INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 18, 1167–1183 (1994)

PARALLEL COMPUTATION OF THE FLOW OF INTEGRAL VISCOELASTIC FLUIDS ON A HETEROGENEOUS NETWORK OF WORKSTATIONS

P. HENRIKSEN AND R. KEUNINGS

Unité de Mécanique Appliquée, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium

SUMMARY

We consider the parallel computation of flows of integral viscoelastic fluids on a heterogeneous network of workstations. The proposed methodology is relevant to computational mechanics problems which involve a compute-intensive treatment of internal variables (e.g. fibre suspension flow and deformation of viscoplastic solids). The main parallel computing issue in such applications is that of load balancing. Both static and dynamic allocation of work to processors are considered in the present paper. The proposed parallel algorithms have been implemented in an experimental, parallel version of the commercial POLYFLOW package developed in Louvain-la-Neuve. The implementation uses the public domain PVM software library (Parallel Virtual Machine), which we have extended in order to ease porting to heterogeneous networks. We describe parallel efficiency results obtained with three PVM configurations, involving up to seven workstations with maximum relative processing speeds of five. The physical problems are the stick/slip and abrupt contraction flows of a K.B.K.Z. integral fluid. Using static allocation, parallel efficiencies in the range 67%–85% were obtained on a PVM network with four workstations having relative speeds of 2:1:1:1. Parallel efficiencies higher than 90% were obtained on the three PVM configurations using the dynamic load-balancing schemes.

KEY WORDS Parallel numerical algorithms Load-balancing schemes Network of workstations Parallel software tools PVM Viscoelastic fluids

1. INTRODUCTION

The numerical simulation of non-Newtonian fluids has been a topic of active research over the last 15 years.[1–4] This field is particularly relevant to material-processing applications involving structured fluids such as polymers and fibre suspensions.[5,6] Significant breakthroughs have occurred recently towards the development of accurate and stable discretization methods for non-Newtonian flow in general and viscoelastic fluids in particular.[2,3] The demands in computer resources remain, however, a difficult issue. Indeed, realistic three-dimensional simulations of industrial viscoelastic flows are currently hardly feasible even with classical vector supercomputers. In this context parallel computing methodologies have an enormous potential.[7,8]

The most realistic constitutive equations for viscoelastic fluids belong to the class of integral models.[9,10] The use of such models in complex flow simulations has proven very difficult for reasons of numerical accuracy and stability, but also because of compute cost. The numerical difficulties have recently been largely overcome, in particular by Crochet and his collaborators.[11–13] The issue of compute cost is such that simulations with integral fluids are currently limited to steady state, two-dimensional flows. In fact, available sequential techniques for integral fluids are not at all appropriate to the vector-processing approach of classical supercomputers.

CCC 0271–2091/94/121167–17

Received December 1992
Revised April 1993

On the other hand, parallel techniques can be exploited fruitfully, as demonstrated by Aggarwal and Keunings[14] in their experiments with a 128-processor INTEL iPSC/860 hypercube.

The present work is part of an integrated effort within our group towards efficient parallel algorithms for simulating rheologically complex flows. Our current research themes include a generic frontal finite element solver for distributed memory parallel computers,[15–17] automatic domain decomposition and load-balancing algorithms,[17] and specific parallel techniques for memory fluids.[14,17] In this paper we explore the use of a heterogeneous network of workstations for the parallel computation of integral viscoelastic fluid flow. The participating workstations are grouped into a single computing entity by means of the public domain PVM software (Parallel Virtual Machine) developed by Beguelin *et al.*[18] Our developments are based on the sequential numerical methods proposed by Crochet and co-workers,[11–13] and implemented in the commercial POLYFLOW package.[19] Although results are reported for this particular field of applications only, the proposed parallel methodology can be applied to other problems in computational mechanics that involve a compute-intensive treatment of internal variables. Three examples of such problems are the deformation of viscoplastic solids,[20] the flow of fibre suspensions,[21] and the solution of differential viscoelastic models by the method of characteristics.[22]

The paper is organized as follows. We briefly review in Section 2 the field equations governing the flow of memory integral fluids. Section 2 also includes an overview of the sequential numerical method developed by Crochet and his collaborators.[11–13] The proposed parallel algorithms are described in Section 3. Section 4 is devoted to PVM implementation issues, while Section 5 defines the PVM configurations used in this work. The issue of measuring parallel efficiency on a heterogeneous PVM configuration is discussed in Section 6. In Section 7 we describe the flow problems selected for our experiments. Finally we discuss in Section 9 the results of the simulations in terms of the efficiency of the parallel algorithms.

2. GOVERNING EQUATIONS AND NUMERICAL TECHNIQUE

We consider steady state, isothermal, creeping 2D flows of incompressible viscoelastic fluids. The conservation laws read

$$\nabla\cdot(-p\mathbf{I}+\mathbf{T})+\mathbf{f}=0, \qquad \nabla\cdot\mathbf{v}=0, \tag{1}$$

where p is the pressure, $\mathbf{I}$ is the unit tensor, $\mathbf{T}$ is the extra stress tensor, $\mathbf{v}$ is the velocity vector and $\mathbf{f}$ is the body force per unit volume of fluid. For integral viscoelastic fluids the extra stress $\mathbf{T}$ is related to the deformation experienced by the fluid through a memory integral along the particle paths.[10] In the present paper we select a K.B.K.Z. integral model of the separable form as used in actual polymer-processing simulations by Goublomme *et al.*[13] We have

$$\mathbf{T}=\int_0^{\infty} M(s)h(I_1, I_2)\mathbf{C}_t^{-1}(t-s)\,\mathrm{d}s, \tag{2}$$

where the integral is computed along particle paths parametrized by the time lapse s relative to current time t. The symbol $\mathbf{C}_t^{-1}$ denotes the Finger strain tensor, namely the inverse of the Cauchy–Green strain tensor.[10] The damping function h depends upon the strain invariants I_1 and I_2, defined as the traces of the Finger and Cauchy–Green strain tensors respectively. Finally, the factor $M(s)$ is a memory function obtained from linear viscoelastic measurements. We have selected for our numerical experiments the particular forms for M and h used by Goublomme *et al.*[13] in their recent paper.

The governing equations (1) and (2) form a highly non-linear problem in view of the fact that pathlines needed to compute the memory integral (2) are *a priori* unknown. Numerical techniques for simulating integral viscoelastic flows are based on a decoupled approach whereby the computation of the extra stress is performed separately from that of the flow kinematics.[2,3] From known kinematics, one first calculates the extra stress by means of the constitutive model (2); the velocity and pressure fields are then updated by solving the conservation equations (1), using the current extra stress as a known pseudobody force term. The procedure is then iterated upon.

Formally, one iteration of the solution procedure involves the following steps.

Step 1	Integrate the constitutive equation (2) to compute the extra stress, using the kinematics calculated at the previous iteration.
Step 2	Update the kinematics (and pressure) by solving the conservation laws (1); the extra stress computed in Step 1 is treated as a pseudobody force.

Step 2 defines a Stokes flow problem, which Crochet *et al.*[11–13] solve by means of a Galerkin, finite element velocity–pressure formulation. This so-called *u–v–p* problem leads to an algebraic set of equations for the nodal unknowns of velocity and pressure, which is solved using a direct frontal solver. In order to compute the generalized load vector of the Stokes problem, values of the extra stress are needed at *all* Gauss integration points of the *u–v–p* finite element mesh.

Step 1 consists of three subtasks, to be performed for each Gauss point:

Tracking	On the basis of the velocity field computed at the previous iteration, determine the upstream trajectory of the integration points.
Strain	Compute the deformation between a discrete number of past configurations and the current one; then evaluate the integrand of (2).
Stress	Compute the memory integral (2) numerically.

Note that these substeps involve computations that are *non-local* to the elements of the *u–v–p* mesh. Highly accurate tracking and strain computation techniques have been developed by Crochet and his collaborators.[11–13] These authors have implemented the above numerical algorithm in the commercial package POLYFLOW developed in Louvain-la-Neuve for the simulation of polymer processing flows.[19] This sequential code is used as the starting point for our 'parallel' developments with integral viscoelastic models.

3. PARALLEL ALGORITHMS

Like any other finite-element-based technique, the numerical method described in the previous section has two potentially compute-intensive phases, namely (i) the element-by-element calculation of the stiffness matrix and load vector and (ii) the solution of the algebraic equations. For complex two-dimensional flows of integral fluids the solution phase is only a very small temporal fraction of the total sequential run. Values in the range 0·4%–5% are reported by Aggarwal and Keunings[14] for typical applications. One should thus first concentrate on the development of efficient parallel strategies for the element-by-element computations.

This seems at first sight to be an 'embarrassingly parallel' problem. It is indeed one if the workload associated with the computation of the element contributions does not change from one element to the other *and* if the available processors have identical hardware characteristics and overall workload. In such a case equidistribution of the elements to the available processors will lead to perfect load balancing. With integral viscoelastic fluids and heterogeneous parallel systems, however, equidistribution is unlikely to produce satisfactory results and load balancing becomes an important issue.[14] Indeed, the CPU time required to compute the element contributions may vary by an order of magnitude from element to element within the same non-linear iteration. In addition, the compute load within each element can vary from one iteration to the next. Finally, important differences in relative speed may exist between the participating processors, and their overall load can vary in time. Dynamic allocation schemes thus appear essential.

In the sequel we present the parallel algorithms in terms of concurrent processes instead of concurrent processors. We have indeed in mind that several processes may co-exist on the same processor within a local time-sharing environment.

Following the work of Aggarwal and Keunings,[14] the finite element equations are solved *sequentially* by means of a direct frontal technique. On the other hand, the computation of the element contributions is distributed to the concurrent compute processes. We consider both static and dynamic distribution strategies.

A simple static allocation strategy amounts to distributing an equal number of elements to the processes in charge of the element computations. The distribution is left unchanged during the course of the non-linear iterations. This strategy has been used by Aggarwal and Keunings[14] in their preliminary work on the 128-processor INTEL iPSC/860 hypercube. As discussed above, it cannot guarantee load balancing. In the dynamic approach the elements are allocated to the compute processes on a *work-on-demand* basis. This involves additional interprocess communications relative to the static approach. The gain is of course that load balancing is achieved automatically. We shall consider two versions of the dynamic allocation procedure.

The proposed algorithms are described in pseudocode in Figures 1–3. Three types of concurrent processes are involved: a single *Host* process, a single *Frontal* process and a number *N* of *Node* processes. In the static approach the Host single task is to initiate the Frontal and

```
PARALLEL ALGORITHM (Static allocation scheme)

Concurrent processes : Host, Frontal, {Node[i] , i=1,2,...,N};
Process Host :
        Initiate process [Frontal, {Node[i], i=1,2,...,N}];
Process Frontal :
        Loop it : start = 1, end = Max_number_of_iterations, step = 1:
                Send kinematics and mesh to all Node processes;
                Loop k : start = 1, end = Number_of_elements, step = 1:
                        Receive contribution of element k from a Node process;
                        Assemble contribution into active system;
                        Perform frontal elimination;
                Compute solution by backsubstitution;
                If solution has converged then stop iterations;
        Send Stop_signal to all Node processes;
Process Node[i] :
        Repeat until reception of Stop_signal:
                Receive kinematics and mesh;
                Loop j : start = i, end = Number_of_elements, step = N:
                        Compute contribution of element j;
                        Send contribution to Frontal process;
```

Figure 1. Pseudocode for the parallel algorithm using static allocation scheme

```
PARALLEL ALGORITHM (Dynamic allocation scheme #1)

Concurrent processes : Host, Frontal, {Node[i], i=1,2,...,N};
Process Host :
        Initiate process [Frontal, {Node[i] , i=1,2,...,N}];
        Repeat until reception of Stop_signal:
                Loop j : start = 1, end = Number_of_elements, step = 1:
                        Wait for request from a Node process;
                        Send element index j to that Node process;
                Loop k : start = 1, end = N, step = 1:
                        Wait for request from a Node process;
                        Send No_more_element_signal to that Node process;
Process Frontal :
        Loop it : start = 1, end = Max_number_of_iterations, step = 1:
                Send kinematics and mesh to all Node processes;
                Loop k : start = 1, end = Number_of_elements, step = 1:
                        Receive contribution of element k from a Node process;
                        Assemble contribution into active system;
                        Perform frontal elimination;
                Compute solution by backsubstitution;
                If solution has converged then stop iterations;
        Send Stop_signal to Host and to all Node processes;
Process Node[i] :
        Repeat until reception of Stop_signal:
                Receive kinematics and mesh;
                Repeat until reception of No_more_element_signal:
                        Send request to Host;
                        Wait for arrival of My_element;
                        Unless My_element is equal
                        to No_more_element_signal:
                                Compute contribution of My_element;
                                Send contribution to Frontal process;
```

Figure 2. Pseudocode for the parallel algorithm using dynamic allocation scheme #1

Node processes. In the dynamic approach the Host has the additional task to distribute work to the Node processes. The Frontal process is responsible for the assembly of the element contributions and the frontal solution of the *u–v–p* finite element equations. It also performs all the usual book-keeping tasks of a sequential finite element code (e.g. input of data, output of results, management of the iterative process). Finally, the Node processes compute the contributions of the elements. In order to ease implementation and in view of the non-local character of the computations, all Node processes have complete knowledge of the finite element mesh and the current flow kinematics. For free surface problems the mesh changes from one iteration to the next.) Data distribution is therefore not an issue here.

In the first version of the dynamic procedure (Figure 2), the Nodes ask the Host for work and then wait for the Host reply before starting to compute. In the second version (Figure 3) the Nodes are in a sense always a step ahead of the Host. Once they have received the task of

```
PARALLEL ALGORITHM (Dynamic allocation scheme #2)

Concurrent processes : Host, Frontal, {Node[i] , i=1,2,...,N};
Process Host :
    Initiate process [Frontal, {Node[i], i=1,2,...,N}];
    Repeat until reception of Stop_signal:
        Loop j : start = N+1, end = Number_of_elements, step = 1:
            Wait for request from a Node process;
            Send element index j to that Node process;
        Loop k : start = 1, end = N, step = 1:
            Wait for request from a Node process;
            Send No_more_element_signal to that Node process;
Process Frontal :
    Loop it : start = 1, end = Max_number_of_iterations, step = 1:
        Send kinematics and mesh to all Node processes;
        Loop k : start = 1, end = Number_of_elements, step = 1:
            Receive contribution of element k from a Node process;
            Assemble contribution into active system;
            Perform frontal elimination;
        Compute solution by backsubstitution;
        If solution has converged then stop iterations;
    Send Stop_signal to Host and to all Node processes;
Process Node[i] :
    Repeat until reception of Stop_signal:
        Initialize My_element to i;
        Receive kinematics and mesh;
        Repeat until reception of No_more_element_signal:
            Send request to Host without waiting for the answer;
            Compute contribution of My_element;
            Send contribution to Frontal process;
            Wait for arrival of My_element;
```

Figure 3. Pseudocode for parallel algorithm using dynamic allocation scheme #2

working for a new element, they immediately ask the Host in advance for the next element; they then compute the contribution of the currently available element before the reply arrives from the Host.

The differences between the two dynamic allocation schemes are illustrated in Figures 4 and 5, which give a typical graphical snapshot of the process activity. These figures were produced by means of the software tool library described in the next section.[23] In these plots, horizontal 'low-level' lines indicate that a process is doing useful work, while 'high-level' lines show that the process is idle, waiting for reception of a message from another process. Non-horizontal lines connecting two processes indicate a communication event. For example, in Figure 4, Node 1 sends the computed element contribution to the Frontal process (line A). It then sends a request for work to the Host (line B), waits for the answer (line C) and receives from the Host the index of the next element to process (line D). Node 1 can now compute the contribution of the new element (line E) and send it to the Frontal process (line F) when the computations are over.

A similar graph is shown in Figure 5 for the second version of the dynamic approach. For

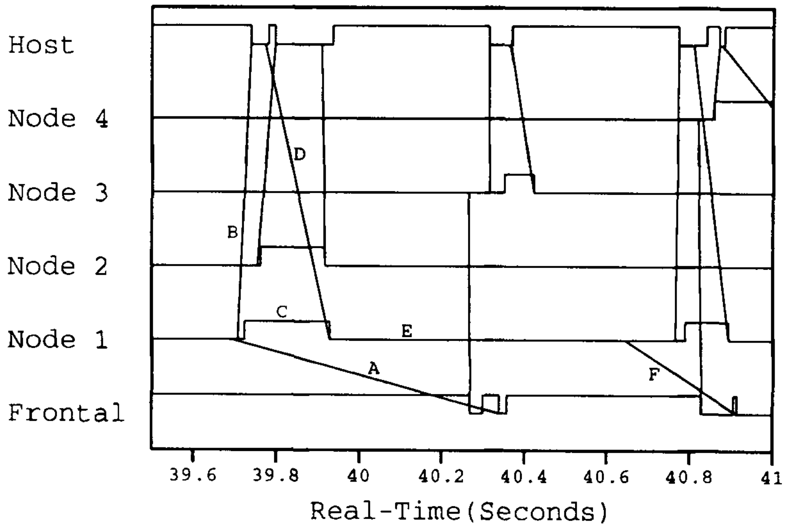

Figure 4. Graphical snapshot of process activity (dynamic scheme #1). Labels A–F mark the following events for Node 1: A + F, sends computed element contribution to Frontal; B, sends request for next element to Host; C, waits; D, receives index of next element from Host; E, computes element contribution; F, sends computed element contribution to Frontal

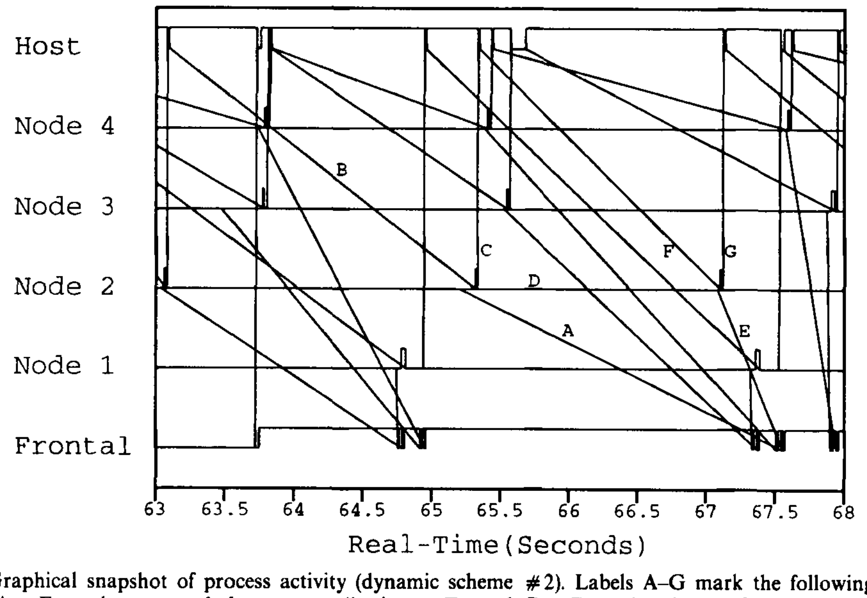

Figure 5. Graphical snapshot of process activity (dynamic scheme #2). Labels A–G mark the following events for Node #2: A + E, sends computed element contribution to Frontal; B + F, receives index of new element from Host; C + G, sends request for next element to Host but does not wait; D, computes element contribution

example, Node 2 sends the element contribution to the Frontal process (line A), receives from the Host the index of the new element (line B), immediately requests the next element (line C) but does not wait for the answer, computes the contribution of the new element (line D), sends it to the Frontal process (line E) and receives the index of the next element from the Host (line F). With this scheme the wait period for the next element is virtually eliminated.

We now describe the issues related to the implementation of the above parallel algorithms on a heterogeneous PVM network of workstations.

4. SOFTWARE TOOLS BASED UPON PVM

The PVM software is a public domain programming environment which allows for the development and execution of parallel distributed applications that consist of many interacting, though mostly independent, components. The software operates on a heterogeneous collection of processing elements that are interconnected by one or more networks. Machines grouped by PVM to form a 'Parallel Virtual Machine' may be workstations, vector or even parallel computers. The PVM software is being developed by the Oak Ridge National Laboratory and is available through netlib. It can easily be installed on most UNIX systems. The main documentation may be found in References 18 and 24. We have used Release 2.4.0 of PVM, which supports a message-passing model of computation. Future releases will allow for the use of physically distributed memory as logically shared memory.

Basic features currently supported by PVM include initiation of processes, non-blocking send, blocking receive, test for reception of message of given type, and synchronization barriers. Parallel algorithms implemented with PVM may be written in the standard procedural languages FORTRAN or C. The PVM software allows for communication between any processes, and any PVM process may start or stop any other PVM process.

In the PVM environment, any number of processes may be started on any of the participating processors. When initiating a process, the user may specify that PVM is allowed to determine itself on what processor to start the process, or that PVM must select a machine with a given processor architecture, or else that PVM must use a specific computer. Through the use of the SUN XDR library, PVM performs data conversion between machines with different data formats. It also recognizes processes running on the same processor, thereby avoiding unnecessary communication over the network.

In order to use PVM and before any interprocess communication may take place, one must initiate the PVM daemon pvmd on every machine in the PVM network. This is done automatically by running pvmd on any one of the participating computers using the name of the 'host'-file as argument. The latter defines all the participating machines.

We have installed the PVM software on a wide variety of workstations without encountering any significant difficult (Table I). Problems arise, however, while porting PVM-based application codes to highly heterogeneous configurations. Henriksen[23] gives a detailed account of such difficulties, which are related to the current version of PVM itself and to portability of the FORTRAN 77 language used in this work.

The above issues, combined with the software development work in our group using the INTEL iPSC/860 hypercube, led Henriksen[23] to develop a new set of software tools, called the Extended Host/Node Hypercube-Model Library. This library is based upon PVM tools. Its programming paradigm is that of a Host/Node structure where a single host process initiates a number of identical node processes. In the case where the number of node processes is an integral power of two, the library assigns a 'cube-dimension' to the PVM configuration, thereby allowing the INTEL iPSC code to view the PVM machine as a 'standard' hypercube. This basic Host/Node Hypercube model is extended to allow any process to initiate additional processes. Only one host process is allowed though.

The library developed by Henriksen[23] consists of 17 user subroutines and more than 50 internal routines. In addition to providing the user with all the PVM flexibility, it offers additional features which include checking of error conditions, improved identification of

Table I. List of workstations on which we have installed PVM, together with the corresponding operating systems and FORTRAN compilers

Workstation	Operating system	FORTRAN compiler
DG AViiON 200	DG/UX 5.4	Hills FORTRAN-88000 1.8.6
DG AViiON 6020	DG/UX 4.31	Hills FORTRAN-88000 1.8.4.09
DECstation 3100	ULTRIX V4.2 (Rev.96)	DEC FORTRAN V3.0-2
DECstation 5000	ULTRIX V4.2 (Rev.96)	DEC FORTRAN V3.0-2
HP 9000/720	HP-UX A.08.07	HP FORTRAN 8.07
HP 9000/730	HP-UX A.08.07	HP FORTRAN 8.07
IBM RS6000/320	AIX 3.2	AIX FORTRAN 2.2
SG IRIS	IRIX System V 3.2.1	FTN 3.3
SUN 3	SunOS 4.1	Sun FORTRAN 1.3.1
SUN SPARC1	SunOS 4.1	Sun FORTRAN 1.3.1
SUN SPARC2	SunOS 4.1	Sun FORTRAN 1.3.1

message sender, global function calls, timing routines, a debugging mode and a graphical communication timer. These features were found very useful in porting codes written for the INTEL hypercube to PVM configurations. They are undoubtedly helpful in the *ab initio* development of PVM programmes as well.

The sequential numerical algorithms described in Section 2 are implemented in the POLYFLOW package developed in Louvain-la-Neuve.[19] This commercial application code contains about 500 k lines of FORTRAN 77. Fortunately, its modular structure allows for a relatively easy implementation of the parallel algorithms described in Section 3. An INTEL iPSC/860 implementation using static allocation has been developed and exploited on a 128-processor configuration by Aggarwal and Keunings.[14] The PVM parallel version of the POLYFLOW code has been developed from its INTEL iPSC counterpart using the tool library developed by Henriksen.[23]

5. HETEROGENEOUS PVM CONFIGURATIONS

The experiments reported below involved three heterogeneous PVM configurations which are defined in Table II. Also listed are the PVM processes assigned to the different workstations. Configurations A and B involve the same workstations but a different allocation of PVM processes. Indicative relative speeds of the various CPUs are quoted for typical sequential POLYFLOW runs.

The participating workstations are linked through Ethernet subnetworks that are part of our departmental network (Figure 6). The TCP/IP protocol is used, with a software bandwidth limitation of 2 Mbit s^{-1}. All workstations have some local disk space for swapping and moderate file storage. A large NFS disk space is attached to the Data General AViiON server. Note that the DEC 3100 workstations access the NFS main disk and communicate with the SG IRIS through two routers, one being itself a participating workstation.

The workstations listed in Table II are heavily used during regular office hours for both teaching and research activities; they are almost never completely idle. Clearly the computing environment shown in Figure 6 has not at all been designed with parallel distributed computing in mind.

Table II. The three PVM configurations used in the experiments

	Workstation	Relative speed
PVM configuration A		
Host	DECstation 5000	2
Frontal	DECstation 5000	2
Node 1: element contributions	DECstation 3100 A	1
Node 2: element contributions	DECstation 3100 B	1
Node 3: element contributions	DECstation 3100 C	1
PVM configuration B		
Host	DECstation 5000	2
Frontal	DECstation 5000	2
Node 1: element contributions	DECstation 5000	2
Node 2: element contributions	DECstation 3100 A	1
Node 3: element contributions	DECstation 3100 B	1
Node 4: element contributions	DECstation 3100 C	1
PVM configuration C		
Host	SG IRIS	5
Frontal	SG IRIS	5
Node 1: element contributions	SG IRIS	5
Node 2: element contributions	DECstation 5000	4
Node 3: element contributions	DECstation 3100 A	2
Node 4: element contributions	DECstation 3100 B	2
Node 5: element contributions	DECstation 3100 C	2
Node 6: element contributions	DG 200 A	1
Node 7: element contributions	DG 200 B	1

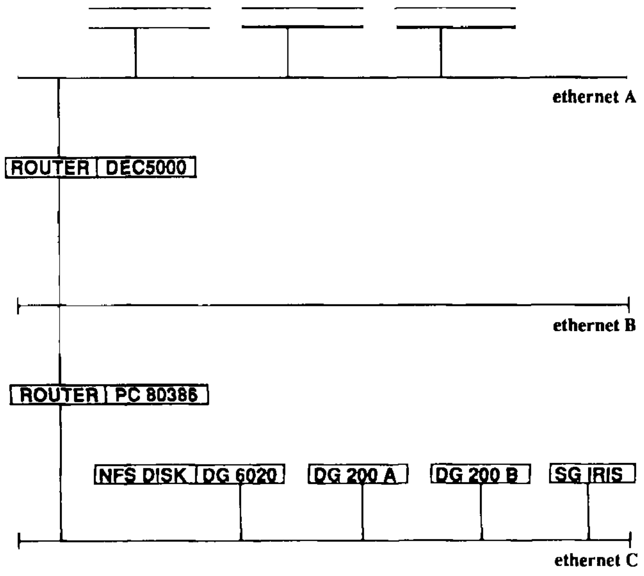

Figure 6. Network topology of workstations used in this work

6. PARALLEL EFFICIENCY AND TIMING MEASUREMENTS

When using a parallel computer with P identical processors, one defines the parallel efficiency of the algorithm as the speed-up divided by P. Speed-up is itself defined as the time for running the sequential algorithm divided by the time needed to run the parallel algorithms on the P processors.[7]

We adopt the following approach to define parallel efficiency when using a heterogeneous PVM network of workstations.

Step 1 For each workstation w_i of the PVM network, measure the time T_i needed to run a typical 'small' flow problem with the sequential algorithm.

Step 2 Select a particular workstation w_j, and estimate in 'w_j' units the compute power CP_j of the PVM configuration:

$$\mathrm{CP}_j = T_j \sum_i \frac{1}{T_i}. \tag{3}$$

Step 3 Measure the time T_j^{seq} needed to run the full problem under investigation on workstation w_j using the sequential algorithm.

Step 4 Measure the time T^{PVM} needed to run the full problem under investigation on the PVM network using the parallel algorithm.

Step 5 Compute the parallel efficiency ε of the algorithm as

$$\varepsilon = \frac{T_j^{\mathrm{seq}}}{T^{\mathrm{PVM}}} \frac{1}{\mathrm{CP}_j}. \tag{4}$$

The classical definition of efficiency is recovered when identical processors are used. In this case the compute power CP reduces to the number P of processors. For heterogeneous systems the compute power CP_j of a PVM configuration is an estimate of the number of 'w_j-equivalent' processors in the configuration. We have measured $\mathrm{CP}_{\mathrm{DEC5000}} = 2{\cdot}6$ for PVM configurations A and B and $\mathrm{CP}_{\mathrm{SGIRIS}} = 3{\cdot}8$ for PVM configuration C (Table II).

The definition (4) of efficiency is based on measurement of elapsed times. It is not easy to obtain accurate timing results on heterogeneous PVM configurations that are themselves part of a wider departmental network. The results reported in the next section were obtained with the participating workstations being entirely devoted to the PVM experiments. Of course, this statement cannot be absolutely correct. Indeed, the UNIX operating system is always managing background processes. In addition, the PVM configurations are not isolated systems; collisions can arise with other workstations that use NFS. In order to illustrate this point, we show in Figure 7 the measured transfer rate of 20 consecutive write-buffer calls (buffer size is 800 kB) from one of the DEC 3100 workstations to the DG 6020 disk. We see that the transfer rate varies approximately by a factor of three over an interval of a few minutes.

Another difficulty is related to the estimation of the compute power CP_j of the PVM configurations. As mentioned above, this measure is based upon 'small' flow problems that are run in sequential mode on all the participating workstations. It would be more accurate to use the timings for the actual (large-size) problem under investigation. Such measurements are not feasible, however, with the less powerful CPUs of the PVM network.

On the basis of many careful experiments, we estimate that the margin of uncertainty in the reported efficiency data is 5%. One should not expect anything near to this satisfactory figure if the PVM code were run in competition with other users' processes.

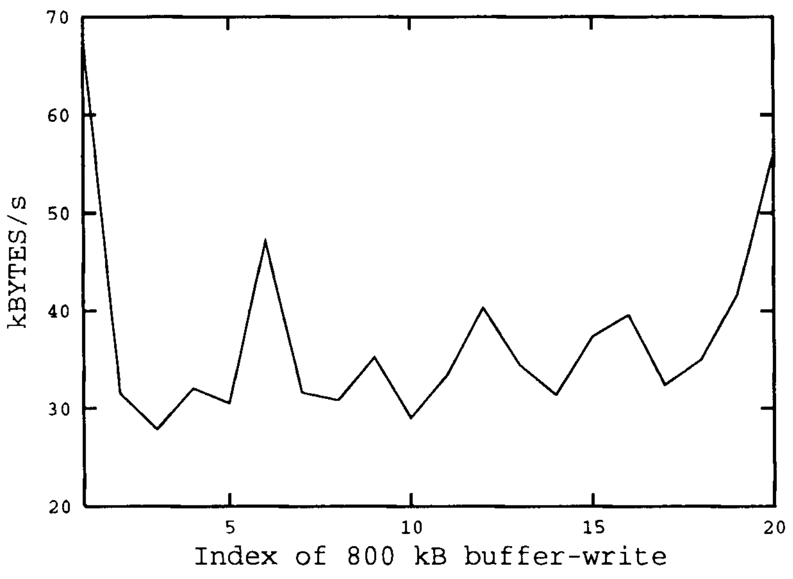

Figure 7. Transmission rates for 20 consecutive writes of an 800 kB buffer from a DEC 3100 to the NFS disk on the DG 6020 (Figure 6)

7. STICK/SLIP AND CONTRACTION FLOW PROBLEMS

The results discussed in Section 8 have been obtained for two complex flows of relevance to the community of computational rheologists, namely the stick/slip and contraction flow problems.[3] Figure 8 shows the computational domains and boundary conditions, together with representative finite element discretizations for the *u–v–p* problem.

Several meshes with up to 512 elements were used. Although this is not an impressive figure for a standard Stokes flow problem, the fact that the memory integral (2) must be computed at each Gauss point during the iterative procedure makes such computations very lengthy. Indeed, this level of mesh refinement is the highest that Goublomme *et al.*[13] could afford in their recent sequential calculations with the K.B.K.Z. model. For example, a single non-linear iteration with a 450-element contraction mesh consumes 20 min of CPU time on the SG IRIS. Typically between 30 and 100 non-linear iterations are needed to achieve convergence in such flow problems.

8. PARALLEL EFFICIENCY RESULTS

Our purpose is to discuss the parallel efficiency data obtained on the three PVM configurations of Table II. Information on the physical and numerical aspects of the simulations can be found in References 11–13.

Figure 9 shows the parallel efficiency obtained for the stick/slip problem using PVM configurations A and B. Results are shown for four finite element meshes. We consider here the results obtained with the static allocation scheme and the first version of the dynamic scheme (Figures 1 and 2). One should note that these efficiency results have been obtained for the first few iterations of the non-linear procedure, starting from the Newtonian flow field. During these early iterations the flow kinematics do not change much from one iteration to the next. This of course is the best possible situation for the static allocation scheme. Nevertheless, inspection of

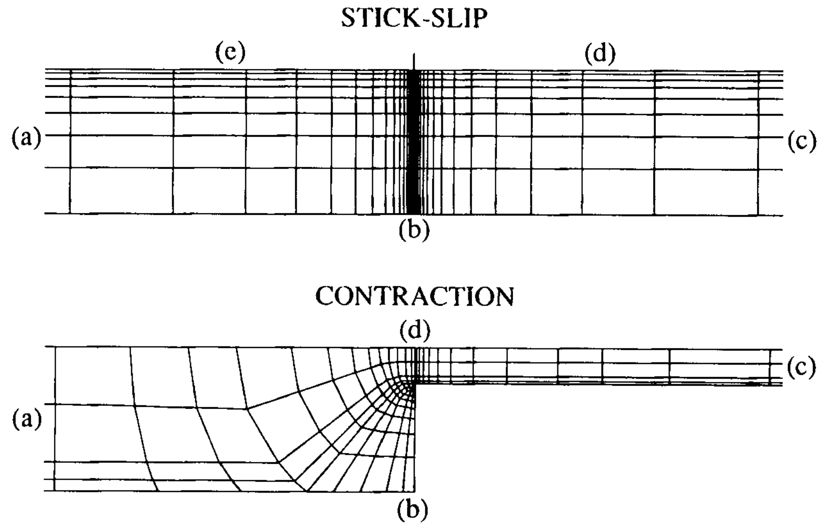

Figure 8. Partial view of typical finite element discretizations. Boundary conditions for stick/slip flow are: fully developed flow at inlet (a) and outlet (c), plane of symmetry (b), slip surface (d) and no-slip wall (e). Boundary conditions for contraction flow are: fully developed flow at inlet (a) and outlet (c), no-slip wall (b) and axis of symmetry (d)

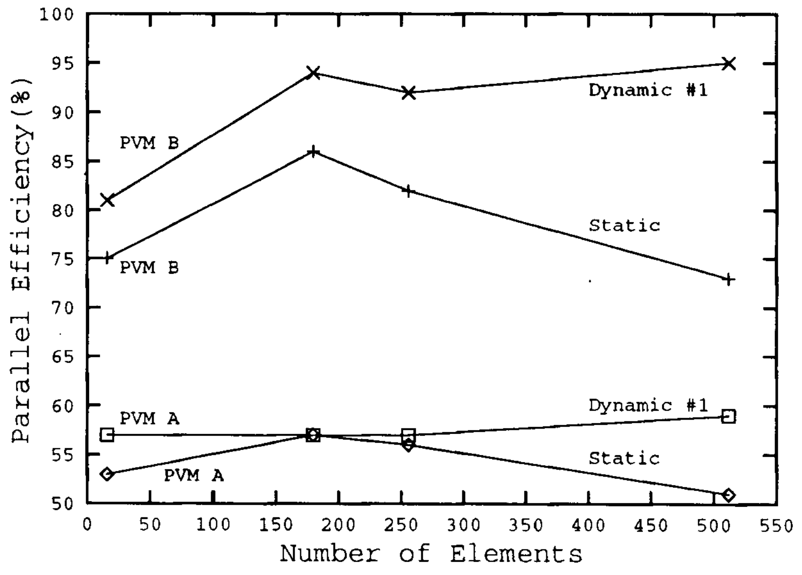

Figure 9. Parallel efficiency as a function of problem size (stick/slip problem)

Figure 9 reveals the superiority of the dynamic allocation approach, especially when the size of the problem (i.e. the number of elements) increases. One also observes that efficiency levels in the range 80%–95% are obtained with PVM configuration B and the dynamic approach.

With PVM configuration A, the efficiency levels off at 59%. The reason for this is easy to grasp. PVM configuration A makes bad use of the available compute resources, since only two very 'light-weight' processes (Host and Frontal) run on the DEC 5000, which is the most powerful

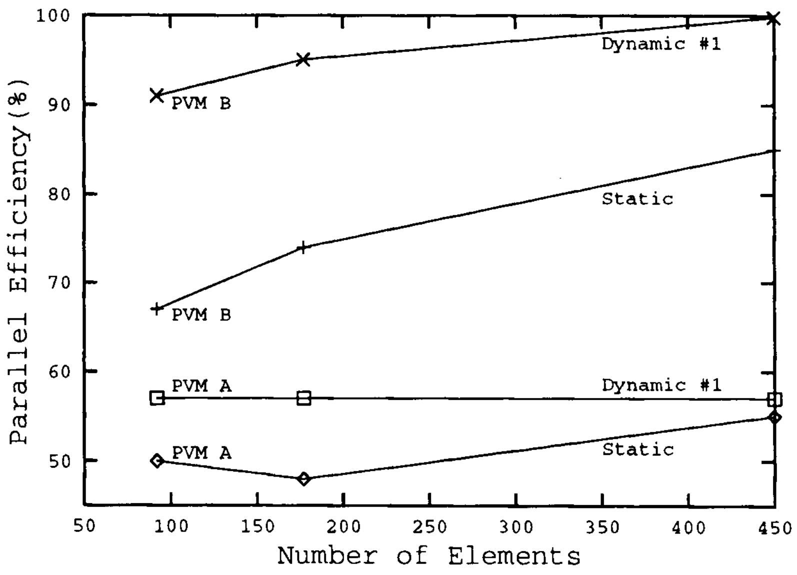

Figure 10. Parallel efficiency as a function of problem size (contraction problem)

CPU of the configuration (Table II). PVM configuration B, on the other hand, allocates also one of the 'heavy-weight' Node processes to the DEC 5000. In fact, the maximum available efficiency that can be reached with PVM configuration A is 60%. This value is computed by disregarding the contribution of the DEC5000 in equation (3).

Figure 10 shows similar results for the contraction flow problem. Here three meshes have been used. Again very high efficiency levels are obtained with PVM configuration B and the dynamic allocation procedure; the static scheme, though less efficient, behaves in a satisfactory manner. With PVM configuration A the dynamic scheme is, with all three meshes, very close to the maximum achievable efficiency (60%). The static scheme is really at its best in this case, since the Node processes run on identical CPUs and not much change in kinematics occurs during the first non-linear iterations.

Let us now compare the results obtained with PVM configurations B and C. In the sequel we shall only consider the two versions of the dynamic allocation approach. Figure 11 shows the efficiency results obtained for the stick/slip problem and dynamic scheme #1. Although frequent interprocess communications occur with the dynamic approach very high efficiency levels are reached with both PVM configurations when the number of elements increases. The intricate load-balancing issue described in Section 3 is thus treated automatically in a rather satisfactory way. When inspecting Figure 11, one should not consider with too much scrutiny the crossing of the two curves obtained with PVM configurations B and C. Indeed, as discussed previously, the level of uncertainty in the reported efficiency values is of the order of 5%. In view of the excellent efficiency results obtained on the stick/slip flow problem with PVM configurations B and C and dynamic scheme #1, we have not deemed it necessary to test dynamic scheme #2 in this context.

Let us now consider the contraction flow problem (Figure 12). With PVM configuration B and dynamic scheme #1, the efficiency is found to increase monotonically when the problem size increases, with values higher than 90%. Here again one does not expect (nor does one need) a major improvement by using dynamic scheme #2. The situation is somewhat different with

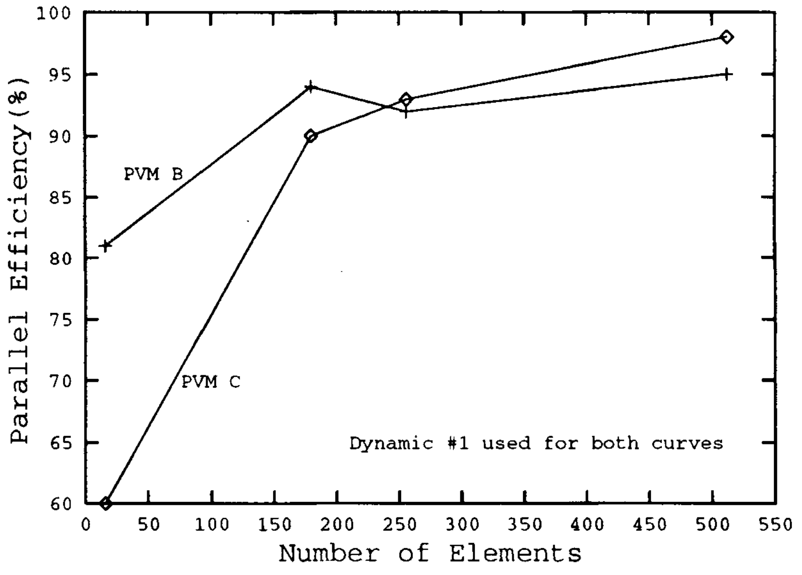

Figure 11. Parallel efficiency as a function of problem size (stick/slip problem): dynamic allocation scheme #1

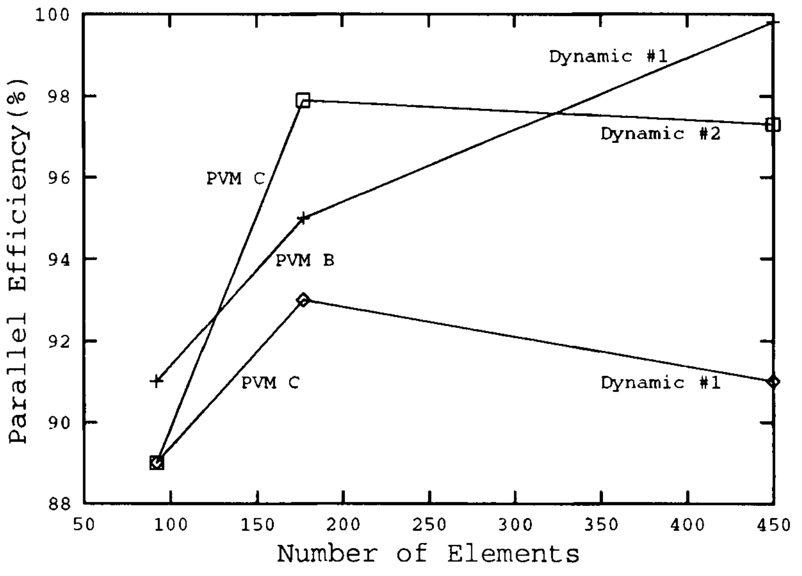

Figure 12. Parallel efficiency as a function of problem size (contraction problem): dynamic allocation schemes #1 and #2

PVM configuration C. Using dynamic scheme #1, we find that efficiency levels off and eventually decreases with increasing problem size. We attribute this behaviour mainly to the idle cycles wasted by the Node processes when waiting for work (see Figure 4). Dynamic scheme #2 virtually eliminates those idle periods (see Figure 5). As a result, efficiency remains monotonically increasing and does reach very high values when the number of elements increases.

9. CONCLUSIONS

This work demonstrates that large-scale computational mechanics simulations can be conducted efficiently on a 'Parallel Virtual Machine' that groups a number of interconnected workstations. Although the present paper focuses on a particular application, namely the simulation of memory fluids described by integral models, the proposed parallel methodology is relevant to other problems that involve a computer-intensive treatment of interval variables.

We have found the public domain PVM software to be a powerful tool that allows the use of a highly heterogeneous network of workstations as a single computing resource. The PVM software is stable, flexible and easy to use. The minor flaws identified during the course of this work have been overcome in the software library developed by Henriksen.[23] Extensions to the PVM package are straightforward to add, resulting in environments that can compete with commercial software products.

Although the idea of grouping available workstations in a 'Parallel Virtual Machine' is very attractive, one should not underestimate the many potential sources for loss of parallel efficiency. In particular, these PVM configurations may not always be viewed as isolated systems and the workload of the participating processors may change during the course of a PVM run. Clearly, dynamic allocation procedures of workload to processors, such as those developed in the present paper, are essential for the efficient use of heterogeneous PVM configurations made of interconnected workstations.

ACKNOWLEDGEMENTS

The post-doctoral research of Dr. P. Henriksen is supported financially by the BRITE/EURAM European Programme. We wish to thank our collaborator Dr. R. Aggarwal for his help with the INTEL iPSC/860 parallel implementation of the POLYFLOW package. We also thank our local UNIX expert Mr. E. Mulemangabo for his assistance. The results presented in this paper have been obtained within the framework of Interuniversity Attraction Poles initiated by the Belgian State, Prime Minister's Office, Science Policy Programming. The scientific responsibility rests with its authors.

REFERENCES

1. M. J. Crochet, A. R. Davies and K. Walters, *Numerical Simulation of Non-Newtonian Flow*, Elsevier, Amsterdam, 1984.
2. M. J. Crochet, 'Numerical simulation of viscoelastic flow: a review', *Rubber Chem. Technol. Am. Chem. Soc.*, **62**, 426–455 (1989).
3. R. Keunings, 'Simulation of viscoelastic fluid flow', in C. L. Tucker III (ed.), *Fundamentals of Computer Modeling for Polymer Processing*, Carl Hanser, 1989, pp. 402–470.
4. R. Keunings, 'Progress and challenges in computational rheology', *Rheol. Acta*, **29**, 556–570 (1990).
5. P. Moldenaers and R. Keunings (eds.), *Theoretical and Applied Rheology*, Vols 1 and 2, Elsevier, Amsterdam, 1992.
6. C. L. Tucker III (ed.), *Fundamentals of Computer Modeling for Polymer Processing*, Carl Hanser, Munich, 1989.
7. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
8. D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice–Hall, Englewood Cliffs, NJ, 1989.
9. R. B. Bird, R. C. Armstrong and O. Hassager, *Dynamics of Polymeric Liquids*, Vol. 1, 2nd edn. Wiley, New York, 1987.
10. R. I. Tanner, *Engineering Rheology*, Clarendon, Oxford, 1985.
11. S. Dupont, J.-M. Marchal and M. J. Crochet, 'Finite element simulation of viscoelastic fluids of integral type', *J. Non-Newtonian Fluid Mech.*, **17**, 157–183 (1985).
12. S. Dupont and M. J. Crochet, 'The vortex growth of a K.B.K.Z. fluid in an abrupt contraction', *J. Non-Newtonian Fluid Mech.*, **21**, 81–91 (1988).
13. A. Goublomme, B. Draily and M. J. Crochet, 'Numerical prediction of extrudate swell of a high-density polyethylene', *J. Non-Newtonian Fluid Mech.*, **44**, 171–195 (1992).

14. R. Aggarwal and R. Keunings, 'Finite element simulation of memory fluids on message passing parallel computers', in R. B. Pelz, A. Ecer and J. Haüser (eds.), *Proc. Parallel CFD '92*, Elsevier, Amsterdam, 1993, pp. 1–8.
15. O. Zone and R. Keunings, 'Direct solution of two-dimensional finite element equations on distributed memory parallel computers', in M. Durand and F. El Dabaghi (eds.), *High Performance Computing II*, North-Holland, Amsterdam, 1991, pp. 333–344.
16. O. Zone, R. Keunings and D. Roose, 'Numerical algorithms for the direct solution of finite element equations on a distributed memory parallel computer', in A. Bode (ed.), *Lecture Notes in Computer Science*, Vol. 487, *Distributed Memory Computing*, Springer, Berlin, 1991, pp. 294–303.
17. R. Aggarwal, P. Henriksen, R. Keunings, D. Vanderstraeten and O. Zone, 'Numerical simulation of non-Newtonian flow on MIMD parallel computers', in Ch. Hirsch, J. Périaux and W. Kordulla (eds), *Computational Fluid Dynamics '92*, Vol. 2, Elsevier, Amsterdam, 1992, pp. 1139–1146.
18. A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek and V. S. Sunderam, 'A user's guide to PVM Parallel Virtual Machine', *Tech. Rep. ORNL/TM-11826*, Oak Ridge National Laboratory, 1991.
19. M. J. Crochet, B. Debbaut, R. Keunings and J. M. Marchal, 'Polyflow: a multi-purpose finite element program for continuous polymer flows', in K. T. O'Brien (ed.), *Computer Modeling for Extrusion and Other Continuous Polymer Processes*, Carl Hanser, Munich, 1992, pp. 25–50.
20. M. Géradin, J. C. Golinval and J. P. Mascarell, 'Three dimensional turbine blade analysis in thermo-viscoplasticity', *Rech. Aérosp.*, **1989–5**, 51–57 (1989).
21. J. R. Rosenberg, M. M. Denn and R. Keunings, 'Simulation of non-recirculating flows of dilute fiber suspensions', *J. Non-Newtonian Fluid Mech.*, **37**, 317–345 (1990).
22. J. Rosenberg and R. Keunings, 'Numerical integration of differential viscoelastic models', *J. Non-Newtonian Fluid Mech.*, **39**, 269–290 (1991).
23. P. Henriksen, 'Software tools based upon PVM for parallel computing on a heterogeneous network of workstations', *CESAME Rep.*, Université Catholique de Louvain, 1992.
24. G. A. Geist and V. S. Sunderam, 'Network based concurrent computing on the PVM system', *Tech. Rep. ORNL/TM-11760*, Oak Ridge National Laboratory, 1991.