Coarse Grain Parallel Finite Element Simulations for Incompressible Flows

P.W. Grant and M.F. Webster[1]
Institute of non–Newtonian Fluid Mechanics
Department of Computer Science
University of Wales Swansea
Swansea, SA2 8PP, UK

X. Zhang[2]
Department of Computer Science and Information Systems
Brunel University
Uxbridge, Middlesex, UB8 3PH, UK

Parallel simulation of incompressible fluid flows is considered on networks of homogeneous workstations. Coarse-grain parallelization of a Taylor–Galerkin/pressure–correction finite element algorithm are discussed, taking into account network communication costs. The main issues include the parallelization of system assembly, and iterative and direct solvers, that are of common interest to finite element and general numerical computation. The parallelization strategies are implemented on a Sun workstation cluster using the PVM (Parallel Virtual Machine) message passing library. Test results are obtained with a maximum of nineteen workstations and various PVM configurations are exhibited. Parallel efficiency close to ideal has been achieved for some strategies adopted. It is suggested that loadbalancing may not always be beneficial on distributed platforms with broadcasting communication connection. © ??? John Wiley & Sons, Inc.

Keywords: Distributed parallelization, finite elements, coarse–grain parallelism

I. INTRODUCTION

Computational Fluid Dynamics (CFD) is well known as a problem domain for its computational intensive nature, with applications found in many different areas of science and

[1]Author for correspondence.
[2]This author was employed by the Department of Computer Science, University of Wales Swansea as a Senior Research Assistant when the research was carried out.

engineering. This is particularly true for the simulation of rheologically complex flows, that arise in many industrial processes and remain an elusive challenge even for the more advanced modern computers. Parallel computing is thus perceived as a promising avenue for future advances in this applied area of science.

A Taylor–Galerkin/ pressure–correction (TGPC) finite element semi–implicit time marching scheme has been developed in Swansea in a sequential form for the simulation of incompressible Newtonian and non-Newtonian flows [1, 2]. Here it is invoked for Newtonian flows. One of its merits for complex flows is its considerable computational accuracy. This algorithm has been used as a case study to develop an efficient parallel implementation, and investigations into implementations using a functional language have also been conducted [4, 5]. This paper presents a detailed study into how parallelism, in this finite element algorithm, can be explored, investigating various schemes through experimental testing. The target hardware platform is that of a network of homogeneous workstations. The parallelization strategies and associated test results are instructive for a wide range of CFD application domains. At this stage, issues such as heterogeneous networks and load balancing are yet to be investigated. PVM [6] software is adopted to support message passing among current software processes.

The paper is organized as follows. Section II. includes a brief review of the equations governing the flow of incompressible fluids and an overview of the TGPC algorithm. Section III. describes a domain decomposition approach, that is commonly adopted to allow the exploitation of data parallelism in field problems for distributed memory platforms. Test CFD problems addressed are described in Section IV. Sections V. to VII. are devoted to the parallelization of individual algorithm components and and their integration. Section VIII. examines the effect of a degree of freedom parallelization approach introduced on top of already tested parallel strategies. A summary is given in Section IX..

II. GOVERNING EQUATIONS AND NUMERICAL SCHEME

The governing equations for generalized Newtonian incompressible flows may be expressed by the Navier–Stokes equations as

$$\rho\frac{\partial \mathbf{u}}{\partial t} \;+\; \rho\mathbf{u}\cdot\nabla\mathbf{u} \;-\; \nabla\cdot(\mu\nabla\mathbf{u}) \;+\; \nabla p \;=\; \mathbf{f} \tag{2.1}$$

$$\nabla\cdot\mathbf{u} \;=\; 0 \tag{2.2}$$

where μ and ρ are the viscosity and density of the fluid, and $\mathbf{u}$ and p are the velocity and pressure field variables.

Finite element methods discretise the spatial problem domain, over which the solution is defined, into a mesh of finite elements (usually triangular or rectangular in two–dimensions). A numerical solution is then obtained at the mesh nodes. The solution at locations other than the mesh nodes is given by interpolation from those at neighbouring nodes. For this purpose we use continuous interpolation functions, with quadratic approximation for $\mathbf{u}$ and linear for p on two–dimensional triangular elements.

The above equations can be discretised by a transient semi–implicit Taylor–Galerkin/pressure–correction element scheme [1, 2] of the form:

$$(\frac{2\rho\mathbf{M}}{\Delta t}+\frac{\mathbf{S}_\mu}{2})(U^{(n+\frac{1}{2})}-U^{(n)})=\frac{F^{(n)}+F^{(n+\frac{1}{2})}}{2}+\{\mathbf{L}^TP-[\mathbf{S}_\mu+\rho\mathbf{N}(U)]U\}^{(n)} \tag{2.3}$$

$$(\frac{\rho\mathbf{M}}{\Delta t}+\frac{\mathbf{S}_\mu}{2})(U^{(*)}-U^{(n)})=\frac{F^{(n)}+F^{(n+1)}}{2}+\{\mathbf{L}^TP-\mathbf{S}_\mu U\}^{(n)}-[\rho\mathbf{N}(U)U]^{(n+\frac{1}{2})} \tag{2.4}$$

$$\frac{\Delta t\mathbf{K}}{2\rho}(P^{(n+1)}-P^{(n)})=\mathbf{L}U^{(*)} \tag{2.5}$$

$$\frac{2\rho\mathbf{M}}{\Delta t}(U^{(n+1)}-U^{(*)})=\mathbf{L}^T(P^{(n+1)}-P^{(n)}) \tag{2.6}$$

where $\mathbf{M}$ is a constant mass matrix; $\mathbf{S}_\mu$ is a diffusion matrix; $\mathbf{K}$ is a pressure difference stiffness matrix; $\mathbf{N}(U)$ is a convection matrix; $\mathbf{L}$ is a divergence/pressure gradient matrix; F is a force function vector due to boundary conditions; U is the velocity solution vectors and P is the pressure solution vector. Typically, the viscosity may be a function of shear rate. The order of computation on each time step is to start with equations (2.3) and (2.4) to solve for the intermediate field $U^{(*)}$ from $(U^{(n)}, P^{(n)})$, then equation (2.5) computes $P^{(n+1)}$, and finally equation (2.6) computes the end of time step velocity $U^{(n+1)}$. Matrices $\mathbf{M}$, $\mathbf{S}_\mu$ and $\mathbf{K}$ are sparse, *symmetric*, and *positive definite* under the appropriate choice of boundary conditions [3].

The entries of the convection matrix $\mathbf{N}(U)$, $\mathbf{N}_{ij}(U)$, appearing in the RHS of equations (2.3) and (2.4) take the form

$$\mathbf{N}_{ij}(U) \;=\; \int_\Omega \phi_i(\phi_l U_k^l \frac{\partial\phi_j}{\partial x_k})rd\Omega \tag{2.7}$$

where x_1 and x_2 indicate radial and axial axes, r and z respectively; ϕ is the interpolation function; $(U_1, U_2) = U$ and the double indices l imply summation. The tensorial entries of the diffusion matrix $\mathbf{S}_\mu$, $(\mathbf{S}_\mu)_{ij}$, appearing in both the RHS and left–hand–sides of equations (2.3) and (2.4) may be expressed as

$$(\mathbf{S}_\mu)_{ij}=[S_{lm}]_{ij}, \quad l,m=1,2, \tag{2.8}$$

$$(S_{11})_{ij}=\int_\Omega \mu(2\frac{\partial\phi_i}{\partial r}\frac{\partial\phi_j}{\partial r}+\frac{\partial\phi_i}{\partial z}\frac{\partial\phi_j}{\partial z}+2\frac{\phi_i\phi_j}{r^2})rd\Omega, \tag{2.9}$$

$$(S_{12})_{ij}=\int_\Omega \mu(\frac{\partial\phi_i}{\partial z}\frac{\partial\phi_j}{\partial r})rd\Omega, \tag{2.10}$$

$$(S_{21})_{ij}=(S_{12})_{ji}, \tag{2.11}$$

$$(S_{22})_{ij}=\int_\Omega \mu(\frac{\partial\phi_i}{\partial r}\frac{\partial\phi_j}{\partial r}+2\frac{\partial\phi_i}{\partial z}\frac{\partial\phi_j}{\partial z})rd\Omega. \tag{2.12}$$

Generally, for the simulation of non–Newtonian flows with non–polynomial functional form for μ, element level evaluation involves quadrature. In our case, loops over seven Gauss integration points per triangular element are employed. This is costly but unavoidable to retain accuracy. The present Newtonian context is a special case where the constant nature of the viscosity renders simplicity. In equations (2.3), (2.4) and (2.6), the RHS and solutions are multiple vectors, corresponding to individual velocity components of the continuous solution.

It is often desirable to use unstructured meshes, which require fine resolution in certain sub–domains, for the simulation of computational fluid dynamics problems, where for example local flow features, such as boundary layers, shocks or solution singularities arise. Since the amount of computation involved in a finite element method is generally proportional to the number of mesh elements, this technique allows optimal solution resolution for a given number of elements. An undesirable side effect of this technique is the production of unstructured irregular grids, which for some nodes have high node connectivity or, equivalently, high data dependencies on neighbouring nodes. This has the implication that resultant system matrices, when explicitly assembled, have complex structure whose storage may be overly demanding on space. In our implementation, two approaches have been adopted to resolve this difficulty. One is to avoid explicit assembly of system matrices, where possible, and the other is to use distributed matrix representation.

A preconditioned version of a Jacobi iterative method [7], that does not require explicit assembly of system matrices, is used for the solution of equations (2.3), (2.4) and (2.6). These equations are all of the same type. The relevant right–hand–sides (RHSs) are assembled and stored for use in the subsequent Jacobi iteration sweeps. This method has been shown to be well–suited to the solution of such augmented mass matrix equations. With their favourable conditioning few iterations are required [8], and also the method is highly parallelizable. Each vector component of the nodal solution is completely independent of the remainder within a single Jacobi iteration sweep.

For the two–dimensional static domain problems investigated here, the matrix $\mathbf{K}$ is constant. Its attributes suggest a direct method of solution for equation (2.5). Here a Choleski method [9] is particularly appropriate, as a large amount of its computation needs only to be performed once per flow simulation in the form of a Choleski factorization. Another reason that we prefer a direct method is that the pressure difference equation (2.5) may be ill–conditioned. Direct methods are often space demanding and may present hurdles to efficient parallel implementations over iterative alternatives. For certain problems, they are preferred to iterative methods due to their superior numerical accuracy and efficiency in speed [10]. In the context of two–dimensional static domain problems, the forward/backward substitution solution operation involved in the Choleski method is computationally insignificant compared with the overall computation time required for the whole TGPC algorithm. The mainly sequential nature of the Choleski method does not therefore cause any major concern. The storage of Choleski factors is distributed, which increases the ability to solve large problems on conventional workstation clusters. For dynamic domain and three–dimensional problems iterative methods may prove more cost–effective than direct methods. This has led to the investigation of Conjugate Gradient (CG) iterative alternatives, which will be considered elsewhere.

The main operations involved in the solution of equations (2.3)—(2.6) comprise

1. assembly of
   - RHS vectors of equations (2.3)—(2.6) accumulating local contributions,
   - system matrices $\mathbf{M}$ and $\mathbf{S}_\mu$ (implicit), and
   - matrix $\mathbf{K}$ (explicit),

2. factorization of matrix $\mathbf{K}$,
3. the Jacobi iteration, and
4. the Choleski solution.

The assembly and factorization of matrix $\mathbf{K}$ are performed only once for each simulation.

A finite element mesh contains two types of entities, namely elements and nodes. Discrete solution vectors are specified with components at the nodes of the mesh, while mesh elements describe data dependency at the element sub–domain level. The co–existence of these two types of entities makes it possible to express the computations involved in either *node–based* or *element–based* operational units. This is true, for example, of the assembly processes.

III. MESH DECOMPOSITION AND INTERPROCESSOR COMMUNICATION

In the continuous equation system, the solution at a particular location is generally dependent on that of the entire problem domain. In the discretised problem domain, data dependency is localized to particular elements. The nature of this localized data dependency renders finite element algorithms amenable to parallel implementations. There are two general parallelization approaches. The first approach, described in [11], is frequently termed *multicolouring* and uncouples local data dependency. A weakness to this approach lies in the difficulty in application to irregular domains [11]. A better alternative is to provide a global uncoupling by dividing the problem domain into sub–blocks, or sub–domains, a commonly used parallelization approach for finite element algorithms. This approach is generally called *domain decomposition* [12, 13]. Malone [14] discusses in detail how parallel computation can be achieved using this strategy. Basically, a sub–domain is used as a basic task distribution unit. If localized data dependency is not destroyed, a solution on the interior elements of a particular sub–domain can be sought entirely in parallel with that on other sub–domains. Then all required data will be local to the processor in control of that specific sub–domain. Interprocessor communication may be necessary only when the solution on boundary elements of a sub–domain is sought. If such communications are initiated at the earliest possible opportunity their cost will be minimized. This is achieved by computing and transmitting contributions in boundary elements, prior to computing those for elements internal to the sub–domain. Such a computation and communication arrangement enables coarse–grained data parallelism to be extracted efficiently and is particularly suitable for platforms of networked computers.

Both direct and iterative solvers can be implemented for parallel computation based on the domain decomposition approach and Farhat [15] presents a concise discussion. Examples of adaptation of classical approaches to iterative solvers can be found in [16, 11]. An innovative and efficient approach suggested by Farhat and Roux [17] introduces Lagrange multipliers to enforce compatibility at interface nodes. Its convergence properties are analysed in [18]. Iterative solvers in general are less economical than direct solvers for repeated right hand side analyses. In [19], a modification to the preconditioned Conjugate Gradient method is suggested to remedy this difficulty. Two investigations of domain decomposition direct solvers are reported in [20, 10]. In section VII., we describe an alternative approach to that presented in [20].

For an unstructured mesh, a boundary node may be shared by a large number of sub–domains. Accumulating boundary contributions from all sub–domains requires the knowledge of boundary node global connectivity and the corresponding interprocessor communication. It is appropriate to assign an extra processor to handle the accumulation of all boundary contributions. As the total number of boundary nodes is unlikely to be large, one boundary processor should be sufficient for this task. This approach has the following two advantages over distributing the task amongst sub–domain processors. Firstly, each sub–domain processor only needs to communicate with the boundary processor, reducing the number of its communication connections to a minimum. On this broadcasting network, establishing a network connection is likely to be expensive and therefore connection reduction should take a high priority. Secondly, a centralized communication configuration is much easier to program and manage than a distributed one. To optimize this process, the computation on boundary elements is performed prior to that on internal elements, so that boundary contributions can be delivered to the boundary processor and accumulated results can be returned in parallel with evaluation on elements internal to the sub–domain. On a broadcasting network where data transmission is sequential, this scheme requires $O(M + P)$ time to connect all sub–domain processors and transmit all boundary data for each process sweep of the entire mesh; M being the total number of boundary nodes and P the total number of sub–domains. The communication cost may be hidden provided there is sufficient computation to perform on internal elements in parallel with data communication.

As pointed out above, an implementation of a finite element algorithm may involve both node–based and element–based operations. In the existing sequential TGPC implementation, the majority of computation is performed element–wise. An element–based problem partition is therefore adopted, where every element is exclusively assigned to a particular sub–domain. For node–based operations, ambiguity arises for the distribution of computation at nodes in a boundary area (shared by elements belonging to different sub–domains). In this implementation, such operations are either duplicated for each processor in question, or performed on a dedicated processor, depending on the nature of the operations. If all operations are implemented in a node–based fashion, it is more appropriate to adopt the node–based partition approach, where every node is assigned exclusively to a particular sub–domain.

With the domain decomposition, static load–balancing is not difficult to organize. In our context the assembly processes, which are element–oriented, consume the majority of the computation time. Hence, for an element–based partition, equitable load–balance can be achieved by ensuring equal numbers of elements in all sub–domains. Under present circumstances and if an alternative node–based partition is adopted, the same balancing criteria should be applied as the number of nodes does not necessary reflect the amount of element–oriented computation. This is most poignant for unstructured meshes. Such a balancing criteria may be achieved by associating appropriate weights to individual mesh nodes. For problems with dynamically changing domains, dynamic load–balancing may be a necessity, see for example [21, 22].

There are many techniques for partitioning spatial domains, the one employed here is a Recursive Spectral Bisection (RSB) method [23]. RSB attempts to minimize the total length of sub–domain boundaries and therefore reduces interprocessor communication.

IV. TEST PROBLEMS AND METHODS

All test runs presented in this paper have been performed on a homogeneous network of nineteen diskless Sun Sparc–1 workstations employing the PVM3 [6] library for network message passing. These workstations run the SunOS (UNIX) operating system, and are interconnected by an *Ethernet* network.

Simulations for steady inelastic flow past a rigid sphere in a tube have been conducted under an axisymmetric frame of reference. For simplicity, in these test cases the shear viscosity is taken as a constant function. Two unstructured meshes of 1535 and 5764

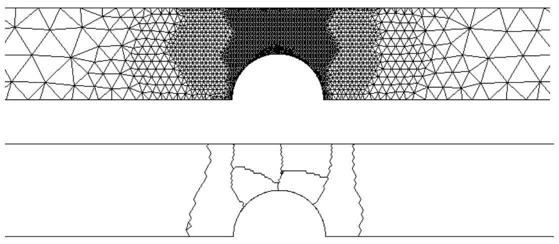
TABLE I. Number of mesh elements after partition

	# sub.	# internal		# boundary	
		min	max	min	max
Mesh 1	1	1535	1535	-	-
	2	744	744	23	24
	4	340	369	15	44
	8	192	184	8	49
	16	38	89	7	58
Mesh 2	1	5764	5764	-	-
	2	2822	2823	59	60
	4	1316	1381	60	125
	8	604	672	48	117
	16	245	338	22	115

TABLE II. Number of mesh nodes after partition

	# sub.	# internal		# boundary	
		min	max	min	max
Mesh 1	1	3452	3452	-	-
	2	1712	1717	23	23
	4	817	884	15	42
	8	395	448	9	48
	16	163	228	9	62
Mesh 2	1	12221	12221	-	-
	2	6078	6084	59	59
	4	2930	3085	61	122
	8	1427	1563	49	119
	16	676	824	23	123

FIG. 1. A typical test mesh and its partition



elements have been used, and quadrature is introduced for evaluation of integrals of element level quantities. Figure 1 shows the 5764 element mesh (partial) together with a representative domain partition. Table I contains various element related parameters of the two discretization meshes after being partitioned into different numbers of sub-domains and the equivalent node related parameters are included in Table II. The second column of these two tables indicates the number of sub-domains. Although the original meshes (`# sub. = 1`) do not possess any sub–domain boundary elements and nodes, they are included for comparison. These tables indicate the element–oriented and node–oriented sub–domain loadbalance. The amount of data involved in interprocessor communication is indicated by the size of boundary nodes given in Table II.

The timing of program execution is conducted in the following manner. Operations such as the assembly and Choleski factorization of matrix $\mathbf{K}$, that are not inside the TGPC time stepping process are not timed. Immediately before and after the execution of the time stepping cycles, each processor working in parallel sends a signal to a common monitor processor, that is lightly loaded. Upon receiving the start or finish signal from all such parallel processors, the monitor processor registers the system clock time. The execution time is taken as the difference between the finish and start times, normalized by the number of executed time steps. As sequential runs on one processor do not involve data communication, we define two parallelization speed–up ratios, R_1 and R_2, for a parallel implementation. R_1 with P processors is defined as the ratio of run times on one processor compared with that on P processors, and R_2 as the ratio of that on two processors compared with that on P processors.

V. SYSTEM ASSEMBLY

In this TGPC algorithm, a large amount of computation is spent on system assembly, both of matrices and RHS vectors. System matrices and RHSs that arise in finite element calculations can be mathematically decomposed into two different forms, described below. For clarity, we analyze only the matrix case, but the same principle applies to the RHS vector case.

Let $\mathbf{A}$ be a system matrix. The first approach is an element–based assembly, where the decomposition is expressed as

$$\mathbf{A} \;=\; \sum_e \mathbf{L}_e^T \mathbf{A}_e \mathbf{L}_e. \tag{5.1}$$

TABLE III. Relative speed (R_1) of parallel RHS assembly

# sub–domains	2	4	6	8	10	12	14	16	18
Mesh 1	2.0	4.0	6.0	7.9	10.0	12.0	14.0	15.2	16.5
Mesh 2	2.0	4.0	6.0	8.0	10.0	12.0	14.0	16.0	18.0

Each element matrix $\mathbf{A}_e$ arises from considering the element e in the finite–element mesh. $\mathbf{A}_e$ in our case is either 6×6 or 3×3 depending on whether quadratic or linear interpolation functions on the two–dimensional triangular element are used. The $\mathbf{L}_e$ matrix is a 6 (or 3) $\times n$ Boolean matrix, that maps the local node numbering (associated with the individual element e) to its global numbering for the complete domain (each column of $\mathbf{L}_e$ contains a single unit entry). Equation (5.1) implies that matrix $\mathbf{A}$ can assembled by first processing all elements to compute their element matrices $\mathbf{A}_e$, and then transmitting these entries into the appropriate location in $\mathbf{A}$. A second approach is a node–based assembly, where $\mathbf{A}$ is decomposed as

$$\mathbf{A} \;=\; \sum_n [\sum_e (\mathbf{L}_e)_n^T \mathbf{A}_e (\mathbf{L}_e)_n] \tag{5.2}$$

where $(\mathbf{L}_e)_n$ denotes a matrix obtained by zeroing all columns of $\mathbf{L}_e$ other than column n. This suggests a different computation order from the element–based assembly. The matrix $\mathbf{A}$ is assembled row–by–row. For each node n and each $\mathbf{A}_e$ connected to it, the nth row of $\mathbf{A}$ can be composed by computing the corresponding rows of $\mathbf{A}_e$. (For those elements that are not connected, their corresponding $(\mathbf{L}_e)_n = \mathbf{0}$).

Although the element–based and node–based assembly approaches differ only in computation order, this difference has wider implications. For the element–based approach, the amount of computation involved in assembling an element is fixed, as the number of nodes associated with each element is fixed. For the same reason, a dynamic data structure is unnecessary for storing element–to–node mapping information. This is an important issue for a FORTRAN implementation. For an unstructured mesh, however, the number of elements attached to an individual node can be unbounded. Therefore a dynamic data structure is necessary for keeping node–to–element mapping information, which is required for node–based assembly [24]. Naturally, the amount of computation involved in the node–based assembly varies across the different nodes. Another implication is related to the synchronization of memory access. On shared–memory systems, the element–based approach requires a control mechanism for synchronizing memory access. Alternatively, the node–based approach does not, as all updates to a specific entry are performed by the same processor. On a distributed–memory platform, the synchronization is irrelevant as no two processors can access the same memory entry.

For this FORTRAN implementation, only the element–based assembly approach has been implemented. In our functional implementation (in Haskell) of the same algorithm, the assembly is performed naturally in a node–based fashion, although forcing assembly in another order is also possible [5].

Assuming that interprocessor communication cost can be hidden in this context, the theoretical optimum parallelization speed–up for system assembly should be very close to the ideal linear target (the parallel implementation incurs a small amount of duplicated computation on boundary nodes). Indeed, Table III shows that results close to the optimum are achieved.

TABLE IV. Relative speed (R_1) of parallel Jacobi iteration

# sub–domains	2	4	6	8	10	12	14	16	18
Mesh 1	2.0	4.0	5.9	7.8	10.0	12.0	14.0	16.0	17.3
Mesh 2	2.0	4.0	6.0	8.0	10.0	12.0	14.0	16.0	17.8

VI. SOLVING FOR THE VELOCITY FIELD

A Jacobi iteration is adopted for the solution of velocity components. Suppose $\mathbf{A}X = B$ is the matrix equation to be solved, where B is a multiple vector comprising of an equal number of vectorial components to that of the velocity vector. The Jacobi method generates a sequence of vectors $X^{(r)}$ for index r through

$$X^{(r+1)} = (\mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{A})X^{(r)} + \omega\mathbf{D}^{-1}B \tag{6.1}$$

where ω is a positive relaxation factor and $\mathbf{D}$ is a chosen diagonal matrix. The choice of ω and $\mathbf{D}$ affects the convergence properties of the scheme. Here, we select ω as unity for convenience and the row sum version of $\mathbf{D}$, see [8]. $X^{(0)} = \mathbf{0}$ is usually adopted as the initial iterative starting vector, where X represents a temporal increment in the primary solution variables.

When solving equations (2.3), (2.4) and (2.6), we do not assemble these matrices explicitly. To economize on space we incorporate such operations into the Jacobi iteration. Using equation (5.1), equation (6.1) can be reorganized as:

$$\begin{aligned} X^{(r+1)} &= (\mathbf{I} - \omega\mathbf{D}^{-1}\sum_e \mathbf{L}_e^T\mathbf{A}_e\mathbf{L}_e)X^{(r)} + \omega\mathbf{D}^{-1}B \qquad (6.2)\\ &= X^{(r)} - \omega\mathbf{D}^{-1}\sum_e \mathbf{L}_e^T\mathbf{A}_e X_e^{(r)} + \omega\mathbf{D}^{-1}B \end{aligned}$$

which necessitates calculating $\mathbf{A}_e X_e^{(r)}$ per element, followed by permutation transposition with $\mathbf{L}_e^T$.

A Jacobi iteration also involves vector addition and subtraction, and diagonal matrix and vector multiplication. These operations are node–oriented, and distributed according to whichever sub–domain a node belongs to.

The theoretical parallelization speed–up for the Jacobi solver, like the system assembly case, is almost linear. This is supported by our empirical tests whose results are presented in Table IV. Within each Jacobi sweep, the costs in these tests of interprocessor communication have been hidden satisfactorily. However, this may not be the case when, for example, the number of sub–domains is sufficiently large. A variation to the above approach may prove advantageous. Initial sub–domains may be overlapped instead of having clear–cut boundaries. On each sub–domain, only solutions on interior nodes are updated by each Jacobi iteration sweep. The data dependency of these sub–domains will therefore contract with each iteration sweep. The overlap of the initial sub–domains must be sufficiently wide so that all nodes are included in the final Jacobi iteration sweep. In this manner, only one interprocessor communication phase is necessary for exchange of data pertaining to the overlapping area, minimizing the communication overhead incurred and tolerating a longer transmission delay. This scheme is feasible because the number of Jacobi iterations sweeps is always small, typically in the order of three. In fact, this scheme relaxes data dependency in boundary areas, and hence synchronization requirements, by duplicating computation. This approach awaits attention in a further study.

VII. SOLUTION FOR THE PRESSURE FIELD

A Choleski direct method is adopted for the solution of the pressure field in equation (2.5). It determines the solution of the system, $\mathbf{A}X = B$, by the forward and backward substitution steps:

$$\mathbf{L}Y = B, \qquad \mathbf{L}^T X = Y \tag{7.1}$$

where the Choleski matrix factor $\mathbf{L}$ is a lower triangular matrix satisfying $\mathbf{L}\mathbf{L}^T = \mathbf{A}$. The elements of $\mathbf{L}$ are given by

$$l_{ij} = (a_{ij} - \sum_{p=1}^{j-1} l_{ip}l_{jp})/l_{jj}, \quad j < i, \tag{7.2}$$

$$l_{ii} = (a_{ii} - \sum_{p=1}^{i-1} l_{ip}l_{ip})^{1/2}. \tag{7.3}$$

The forward/backward substitution steps can be described by the following computations:

$$y_i = (b_i - \sum_{p=1}^{i-1} l_{ip}y_p)/l_{ii}, \quad i = 1, 2, ..., N, \tag{7.4}$$

$$x_i = (y_i - \sum_{p=i+1}^{n} l_{pi}x_p)/l_{ii}, \quad i = N, N-1, ..., 1, \tag{7.5}$$

where $N \times N$ is the size of $\mathbf{A}$.

One major difficulty relating to the use of the Choleski method is the explicit storage of Choleski factors. For structured meshes, a standard fixed bandwidth storage scheme is a reasonable choice as it is straightforward to implement and can be space efficient. For unstructured meshes, a variable bandwidth or profile storage scheme is preferred as, in this case, the minimum bandwidths of assembled system matrices tend to be large. The matrix profile can normally be significantly reduced by adopting a profile reduction scheme such as that proposed by Sloan [25]. This has been adopted in this investigation.

Here, it is appropriate to reorder the system matrix, based on the concept of domain decomposition, to allow both distributed storage of its Choleski factor and distributed parallel Choleski forward/backward substitution. Without reordering, it can be very difficult to extract any meaningful parallelism on a distributed platform with broadcast data communication. The system matrix can be reordered into the following block form:

$$\mathbf{A} = \begin{bmatrix} A_{11} & & & A_{1,P+1} \\ & A_{22} & & A_{2,P+1} \\ & & \cdot & \cdot \\ & & \cdot & \cdot \\ A_{P+1,1} & A_{P+1,2} & \cdot\ \cdot & A_{P+1,P+1} \end{bmatrix}, \tag{7.6}$$

where $A_{i,P+1} = A^T_{P+1,i}$. Subscripts $i = 1, 2, ..., P$ refer to the ith sub–domains and $P + 1$ refers to the complete sub–domain boundary. A corresponding Choleski factor $\mathbf{L}$ will then have a block form of

$$\mathbf{L} = \begin{bmatrix} L_{11} & & & & \\ & L_{22} & & & \\ & & \cdot & & \\ & & & \cdot & \\ L_{P+1,1} & L_{P+1,2} & \cdot & \cdot & L_{P+1,P+1} \end{bmatrix} \tag{7.7}$$

which allows a certain amount of coarse–grained parallelism to be accessed. Forward/backward substitutions can be performed in parallel on blocks $\{L_{ii}, L_{P+1,i}\}$, $i = 1, 2, ..., P$. This formation is not difficult to achieve when we take advantage of a domain decomposition procedure. Conceptually, just as with the parallel assembly process described in Section V., it is possible to conduct forward/backward substitutions for interior areas of sub–domains in parallel, as these areas are not directly connected. This suggests that block L_{ii} should only be associated with internal nodes of the ith sub–domain. Operationally, this formation can be achieved by using the following local domain node numbering scheme:

1. number all internal nodes before any sub–boundary node, and
2. number nodes in the same sub–domain consecutively.

As a result, $L_{P+1,i}$ will represent the dependencies between internal and boundary nodes within the ith sub–domain, and $L_{P+1,P+1}$ the dependencies among all boundary nodes. Fortunately, this numbering scheme is not in conflict with that required for Sloan's profile reduction scheme, and a combined form is adopted that economizes on storage. By employing Sloan's heuristic function, this scheme first numbers internal nodes and then boundary nodes.

Average sizes of the sparse submatrices can be estimated as follows. For the diagonal blocks, $L_{i,i}$, the storage requirement is approximately $N/P \times h_{band}$. The blocks along the bottom row, $L_{P+1,i}$ for $i \leq P$ require $(n_B^{(i)}) \times N/P$ where $n_B^{(i)}$ is the number of boundary nodes in the sub–domain and this can be approximated by $(M.N)/P^2$. For the bottom corner block $L_{P+1,P+1}$ we have an estimate of $\Sigma_{j=1}^{P} n_B^{(i)} \times k_{conn}$ which approximates to $M.k_{conn}$, where k_{conn} is a measure of connectedness of the boundaries, which will be small for regular meshes.

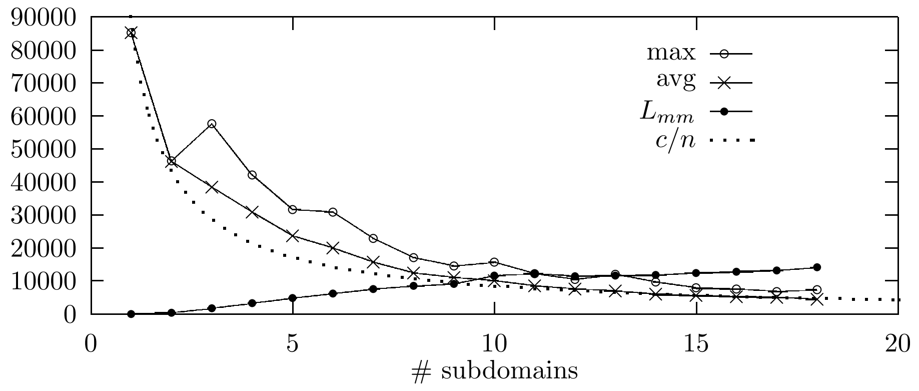
This Choleski domain decomposition scheme somewhat resembles the parallel scheme described in [20]. The main distinction is that the scheme in [20] does not factorise $A_{P+1,i}, i = 1, 2, 3, ..., P$, whereas ours does. Their approach has the advantage of avoiding a large number of fill–ins. The penalty is that the sub–domain processor has then to solve an equation with a $n_B^{(i)} + 1$ column right hand side,whereas our right hand side has only one column. It is also necessary to multiply all columns of the solution with the off-diagonal block (no fill–in), somewhat equivalent to our forward substitution involving $L_{P+1,i}$. The amount of data involved in interprocessor communication is $(n_B^{(i)})^2$ comparing less favourably with our $n_B^{(i)}$. Although the same amount of communication must be performed in our Choleski factorisation phase, we need only invoke it once.

The block $\{L_{ii}, L_{P+1,i}\}$, together with the associated computation, is distributed to the ith parallel processor and $L_{P+1,P+1}$ is distributed to the boundary processor. Again, only P communication connections are required. This is not as parallelizable as the assembly processes and Jacobi iteration, as communication and forward (or backward) substitution on boundary nodes can only start after (or before) the same operation has finished (or is started) on internal nodes. It should be noted that, of the overall computation, the Jacobi component is the dominant part. This storage scheme can be viewed as a distributed version of Liu's generalized envelope method [26], with the exception that nodes are not ordered by a minimum degree node numbering scheme. Liu's method provides a convenient way to exploit all zero entries in a Choleski factor using a conventional variable bandwidth storage scheme. In our case, this means all zero entries within $\{L_{ii}, L_{P+1,i}\}$ will be removed from storage when all entries on the second diagonal of L_{ii} are non-zero.

In this implementation, the same domain partition used for the assembly processes and Jacobi iteration is adopted. Since forward/backward substitutions are not element-oriented operations, such a partition does not necessarily lead to a balanced work load. Actually, the amount of computation required on a sub–domain is governed by the profile of its corresponding matrix block $\{L_{ii}, L_{P+1,i}\}$. For unstructured meshes, although a sub–domain containing nodes with higher connectivity tends to have less columns in block $\{L_{ii}, L_{P+1,i}\}$, this block is likely to be denser than for a structured mesh equivalent. Hence, this situation is somewhat self regulating as regards load-balancing.

Figure 2 displays the run profiles for Choleski factor blocks, where `avg` and `max` indicate the average and maximum number of entries in $\{L_{ii}, L_{P+1,i}\}$ respectively, and `c` indicates the overall size of the Choleski factor for the unpartitioned mesh. In Figure 2, both the

FIG. 2. # entries of Choleski factor blocks (`Mesh 2`)



average and maximum profiles of $\{L_{ii}, L_{P+1,i}\}$ are approximately inversely proportional to P, the number of sub–domains or parallel processors. The consistent peak at $P = 3$ is due to the fact that there is a significant $L_{P+1,i}$ profile increase at this point. The profile of $L_{P+1,P+1}$ is a major factor that influences the scalability of this implementation, and in both test cases, the run profiles for $L_{P+1,P+1}$ exhibit a linear increase with P. We can also observe from Figure 2 the load-balance of the sub–domain tasks associated with the forward/backward computations, by noting the difference between the maximum and average profiles of $\{L_{ii}, L_{P+1,i}\}$.

The profiles in Figure 2 indicate the amount of computation required to perform Choleski forward/backward substitutions, this information can be used to predict ideal parallel computation speed-up with zero communication cost, by computing the normalized total profile of $L_{P+1,P+1}$ and the maximum profile of $\{L_{ii}, L_{P+1,i}\}$. This is

TABLE V. Relative speed (R_1) of parallel Choleski method

# sub–domains		2	4	6	8	10	12	14	16	18
Mesh 1	ideal	1.9	1.6	2.2	2.4	3.1	3.3	3.2	2.9	3.0
	actual	1.5	1.6	1.6	1.3	1.1	0.9	0.8	0.7	0.6
Mesh 2	ideal	1.8	1.9	2.3	3.3	3.1	3.9	4.0	4.2	4.0
	actual	1.8	1.9	2.3	3.1	2.8	3.0	2.9	2.6	2.3

TABLE VI. Overall relative speed (R_1)

# sub–domains	2	4	6	8	10	12	14	16	18
Mesh 1	2.0	4.0	6.0	7.8	9.8	12.0	13.2	15.7	16.9
Mesh 2	2.0	4.0	6.0	8.0	10.0	11.9	13.9	15.8	17.7

introduced in Table V for comparison with the actual tests results. In Table V, the best actual speed-up achieved is only about three. For smaller numbers of sub–domains, the actual parallel performance is very close to ideal. For larger numbers of sub–domains, the actual performance departs from the ideal prediction due to the $O(M + P)$ network communication cost described in Section III.. The situation can be even worse if a temporal network congestion is created, as this tends to increase the overhead in establishing a connection. In such a context, a slightly unbalanced workload may prove favourable, allowing the $O(M+P)$ communication to start earlier and avoiding a network congestion.

As pointed out in Section II., sometimes an iterative solver is preferred for the solution of equation (2.5). A preconditioned CG method is an obvious candidate of choice. One approach is to apply the method to the global system matrix described in expression (7.6), which can be very slow compared with direct methods. A compromise is to use the classical Schur Complement method, which treats the interface problem iteratively using independent direct solvers for the sub–domains [15]. The second approach can be an order of magnitude faster than the first approach [10].

The overall performance of the algorithm is reflected in the results of Table VI, that show the relative speed when all previously described parallelization approaches are adopted simultaneously. Approximately linear speed-up has been achieved. Thus the overall speed performance of the algorithm has not been retarded in any significant manner by the Choleski component phase. This is due to the fact that the Choleski component is computationally insignificant in overall run time. Such a speed-up in performance is considered most satisfactory. For three–dimensional flows, the Choleski sub–blocks may become large if the domain is decomposed into P sub–domains. This can be partially alleviated in two ways, by increasing the number of processors and the local memory. Nevertheless, for large problems, in practice the domain would have to be decomposed into a larger number of sub–domains, and each processor would deal with several sub–blocks sequentially. The overall time however, would still be dominated by the Jacobi iteration stages.

These issues are considered in some more detail. The Jacobi iteration stages have a time complexity which is linear in N/P, and if the diagonal Choleski factors $L_{i,i}$ can be stored in the local memory of the processors then the time complexity of these stages are also linear in N/P. Let us suppose the vertex node size of our two–dimensional problem is, say, Q^2 then the scale–up to the corresponding three–dimensional problem can be taken as Q^3. The time for the Jacobi stages of the algorithm will then increase by a factor of Q translating from the two–dimensional to the three–dimensional problem.

In the Choleski stages, if P processors are used, then the size and scale–up of the factors $L_{i,i}$ will go from $(Q^2/P)h_{band}$ to $(Q^3/P)h_{band}$, and this may no longer fit within the memory (RAM) of a single processor. We could then view the Choleski stages as being solved by splitting the domain into QP sub–domains (or processes). The Choleski diagonal factors then have size $(Q^3/(QP))h_{band} = (Q^2/P)h_{band}$, identical to the situation for the two–dimensional problem. Each processor will then deal with Q sub–domains. The time for Choleski stages on one processor is then $Q(Q^2/P)h_{band}$ and so is also an increase by a factor of Q on the two–dimensional problem time. This has not taken into account the extra time needed for disk access (which can be local) and the extra network communication.

The disk access on one processor requires copying, some Q times, factors of size $(Q^2/P)h_{band}$. This gives an overall time of $O(Q^3/P)h_{band}$, the same order as before. The network communication for processor i involves passing the part solution vector at the internal nodes and in the three–dimensional case is of $O(Q^2/P)$. This can mostly be hidden when other processing is being performed.

The final part of the Choleski stage, involves computation using the sparse blocks $L_{P+1,j}$ of size $(Q^2/P)^2$ (which would require particular sparse storage considerations, presuming internal sub–domain node-numbering optimised before boundary nodes are numbered), and $L_{P+1,P+1}$ of total size $O(Q^2.k_{conn})$, being proportional to the size of the boundaries and their local connectivity banding factor k_{conn}. If necessary this could be distributed over the P available processors and so reduce the size to $(Q^2/P)k_{conn}$. It is also likely that extra memory or extra processors could be made available for three–dimensional problems. The implication is that the problem domain can be split into less than QP sub–domains, each associated problem being larger in size according to the increase in memory resource. This would reduce network communication time. To take advantage, timewise, of increase in the number of processors, the converse situation is sought of more sub-problems. A suitable compromise must be reached. Clearly if memory is more of a bottleneck, then increasing QP sub–domains, with fixed number of processors, would also be a possibility (as the allowable memory/processors increases), at the compromise of time.

From the above deliberations, it clearly can be arranged that the Choleski stage time consumption also increases by a factor of Q or less, and hence this method is certainly feasible for three–dimensional problems. One may wish to consider even tighter recommendations and analyse the possibilities of a generalised scale–up factor $1 \leq \beta_{size} \leq Q$; unity for a naive implementation and Q following the arguments above. Such a factor $\beta_{size} = memory\_increase\_factor * increase\_in\_processes$. In turn, time may be reduced by a factor $\beta_{time} = increase\_in\_processors * increase\_in\_processor\_speed$. The expectations here in scale–up somewhat modify the over–pessimistic view of a Q times factor on time degradation; it is not unreasonable to achieve Q' closer to $O(P)$, if one assumes $Q = O(P^2)$, $\beta_{time} = O(P^{1/2})$ and $\beta_{size} = O(P^{1/2})$. This is true as it may be established that $Q' = \beta_{time} * \beta_{size} = O(P^{1/2}.P^{1/2}) = O(P)$. If we take $P = O(10)$, the implication is that the vertex node problem size considered is $O(10^4)$ for two–dimensional and $O(10^6)$ for three–dimensional problems. This we believe is realistic in current practical implementations.

TABLE VII. Relative speed measures (R_1 and R_2) of DOF approach

# sub–domains		1	2	3	4	5	6	7	8	9
Mesh 1	R_1	1.1	2.3	3.4	4.5	5.6	6.5	7.7	8.6	9.8
	R_2	1.0	2.0	3.0	3.9	4.9	5.7	6.8	7.6	8.7
Mesh 2	R_1	1.0	2.0	3.0	4.0	4.9	6.0	7.0	7.9	8.9
	R_2	1.0	2.0	3.0	4.0	4.9	5.9	6.9	7.8	8.8

VIII. INCORPORATING A DEGREE OF FREEDOM APPROACH

It is possible to extract parallelism over multiple degrees of freedom in our algorithm. To do this, we assign statically one processor to each degree of freedom (refereed to as a DOF approach below). This is a hybrid approach — imposed upon the domain decomposition approach, described earlier, in a direct fashion. As the assembly and solution of equation (2.5) requires only one processor per sub–domain, the remaining processors lie idle. Under present circumstances this does not lead to a significant waste of resource as the duration of these work tasks is comparatively short.

More interprocessor communication must be introduced to implement the DOF approach. The first step for the solution of the equations is the assembly of RHSs. The fact that each RHS component in equations (2.3)—(2.6) depends on all solution components of its previous fractional step, and this implies that the solution computed by a particular processor should generally be broadcast within its sub–domain (including that on boundary nodes). Notice that each $(\mathbf{S}_\mu)_{ij}$ itself is a 2×2 matrix. This implies that terms appearing in equations (2.3) and (2.4) are dependent on all degrees of freedom of (ΔU) through products with components of $\mathbf{S}_\mu$. Therefore similar broadcasts are introduced at the beginning of each Jacobi sweep, when equations (2.3) and (2.4) are solved.

As the number of degrees of freedom is never large, we allow direct dialogue between each pair of processors within a sub–domain. Unfortunately, in this application, it is difficult to perform such communication in parallel with numerical computation. Since the overall message length of all such communications is proportional to the size of the problem domain, for large test problems, this type of communications may incur a measurable communication cost.

By using this DOF approach, we wish to access a D times parallelization speed–up, equivalent to the number of separate parallelizable degrees of freedom. However, this is impractical. The first reason is that the Choleski component is not parallelizable in this manner. The same is true for the time consuming evaluations of equations (2.7) and (2.8). The major part of the computation involved is necessary for each degree of freedom. In this implementation, the computation is actually duplicated on each processor.

Table VII reveals the performance of the DOF approach. By dividing R_1 by R_2 results, we can observe the net benefit of introducing the DOF approach. For `Mesh 1` the improvement is about 10% and for `Mesh 2` it is close to 0% for all sub–domain numbers. These figures are very low compared with the ideal performance of 100% for two–dimensional test problems. We attribute this to the components not parallelizable by the DOF approach. Since the total amount of extra communication introduced by the DOF approach between the processors, associated with each individual degree of freedom, is proportional to the mesh size, `Mesh 2` results in more data traffic. It is this that we link with the further performance degradation (10% to 0%).

However, it has been found that the DOF approach does not introduce any significant degradation on the effectiveness of the domain decomposition approach. In other words, a hybrid DOF test run with P (≤ 9) sub–domain partitions is roughly P times as fast as that without domain partitioning (R_2).

In conclusion, this hybrid DOF approach has not been found an effective strategy for this particular application on networked computers, although the approach extracts coarse–grained parallelism. The fundamental reason is that this application introduces extra network communication due to the fact that some terms in equations (2.3)—(2.6) are dependent on all degrees of a solution vector at a previous stage. Provided each degree of freedom is distributed to a different processor, the total amount of data involved in such communication within a time step is of $O(D^2N)$, where D is the total number of degrees of freedom and N the total number of mesh nodes. This is independent of the number of sub–domains introduced into the original problem domain through partitioning. Transmitting this amount of data on computer clusters, connected by broadcasting networks, does not make the approach easily scalable to problems with larger D or N values. There are many engineering circumstances which involve the simulation of large–scale Newtonian flows governed by the Navier–Stokes equations. For the simulation of such flows, their corresponding diffusion matrix may be represented in block diagonal form, which in turn avoids the need for data communication at the beginning of each Jacobi sweep. In these cases, the DOF approach would reflect better performance characteristics. In contrast on a shared–memory platform, such data dependency does not present a significant problem.

IX. CONCLUSION

In this article we have brought together the strengths and weaknesses of various parallelization strategies imposed upon a fractional–staged time stepping finite element algorithm. Most of the parallelization strategies are based upon a domain decomposition technique. Both direct and iterative solvers are considered within separate and unified frameworks. The predominance of the computation falls to the iterative solver, and as such almost ideal linear speed–up is attained via the coarse–grained strategies proposed. This is achieved through the effective masking of the communication overheads with computation performed in parallel.

Principally an element–based approach is utilized through a Jacobi iteration and during assembly processes. A domain partitioning strategy is found to be a most effective parallelization method that can be organized to complement a profile reduction scheme of Sloan. This achieves the desired block structure of the associated Choleski factors that enables parallelization of the direct solver. It is observed that a less balanced workload may sometimes be more beneficial to avoid communication congestion and improve overall parallel performance.

To further improve parallel performance, other strategies can also be incorporated. These include the parallelization of the quadrature over the Gauss points in domain integral evaluations by associating parallel processors with particular Gauss points over all elements. Another strategy is the adoption of parallel Conjugate Gradient methods.

This research was supported by a grant from the UK EPSRC (GR/J12321). The authors are grateful for the detailed comments and constructive advice from anonymous referees which has led to many improvements in this paper.

REFERENCES

1. D.M. Hawken, H.R. Tamaddon-Jahromi, P. Townsend, and M.F. Webster. A Taylor-Galerkin-based algorithm for viscous incompressible flow. *Int. J. Num. Meth. Fluids*, 10:327–351, 1990.
2. D. Ding, P. Townsend, and M.F. Webster. Computer modelling of transient thermal flows of non-Newtonian fluids. *J. Non-Newtonian Fluid Mechanics*, 47:239–265, 1993.
3. D. Ding, P. Townsend, and M.F. Webster. On Computation of Two and Three-Dimensional Unsteady Thermal Non-Newtonian Flows. *J. Num. Meth. Heat Fluid Flow*, 5:495-510, 1995.
4. P.W.Grant, J.A.Sharp, M.F.Webster, and X.Zhang. Sparse matrix representations in a functional language. *J. Functional Programming*, 6(1):1–28, January 1996.
5. P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Experiences of parallelising finite element problems in a functional style. *Software – Practice and Experience*, 25(9):947–974, September 1995.
6. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sundream. PVM3 user's guide and reference manual. Technical report, Oak Ridge National Laboratory, 1993.
7. L.A. Hageman and D.M. Young. *Applied Iterative Methods*. Academic Press, London, 1981.
8. D. Ding, P. Townsend, and M.F. Webster. Iterative solutions of Taylor-Galerkin augmented mass matrix equations. *Int. J. Num. Meth. Eng.*, 35:241–253, 1992.
9. J.H. Wilkinson and C. Reinsch. *Handbook for Automatic Computation, Linear Algebra*, volume II. Springer-Verlag, New York, 1971.
10. R. Keunings. Parallel finite element algorithms applied to computational rheology. *Computers Chem. Engng*, 19:647–669, 1995.
11. C. Farhat and E. Wilson. Concurrent iterative solution of large finite element systems. *Communications in Applied Numerical Methods*, 3:319 – 326, 1987.
12. P. Le Tallec, Y. H. De Roeck, and M. Vidrascu. Domain decomposition methods for large linear elliptic three dimensional problems. *J. Computat. Appl. Math.*, 34:93–117, 1991.
13. D. E. Keyes, T. F. Chan, G. Meurant, J. S. Scroggs, and R. G. Voigt, editors. *Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, 1992.
14. J. G. Malone. Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers. *Computer Methods in Applied Mechanics and Engineering*, 70:27 – 58, 1988.
15. C. Farhat. Which parallel finite element algorithm for which architecture and which problem? *Eng. Comput.*, 7:186 –195, September 1990.
16. B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In A. K. Noor, editor, *Parallel Computations and Their Impact on Mechanics*, pages 209 – 228. ASME, New York, 1987.
17. C. Farhat. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32:1205 – 1227, 1991.

18. C. Farhat, J. Mandel, and F. X. Roux. Optimal convergence propertities of the FETI domain decomposition method. *Comput. Methods Appl. Engrg*, 115:365 – 385, 1994.

19. C. Farhat, L. Crivelli, and F. X. Roux. Extending substructure based iterative solvers to multiple load and repeated analyses. *Comput. Methods Appl. Mech. Engrg.*, 117:195 – 209, 1994.

20. C. Farhat, E. Wilson, and G. Powell. Solution of finite element systems on concurrent computers. *Engineering with Computers*, 2:157 – 165, 1987.

21. C. H. Walshaw and M. Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. Technical Report 92.32, Division of Computer Science, University of Leeds, 1992.

22. P. Henriksen and R. Keunings. Parallel computation of the flow of integral viscoelastic fluids on a heterogeneous networks of workstations. *Int. J. Num. Meth. Fluids*, 18:1167–1183, 1994.

23. H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computer Systems in Engineering*, 2:135–148, 1991.

24. D.M. Hawken, P. Townsend, and M.F. Webster. Use of dynamic data structures in finite element applications. *Int. J. Num. Meth. Eng.*, 33:1795–811, 1992.

25. S. W. Sloan. An Algorithm for Profile and Wavefront Reduction of Sparse Matrices. *Int. J. Num. Meth. Eng.*, 23:239–251, 1986.

26. Joseph W.H. Liu. A generalized envelope method for sparse factorization by rows. *ACM Trans. on Math. Software*, 17:112–129, March 1991.