradigm. However, its implementation in a multiphysics code with a range of possible discretization strategies on a single mesh presents a series of challenges, which will become apparent below.

3.1 THE BASIC STRATEGY

When dealing with multiphysics simulations, there are a range of issues to address that affect the strategy for mesh partitioning. These include (Cross, 1996) the following:

- the relative computational effort of each physical phenomenon in each element;
- the presence or absence of a specific physics phenomenon in an element;
- the algorithmic structure, which determines whether each phenomenon can be solved simultaneously.

It is tempting to develop a partition strategy for multiphysics that performs separate partitions for each distinct physics procedure or module independently. However, if an element has a number of phenomena active simultaneously, this could involve a substantial amount of additional interprocessor communication, unless the solution of each variable is coincidentally on the same processor.

The real challenge in parallel multiphysics simulation is to devise a partitioning strategy that ensures the following:

- all the distinct physics operating in an element is essentially located on the same processor,
- the partition of the single mesh reflects the weights associated with the physics active at each element and load balances accordingly (as a weighted graph).

The approach adopted here was first suggested by McManus, Cross, and Johnson (1995) and may be summarized as the following:

- The code is restructured on the basis of partitioned single mesh for the whole solution domain as a single entity (i.e., without regard to the physics).
- The partitioning strategy (i.e., tool) recognizes the different weights depending on the physics active in that element and constructs partitioned subdomains (which may be disjoint) that ensure a computational load balance.

Actually, in the early stages of this work, there were no mesh-partitioning tools with this capability. Now there are at least two—METIS (Karypis and Kumar, 1998) and JOSTLE (Walshaw, Cross, and Everett, 1995); the latter is used in this work (see below). Even though a single mesh is used in the analysis, the procedures for each physics component do not necessarily use the same discretization process. In fact, the flow, heat transfer, and solidification procedures are solved at the cell center where the element is the control volume, while the solid-mechanics procedures are solved at the vertex, and the control volume is assembled from components of the neighboring elements (see Figure 1). Hence, to ensure that the above strategy can be implemented, it is necessary to construct secondary partitions (for the vertex-based graph from the element-based graph) that maximize the locality of element and associated vertex discretizations on one processor.

The key advantage of the above strategy is twofold:

1. The approach to parallelizing the multiphysics code is essentially similar to that for a single discipline code (such as CFD) with the added complexity of the secondary partitioning issue.
2. The problem of ensuring a partition that yields a load balance for any kind of multiphysics problem is essentially addressed by the distinct partitioning/load-balancing tool.

3.2 MESH PARTITIONING

A key to success is the quality of the mesh partitioner; what is required here is

- a partitioned mesh that is well load balanced (i.e., the same workload on each processor),
- one that is well balanced with regard to interprocessor communications (i.e., the data exchanged between processors is minimized with respect to both volume and distance over the processor network), and
- one that runs very rapidly with respect to the parallel multiphysics simulation so that partitioning is a nominal overhead cost.

There is a vast literature on mesh partitioning and dynamic load balancing in the engineering analysis context. It is not the objective of this paper to review this area but simply to identify the approach taken to address the challenge of multiphysics. However, an excellent review of mesh partitioning and dynamic load balancing has recently appeared in Hendrickson and Devine (forthcoming), who raise some of the issues discussed above and highlight various possible approaches. The tool used in the mesh-partitioning task in this work is JOSTLE, also developed at Greenwich by Walshaw, Cross, and Everett (1995). This tool is a state-of-the-art partitioner. It employs a two-phase procedure and optimizes at various levels to rapidly produce high-quality partitions using O(N) process. In particular, the latest version of JOSTLE

- runs in parallel,
- load balances with respect to computational load and interprocessor communication,
- exploits the use of disconnected subdomains to achieve a load balance,
- enables dynamic load balancing.

In this context, it is necessary to know that a graph is built from the physical domain mesh that describes the el-

ement connectivity. The JOSTLE tool then uses the dual of this graph as the basis for its partitioning process. It can produce partitions for an arbitrary number of processors and also match the partition connectivity of the resulting subdomains to specified processor topologies. We will return to this later.

3.3 THE ISSUES OF MESH DECOMPOSITION

Having obtained a partition of the mesh into P parts, the partition is used to decompose the mesh into P subdomains that can be allocated one per processor. The elements, nodes, and faces that are allocated uniquely to a processor are referred to in this paper as the core mesh components. These components are said to be "owned" by a processor. Each subdomain is extended with a layer of points, faces, and elements that overlaps the subdomains along the interprocessor boundaries, as illustrated in Figure 4. These overlap or halo regions carry variable values from neighboring subdomains that are required for the solution of variables inside the subdomain.

Decomposition of the mesh into a set of extended submeshes consists of five essential steps:

1. Find a partition of the mesh (primary).
2. Derive secondary partitions from the primary partition.
3. Determine the mesh overlaps to the neighboring subdomains.
4. Renumber the mesh in each subdomain.
5. Construct templates for overlap data exchange.

This strategy is an extension of what needs to be done for a primary-only partition and involves some careful housekeeping.

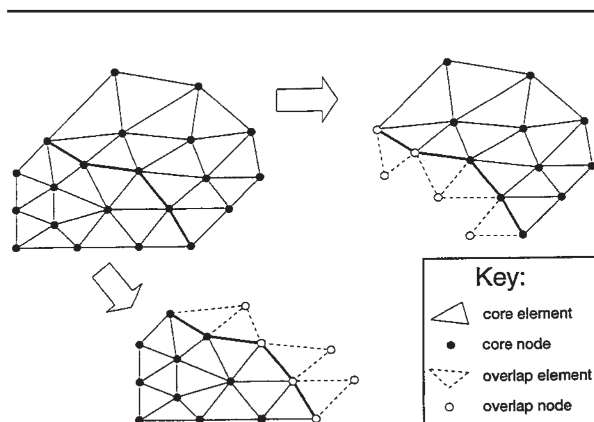

Fig. 4 A mesh of 28 triangles divided into two subdomains with the overlaps required for the flow scheme

"The mesh entity that provides the dominant spatial reference used by the code to be parallelized is ordinarily chosen as a basis for mesh partitioning."

3.4 DERIVING SECONDARY PARTITIONS

The mesh entity that provides the dominant spatial reference used by the code to be parallelized is ordinarily chosen as a basis for mesh partitioning. This partition is referred to as the primary partition. Secondary partitions may be derived from the primary partition for the other mesh entities used in the code. The compute time for a CM code is largely dominated by the time spent in the solution of an equation of the form $A\underline{x} = \underline{b}$. It is consequently important for load balance to obtain an equal number of rows and an equal number of coefficients in each of the distributed A matrices. This inevitably results in some compromise. With an element-based unknown vector, for exam-

ple, a primary partition based on elements will keep the vector length and hence the number of rows in the A matrix balanced across each subdomain. But the number of off-diagonal coefficients in each A matrix depends on the number of internal faces in the subdomain. Balancing elements will not necessarily balance matrix coefficients. In the case of the two-dimensional flow procedure in UIFS, the primary partition is based on elements, and there is only one secondary partition, that being for grid points. For reasons of clarity, the following discussion is based on an element-based primary partition. The discussion is nonetheless applicable to other mesh entity partitioning orders.

Secondary partitions are inherited from the primary partition in accordance with the connectivity between the entities. For example, each node is connected to a number of elements, each of which belongs exclusively to one subdomain. This provides a basis for the allocation of nodes to a subdomain. The most obvious and simple partition inheritance scheme is to allocate the node to the subdomain that owns the majority of the connected elements. In the case of an equal number of connected elements being owned by two or more subdomains, the node is allocated to the domain that owns the least number of nodes. This simple, inexpensive scheme gives a good alignment between the primary and secondary partitions but can lead to a high load imbalance in the secondary partition. It does not follow that two unstructured meshes with equal numbers of elements will have the same number of nodes; indeed, there may be a large discrepancy between the two node counts. When the two meshes are subdomains to be operated on in parallel, this can produce an unacceptably high degree of load imbalance for element-based matrix computations, as discussed earlier, and possibly even greater imbalance for node-based calculations. If, however, the node allocation between the subdomains is forced to be balanced, the element and node partition may not be well aligned, which can result in an undesirably large and imbalanced overlap layer. This will consequently lead to large and unbalanced communications between the subdomains.

Load imbalance may be redressed to an extent through the use of more elaborate schemes. In this work, all nodes are allocated to the subdomain with the highest number of elements connected to the node. Nodes with an equal number of connected elements in each connected subdomain are not allocated until all other nodes have been allocated, at which point they are assigned in sequence to the least loaded subdomain.

It is conceivable that the nodal imbalance may become unmanageably large, in which case some nodes may require allocating to subdomains that own none of the connected elements to redress the balance. The resulting communication imbalance may or may not be significant depending on the characteristics of the hardware platform. The quality of the secondary partitions then becomes a platform-dependent optimization issue.

These schemes may be seen as an attempt at solving a graph problem by the application of simple heuristics. It may therefore be worthwhile to use graph-based techniques to derive the secondary partitions. One possible scheme is to produce a weighted graph of the nodes that clusters the nodes for which all connected elements lie on one partition. This graph can then be partitioned using one of the graph-partitioning algorithms developed for obtaining the primary partition. The amount of effort that is worthwhile devoting to the derivation of a secondary partition is problem dependent. Like the search for a primary partition, there may be no singular optimal solution, and a near-optimal solution will, in the majority of cases, provide a sufficiently good solution.

3.5 HALO LAYER/OVERLAP CONSTRUCTION

The overlaps between the subdomains are determined in accordance with the data dependency required by the code. For example, if the solution for an element-based variable requires the values in all adjacent elements, then the adjacent elements that lie in neighboring subdomains are added as overlaps to the list of elements. Similarly, if the nodes that compose the overlap elements are also required, then they too are added to the list of overlap node. In this way, the description of the mesh for each subdomain is extended to include all data that are required for the solution of the subdomain. The utility used to construct overlaps for the codes discussed in this paper employs a simple set of rules to determine the elements and nodes that are to be included in the overlaps.

When using only the element flow and heat code:

Overlap elements are defined as all elements that are adjacent to a core element.

Overlap nodes are defined as nodes of all elements, including overlaps that are not core nodes.

However, the node-based stress code involves a more extensive data dependency, and the required overlap layers become deeper so that:

Additional overlap elements are defined as elements that contain at least one core node.

Additional overlap nodes are defined as nodes that are connected to core nodes.

An example of the overlaps required for the flow code is shown in Figure 4. The same mesh is shown in Figure 5 with the additional elements and nodes in the overlaps required for the stress code.

Given that the mesh data structure is either one-dimensional linked or indexed lists or stored as multidimensional arrays, then the number of entities is the highest index (last in F77, first in C), and the overlaps may be stored as extensions to existing data structures. This allows them to be passed to subroutines and addressed in the parallel code in the same manner as the original data structures. This hides the parallelism and results in only small source code changes being required to extend the mesh as it is implemented in the serial code. For example, the array of grid points in the Fortran code of UIFS may be declared as the following:

```
INTEGER DIMENS, NO_OF_GRID_POINTS
REAL GRID_POINTS(1:DIMENS, 1:NO_OF_
   GRID_POINTS)
```

This array may be easily extended to include overlaps such as the following:

```
INTEGER DIMENS, EXTD_NO_OF_GRID_POINTS
REAL GRID_POINTS(1:DIMENS, 1:EXTD_NO_
   OF_GRID_POINTS)
```

Clearly, this structure will still be correctly declared in all subsequent subroutines calls without any code modification. Subroutines may be called with either the original or the extended point count, and the declaration will remain consistent. If, however, the array of grid points is declared as

```
REAL GRID_POINTS(1:NO_OF_GRID_POINTS,
   1:DIMENS)
```

then the array may also be extended as

```
REAL GRID_POINTS(1:EXTD_NO_OF_GRID_
   POINTS, 1:DIMENS)
```

But now each subroutine must declare grid points to the extended size to remain consistent. It may prove less invasive to change the serial code to reverse such declarations and subsequently all occurrences of the variable. Apart

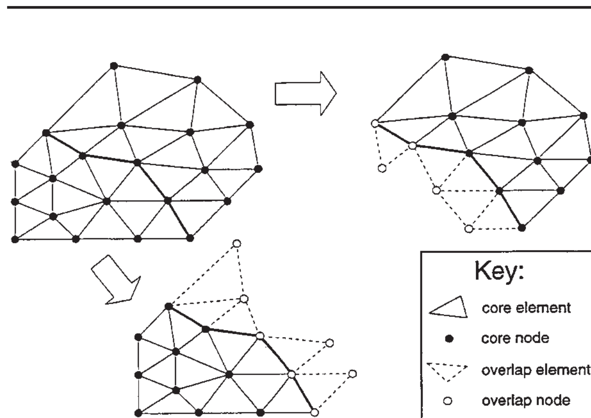

Fig. 5 A mesh of 28 triangles divided into two subdomains with the overlaps required for the stress scheme

from cache effects, such a modification will not affect the serial code and is unlikely to raise objections from the serial code authors.

3.6 PARALLEL EXECUTION CONTROL AND DATA LOCALIZATION

Consider the following code fragment that loops over each grid point in each element:

```
INTEGER NUMBER_OF_GP_IN_ELEMENT(1:NUMBER_
   OF_ELEMENTS)
INTEGER GP_IN_ELEMENT(1:MAX_NUM_GP_PER_
   ELE,1:NUMBER_OF_ ELEMENTS)
REAL    XELE(1:NUMBER_OF_ELEMENTS)
REAL    YGP(1:NUMBER_OF_GRID_POINTS)
DO I=1, NUMBER_OF_ELEMENT
     DO J=1,   NUMBER_OF_GP_IN_ELEMENT(I)
         XELE(I) = XELE(I) + YGP(GP_IN_
             ELEMENT(J,I)
     END DO
END DO
```

Two arrays are used in this example to describe the element topology: `NUMBER_OF_GP_ELEMENT` is a vector that contains the number of grid points that are in each element. `GP_IN_ELEMENT` is a two-dimensional array that contains the grid point number for each grid point in each element. Two data items are involved: an element- based variable `XELE` and a grid point-based variable `YGP`.

This code fragment can be implemented in parallel by using "owner computes" execution control masks that are conditionals to control the scope of operations for each processor. To achieve scalability in both computation performance and memory, a processor needs to both act on and only store its own data, plus copies of other processors' data in the overlap areas. Hence, the fundamental mesh entity (the grid point described as a set of coordinates) will be locally renumbered through the simple process of being packed into memory as a consecutive list of coordinates for each grid point in the subdomain. So the core grid points are packed into the first 1 to `LOCAL_NUMBER_OF_GRID_POINTS` locations and the overlap grid points as `LOCAL_NUMBER_OF_GRID_POINTS+1` to `EXT_LOC_NUMBER_OF_GRID_POINTS`, where `LOCAL_NUMBER_OF_GRID_POINTS` is the number of grid points in the subdomain core, and `EXT_LOC_NUM_OF_GRID_POINTS` is the number of grid points in the entire subdomain. Similarly, extracting and storing (packing) only the local entries for the variables `XELE`, `YGP`, and `NUMBER_OF_GP_IN_ELEMENT` is straightforward. Moreover, through the exploitation of the local subdomain mesh information, the loop can localize the data usage so that the above code fragment is transformed to

```
INTEGER NUMBER_OF_GP_IN_ELEMENT(1:EXT_
   LOC_NUM_OF_ELEMENTS)
INTEGER
GP_IN_ELEMENT(1:MAX_NUM_GP_PER_ELE,1:
INTEGER PTR_ELE(1:NUMBER_OF_ELEMENTS)
INTEGER PTR_GP(1:NUMBER_OF_GRID_POINTS)
REAL      XELE(1:EXT_LOC_NUM_OF_ELEMENTS)
REAL      YGP(1:EXT_LOC_NUM_OF_GRID_
            POINTS)
DO I=1, LOCAL_NUMBER_OF_ELEMENTS
    DO J=1, NUMBER_OF_GP_IN_ELEMENTS(I)
            XELE(I) = XELE(I) + YGP(GP_
              IN_ELEMENT(J,I))
    END DO
END DO
```

If this code fragment exists inside a subroutine where `NUMBER_OF_ELEMENTS` is passed into the subroutine as an argument, then the calling routine can be modified to call the subroutine with `LOCAL_NUMBER_OF_ELEMENTS`, so that no code modification is required in the subroutine.

This paper follows the option of renumbering each entire subdomain to a local numbering scheme. Each processor "sees" its renumbered subdomain as a complete mesh consisting of 1 to n_e elements and 1 to n_p grid points, where n_e and n_p are the local number of elements and grid points, respectively. This can be carried out at the highest possible level in the code, that is, where the problem specification is read from file. A record of the global (serial) numbers for each local mesh entity (referred to as a local-to-global index) is stored on each processor to allow reconstruction of data back into the original global form. Translation from local to global numbering using this record is only required as an I/O process when writing variables to file (or dynamically load balancing). Rebuilding of global variables is carried out by the I/O (master) processor, and so this is where the local-to- global indices are required. However, the indices are distributed with the subdomains to maintain scalability of memory. This scheme can encounter difficulty when the problem size increases to the point at which the geometry description will no longer fit into the memory processor. This is not, however, insurmountable and is discussed further elsewhere (McManus, Cross, and Johnson, 1995). The effect of renumbering is illustrated in Figures 6 and 7. Consider the element partition in Figure 7. The partition list P_e of

processor numbers that own each element as returned from the partitioner is as follows:

1 1 1 1 1 12 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2

Renumbering the core elements in the first 14 elements of each partition allows transformation to local loop limits. The implications of renumbering are discussed further below.

3.7 OVERLAP COMMUNICATION

The notion of the mesh overlaps is that each processor calculates only the values of core variables. That is, variables associated with mesh entities within their own domain, with no computation being performed on the overlaps. Variable values are then copied into the overlap from the processors on which the variables are calculated, as shown in Figure 8. This is a one-way communication process between all adjacent subdomains. Data travels only from the core of the subdomains (where they are calculated) into the overlaps of adjacent subdomains (where they are used). However, there are some rather obvious exceptions in which data operations are so trivial that it is faster to perform the operation locally on the overlap than to import the new values from a neighbor. For example, setting a variable to a fixed value (e.g., zero) requires a processor only to write a register to memory. This will undoubtedly be faster than reading data from the communication port and writing the data back to memory. Implementation of such exceptions may be seen as an optimization of the parallelization. Indeed, such optimizations may produce an improvement in performance on some platforms and not others. Overlap values are generally exchanged between processors as soon as practically possible, usually whenever a variable has been fully updated (e.g., at each iteration of the solver). Asynchronous communication schemes may be used to improve the parallel performance by overlapping communication with calculation. This is discussed further below. The coordination of overlap data exchange requires a communication template for each subdomain that holds the mesh entity numbers to be sent and the processor number to which they are to be transmitted. A corresponding template records the entity numbers to be received and the processor number from which they will arrive. These templates must be matched across each subdomain boundary so that the data sent from one subdomain are received in the anticipated order in the ad-

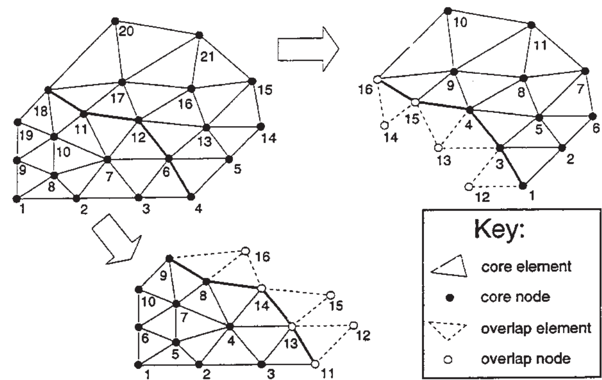

Fig. 6 A mesh of 28 triangles divided into two subdomains showing the renumbering of grid points from global to local numberin.

"The notion of the mesh overlaps is that each processor calculates only the values of core variables. That is, variables associated with mesh entities within their own domain, with no computation being performed on the overlaps."

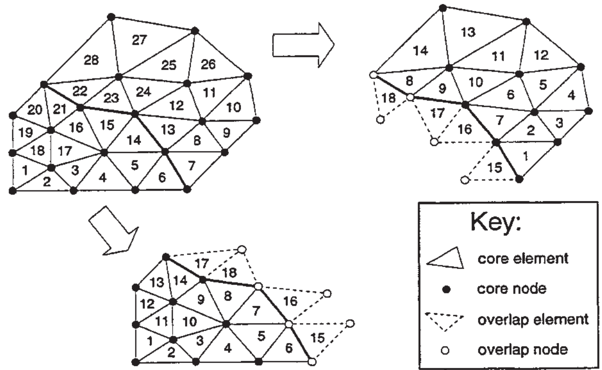

Fig. 7 A mesh of 28 triangles divided into two subdomains showing the renumbering of elements from global to local numbering

Table 1
Communication Operations Required for a Simple Chain of Processors

Processor Number	Odd	Even
	Send right	Receive left
	Receive right	Send left
	Send left	Receive right
	Receive left	Send right

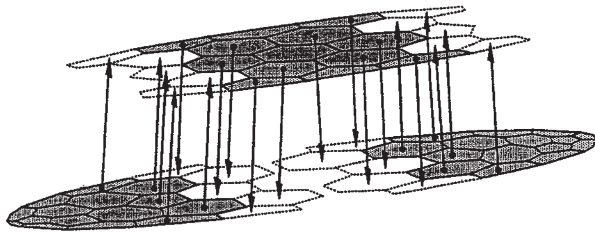

Fig. 8 Overlap update communication scheme

jacent subdomain. This is relatively simple as the decomposition preserves the global ordering of the mesh entities. For a simple processor interconnection topology such as a pipeline (a one-dimensional chain), in which the partition can guarantee mapping to the processor topology, the template becomes reasonably straightforward. Exchange of data between processors can be synchronized by the template on an odd-even alternate-pair basis. This is a four-cycle process described in Table 1.

This simple scheme enables the exchange to be carried out as a parallel process. More elaborate processor topologies can be handled with variations on such a scheme. Regular two-dimensional processor arrays, for instance, can use red-black checkerboard-type schemes. It cannot, however, be assumed that the mesh can be partitioned in such a way as to map perfectly onto the processor interconnection topology. A scheme is required that can cope efficiently with an unstructured partition of an unstructured mesh mapped imperfectly to an array of processors. This is a scheduling problem of the type familiar to operational research.

The scheme adopted involves constructing the graph $G(P, C)$ of processors P and subdomain (processor) interconnections C and weighting the interconnects according to the size of the interface. This graph is initially sorted by weight, with the processor pair having the largest amount of data to communicate being first. The graph is then scheduled to provide a sequence in which exchanges occur as a parallel process with the largest exchanges first. Starting with the heaviest node pair, the processor numbers are recorded. The graph is then searched for the next heaviest weight that does not use one of the already recorded processors. When found, this processor pair is sorted to be the next entry in the graph. This operation is carried out until either all processors are involved in communication or an unrecorded pair is no longer available for scheduling. If there are still entries in the graph that have not been scheduled, the list of recorded processors is cleared and the process repeated until all processor pairs have been scheduled. This results in a layering of exchange communication processes that should be (but is not guaranteed to be) no deeper than the maximum node degree of the processor graph $G(P, C)$.

Consider the simple mesh of 42 triangles illustrated in Figure 9, decomposed into three renumbered subdomains, as shown in Figure 10.

Here the overlap renumbering has followed the original global numbering scheme. Processor (a) must receive data for overlap elements 17 and 18 from processor (b), where they are numbered 6 and 9, respectively. Similarly, proces-

sor (b) must receive data for overlap elements 15 and 16 from processor (a), where they are numbered 3 and 8, respectively. The communications for this example may be carried out in three exchanges as follows:

Processor (a)
- 1 Send to processor (b) elements 3 and 8
- 1 Receive from processor (b) elements 17 and 18
- 2 Send to processor (c) elements 9 and 12
- 2 Receive from processor (c) elements 15 and 16

Processor (b)
- 1 Receive from processor (a) elements 15 and 16
- 1 Send to processor (a) elements 6 and 9
- 3 Send to processing (c) elements 5, 7, 10, and 13
- 3 Receive from processor (c) elements 17, 18, 19, and 20

Processor (c)
- 2 Receive from processor (a) elements 15 and 19
- 2 Send to processor (a) elements 5 and 6
- 3 Receive from processor (b) elements 16, 17, 18, and 20
- 3 Send to processor (b) elements 6, 8, 10, and 14

Note that these element numbers are always in increasing order both globally and locally. The send is carried out first to allow parallelism in packing.

Data that are to be transmitted from a subdomain core are collected into a data buffer, which allows one transmission and therefore only one latency to complete the transfer. Unpacking of data from a buffer is an overhead that is not necessary for data reception. Arranging for the overlap-renumbering scheme to consecutively number overlap entities that are owned by the same processor allows incoming data to be received directly into the overlap memory. So the global number ordering is preserved for each interface to other processors but not throughout the overlap. In the above example, elements 15 and 19 on processor (c) are in the core on processor (a) and so should be numbered consecutively. This involves renumbering overlap element 19 on processor (c) to be 16 and then overlap elements 16, 17, 18, and 20 to be 17, 18, 19, and 20, respectively.

3.8 MATRIX DECOMPOSITION AND PARALLEL SOLVERS

UIFS, in common with most other CM codes, requires the solution of a number of linear systems of the form $A\underline{x} = \underline{b}$. The mesh partitioning, as well as the renumbering strat-

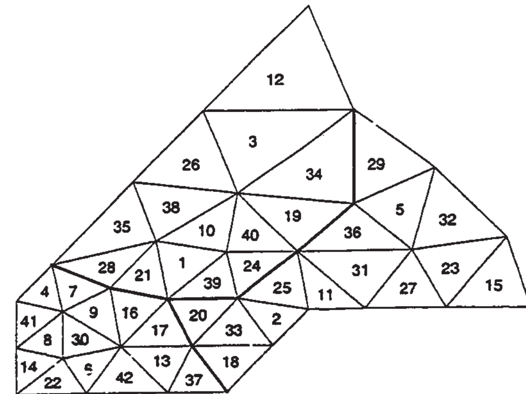

Fig. 9 Mesh of 42 triangular elements

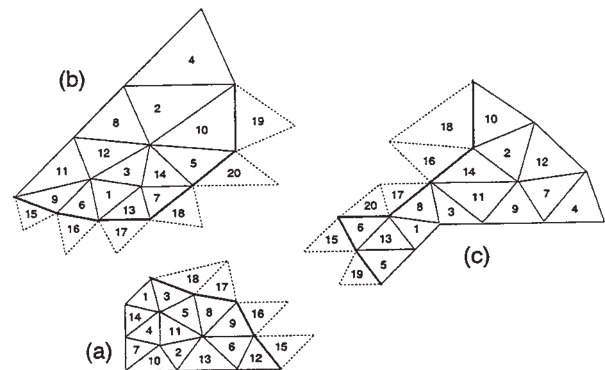

Fig. 10 Mesh of 42 triangular elements partitioned into three renumbered domains

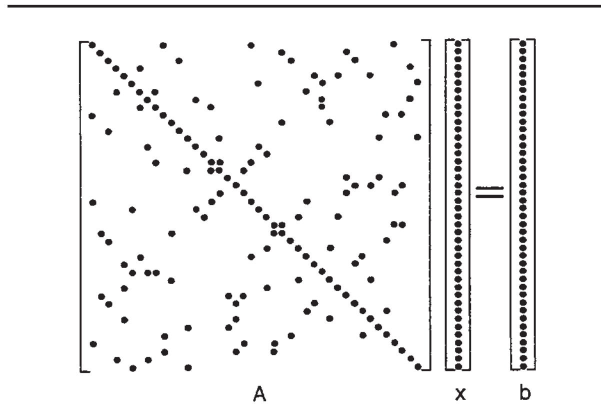

Fig. 11 Matrix for the 42-triangle mesh

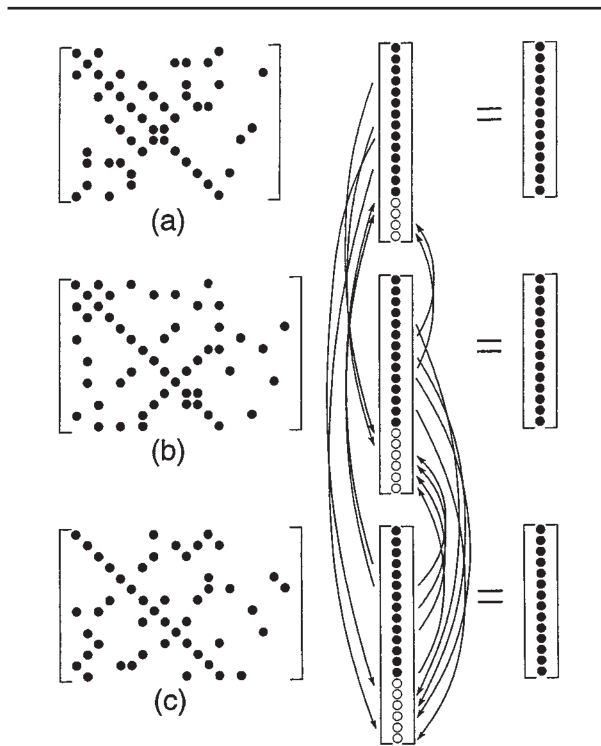

Fig. 12 Matrix for the 42-triangle mesh partitioned into three subdomains

egy, obviously has an impact on how the submatrices are represented. The first-order element matrix for the mesh in Figure 9 is shown in Figure 11. Decomposition of this mesh into the three subdomains in Figure 10 gives rise to the three subdomain matrices in Figure 12. This illustrates how the matrices resulting from the decomposition are not square. Rows of the matrices that correspond to the interprocessor (subdomain) boundaries contain coefficients that address elements in the overlap layer. Figure 12 also shows the data transport from the core of each $\underline{x}$ vector into the overlap region of x on the neighboring processor.

The UIFS code uses three iterative solvers, and their parallel form is summarized below.

Parallel Jacobi, Gauss-Seidel, and SOR Algorithms

Obviously, all of these algorithms attempt to find a solution of $A\underline{x} = \underline{b}$ by generating $x^{(k+1)}$ from components of $\underline{x}^{(k)}$ for $k \geq 0$ in an iterative fashion. The scalar form of these algorithms can be expressed as

$$x_i^{(k+1)} = \alpha\left(\frac{\sum_{j=1, j\neq i}^{n}\left(-a_{ij}x_j^{(k+s)}\right) + b_i}{a_{ii}}\right) + (1-\alpha)x_i^{(k)} \quad (19)$$

$$i = 1, \ldots, n$$

where $s = 0$, $\alpha = 1$ for the Jacobi scheme, $s = 1$ for the others, and generally, $\alpha < 2$.

Parallelizing the Jacobi algorithm is trivial. Much work also has been done to develop parallel versions of Gauss-Seidel procedures. However, in CFD calculations, the GS-SOR algorithms are used in the calculation of the velocity components, and there is a fair amount of flexibility here. In the SIMPLE-type procedures (Patankar, 1980; Versteg and Malalasekera, 1995), the computational effort is dominated by the pressure correction calculation, and only a relatively few solver GS-SOR cycles are required for the velocity components each time through the full iterative cycle. Hence, the above algorithm can be implemented in parallel as

$$x_i^{(k+1)} = \alpha\left(\frac{\sum_{j=1}^{i-1}\left(-\alpha_{ij}x_j^{(k+s)}\right) + \sum_{j=i+1}^{m_p}\left(-\alpha_{ij}x_j^{(k)}\right) + b_i}{a_{ii}}\right), \quad (20)$$
$$+ (1-\alpha)x_i^{(k)}$$

where $i = 1, \ldots, n_p$ and $p = 1, \ldots, P$; n_p is the number of elements (coefficients) in subdomain p; P is the number of

processors; and m_p is the number of elements (coefficients), including the overlap/halo elements in sub- domain p.

Obviously, even though there is a renumbering of the mesh within a partitioned subdomain, the update will require the latest values of $\underline{x}$ located in the overlap/halo layers (and so calculated on other processors). However, using the above formula, which involves using some old values in overlap/halo layers, appears to have little practical effect on the convergence behavior of the overall flow, heat, and solid-mechanics procedures (and so for the full multiphysics loop). Formally, results in McManus's (1996) thesis show that variations in the values of serial and parallel variables and differences in the number of iterations required to converge are both insignificant. In practical terms, the variations between the serial and parallel results are significantly less than the variations caused by running the serial code on different processors (Sparc, i860, MIPS, etc.). Even with processors using IEEE arithmetic, differences in rounding modes give rise to subtle variations in numerical behavior.

Parallel PCCG Algorithm

CGM is an established nonstationary iterative method for symmetric positive-definite systems providing rapid convergence and $0(m)$ computational efficiency, where m is the number of nonzero components of A. Preconditioning is frequently used to improve the condition number of the matrix A (Golub and Van Loan, 1989). For a positive-definite matrix preconditioner M,

$$A\underline{x} = \underline{b} \equiv M^{-1} A\underline{x} = M^{-1}\underline{b}. \tag{21}$$

If the eigenvalues of $M^{-1}A$ are clustered better than the eigenvalues of A, then the preconditioned problem may converge in fewer iterations than the original problem. The Jacobi preconditioner is the diagonal of the A matrix that has the effect of scaling the quadratic form along the coordinate axes. While not the most effective preconditioner in reducing iterations, its simplicity provides computational efficiency. A modification that transforms $Ax = b$ into $\breve{A}\breve{x} = \breve{b}$ where the components of $\breve{A}, \breve{x}$ are $\breve{b}$

$$\breve{a}_{ij} = \frac{a_{ij}}{\sqrt{a_{ii}a_{jj}}}, \; \breve{x}_i = x_i\sqrt{a_{ii}}, \; \breve{b}_i = \frac{b_i}{\sqrt{a_{ii}}} \tag{22}$$

results in the diagonal of $\breve{A}$ being the identity matrix I and hence the Jacobi preconditioner $M = I$. The CGM iteration can then be expressed as

$$\underline{u}^{(k)} = A\underline{p}^{(k-1)}, \tag{23}$$

$$\alpha^{(k)} = \frac{\rho^{(k-1)}}{\underline{p}^{(k-1)T}\underline{u}^{(k)}}, \tag{24}$$

$$\underline{x}^{(k)} = \underline{x}^{(k-1)} + \alpha^{(k)}\underline{p}^{(k-1)}, \tag{25}$$

$$\underline{r}^{(k)} = \underline{r}^{(k-1)} - \alpha^{(k)}\underline{u}^{(k)}, \tag{26}$$

$$\rho^{(k)} = \underline{r}^{(k)T}\underline{r}^{(k)}, \tag{27}$$

$$\beta^{(k)} = \frac{\rho^{(k)}}{\rho^{(k-1)}}, \tag{28}$$

$$\underline{p}^{(k)} = \underline{r}^{(k)} + \beta^{(k)}\underline{p}^{(k-1)}. \tag{29}$$

This method involves three basic computational processes: matrix-vector product, vector inner product, and AXPY ($a\underline{x}$ plus $\underline{y}$).

Recall that each distributed A matrix is no longer square as it now addresses coefficients in the overlaps. So an overlap exchange communication is required to obtain the values of $\underline{p}$ in the overlaps before evaluating the matrix vector product $A\underline{p}$ in equation (23).

The inner products in equations (24) and (27) are calculated in parallel as a sum of local partial inner products. Equation (27), for example, is evaluated as

$$\rho = \sum_{p=1}^{p=P} \sum_{j=1}^{j=n_p} r_{pj}^2. \tag{30}$$

This requires a global summation and hence synchronization across all processors.

The AXPY in equations (25), (26), and (29) is an ideally parallel process requiring no interprocessor communication.

The diagonally preconditioned conjugate-gradient (DPCG) algorithm, along with most other preconditioning schemes, is explicit in that it uses only old variable values within each iteration. It may therefore be expected to give identical results from both serial and parallel versions. In practice, however, finite numerical precision gives rise to variations of the inner products with P. As the solutions are highly sensitive to α and β, these small variations lead to quite noticeable numeric differences between the serial and parallel solution. In this case, it could be argued that both serial and parallel solutions are equally valid solutions to the original problem. Clearly, all solutions should be quantitatively identical.

3.9 PARALLEL UTILITIES: SCALABILITY ISSUES

Parallel utilities developed for this work build on the established CAPLib communication library that provides a useful portability layer to a range of transports. Together with JOSTLE, utility routines to partition, decompose, and recompose the mesh have been developed.

The key communication utility, `OVERLAPX VARIABLE, SPATIAL_REFERENCE)`, performs an exchange of overlap data for the input `VARIABLE` between all processors in accordance with the communication schedule defined by the argument `SPATIAL_REFERENCE` (i.e., element or grid point). Overlap exchange is a highly parallel process that involves a matching send-and-receive operation across all subdomain boundaries (Section 3.7). The time required for OVERLAPX is approximated as $2s_{max}t_m$, where s_{max} is the maximum node degree in the processor (subdomain) communication graph $G(P, C)$, and t_m is the average time to send a message. The important point here is that the number of processors P does not feature highly in this approximation, and so OVERLAPX scales well, the time required being largely independent of P.

"Global commutative operations (PVM reduce, MPI collective) are used to obtain global values of commutative functions by combining local partial evaluations of the function and broadcasting the results to each processor."

Global commutative operations (PVM reduce, MPI collective) are used to obtain global values of commutative functions by combining local partial evaluations of the function and broadcasting the results to each processor. The time required for a global commutative operation is dependent on the actual implementation of the operation, which can vary with partition strategy, communication harness, and platform hardware. For example, a global commutative may be implemented on a chain of processors by passing all partial evaluations to the master processor, where the global value can be evaluated and broadcast to all processors. The time required for this operation will consequently be something like $2(P - 1)t_1$, where t_1 is the communication start-up time (latency). With a mesh of $p \times q$ processors, a similar strategy will require $2(p + q - 2)t_1$. No matter how a global operation is implemented, the time required increases with P and so does not scale particularly well. Care is therefore required in avoiding as far as possible such operations. Some global commutative strategies do not ensure that an identical result reaches all processors. It must be remembered that floating-point operations have finite precision, and so floating-point arithmetic commutative operations are not truly commutative. So, for example, a floating-point global summation operation based on a ring of processors that accumulates partial summations by passing the partial results around the ring of processors will complete in (P –

$1)t_1$, but the values left on each processor will have different rounding errors. This can cause severe problems. If, for instance, the result is tested to determine convergence, some processors may test true and others false, and the code will consequently fail. Execution of a global summation in parallel must produce a different result to the serial summation, but both results are valid. It is only required that a global commutative produces an identical result across all processors, not an identical result to the serial commutative operation. Global commutative operations are now implemented as a binary tree or hypercube communication that returns a numerically identical result on each processor. This requires only $1 + [\log_2 (P - 1)]$ latencies and consequently scales to large P.

A scatter routine is used to distribute a variable across the processors, again in accordance with the given spatial reference. Similarly, a gather is used to rebuild variables from components on each processor. Scatter/gather operations are costly of both time and memory, requiring a number of messages proportional to P and globally dimensioned data space. The negative impact of these operations, however, is not particularly significant as they are only required for I/O operations.

4 Tools and Equipment Employed

In this paper, we will focus on the parallel performance of the UIFS code on a PARAMID-i860XP system because it has a number of parallel facilities that enable a range of issues to be assessed that could affect scalability. The PARAMID is essentially a distributed-memory parallel system with a flexible interconnection topology and a reasonable latency, bandwidth, and asynchronous functionality. As such, we will investigate the impact on performance of the following:

- the quality of the mesh partition,
- the extent to which the mesh partition connection topology matches that of the processors in the parallel system, and
- the impact of asynchronous communications.

With regard to scalability, our experiments have involved both relative speedup for a fixed problem size, as well as increasing the problem size in proportion to the number of processors available. However, our results have focused on the former because they are more discriminating in showing the effective limits of scalability for a particular problem size.

4.1 EXPLOITING THE JOSTLE TOOL TO EXPLORE THE EFFECT OF THE MESH PARTITION QUALITY

The JOSTLE strategy is to derive an initial partition as quickly and cheaply as possible and then use optimization techniques to improve the quality of the partition. Two alternative methods are provided to produce the initial partition. One method is a variation of the greedy algorithm—in this case, a graph-based variant on the original mesh-based algorithm proposed by Farhat (1988). The other method is geometric sorting, which operates in a similar manner to orthogonal coordinate bisection. This method provides a crude mapping to a $p \times q$ processor grid ($p \geq q$). The nodes, N, are sorted on the longest axis and split into sets of N/pq. The nodes in these sets are then sorted in the orthogonal axis and split into sets of N/pq. Having used one of the above methods to obtain an initial partition, one of two optimization methods can be applied to improve the partition. Uniform optimization is a technique in which each partition attempts to minimize its own surface energy analogous to the way that bubbles pack together. The technique works by calculating the center of each partition in a graphical sense and determining the radial distance of each node from the center. Nodes that are most distant from the center can then be migrated between neighboring partitions. Grid optimization is a similar technique to uniform optimization, except that nodes are allowed to migrate only between neighbors in the processor grid. Four partitioning (mapping) strategies are provided by JOSTLE. Unmapped partitioning ignores the processor interconnection topology throughout the entire partitioning process. A postmapped partition is an unmapped partition that has been mapped to the processor topology with a simple mapping algorithm applied after partitioning. The premapped partition begins with a partition that is crudely mapped to the process topology and then is optimized, ignoring the processor topology to minimize the number of cut edges. The mapped partition acknowledges the processor topology throughout the partitioning process. The mapping strategies are summarized in Table 2.

JOSTLE operates on a graph that, in the case of UIFS, represents the mesh and returns a partition of that graph. For parallel PUIFS, the dual graph of the mesh is used to obtain a partition based on elements. The dual graph is where the nodes or vertices of the graph represent the elements of the mesh, and the graph edges represent the element adjacency (connectivity). For the purposes of experimentation, JOSTLE can be run as a stand-alone program

Table 2
Partition Mapping Strategies Provided by JOSTLE

Strategy	Initial Partition	Optimization	Processor Allocation
Unmapped	Greedy	Uniform	No
Postmapped	Greedy	Uniform	Yes
Premapped	Geometric sort	Uniform	No
Mapped	Geometric sort	Grid	No

that produces a file describing the mesh partition. This allows for flexibility in adjusting the parameters used to control the partition and visualization of the partition produced. JOSTLE also has been embedded into PUIFS so that a partition may be produced rapidly at runtime. The partition produced by JOSTLE (primary partition) is used to generate a secondary partition for the mesh grid points. The primary and secondary partitions are inverted to generate lists of the global element and grid point numbers that exist in each subdomain. The rules for overlap generation given in Section 3.5 are applied to produce descriptions of the halo layers/overlaps in a global numbering scheme. The element and grid point lists are extended to contain the global element and grid point numbers for the overlaps. Boundaries in UIFS are described as a set of grid points, and boundary conditions are described in a file as a set of "patches." This allows the boundary points along with the associated boundary patch numbers to be partitioned in accordance with the extended grid point partition. The boundary patch descriptions and material properties are not partitioned. These parameters are read at runtime and distributed to all processors whether or not they are needed on that processor. For small numbers of processors ($P < 500$), this is an insignificant memory overhead for PUIFS, which is worthwhile because it simplifies any code modifications.

4.2 HARDWARE EMPLOYED: TRANSTECH PARAMID SYSTEM

Most of the computational experiments reported here were performed on this system, and so the description is worth describing in more detail.

This machine has 28 i860XP-based processor elements, 16 of which are equipped with 32 Mbytes and 12 of which are equipped with 16 Mbytes of fast (40 ns) DRAM memory. Each i860 is equipped with a T800 communication coprocessor with 8 or 4 Mbytes of memory. The PEs are hard connected in pairs with Inmos C004 multistage crossbar switches, providing interconnection between the PE pairs. This configuration allows great versatility in PE interconnection topology. An obvious and simple arrangement for the Paramid topology is a two-dimensional grid, which is the arrangement used for these results. A virtual channel router resident on each processor allows message passing between all of the processors in the machine, allowing the machine to be programmed as though the machine were a fully connected network. Parmacs, PVM, and C Toolset-style communication libraries are all available on the Paramid. These results have been obtained using the C Toolset library as this library gives better perfor-

mance than the alternatives on this platform. In this context, the latency was measured as 33 μs with a bandwidth of 1.7 Mb/s.

4.3 MEASURING PARALLEL PERFORMANCE

Strictly speaking, the runtime of the original serial code should be used as a measure of the runtime on one processor. This is, however, not always the most practical scheme (Fox, Williams, and Messina, 1994). It is often the case that in scrutinizing a code for parallelization, there arise instances when optimizations of the serial code may or must be made to achieve honest comparisons. One common occurrence in computational mechanics (CM) codes is the printing of end-of-sweep residuals, principally as a means of imparting confidence to the code user. Interrupting an operating system to print can carry a significant overhead, and so silencing a code gives a reduced runtime. This effect is of greater importance in parallel, where for many systems, the operating system interrupt can carry a significant overhead. We are left with a dilemma as to what we consider to be the runtime on one processor and what is the runtime on many processors. Many CM codes incorporate a timer to report the elapsed CPU time for a run. It has become normal practice for such timers to start after reading the problem specification from the file and stop before writing results to the file. This is reasonable because file access times can be dependent on other traffic on the systems. Timing only the CPU activity gives an optimistic view of parallel performance as parallel I/O hardware is rare, and so I/O activity seldom scales. The order of CM codes tends to be somewhere between linear $0(N)$ and quadratic $O(N)^2$, so measuring only CPU time is not unreasonable as this forms the asymptotic bound on runtime for large problems.

"One common occurrence in computational mechanics (CM) codes is the printing of end-of-sweep residuals, principally as a means of imparting confidence to the code user."

The results presented in this paper use the CPU time of the parallel code on one processor for t_1, which in this case is less than the runtime of the original code because of a number of serial optimizations. The overhead of the parallel version on a single processor is only the cost of the call to the communication routines in which no communication occurs. This has proved to have an insignificant impact on the runtime in numerous parallelized codes.

Parallel speedup S_P is the ratio of the runtime on one processor t_1 to the runtime on P processors t_P.

$$S_P = \frac{t_1}{t_P}. \tag{31}$$

If the parallelization is 100% efficient, then $S_P = P$, but this is rarely the case for real CM problems. There is always

some fraction of the code f_s $(0 < f_s < 1)$ that is inherently serial. This limitation on the maximum possible speedup is summarized as Amdahl's law,

$$S_P^{\max} = \frac{t_1}{\frac{(1-f_s t_1)}{P} + f_s t_1}. \quad (32)$$

The asymptotic limit of Amdahl's law as $P \to \infty$ gives

$$S_P^{\max} = \frac{1}{f_s}. \quad (33)$$

This clearly places a finite limit on the maximum achievable speedup from a parallel code. Amdahl's law was originally cited as a strong reason to doubt the usefulness of massively parallel systems. For a fixed problem size, f_s is constant, and so scalability is restricted. Scalability can only be possible if f_s reduces with an increasing problem size. It is now well established that, in practice, f_s for a CM code is often extremely small, even with modest problem sizes. CM codes tend to be somewhere between $0(N)$ and $0(N^2)$, whereas f_s is somewhere between constant and $0(N)$. Consequently, f_s tends toward insignificance as the problem size increases, and so scalability becomes possible. The communication cost and the idle time inevitably in a parallel code deteriorate the performance further. However, other factors not included in Amdahl's law, such as better cache usage for each subdomain in comparison with the global problem, can have a beneficial effect.

Parallel efficiency, E_P, is often used as the performance measure for a parallel code and is simply the ratio of the parallel speedup S_P to the number of processors P:

$$E_P = \frac{S_P}{P} \times 100\%. \quad (34)$$

In principle, parallel efficiency cannot exceed 100%. However, there are two instances when parallel efficiency may become superlinear and exceed 100%. One possibility is to break some data dependency in the parallel code that is not actually required. The implication here is that the serial code is open to some form of optimization. Having applied the optimization to the serial code, a superlinear parallel efficiency should no longer be achievable. The other cause of superlinear performance is cache usage. Decomposing a large problem that does not fit well into cache into a number of small problems may allow the decomposed problems to fit into cache. Cache success is an important factor in CPU performance, especially for high clock rates (>100 MHz) in the current generation of processors that are able to process data far faster than conventional DRAM memory may be accessed.

4.3.1 Scalability. Here we are concerned with two aspects of scalability—computation and memory.

Briefly, computational scalability is the extent to which the computation time is reduced for a problem of the same size as the number of processes is increased. Another way to estimate computational scalability is to consider how the parallel performance varies as the problem size per processor is kept constant but the number of processors increases.

The second scalability issue is concerned with memory—can the size of the problem be scaled linearly with the number of processors used? This requires that there are no globally sized data items and no significant arrays that have the number of processors as an index.

5 Results and Discussion

5.1 IRREGULAR SHAPE TEST CASE

The geometry of the irregular test case is shown in Figure 13 for the 3034 triangular mesh case. In fact, this geometry was configured for 5 mesh densities, with 3034, 10,027, 30,064, 60,005, and 119,822 triangles. The meshes were all partitioned using JOSTLE, and Figure 13 illustrates the use of different mapping strategies. In fact, five different mapping strategies were employed in the mesh-partitioning process:

1. Unmapped: Machine topology is ignored throughout the partitioning process.
2. Postmapped: The unmapped partition is postmapped to match the machine topology as a $p \times 2$ grid.
3. Premapped: Initially mapped 2-D partition optimized to reduce the number of cut edges.
4. Mapped 1-D: Mapped to a 1-D processor array.
5. Mapped 2-D: Mapped to a 2-D processor array.

The effect of the partitioning strategy on the cut edge count is illustrated in Figure 14. Through all of the mesh sizes, the lowest cut edge count is obtained using the unmapped (postmapped) partitioning strategy. The mapped 1-D and mapped 2-D partitions give the highest cut edge count, with the mapped 1-D partition having approximately twice the cut edge count of the other partitions. Conventional wisdom, of course, would indicate that the lowest edge cut is the best partition to use. However, this has to be balanced against the additional cost of communication if the mesh connectivity is not on a neighboring processor.

5.2 THE TEST CASES

Three cases were used to investigate the performance of each algorithm and the impact of physical coupling on the parallel performance.

5.2.1 Fluid Dynamics Test Case. To provide a fluid dynamics test case, the shape is filled with liquid gallium at 80°C. The boundary is set at 80°C with the exception of the top surface, which is cooled to 30°C. The test case is run to steady state to produce the convection currents illustrated in Figure 3a. The momentum, pressure, and heat solvers are used only for this test case, with the Jacobi method used for each solver. The Jacobi method is used simply because the parallel results with a Jacobi solver are identical regardless of the number of processors used. This makes it easier to detect any errors in the test runs. The Jacobi method does not give the best serial performance, and a Gauss-Seidel SOR solver would ordinarily be used for the pressure and heat solvers for such a problem. However, this issue is irrelevant for the purpose of evaluating speedup; performance results should be the same if Gauss-Seidel SOR is used.

5.2.2 Solid-Mechanics Test Case. To provide a solid-mechanics test case, the mesh was left free to move in all directions with the exception of the top surface, which was fixed. Material properties used were for gallium. A uniform fixed thermal load of 10°C was applied to an initial temperature of –30°C. This load was applied for four 2-second time steps. Four time steps were used simply to provide a convenient runtime for the purposes of measurement. An exaggerated mesh displacement is shown in Figure 15. Only the displacements are solved in this test case, with stresses being calculated from the displacements. The diagonally preconditioned conjugate-gradient method is used in the displacement solvers.

5.2.3 Solidification Test Case. The solidification test case starts with liquid gallium close to solidification at 30°C. The boundary is held at 20°C with the exception of the top surface, which is held at 0°C. The case is run until the gallium is largely solidified with small regions of recirculating liquid remaining. The residual stress contours, mesh displacement, and flow vectors are illustrated in Figure 3b. This case uses the larger stress overlaps for both the flow and the stress portions of the problem. At the start of the run, there is negligible work for the stress solver as most of the domain is liquid. At the end of the run, only a small portion of the problem remains liquid, yet the majority of the compute time is still required in the flow solvers. All of the solvers are enabled for this test case, momentum, pressure, heat, and displacement. The Jacobi method

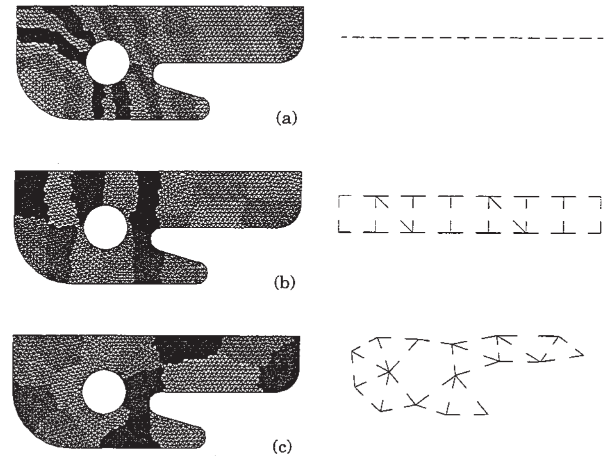

Fig. 13 Partitions of a 2-D mesh into (a) 1-D, (b) 2-D, and (c) uniform topologies with the corresponding subdomain connectivity graphs

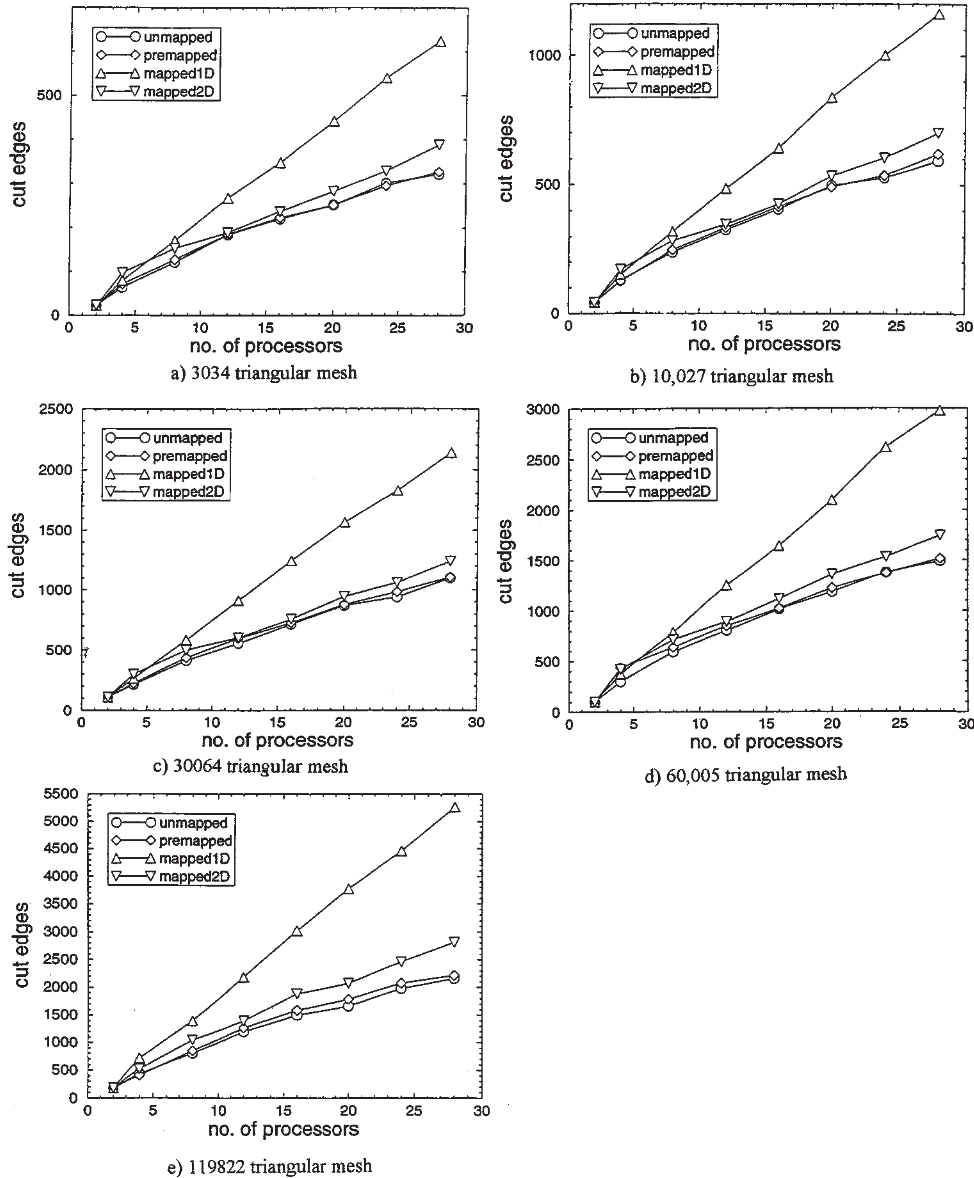

Fig. 14 The number of cut edges against the numbers of partitions for a range of partition strategies for the (a) 3034, (b) 10,027, (c) 30,064, (d) 60,005, and (e) 119,822 triangular meshes

is used for the momentum, pressure, and heat solvers as in the fluid dynamics test case. The diagonally preconditioned conjugate-gradient method is used in the displacement solvers as in the solid-mechanics test case.

5.3 PERFORMANCE ON A RELATIVELY HIGH-LATENCY SYSTEM

The following parallel performance measurements were obtained using the Transtech Paramid at the University of Grenwich, which communication delivers a start-up latency of 33 μs. This is quite high in comparison with current technology that provides around 5 μs latency.

The 30,064-element test case is the largest of the test cases that can fit into the memory of one 32 MByte processor node. The serial runtime for the 60,005-element case was regressed from the two-processor runtime, and for the 119,822-element test case, the four-processor runtime was used. Clearly, this affects the absolute accuracy of the graphs but does not change the trend in providing a comparison between partitioning techniques, as shown in Figures 16 to 30.

The lowest number of cut edges and (in this case) the lowest amount of communication for each mesh size is given by the unmapped (postmapped) partition, but this partition clearly does not give the best speedup performance. The unmapped and postmapped partitions are actually the same partition, the postmapped partition having had an additional optimized mapping of partitions to processors applied to it. Where the two partitions give a similar speedup, this reflects an unintentionally fortuitous mapping of the unmapped partition to the processor topology. It is possible that the unmapped and postmapped partitions may by chance be identical. It is, however, highly unlikely that the unmapped partition would ever give a better speedup than the postmapped partition; in such a case, the processor allocation strategy would have failed. Of course, any performance differences between the unmapped and postmapped partitions are unlikely to be significant for small numbers of processors.

The best overall speedup performance in the graphs is given by the mapped partitions, despite the cut edge count being higher than the other partitions. This confirms the proposition that partitioning in accordance with the machine topology will result in improved performance when the interprocessor communication:processor speed ratio is sufficiently high.

Using a pipelined (mapped 1-D) partition leads to a significantly higher number of cut edges, and consequently the message length is far greater, although fewer messages are required. A mapped 1-D partition requires only two messages and hence two latencies for each overlap update (one to each neighbor), which explain the perhaps unexpectedly good speedup results for the pipeline partition.

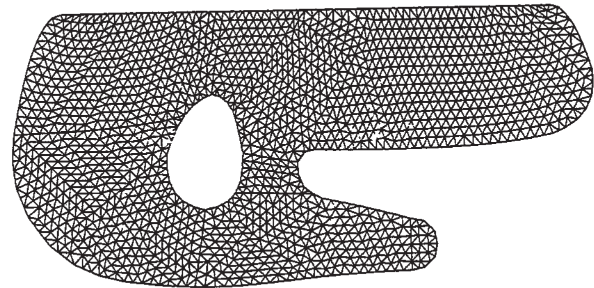

Fig. 15 Mesh displacement for the solid-mechanics test case

The mapped 2-D partition in Figure 13 shows the maximum node degree of the processor communication graph to be 4. However, the edges in this communication graph represent only element adjacency, but the data dependency is actually more extensive than merely adjacent. Adding overlaps to the subdomains therefore increases the maximum node degree of the processor communication graph to 5 as the overlaps reveal dependencies between subdomains previously shown as unconnected. Consequently, five messages are required for each overlap update. Given that the imbalance of elements between the subdomains for all cases is less than 0.25% (and for the secondary grid point partition, the imbalance is less than 0.75%), the effect of load imbalance for the test cases is insignificant (i.e., with constant element shape with near-constant mesh density).

These results indicate that the machine performance with this code is latency bound for the smaller test cases and bandwidth bound for the larger flow-dominated test cases. Consider Figure 16a, in which the best speedup is given with the mapped 1-D partition; this partition has the greatest amount of data to communicate but the lowest number of messages (latencies) per processor. Clearly, latency is the bound on performance with this problem. For the larger fluid dynamic test case shown in Figure 16b, the mapped 2-D partition gives the best speedup. Here, the large amount of data communication required for the mapped 1-D partition is eroding the advantage of fewer latencies, allowing the mapped 2-D partition to outperform it. Clearly, the interprocessor bandwidth is the bound on this problem. For the graphs between the small and large test case, the transition from latency to bandwidth bound can be seen. Figure 16c is an encouraging result that demonstrates that scalability is achievable given a large enough problem size. The slowdown exhibited with the small test cases is a direct consequence of the communication dominating the calculation, as the number of processors increases the time required for calculation falls, but the time required for communication remains more or less constant (see Figure 17).

Investigation shows that the relatively poor results for the solid-mechanics test cases (see Figures 18 and 19) are primarily a consequence of the two global commutative operations required in every iteration of the CG solver as implemented in the serial code. Each global commutative operation incurs a number of communication start-up latency costs; a high-latency cost leads to poor performance. This is revealed by profiling the parallel code execution in which the global commutative summations dominate the runtime. The solidification test case uses the larger overlaps required for the stress code, but this has only a slight effect on the speedup in comparison with the flow-only results. This confirms that the predominant limiting factor for performance on the Transtech Paramid is the communication start-up latency. Part of the solidification test case involves the CG solver, but again this only marginally affects the results as the time required for the stress calculation is considerably less than the time required for the flow and heat calculation, as can be seen in Figures 20 and 21.

Communication on the Transtech Paramid has been measured with a peak bandwidth of 1.7 MBytes per second. This bandwidth is not sustained with virtual channel routing and degrades to around 1.3 for near-neighbor communication and can get as low as 0.9 for nonlocal messages. This can deteriorate further to around 0.3 MBytes per second if the communication channels are saturated, as they will be for real problems with unmapped partitions. Similarly, the start-up latency degrades with increasing network traffic. While this bandwidth is low in comparison with other parallel machines (Dongarra and Dunigan, 1995), the latency appears reasonable. Similar performance may therefore be expected from other parallel platforms for the test cases that run to a latency bound. The test cases show that what is bandwidth limited on the Paramid would be expected to run slightly faster on other platforms and become latency bound.

Partitioning onto a $p \times q$ processor array for $q > 2$ has yet to be tested but is not expected to improve performance on the Paramid (or, indeed, other machines) with these test cases because of the latency bound. While a $q = 2$ mapped partition is likely to incur five latencies, a $q > 2$ mapped partition will incur eight latencies but will not significantly reduce the number of cut edges until P (and N) increases considerably.

5.4 IMPROVING PERFORMANCE ON THE HIGH-LATENCY SYSTEM

The results given in Section 5.3 demonstrate a range of results from poor to good with moderate parallelism. It is fair to say that the poor results reflect poor communication performance, especially in terms of the communication start-up latency. This, coupled with the reasonably good calculation performance of the parallel platform, leads to a poor calculation-to-communication ratio. Given that a parallel machine is unlikely to ever return perfect performance, all possible optimizations of the code should be sought. Two simple-to-implement optimizations that may be expected to realize a significant performance improvement became apparent. One is to re-

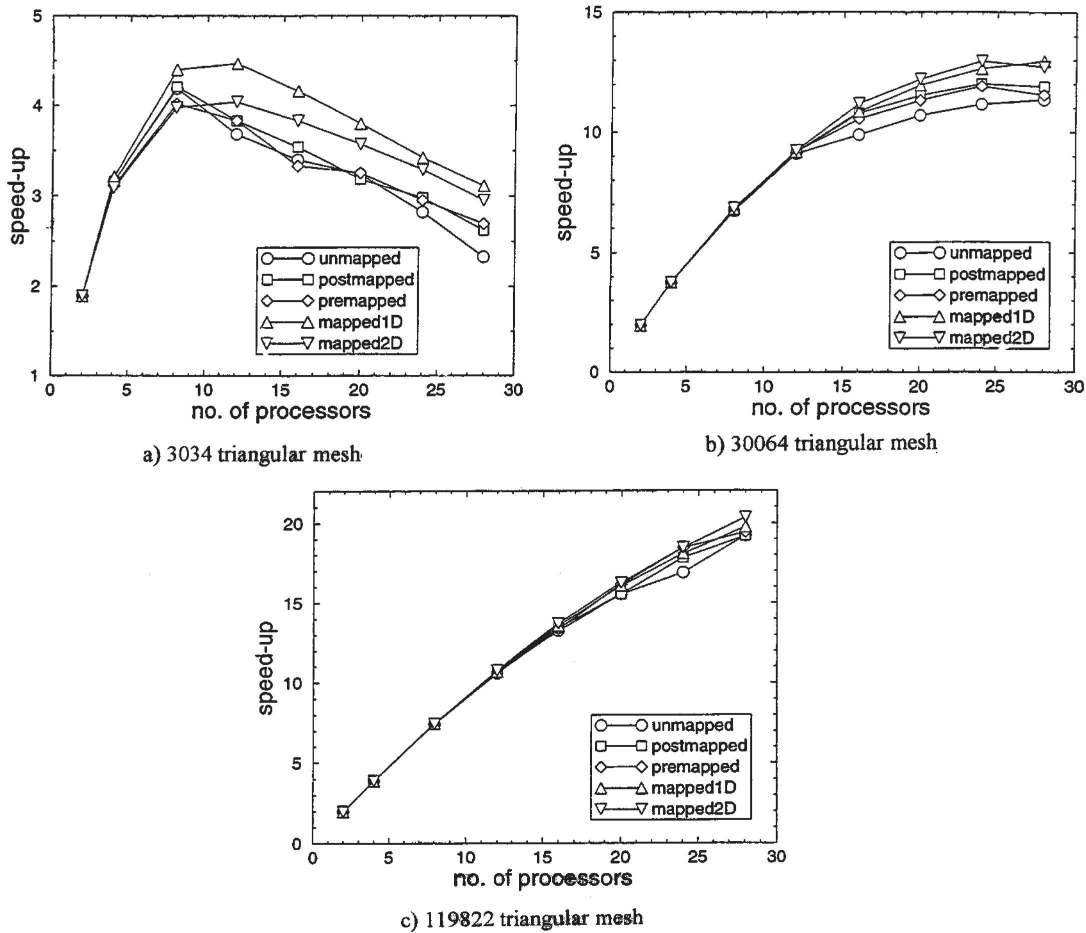

Fig. 16 Speedup for the fluid dynamic test case against the number of processors for a range of partition strategies for the (a) 3034, (b) 30,064, and (c) 119,822 triangular meshes

duce the start-up latency overhead of global commutative operations; the other is to overlap communication with calculation.

5.4.1 Latency Reduction. As communication start-up latency is the dominant component of the communication overhead, it seems reasonable to tackle this problem first. Profiling code execution provides a reasonably accurate view of where time is being spent in the code. For the test cases presented above, the profiles present a clear picture of the nature of the execution. The overriding proportion of the runtime was taken up in the linear solvers, and a significant portion of that time was spent in communication. Of the time spent in communication, it took approximately the same amount of time to

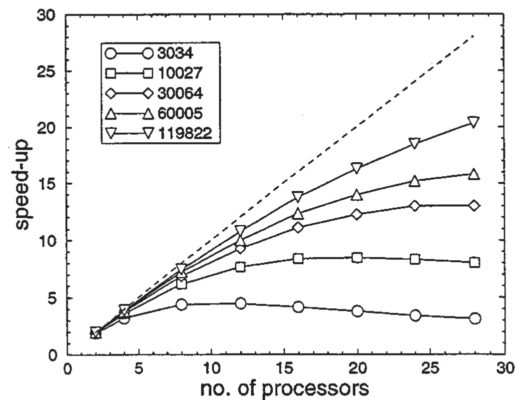

Fig. 17 Best speedup obtained for the fluid dynamic test case against the number of processors for a range of mesh sizes

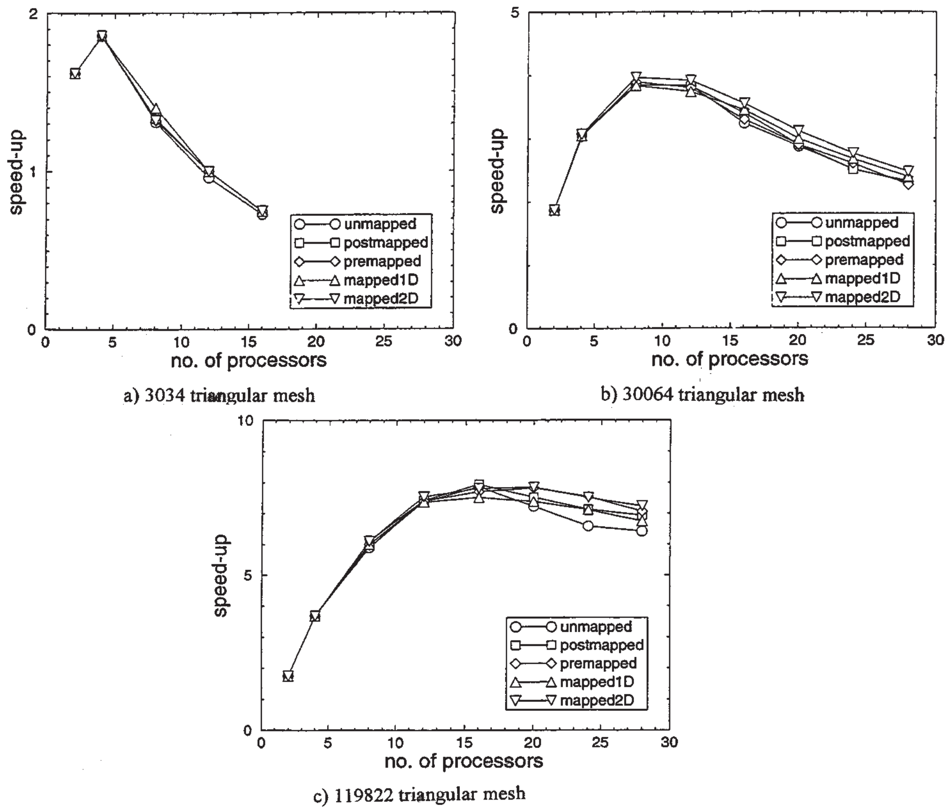

Fig. 18 Graph of speedup for the solid-mechanics test case against the number of processors for a range of partition strategies for the (a) 3034, (b) 30,064, and (c) 119,822 triangular meshes

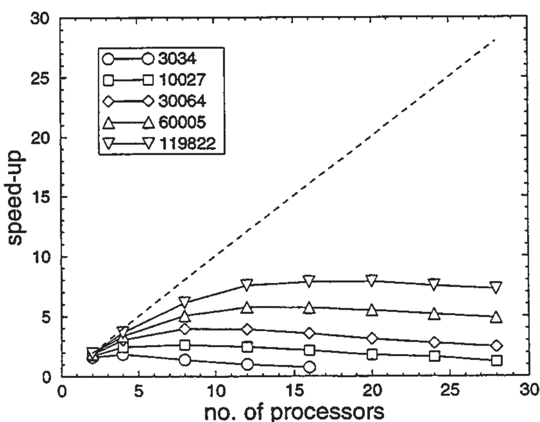

Fig. 19 Best speedup obtained for the solid-mechanics test case against the number of processors for a range of mesh sizes

carry out an overlap update as it did to carry out a global commutative operation.

5.4.2 Flow and Heat Solvers. Looking closely at the Jacobi and GS-SOR solvers, it becomes apparent that the preferred mode of operation in UIFS is to run these solvers to some preset maximum number of iterations, usually set at less than the amount required for convergence, and then loop over all solvers until an overall convergence criterion is reached. The logic here is that no one solver should take precedence in the path to convergence. The relative importance of each component in the solution is then reflected by the number of iterations set for each solver (e.g., 2 for each momentum, 10 for enthalpy, 20 for pressure correction). It is therefore not necessary to evaluate the residual norm at each iteration. A flag TOMITR in the original serial code is passed into each of the solvers to specify whether to run to the specified maximum number of itera-

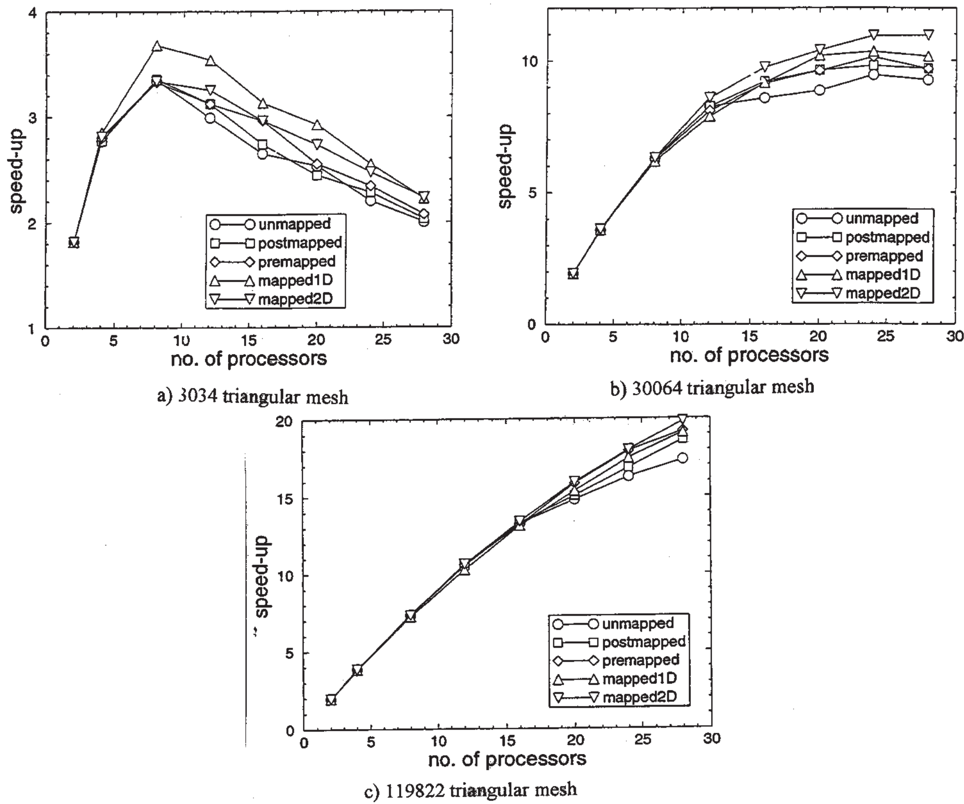

Fig. 20 Speedup for the solidification test case against the number of processors for a range of partition strategies for the (a) 3034, (b) 30,064, and (c) 119,822 triangular meshes

tions. For the test cases, TOMITR is always true. A simple conditional test of TOMITR allows the norm evaluation and hence global commutative operation to be omitted. This reduces the serial runtime by a small amount but has a significant effect on the parallel runtime.

The effect of this modification on the fluid dynamics test case is shown in Figure 22. In comparison with the performance of the unoptimized solver, the degree of improvement in the speedup is more pronounced with large numbers of processors as the proportion of communication to calculation increases with the number of processors. Also, the effect is more apparent with the smaller test cases as the proportion of communication to calculation is greater on the smaller latency-bound cases.

5.4.3 Solid-Mechanics Solver. The conjugate-gradient solver used in the solid-mechanics code has two inner-product operations. These operations appear in the

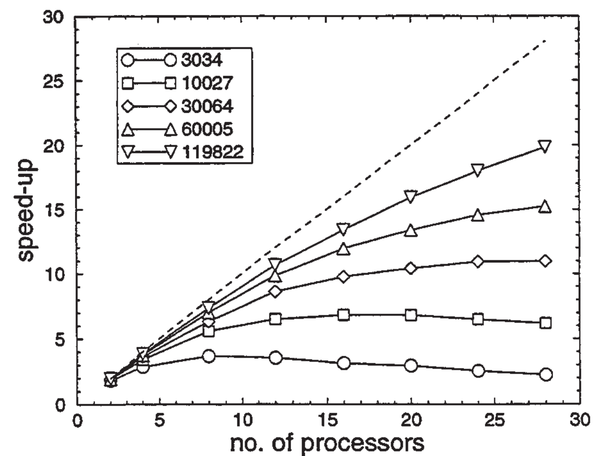

Fig. 21 Best speedup obtained for the solidification test case against the number of processors for a range of mesh sizes

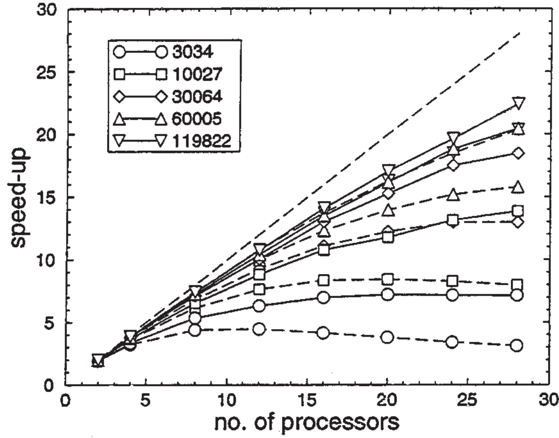

Fig. 22 Speedup obtained with the optimized (solid lines) and unoptimized (dashed lines) Jacobi solver for the fluid dynamics test case with a range of mesh sizes

source as two separate global summation operations. Close inspection of the code reveals that it is possible to rearrange the code to bring the summations to the same point in the code.

Substituting for $\underline{r}^{(k)}$ in equation (27) gives

$$\rho^{(k)} = (\underline{r}^{(k-1)} - \alpha^{(k)}\underline{u}^{(k)})^T(\underline{r}^{(k-1)} - \alpha^{(k)}\underline{u}^{(k)}), \tag{35}$$

which can be expanded to produce

$$\rho^{(k)} = \underline{r}^{(k-1)^T}\underline{r}^{(k-1)} + \alpha^{(k)2}\underline{u}^{(k)T}\underline{u}^{(k)} + 2\alpha^{(k)}\underline{r}^{(k-1)^T}\underline{u}^{(k)}. \tag{36}$$

Substituting $\underline{r}^{(k-1)T}\underline{r}^{(k-1)}$ from equation (27) and gathering terms now gives

$$\rho^{(k)} = \rho^{(k-1)} + \alpha^{(k)}(\alpha^{(k)}\underline{u}^{(k)T}\underline{u}^{(k)} + 2\underline{r}^{(k-1)T}\underline{u}^{(k)}). \tag{37}$$

Calculation of $\rho^{(k)}$ now requires two inner products instead of one but no longer requires $\underline{\boldsymbol{r}}^{(k)}$ and so may be moved forward in the loop to the same point at which $\alpha^{(k)}$ is calculated. So the algorithm becomes

$$\underline{u}^{(k)} = A\underline{p}^{(k-1)}, \tag{38}$$

$$\alpha^{(k)} = \frac{\rho^{(k-1)}}{p^{(k-1)T}\underline{u}^{(k)}}, \tag{39}$$

$$\rho^{(k)} = \rho^{(k-1)} + \alpha^{(k)}(\alpha^{(k)}\underline{u}^{(k)T}\underline{u}^{(k)} + 2\underline{r}^{(k-1)T}\underline{u}^{(k)}), \tag{40}$$

$$\mathbf{x}^{(k)} = x^{(k-1)} + \alpha^{(k)}\underline{p}^{(k-1)}, \tag{41}$$

$$\underline{r}^{(k)} = \underline{r}^{(k-1)} - \alpha^{(k)}\underline{u}^{(k)}, \tag{42}$$

$$\beta^{(k)} = \frac{\rho^{(k)}}{\rho^{(k-1)}}, \tag{43}$$

$$\underline{p}^{(k)} = \underline{r}^{(k)} + \beta^{(k)}\underline{p}^{(k-1)}. \tag{44}$$

The three inner products in equations (39) and (40) may now be calculated using only a single commutative operation to perform the three global summations.

This is similar to the work of D'Azevedo, Eijkhout, and Romine (1993) but involves no algorithmic modification whatsoever and so has no effect on the stability or convergence of the method. The time required for a global summation t_{gs} is dominated by the communication start-up latency, and so the time for three merged global summations

is approximately equal to the time required for a single global summation. This modification is trading the time required for an inner product t_{ip} against the time required for a global summation. When t_{gs} increases with increasing P and t_{ip} decreases with increasing P, with increasing P, there rapidly comes a point where this modification is beneficial. The effect of this modification on the solid-mechanics test case is shown in Figure 23. These results use the one-processor runtime for the faster unmodified CG solver to give a correct evaluation of the speedup. What is immediately apparent from Figure 23 is the improvement across a range of test case sizes for four or more processors. Close examination shows that the largest test case does not show improvement until more than four processors are used. This is consistent with t_{gs} being a function of P only; however, t_{ip} is a function of both P and the problem size N. Further increases in problem size would be expected to more clearly reveal this effect.

Figure 23 represents a significant improvement on the speedup results for the unoptimized solver, but the one remaining commutative operation remains an undesirable overhead. This prompts a closer examination of the global summation operation. A global summation operation has a great deal of parallelism as each processor evaluates its own partial sum. The original global summation algorithm was developed before the virtual channel router provided all-to-all communication. For this reason, the global summation operates in a chain fashion where each processor number P receives a sum from processor $P+1$, adds its own partial sum, and passes the result to processor number $P-1$. After $P-1$ messages, processor 1 has the global summation that can be broadcast to all processors; this will therefore involve $2(P-1)$ latencies overall (i.e., the latency overhead increases with the number of processors). This scheme ensures that each processor receives an identical copy of the global sum regardless of rounding errors.

As the C Toolset transport was lacking the functionality of a PVM reduce or MPI collective operation, a hypercube-based global commutative was implemented in CAPlib as discussed in Section 3.9. This scheme gives the lowest possible number of latencies and it is apparent from the results in Figure 24 that the effect of the hypercube commutative is highly significant. This confirms the proposition that communication start-up latency is an overridingly important factor in the achieved performance of a parallel system. The hypercube global commutative has since been found to provide superior performance to both PVM reduce and MPI collective on a wide range of platforms.

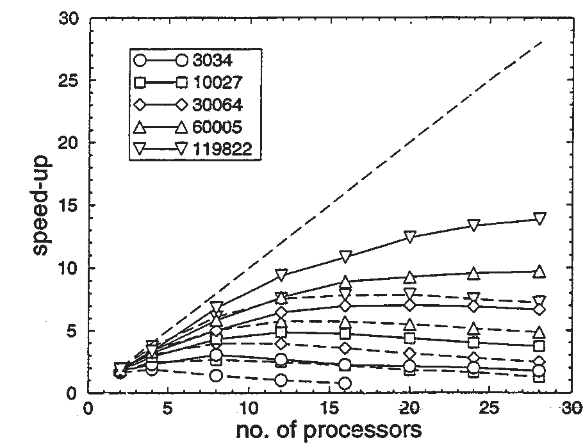

Fig. 23 Graph of speedup obtained with optimized (solid lines) and unoptimized (dashed lines) conjugate-gradient solver for the solid-mechanics test case with a range of mesh sizes

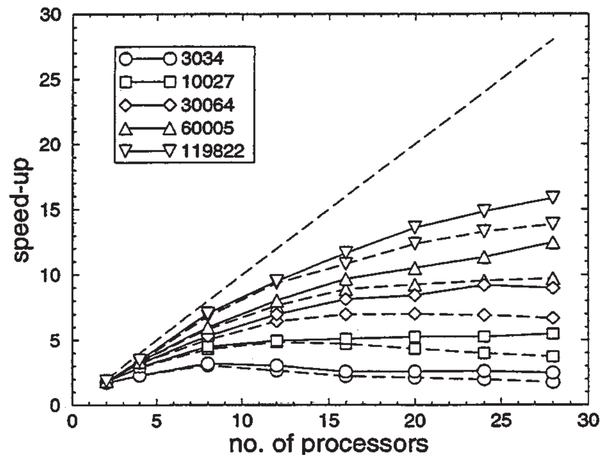

Fig. 24 Speedup obtained with the optimized conjugate-gradient solver using a hypercube (solid lines) and a pipeline (dashed lines) global commutative for the solid-mechanics test case with a range of mesh sizes

5.4.4 The Effect of Optimized Solvers on the Solidification Test Case. Figure 25a shows the effect of the optimized solvers and global commutative functions on the solidification test case. The reduction of latency-based communication overheads in the optimized solvers has had three important effects. Comparing Figure 25a with the plot in Figure 25b for the unmodified code clearly shows the effects. First, the overall level of speedup has increased; speedup that was in the range 12 to 15 for 28 processors has increased to 15 to 21. Second, the separation of the performance from the different partitions is more pronounced. Most noticeably, the lines for the mapped 1-D and mapped 2-D partitions have separated; this is a direct consequence of the bandwidth becoming more relevant as the latency is reduced in the solvers. The mapped 1-D partition has a larger amount of data to communicate and fewer communications than the mapped 2-D partition. Third, the gradient of the mapped 2-D partition line is much steeper in Figure 25a. Further, speedup could therefore be expected if more processors were available.

5.4.5 Asynchronous Communication. Many parallel platforms provide asynchronous or nonblocking communication calls to allow calculation to overlap communication. This allows subroutines to initialize a communication and return from the subroutine call before completion of the communication. The communication can then be tested for completion (synchronized) at some future point in the code. In an ideal case, unrelated code can be executed immediately after an asynchronous communication call and synchronization effected prior to the point at which the communicated data are used. This allows the execution of unrelated code to be overlapped with the communication. Often, this is not possible since the communicated data are immediately required. With PUIFS, asynchronous communication can be exploited within the solvers by splitting the computation into two parts. The Jacobi and Gauss-Seidel solvers first calculate the values around the perimeter of the subdomain where it isrequired in the overlaps of the neighboring subdomains. Once the perimeter calculation is complete, asynchronous communications of these variables is initiated. This leaves the time required to calculate the values in the rest of the subdomain for the asynchronous communication to complete. Completion of the communication is tested at a synchronization point before proceeding to the next iteration. The conjugate-gradient solver operates in a similar manner, splitting two loops so that calculation of u and p over the independent grid points is overlapped with the communication. These schemes amount to a renumbering

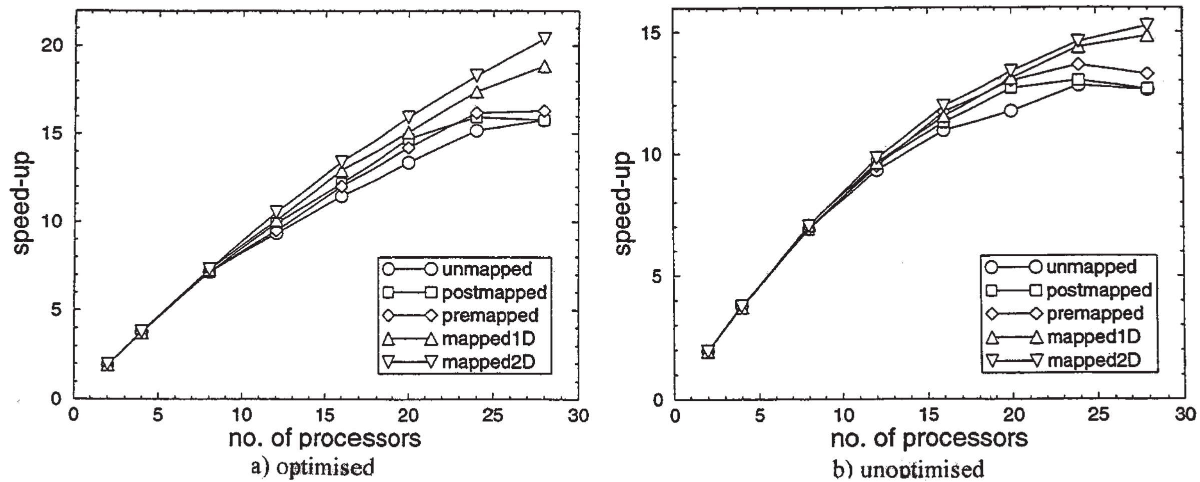

Fig. 25 Speedup obtained with the solvers for the solidification test case with a range of partition strategies for the 60,005 mesh when (a) optimized and (b) unoptimized

of each subdomain core so that entities that are required by the overlaps of neighboring subdomains are numbered before the rest of the core. Such renumbering is generally acceptable as partitioning has already changed the original numbering that was often merely a consequence of the mesh generation in the first instance (Jacobi and CG algorithm are order-independent anyway). The effect of this renumbering scheme on the mesh of 42 triangles is illustrated in Figure 26. Two changes in the numbering are apparent from the original local numbering scheme in Figure 10. First, the overlaps have been numbered as described at the end of Section 3.7, so that overlap elements that are owned by the same subdomain are numbered consecutively. This allows an overlap exchange to write the received overlap variables directly into memory without the need to unpack a buffer. Second, the elements within each subdomain that are overlap elements on neighboring subdomains have been numbered before the rest of the subdomain. Figure 27 shows the effect of the renumbering on the overlap communications. In contrast with the matrices shown in Figure 12, the communications now originate in the first few rows of each subdomain's matrix. In the iterative solver, these rows are evaluated first, and then the asynchronous communication of overlaps is initiated. Evaluation of the remainder of the rows in the matrix equation can then proceed for that iteration while the communication is being carried out. Completion of the com-

munication is tested before continuing on to the next iteration.

On the Transtech Paramid, asynchronous communication is achieved through exploitation of the T-800 coprocessors to manage this task. For workstation networks, notorious for their high latency, this is effected through communication buffers. The results of using asynchronous modified solvers for the fluid-dynamic test case and the solid-mechanics test case are presented in Figures 28 and 29. Here, the improvement in performance over the synchronous results is clear. These results paint a very different picture of parallel performance on a Transtech Paramid than those shown, for instance, in Figures 17 and 19. These results reinforce the common assertion that parallel performance is highly code, problem, and machine dependent. Overlapping the communication with calculation effectively hides the communication overhead as long as there is enough calculation to conceal the communication. The curves for the 3034- and 10,027-element test cases in Figure 28 show a drop in performance in comparison with the synchronous results for 28 processors. With large P and a small problem, the amount of calculation may not be sufficient to overlap all of the communication. Figure 30 clearly shows how effective this hiding is for the 60,005-element solidification test case. Here, the spread in performance between the partitioning strategies is far less apparent than the synchronous case in Figure 25. The mapped 1-D and mapped 2-D partitions still have a performance advantage, but the performance from the other partitions is now comparable with the mapped partitions. The premapped, postmapped, and unmapped partitions now look capable of returning further speedup beyond the 28 available processors, which is clearly not the case in Figure 25. The mapped 1-D and mapped 2-D partitions return near-identical performance as the bandwidth overhead of the mapped 1-D partition is effectively concealed and the advantage of the lower latency requirement for the 1-D partition becomes significant. These results invite investigation of the performance at higher numbers of processors. There must inevitably come a point at which the performance returned from the different partitions becomes apparent as the amount of computation in each subdomain core will no longer be sufficient to fully overlap the communication.

The scalability of this parallelization strategy is confirmed by the data below, which summarize the efficiency per processor when the problem size is scaled with the number of processors:

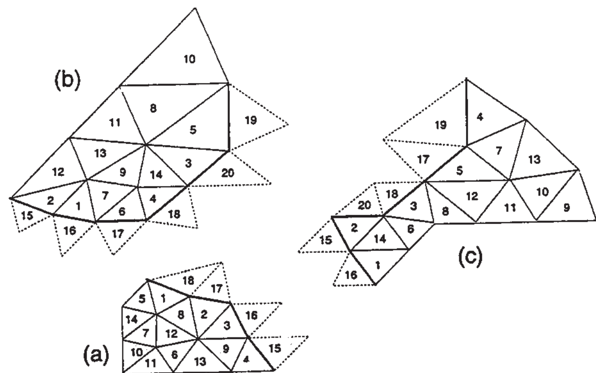

Fig. 26 Mesh of 42 triangular elements partitioned into three subdomains renumbered for asynchronous communication

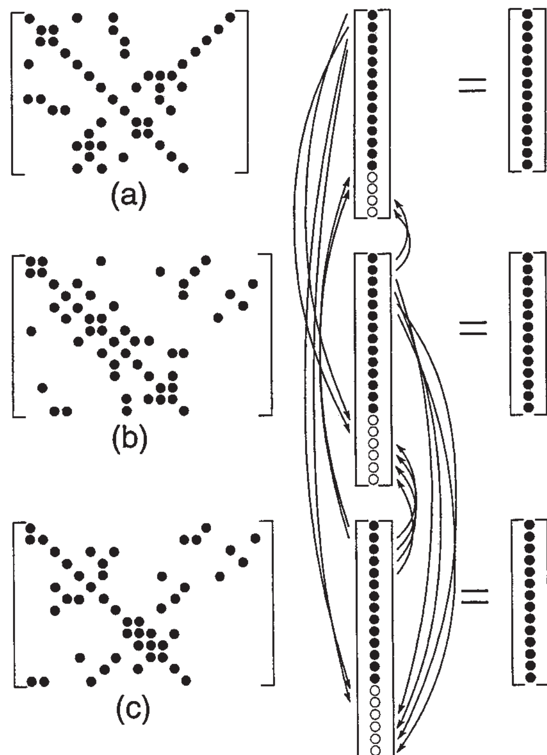

Fig. 27 Matrices of the 42-element mesh partitioned into three subdomains renumbered for asynchronous communication

Number of processors	1	2	6	12	24
Efficiency/processor	1	0.96	0.93	0.93	0.87

These results are based on 5000 nodes per processor (± 0.1%) with optimized DPCG solver, hypercube communative and asynchronous communications, and a mapped partition. The scalability is fading fairly gracefully and indicates a 65% efficiency per processor on a ~200 processor system.

4 Conclusions

In this paper, the issues associated with parallelizing codes that solve multiphysics problems on unstructured meshes have been examined. The focus is on codes that involve closely coupled physical interactions. Arguably, there are a range of alternatives for delivering the multiphysics functionality in scalar, let alone in parallel. However, the contextual software engineering strategy pursued for the scalar code is to employ a suite of procedures (one for each physical phenomenon) on a single mesh in a single code.

A parallelization strategy has been proposed that

- takes a partitioning approach that essentially disregards the physics during the code transformation into parallel form,
- requires the mesh-partitioning/load-balancing tool to provide a suitable partition based on a combination of the mesh geometry and distribution of the physics.

The first component above simplifies the code restructuring essentially to that of the conventional single discipline (e.g., CFD) code. There are a few additional complexities, such as secondary partitions and the associated interprocessor communication issues, but this approach appears to have significant potential provided there are mesh-partitioning/load-balancing tools that can deliver the functionality required. Such tools are now emerging, and the one exploited in this work is JOSTLE.

The above strategy has been explored in the context of the 2-D code, UIFS, which uses finite-volume techniques on an unstructured mesh and has "physics" solution procedures for fluid flow, heat transfer, phase change, and solid mechanics. Using this strategy, the entire UIFS code has been parallelized with only minimal changes to the code and the algorithm being required. Many of the subroutines required no change whatsoever. The majority of programming effort was required for the implementation of the initial decomposition of the code.

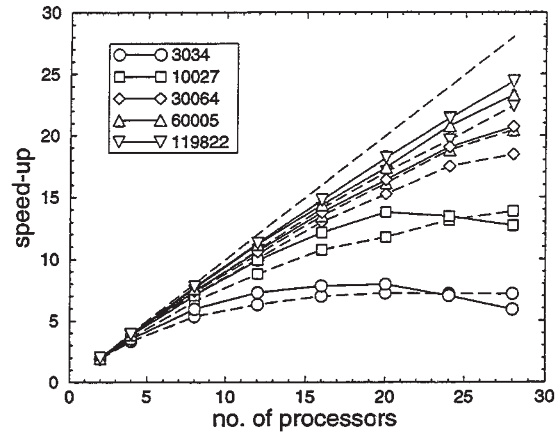

Fig. 28 Speedup obtained with the asynchronous (solid lines) and synchronous (dashed lines) optimized solvers for the fluid dynamic test case with a range of mesh sizes

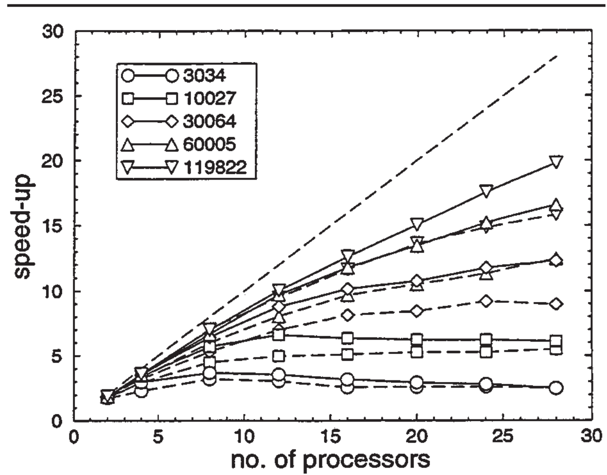

Fig. 29 Speedup obtained with the asynchronous (solid lines) and synchronous (dashed lines) optimized solvers for the solid-mechanics test case with a range of mesh sizes

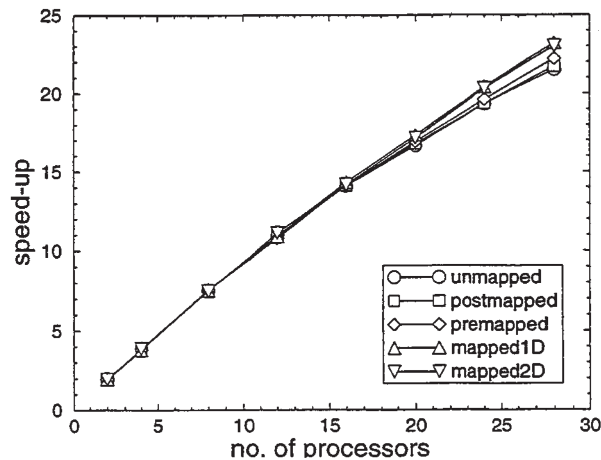

Fig. 30 Flow vectors for the fluid dynamics test case and residual stress contours with flow vectors for solidification test case

"Convergence is much more of a challenge in multiphysics simulation than for single-discipline codes. Preserving the convergence behavior in parallel without destroying the parallel efficiency requires great care."

One issue that is significant in the parallelization of multiphysics codes is the fact that procedures for distinct phenomena may well use different discretization procedures (and so use a range of data dependence stencils). This means that secondary (and possibly tertiary) partitions are required, which need to be structured so that as much interacting physics as possible in the same element is actually located on the same processor in parallel. This issue is covered in some detail here.

Convergence is much more of a challenge in multiphysics simulation than for single-discipline codes. Preserving the convergence behavior in parallel without destroying the parallel efficiency requires great care. The linear solvers used in this work are essentially parallel. The only one that is not, Gauss-Seidel SOR, can easily be made parallel by using "old" values in the overlap areas. Surprisingly, this has little effect on the convergence behavior when it is used on the velocity components and scalar quantities, in the context of the SIMPLE-type flow procedure. However, it turns out that the global summations involved in the conjugate gradient algorithm can affect the numerical result; this can be (and is) caused by a simple arithmetic process such as summing n real numbers, which is affected by rounding error. Even the ordering can affect the result given by summing from 1 to n as against summing from n to 1. In parallel, with two processors, the summation would be executed as something like $\sum_1^{\frac{n}{2}} + \sum_{\frac{n}{2}+1}^{n}$, which can give a different result to $\sum_1^n$. In practice, the coefficients that constitute the system matrix are also subject to numerical differences arising from rounding errors, which can actually mask the rounding effects from the solver. If the original serial algorithm is stable, then these effects have no actual significance on the results. If rounding effects lead to divergence of the parallel results, then suspicion must fall on the validity of the serial case.

Finally, we have examined the performance of this parallelization strategy. Looked at in the cold light of day, initial results were rather disappointing despite the careful consideration given to all the key issues identified above. Small, rather localized optimizations achieved very significant returns with regard to parallel performance. It is not that all the details in the broad strategy were wasted; it is more that without local optimizations, its full potential cannot be realized.

In the belief that reducing interprocessor communication time was vital to optimizing parallel performance,

Walshaw, Cross, and Everett (1997) worked hard at mesh-partitioning algorithms to map the submesh connectivity to that of the processor topology. This proved to be significant for a system in which the communications:processor speed ratio is relatively low, and only synchronous communications can be used. If asynchronous communications can be employed, these provide a significant advance in parallel performance; there is still some advantage afforded by the processor mapping, but its effect is much reduced. However, when running on systems in which the communications:processor speed is relatively high, there is little advantage served by exploiting either asynchronous communications or processor mapping. The key issue, then, is reduced to the generation of a high-quality partition very rapidly in parallel, so very large problems can be run as cost-effectively as possible. This issue has recently been addressed explicitly in parallelizing the multiphysics code, PHYSICA.

In this paper, we have focused on multiphysics in which each node/element/cell has the opportunity to be in the fluid or solid state. Here, the parallelization process is simplified to a certain extent—the same algorithms are applied over the whole mesh in turn. The next level of complexity is to address problems in which one phenomenon is active in one part of the mesh and another is active elsewhere. This class of problem forms the basis of current work.

LIST OF NOMENCLATURE

a_{ij}	Elements of the matrix A
d	Displacement in solid-mechanics equations
E_p	Parallel efficiency
E	Young's modulus
f	Liquid fraction
f_s	Code fraction inherently serial
g_o	Gravitational acceleration
h	Reduced enthalpy
h_{eff}	Effective heat transfer coefficient
k	Thermal conductivity
K	Permeability
K_{gap}	Thermal conductivity of metal-mold gap
L	Latent heat
M	Preconditioner
N_I	($i = x, y$) weighting function
p	Pressure
S_ϕ	(ϕ = solved for variable) source term
S_P	Speedup
T	Temperature
t_1, P	Time on 1 and t_P processors
u_I	Fluid velocity components
v	Fluid velocity vector
V_p	Cell volume
Y	Yield limit
α	Relaxation coefficient
γ	Flow resistance parameter
ΔF_I	Body force ($i = x, y$)
Δ_{gap}	Metal-mold gap distance
ε_{ij}	Strain ($ij = xx, xy, yy$)
σ_{ij}	Stress ($ij = xx, xy, yy$)
$\theta^{(eff)}$	Effective stress
μ	Viscosity
μ	Poisson's ratio
ϕ	Scalar-transported variable
Γ	Diffusivity of ϕ
ρ	Density

BIOGRAPHIES

Kevin McManus is a senior research fellow in the School of Computing and Mathematical Sciences at the University of Greenwich. After graduating from Warwick University with a B.Sc., he worked in the electronics industry before coming to Greenwich in 1991 to pursue research in computational science and engineering. An M.Sc. in scientific and engineering software technology in 1993 and a Ph.D. on strategies for the parallelization of multiphysics software followed in 1996. He has remained focused on this very challenging topic, especially in the context of the parallelization of the multiphysics modeling software framework, PHYSICA. He is the author of about 15 research papers.

Mark Cross is a professor of numerical modeling and director of the Centre for Numerical Modelling and Process Analysis in the School of Computing and Mathematical Sciences at the University of Greenwich. The center has about 100 staff and graduate students, of which about 10 are associated with the Parallel Processing Group, whose work is focused on the development of software tools to support the exploitation of such systems by computational modeling software. He was educated at the University of Wales, Cardiff and received a Ph.D. in 1972 for work on the modeling of semiconductor lasers. Since then, he has worked in both the United Kingdom and United States and has been at Greenwich since 1982. His research interests cover computational modeling of metals/materials processes, computational mechanics algorithms, and software tools and the exploitation of HPC systems. The editor of the archival journal, *Applied Mathematical Modelling*, he is the author of some 300 research publications.

Chris Walshaw is a senior research fellow in the School of Computing and Mathematical Sciences at the University of Greenwich. He graduated from Bath University with a B.Sc. in Mathematics and then moved to Edinburgh, where he gained an M.Sc. from Edinburgh University and a Ph.D. from Heriot-Watt

University, where his doctoral thesis concerned parallel algorithms for systems of differential equations. His postdoctoral work has extended this theme into parallel methods for adaptive unstructured meshes and, in particular, mesh partitioning. Since joining the University of Greenwich in 1993, he has developed the publicly available JOSTLE mesh-partitioning software. He is the author of some 45 research papers.

Steve Johnson is a reader in the School of Computing and Mathematical Sciences at the University of Greenwich. His entire career has been at Greenwich. After completing a B.Sc. in mathematics, statistics, and computing in 1987, he pursued a Ph.D. on techniques and tools for the parallelization of CFD (and similar) codes. Completed in 1992, his research on the interprocedural analysis of FORTRAN codes formed the basis of a large program of research on semiautomatic parallelization tools, which culminated in the development of the CAPTools software. He is the author of 30+ research publications and has supervised a number of Ph.D. candidates.

Peter Leggett is a senior research fellow in the School of Computing and Mathematical Sciences at the University of Greenwich. After completing a B.Sc. in mathematics, statistics, and computing in 1985, he began research in finite element methods for CFD. However, he became distracted by the then-emerging area of parallel computing and has since focused his attention on parallelization techniques and tools for computational science and engineering (CSE) applications. He wrote the intelligent user environment of CAPTools and also developed CAPlib, a portable thin-layer message-passing library specifically targeted at CSE applications, for which he earned his Ph.D. He is the author of 20+ research publications.

REFERENCES

Bailey, C., Chow, P., Cross, M., Fryer, Y., and Pericleous, K. 1996. Multiphysics modelling of the metals casting processes. *Pro. R. Soc. Lond. A* 452:459-486.

Bjørstad, P. E., Coughran, W. M., and Grosse, E. 1994. Parallel domain decomposition applied to coupled transport equations. In *Domain decomposition methods in scientific and engineering computing*, eds. D. Keyes and J. Xu, 369-380. Providence, RI: AMS.

Chow, P., and Cross, M. 1992. An enthalpy control volume unstructured mesh (CV-UM) algorithm for solidification by conduction only. *Int. J. Num. Methods in Eng.* 35:1849-1870.

Chow, P., Cross, M., and Pericleous, K. 1995. A natural extension of standard control volume CFD procedures to polygonal unstructured meshes. *Appl. Math. Modelling* 20:170-183.

Cross, M. 1996. Computational issues in the modelling of materials based manufacturing processes. *J. Comp. Aided Mats. Design* 3:100-116.

D'Azevedo, E., Eijkhout, V., and Romine, C. 1993. Reducing communication costs in the conjugate gradient algorithm on distributed memory multiple processors. Technical Report S-93-185, ORNL.

Dongarra, J., and Dunigan, T. 1995. Message passing performance of various computers. Technical report, ORNL.

el Dabaghie, F., et al. 1998. *Approximations and numerical methods for the solution of Maxwell's equations*. London: Clarendon.

Farhat, C. 1988. A simple and efficient automatic FEM domain decomposer. *Computers and Structures* 28:579-602.

Fox, G. C., Williams, R. D., and Messina, P. C. 1994. *Parallel computing works*. San Francisco: Morgan Kauffman.

Golub, G. H., and Van Loan, C. F. 1989. *Matrix computations*. 2nd ed. Baltimore, MD: John Hopkins University Press.

Hendrickson, B., and Devine, K. Forthcoming. Dynamic load balancing in computational mechanics. *Comp. Methods Applied Mech. & Eng.*

Karypis, G., and Kumar, V. 1998. Multilevel algorithms for multi-constraint graph partitioning, University of Minnesota, Report No. AHPCRC 98-019.

McManus, K. 1996. A strategy for mapping unstructured mesh computational mechanics programs onto distributed memory parallel architectures. Ph.D. diss., University of Greenwich.

McManus, K., Cross, M., and Johnson, S. 1995. Integrating flow and stress using an unstructured mesh on distributed memory parallel systems. In *Parallel computational fluid dynamics: New algorithms and applications*, 287-294. Amsterdam: North Holland.

Patankar, S. V. 1980. *Numerical heat transfer and fluid flow.* Washington, DC: Hemisphere.

Rhie, C. M., and Chow, W. L. 1982. A numerical study of the turbulent flow past an isolated aerofoil with trailing edge separation. *JAIAA* 21:1525-1532.

Taylor, G. A., Bailey, C., and Cross, M. 1995. Solution of the elastoviscoplastic constitutive equations: A finite volume approach. *Appl. Math. Modelling* 19:746-770.

Versteg, H. K., and Malalasekera, M. 1995. *An introduction to computational fluid dynamics: The finite volume method*. London: Longman.

Walshaw, C., Cross, M., and Everett, M. G. 1995. A localised algorithm for optimising unstructured mesh partitions. *Int. J. Supercomputer Appl.* 9:280-295.

Walshaw, C., Cross, M., and Everett, M. G. 1997. Dynamic load balancing for parallel adaptive unstructured meshes. In *Parallel processing for scientific computing*, eds. M. Heath et al. Philadelphia, PA: SIAM.

Zienkiewicz, O. C., and Taylor, R. L. 1991. *The finite element method*. 4th ed., vols. 1-2. New York: McGraw-Hill.