Object-oriented parallelization of explicit structural dynamics with PVM

Petr Krysl[*]and Ted Belytschko[†]

May 20, 1997

Abstract

Explicit finite element programs for non-linear dynamics are of rather simple logical structure. If the inherent characteristics of this logic are exploited in the design and implementation of a parallel computer program, the result can be a lucid, extendible, and maintainable code. The design of an explicit, finite element, structural dynamics program is discussed to some detail, and it is demonstrated that the program lends itself easily to parallelization for heterogeneous workstation clusters, or massively parallel computers, running the PVM software. The design is documented by C-language fragments.

Keywords: finite element method, explicit dynamics, parallelization, PVM

Introduction

As users of finite element programs try to obtain solutions to larger and larger problems, they encounter major technical barriers; limitations in memory or in CPU speed, or both. One of the remedies is parallelization [13]. Conversion of an existing program to run on a network of workstations is in many cases the least expensive solution (compare with Baugh and Sharma [2]).

In the present paper, we deal with a non-linear finite element program for explicit time integration of the momentum equations in structural dynamics. The program can be run on a heterogeneous cluster of workstations, and/or nodes of multiprocessor machines, or on massively parallel machines such as the SP-2 or Paragon. We apply the usual message-passing parallelization technique based in our case on the message-passing library PVM, Parallel Virtual Machine; see Geist *et al.* [17]. One of the reasons for this choice was that PVM is becoming a *de facto* standard among message-passing libraries due to its widespread usage and its support by many major computer vendors.

[*]Research Associate, Civil Engineering, Northwestern University

[†]Walter P. Murphy Professor of Civil and Mechanical Engineering, Northwestern University

A network of workstations can be viewed as an MIMD (multiple-instruction, multiple-data) machine with distributed memory. Parallelization of serial programs for this type of parallel machine is notoriously difficult. One of the reasons is that the parallelization is mostly done manually, by augmenting the serial program control by message-passing directives. Automatic parallelization tools are currently not available [13].

In order to preserve the investments in the serial finite element program, it is clearly desirable to parallelize the existing code, at the smallest possible cost, and in such a way that it is possible to maintain both the serial and the parallel versions of the program. The serial finite element code parallelized in the present effort was written by the first author in an "object-oriented" programming style [10]. We wish to show how this can lead to a clean implementation of the parallel version of this program.

The outline of the paper is as follows. First, we review the related research in Section 1. We briefly discuss domain decomposition and the characteristics of computer networks with respect to parallelization. Next, in Section 2, we show how the abstract algorithm of an explicit time-stepping scheme can be transformed into a high-level representation in terms of programming design and implementation. We construct an *integrator* object, which provides the appropriate behavior, while encapsulating the "knowledge" corresponding to the mathematical algorithm. The implementation of the serial time-stepping driver is described in Section 3. The parallel time-stepping algorithm is then formulated in Section 4. It is shown that the star-shaped configuration of a central "master" program communicating with a number of "workers" is well suited to the parallel formulation in the target hardware and software environment. The implementation of the parallel integrator is then described. It is demonstrated that the master-worker distinction is cleanly reflected in two different implementations of the integrator, which correspond either to the master (Section 5), or to the worker (Section 6). Section 7 discusses two algorithms for the computation of the inertial properties of the structural system, the ghost elements and the exchange algorithm. Since maintainability is considered vital, we implement the parallel version by minimal modifications of the serial program. This is achieved by using the macro-facilities of the C-language. It is shown that as a result only a single source code needs to be maintained.

1 Related research

1.1 Domain decomposition

The underlying concept of parallelization in the form considered here, i.e., *data decomposition*, is currently a subject of lively research. Data decomposition is represented by a splitting of the finite element mesh. The decomposition, or partitioning, can be dynamic, i.e., it may change during the computation, or it can be static. Since we are dealing with non-linear problems, it would be advantageous to use dynamic partitioning; the effort to compute the internal forces may change dramatically during the computation, e.g., because of inelastic constitutive laws. However, dynamic load balancing is a rather complicated and an evolving issue, for which no simple solutions exist; see, e.g., Farhat and Lesoinne [16], Ecer *et al.* [14], Chien *et al.* [9], and Özturan *et al.* [26], and Vanderstraeten and Keunings [29]. Moreover, it is not essential with respect to the goals of the present paper. Therefore, we restrict our presentation to static domain decomposition.

1.2 Parallelization for computer networks

Let us first note that local area networks, which are assumed to be used for execution of the present program, have certain properties with respect to communication. These characteristics should be taken into account to devise an efficient algorithm [2, 13]. The time to send a message through the network can be approximately written as a linear function [see, e.g., Abeysundara and Kamal [1]]

$$T_m = T_l + \bar{T}_b B \ , \tag{1}$$

where T_m is the time needed to communicate a message of B bytes, T_l is the network latency time (usually several milliseconds), and $\bar{T}_b$ is the time needed to transfer one byte of data. For Ethernet networks, which are the most frequently used network type for connecting engineering workstations, $T_l \approx 1000 \times \bar{T}_b$. Thus, the larger the data packets, the better the efficiency achieved (the goal is to "eliminate" the effect of network latency).

While some researchers have opted for RPC (remote procedure call) based message passing [18], or even for direct use of the TCP/IP communications level [2], we are of the opinion that the programming effort is much smaller with a higher-level communication library. Also, ease of maintenance and portability are enhanced. The parallelization in our case was supported by the PVM library. The underlying mechanisms of PVM are described in Geist *et al.* [17]. PVM enables a collection of workstations and/or nodes of multiprocessor machines to act as a single parallel computer. It is a library, which handles the message-passing, data conversions, and task scheduling. The applications are decomposed into a set of tasks, which execute in parallel (with occasional synchronization). The programmer has to write the "prototypes" of the tasks as serial programs. The tasks are instantiated on the machines to be included in the parallel virtual machine by starting up the prototypes. Data exchanges and communication are explicitly programmed.

Although efforts to parallelize linear static finite element analysis have rather little in common with the present work (the main concern is the solution of large systems of linear equations), they are of conceptual interest. Hudli and Pidaparti [18] have dealt with distributed linear static finite element analysis using the client-server model. They have used the RPC library together with (non-portable) lightweight processes to implement their algorithms. The performance issues of the network-based parallel programs were discussed in Baugh and Sharma [2] on the example of linear statics computation. In order to isolate effects, they implemented their parallel algorithm directly on the TCP/IP layer. They conclude that (as expected) the parallelization on local area networks should be rather coarse grained, because of the large communication overhead associated with the relatively slow networks. Also, they note difficulties associated with dynamic load balancing, because of the widely differing characteristics of the participating computers.

1.3 Non-linear parallel dynamics

For a comprehensive account of parallel non-linear dynamics, the reader is referred to the review by Fahmy and Namini [15]. Due to the fundamental differences in hardware characteristics, we choose not to discuss implementations of parallel finite element algorithms on vector, or shared-memory machines.

Yagawa *et al.* [30] have investigated non-linear implicit dynamics with domain decomposition on a network of engineering workstations or supercomputers. Although aimed at implicit structural dynamics, this paper is of interest here, as it illustrates the fundamental differences between data structures, and algorithms used in implicit and explicit analysis. The authors have also addressed the issues of dynamic load balancing through a special processor management.

Namburu *et al.* [24] have investigated explicit dynamics on a massively parallel machine. They have used their own variant of an explicit algorithm.

Malone and Johnson [20, 21] have dealt with explicit dynamics of shells on massively parallel machines (IPSC/i860). They have concentrated on the formulation of a contact algorithm. The mesh is in their algorithm split between processors, and the individual interface nodes are assigned uniquely to processors. Thus it is the responsibility of the processors to keep track of which data to send to which processor, and, more importantly, many small messages need to be sent.

Chandra *et al.* [8] have investigated an object-oriented methodology as applied to transient dynamics. They show how to find inherent parallelism in the interaction of a large number of particles, and they establish an object-oriented data structure for this kind of computation.

2 Explicit integration in time

In this section, we inspect the general properties of the (explicit) central difference formula. We construct an abstraction, which cleanly translates the mathematical design into the programming language implementation. The resulting *serial time-stepping* driver, the *integrator*, is then reformulated to incorporate parallelism.

Interestingly, there are several alternative formulations of the explicit central difference integration; see, for instance, Belytschko [3], Hughes and Belytschko [19], Ou and Fulton [25], Belytschko *et al.* [4], Dokainish and Subbaraj [12], Simo *et al.* [28], and Chung and Lee [11]. In order to be able to exploit fully the potential of the explicit time stepping scheme, matrix inversions must be avoided. Thus the goal is to obtain the primary unknowns by solving a system with a diagonal system matrix. When we consider a general damping, the above goal can be achieved only if the damping terms are not present on the left-hand side. If damping is not present (or if it is expressed by a diagonal matrix), more specialized forms of the central difference algorithm can be used. The difficulty of the coupling due to the presence of both the velocity and the acceleration in the equations of motion is then avoided ab initio.

2.1 Central difference formulas

The symbols in the below formulas are: $\boldsymbol{M}$ the mass matrix (constant, and diagonal in all cases), $\boldsymbol{C}$ the damping matrix, which can be in general function of the velocities, $\boldsymbol{u}_\tau$, $\dot{\boldsymbol{u}}_\tau$, $\ddot{\boldsymbol{u}}_\tau$, the vectors of displacements, velocities, and accelerations, respectively, $\boldsymbol{f}_\tau^{ext}$ the external loads, and $\boldsymbol{f}_t^{int}$ the nodal forces corresponding to the stresses, all at the time τ.

The first variant of the central difference scheme can be obtained from the Newmark β implicit algorithm by inserting limit values of the parameters[28]. The Newmark algorithm is summarized in equations (2)–(3).

$$\boldsymbol{u}_{t+\Delta t} = \boldsymbol{u}_t + \Delta t \dot{\boldsymbol{u}}_t + \Delta t^2 \left[\left(\frac{1}{2} - \beta \right) \ddot{\boldsymbol{u}}_t + \beta \ddot{\boldsymbol{u}}_{t+\Delta t} \right] , \tag{2}$$

$$\dot{\boldsymbol{u}}_{t+\Delta t} = \dot{\boldsymbol{u}}_t + \Delta t \left[(1-\gamma) \ddot{\boldsymbol{u}}_t + \gamma \ddot{\boldsymbol{u}}_{t+\Delta t} \right] . \tag{3}$$

The algorithm becomes explicit (central difference scheme) for $\gamma = \frac{1}{2}$ and $\beta = 0$. Its central difference variant can then be written as in equations (4) to (9).

Variant 1 (Newmark):

1. Calculate velocities at time t:

$$\dot{\boldsymbol{u}}_t = \dot{\boldsymbol{u}}_{t-\Delta t} + \frac{\Delta t}{2} \left(\ddot{\boldsymbol{u}}_t + \ddot{\boldsymbol{u}}_{t-\Delta t} \right) . \tag{4}$$

2. Calculate displacements at time $t + \Delta t$:

$$\boldsymbol{u}_{t+\Delta t} = \boldsymbol{u}_t + \Delta t \dot{\boldsymbol{u}}_t + \frac{\Delta t^2}{2} \ddot{\boldsymbol{u}}_t \tag{5}$$

3. Calculate effective loads:

$$\widehat{\boldsymbol{f}}_{t+\Delta t} = \boldsymbol{f}^{ext}_{t+\Delta t} - \boldsymbol{f}^{int}_{t+\Delta t} - \boldsymbol{C} \dot{\boldsymbol{u}}^p_{t+\Delta t} , \tag{6}$$

with either

$$\dot{\boldsymbol{u}}^p_{t+\Delta t} = \dot{\boldsymbol{u}}_t + \Delta t \ddot{\boldsymbol{u}}_t , \tag{7}$$

or

$$\dot{\boldsymbol{u}}^p_{t+\Delta t} = (\boldsymbol{u}_{t+\Delta t} - \boldsymbol{u}_t)/\Delta t \tag{8}$$

4. Solve for accelerations at time $t + \Delta t$ from:

$$\boldsymbol{M}\ddot{\boldsymbol{u}}_{t+\Delta t} = \widehat{\boldsymbol{f}}_{t+\Delta t} \ . \tag{9}$$

The second time-stepping formula is the central difference algorithm in the form as presented for example by Park and Underwood [27]. The formulas correspond to an iterative correction of the velocity present in the equations of motion. The parameters β and γ are integration constants, and α is an averaging factor ($\alpha = 1$, $\beta = 1$, $\gamma = 0$ for a lightly damped structure, $\alpha = 1/2$, $\beta = 1$, $\gamma = 0$ for a heavily damped structure).

Variant 2:

1. Predict velocity $\dot{\boldsymbol{u}}_t$ as

$$\dot{\boldsymbol{u}}_t = \alpha\dot{\boldsymbol{u}}_t^c + (1-\alpha)\dot{\boldsymbol{u}}_t^p \ , \tag{10}$$

with

$$\dot{\boldsymbol{u}}_t^c = \dot{\boldsymbol{u}}_{t-\Delta t/2} + \frac{\Delta t}{2}\left[\beta\ddot{\boldsymbol{u}}_t^p - (1-\beta)\ddot{\boldsymbol{u}}_{t-\Delta t}\right] \ , \tag{11}$$

and

$$\dot{\boldsymbol{u}}_t^p = \dot{\boldsymbol{u}}_{t-\Delta t/2} + \gamma\Delta t\ddot{\boldsymbol{u}}_{t-\Delta t} \tag{12}$$

$$\boldsymbol{M}\ddot{\boldsymbol{u}}_t^p = \boldsymbol{f}_t^{ext} - \boldsymbol{f}_t^{int} - \boldsymbol{C}\dot{\boldsymbol{u}}_t^p \quad \Rightarrow \quad \ddot{\boldsymbol{u}}_t^p \ . \tag{13}$$

Then compute the effective loads at time t:

$$\widehat{\boldsymbol{f}}_t = \boldsymbol{f}_t^{ext} - \boldsymbol{f}_t^{int} - \boldsymbol{C}\dot{\boldsymbol{u}}_t \ . \tag{14}$$

2. Solve for accelerations at time t from:

$$\boldsymbol{M}\ddot{\boldsymbol{u}}_t = \widehat{\boldsymbol{f}}_t \ . \tag{15}$$

3. Evaluate velocities at time $t + \Delta t/2$, and displacements at time $t + \Delta t$:

$$\dot{\boldsymbol{u}}_{t+\Delta t/2} = \dot{\boldsymbol{u}}_{t-\Delta t/2} + \Delta t\ddot{\boldsymbol{u}}_t \tag{16}$$

$$\boldsymbol{u}_{t+\Delta t} = \boldsymbol{u}_t + \Delta t\dot{\boldsymbol{u}}_{t+\Delta t/2} \ . \tag{17}$$

The third variant, used for example by Warburton [22], can be obtained by using a backward difference stencil for the velocities $\dot{\boldsymbol{u}}_t = \Delta t^{-1}(\boldsymbol{u}_t - \boldsymbol{u}_{t-\Delta t})$; see equations (18) to (21).

Variant 3:

1. Calculate effective loads at time t:

$$\widehat{\boldsymbol{f}}_t = \boldsymbol{f}_t^{ext} - \boldsymbol{f}_t^{int} - \boldsymbol{C}\dot{\boldsymbol{u}}_t \ . \tag{18}$$

2. Solve for accelerations at time t from:

$$\boldsymbol{M}\ddot{\boldsymbol{u}}_t = \widehat{\boldsymbol{f}}_t \ . \tag{19}$$

3. Evaluate displacements and velocities at time $t + \Delta t$:

$$\boldsymbol{u}_{t+\Delta t} = -\boldsymbol{u}_{t-\Delta t} + 2\boldsymbol{u}_t + \Delta t^2\ddot{\boldsymbol{u}}_t \ , \tag{20}$$

$$\dot{\boldsymbol{u}}_{t+\Delta t} = \Delta t^{-1}(\boldsymbol{u}_{t+\Delta t} - \boldsymbol{u}_t) \ . \tag{21}$$

2.2 Integrator

Note that the algorithms in equations (4) to (21) were specified without explicitly defining the structure of the vectors of displacements, internal and external forces (loads), and the mass (damping) matrices of the structure. These objects were defined by their abstract properties, e.g., the vector of internal forces represents the stresses within the structure without any mention of finite element nodes, numerical integration etc. Further, the algorithms were formulated with a physical domain in mind, or rather a discrete model of a physical domain. It is therefore possible to encapsulate the time-stepping algorithms by formulating them at the logical level of (10) to (9). The algorithms thus "operate" only on the physical domain, and all representational details concerning the objects $\boldsymbol{f}$, $\boldsymbol{u}$ etc. are left to the domain. The domain is free to choose the representation of the mathematical, or computational objects involved. For instance, the domain may use for the storage of $\ddot{\boldsymbol{u}}_t$ and $\widehat{\boldsymbol{f}}_t$ either two vectors, or only a single vector. There is even wider freedom in their implementation (single or double precision, static or global, private or public access etc.).

We are now in a position to describe the *integrator* object; compare with Fig. 1. The integrator is created by the model of the physical domain, and consists of data and callback procedures. The concept of a *callback procedure* is best explained by an example from the everyday life[1]: Person A calls a person B and leaves their phone number. This enables the person B to call person A when needed, with the purpose of obtaining or supplying information. A callback procedure is equivalent to the phone number person A left with person B.

An integrator refers to the *model of the physical domain.* It also maintains the current time t and the time step Δt. The behavior of the integrator manifests itself through callback procedures. Note that the callback procedures are *not* defined by the integrator. Rather, the model of the physical domain defines these procedures, and they reflect all the peculiarities of a given domain model. The model allows the integrator to invoke the callbacks on the domain, i.e., the integrator activates the callback procedure and passes the reference to the domain as an argument. The callback procedures required are:

(a) `calc_eff_loads()`, to compute the effective loads,

(b) `solve_for()`, to solve for the primary unknowns,

(c) `update_config()`, to update the configuration (displacements, velocities etc.), and

(d) `change_dt()`, to change the time step Δt during the time integration.

[1]The On-line Computing Dictionary (URL `http://wombat.doc.ic.ac.uk`) defines a callback as "A scheme used in event-driven programs where the program registers a callback handler for a certain event. The program does not call the handler directly but when the event occurs, the handler is called, possibly with arguments describing the event."

Figure 1

The integrator responds to the following messages (i.e., these procedures can be invoked on the integrator):

- `get_t()`, and `get_dt()` to access the current time and current time step, and
- `advance()` to advance the integrator in time by Δt.

The implementation of the `advance()` method is documented for the Warburton and Newmark variants as shown in Fig. 2. Note that the `i->calc_eff_loads` is the callback procedure (pointer to a function in the C-language), and `i->calc_eff_loads(i->domain)` is an invocation of this procedure. Pursuing further the analogy of Section 2.2, `i->calc_eff_loads` is the phone number, and `i->calc_eff_loads(i->domain)` is the call.

Figure 2

3 Serial time-stepping driver

3.1 General remarks

To make the code cleaner (i.e., less cluttered with details which are immaterial to the goal of describing the parallelization of the code), some code fragments were rewritten with C preprocessor macros. For instance, the construct `ELEM_LOOP(expr)` means that expression `expr` is executed for each finite element `theELEM`, where `theELEM` is expanded as `d->elements[vindex]`. Thus, e.g., the construct

```
ELEM_LOOP(dt = min(dt, WE_suggested_time_step(theELEM)));
```

gets expanded as

```
{
  int vindex = 1, vend = d->num_of_elements;
  for (; vindex <= vend; vindex++) {
    dt = min(dt, WE_suggested_time_step(d->elements[vindex]));
  }
}
```

In this particular case, `d` is a reference (pointer) to the model of the physical domain, the references to elements (pointers) are stored in an array called `elements`, which is a field in the record collecting the data kept by the domain. Similarly for the constructs `LOAD_LOOP`, `NODE_LOOP` etc. The construct

```
EQN_LOOP(theUPDATE_VECTOR = theEFFECTIVE_LOADS / theMASS_MATRIX);
```

can be transcribed without the use of macros as

```
for (i = 1; i <= number_of_equations; i++)
   update_vector[i] = effective_loads[i] / mass_matrix[i];
```

with the pointers `update_vector` etc. being initialized to point at appropriate memory locations.

3.2 Callbacks

3.2.1 WD_integrate

The actual computation is carried out by the domain when the procedure `WD_integrate()` is invoked on it. This routine creates the appropriate integrator (the Newmark integrator has been hardwired into the code in Fig. 3 for simplicity), and sets up the initial conditions (and other data structures, if necessary). Then the integrator is advanced in time until the target time is reached. Note, that the procedure to change the time step, `change_dt()`, has not been specified (it has been set to `NULL` meaning "not defined"). To simplify matters, it is not discussed here.

Figure 3

3.2.2 WD_compute_dt

The initial time increment is computed from the shortest time a wave needs to travel between two finite element nodes. The usual estimate based on the highest eigenvibration frequencies of individual elements is used. The function `WD_compute_dt()` returning the estimated time step computed element-by-element is given in Fig. 4. Here, `WE_suggested_time_step()` is a function invoked for each element to get an estimate of Δt computed on the isolated element.

Figure 4

3.2.3 WD_calc_eff_loads

The routine `WD_calc_eff_loads()` is the first of the procedures used by the integrator to communicate with the model of the physical domain. It loops over active loads, elements and nodes to assemble external loads, restoring (internal) and damping forces, respectively, into the vector of effective loads; *cf.* Fig. 5.

Figure 5

3.2.4 WD_solve_for

The routine `WD_solve_for()` of Fig. 6 is very simple: It consists of a single loop over all the equations, solving for the global unknowns. Note that it is the same for all variants of the integrator. However, the order in which it is invoked differs (*cf.* Fig. 2).

Figure 6

3.2.5 WD_update_config_xxx

The physical domain has to define a procedure to update the configuration for each variant of the integrator. The procedures in our case delegate the task to the finite element nodes, since the nodes maintain the displacement (velocity) data; compare with Fig. 7.

Figure 7

4 Parallel time-stepping algorithm

The parallelization by data decomposition requires a partitioning of the structure, which is here assumed to be static. In other words, we assume that a decomposition of the finite element mesh is available, and we do not allow it to change during the computation. This assumption is not necessary (our design remains valid conceptually), but it simplifies the discussion by separating issues.

4.1 Parallel algorithm with neighbor-to-neighbor communication

The partitions are assigned to processors, which operate on them, exchanging information with other processors as necessary. The most commonly employed scheme uses a communication of the forces on the contacts between two partitions directly between the processors responsible for the two partitions. Such an algorithm has been reported by Belytschko *et al.* [7], and Belytschko and Plaskacz [6] for explicit dynamics on SIMD machines. The communication between neighbors on this type of machine is rather efficient, and the message size can be much smaller due to relatively low overhead latency per message.

Malone and Johnson [20, 21] have used similar communication pattern in their parallel algorithm for massively parallel machines, i.e., the internal forces at nodes at the interfaces are exchanged directly between the processors sharing the node. Again, the hardware targeted by these researchers provides very fast communication channels.

Let us summarize the characteristics of the described scheme:

- The communication consists of a large number of small-grain messages (typically only several floating point numbers).
- The processors must do the book-keeping necessary to keep track of which processor is responsible for which interface node. Also, note that the number of processors interested in a given node varies. For example, in a regular hexahedral mesh, an interface node may belong to up to eight different partitions.

- The communication of the information between the neighbors can proceed in some cases concurrently, which requires independent communication paths between different processors.

4.2 Parallel algorithm with master-worker communication

As discussed in the Section 1.2, parallelization for computer networks requires much larger messages to ameliorate the effects of large latencies. Also parallel communication between several workstations on the network is not possible (the communication paths are shared). Thus we chose an alternative communication pattern, which is based on a star-shaped configuration of workers, synchronized by a master processor; see Fig. 8.

Figure 8

4.2.1 Variants with superelements

One possible way to achieve parallelism with the master-worker communication pattern is based on the concept of a superelement with internal degrees of freedom. The partitions become the superelements, and the nodes which are common to the partitions are the only global nodes in the mesh. The nodes which belong to only one partition are internal to this partition. The global nodes interface nodes are handled as the only global finite element nodes present in the mesh. Thus the "master" works only with these global nodes, and no "regular" finite elements, only the superelements. The worker operates on the finite elements of its partition, and communicates the computed results to the master. While this may seem a natural solution (it is in fact a pure example of the server-client architecture), it has serious drawbacks: It makes the coexistence of the serial and the parallel version more difficult, because the parallel version requires the implementation of a special "internal" node, and of a special remote superelement. Additionally, the operations on the interface nodes create a considerable amount of small-grain communication (recall that the operations on the global nodes have to be done "remotely", i.e., on the master). For these reasons this variant is not suitable for a program meeting our specifications, and was not considered further.

4.2.2 Master-workers variant

The basic idea is that the parallel algorithm is (i) synchronized at each time step by assembling the interface forces, and (ii) dependent on the presence of a driving program which mediates in the necessary communication. Therefore, it is quite natural to consider a "star" configuration of a master program at the center and a set of worker programs. The master is responsible for starting up the workers, and for the assembly and distribution of the interface forces. The workers manipulate the partitions as if they were ordinary serial programs, hiding the few points at which a communication with the master is necessary in well-protected "parallel" interfaces.

The algorithm can be specified as:

- Split the original finite element mesh and assign the partitions to different processors. Define interfaces between the partitions as such nodes that are shared by two or more partitions.
- For each time step:
  - Compute the solution for the partitions in parallel as if they were individual models.
  - For the nodes at the interfaces, assemble the nodal forces globally on the master, i.e., sum the forces by which the partitions act on the interface nodes.
  - Distribute the forces assembled by the master for the interface nodes to partitions. Note that the interface nodal forces at the workers are overwritten by the forces received from the master.

The implementation of the above parallel algorithm can be done in a very clean and concise manner based on the preceding high-level description of the serial time integration. The idea is to create the prototypes of the master and of the workers by modifying the integrator callbacks to account for the distributed character of the concurrent computation.

4.2.3 Possible extensions

One further issue deserves mention at this point. All the workers integrate the equations of motion with the same time step. To achieve a more efficient algorithm, it is possible to use different time steps on different partitions; see, e.g., Hughes and Belytschko [19], Belytschko *et al.* [4, 5], or Chandra *et al.* [8]. The synchronization and communication pattern is not really affected, and the design principles presented here remain valid.

4.3 Implementation remarks

Since one of our goals was to preserve both the serial and the parallel program, and to have only one code to maintain, the implementation of the parallel version modifies the serial code by inserting conditional compilation units. Conditional compilation is achieved by defining preprocessor directives. Thus we are able to generate three different programs from a single source code: The serial program, and the master's and the worker's versions of the parallel program.

Code valid for the master is embedded between the directives

```
#if defined(PVM_WASP_MASTER)
...
#endif
```

`INITSEND()`	`pvm_initsend(PvmDataDefault)`
`RECEIVE(tid, tag)`	`pvm_recv(tid, tag)`
`SEND(tid, tag)`	`pvm_send(tid, tag)`
`BCAST(group, tag)`	`pvm_bcast(group, tag)`
`UNPACK(type, item)`	`pvm_upk##type(item, 1, 1)`
`PACK(type, item)`	`pvm_pk##type(item, 1, 1)`
`UNPACK_ARRAY(type, array, nitem)`	`pvm_upk##type(array, nitem, 1)`
`PACK_ARRAY(type, array, nitem)`	`pvm_pk##type(array, nitem, 1)`

Table 1: Correspondence between C preprocessor macros and PVM functions

and similarly for the code valid for the worker by exchanging `MASTER` for `WORKER`. The code which needs to be modified for the parallel implementation is confined to five files (700 lines) out of 45 files (17500 lines of C-language code).

The details of the manipulation of the PVM software were also hidden behind macros to facilitate maintenance. In particular, the PVM functions return values that indicate whether the action requested can be performed, and other details. To avoid clutter, the global variable `pvm_status` was used to hold the returned information. In this way it was possible to write

```
RECEIVE(ANY_TASK, ANY_TAG);
```

which is expanded by the preprocessor into (assuming the source file name `master.c`)

```
if ((pvm_status.bufid = pvm_recv(ANY_TASK, ANY_TAG)) < 0) {
  PVM_err_exit("pvm_recv", "master.c");
};
```

The macros and the corresponding PVM functions are listed in Table 1. The symbols `##` mean "paste". Thus, assuming that `type` is `double`, the preprocessor expands `pvm_pk##type` into `pvm_pkdouble`.

The message-passing is effected in the following way under PVM. Both sending and receiving of data is done via buffers. Thus, to send some data, the programmer first prepares a transmission buffer with the macro `INIT_SEND()`, and packs the data with the macro `PACK()` (or `PACK_ARRAY()`). When all data are packed, the buffer is routed by `SEND()`. The data is received by the macro `RECEIVE()` (blocking operation; the program waits for the message to arrive). The data is then unpacked by the macro `UNPACK()` (or `UNPACK_ARRAY()`) in the same order and as the same data type as packed. When sending or receiving, it is possible to cooperate either with a specific task, or with any task. Also, it is possible to work with a message of a particular type (tag), or with a "typeless" message (denoted by `ANY_TAG`).

4.4 Input data

The ideal was to preserve not only the structure of the serial program, but also of its input data. However, there is clearly a need for additional information to be supplied for the parallel case. The input to the serial program is based on input blocks (nodes, elements, time functions, loads etc.), so it was easy to add another block (parallel-run control block) of definitions needed for the parallel run. It is thus feasible to use the input data to a worker as input to a serial program, and vice versa. (Correct answers are obtained only for a single partition, of course.)

The master's parallel-run control block consists of the names of the input files to the worker's. The worker's parallel-run control block consists of the specification of the interfaces between the partitions. Thus the mesh of the domain is split into submeshes (partitions). For each submesh, the nodes are numbered $1 \dots N_t$ (N_t being the total number of nodes in the submesh). Further, the nodes on the interfaces (i.e. nodes shared by two or more partitions) are numbered $1 \dots N_i$ (N_i being the total number of interface nodes).

Figure 9 shows sample finite element mesh, which has been subdivided into three partitions. One of the partitions is shown on the left. The filled dots represent nodes, filled rectangles stand for elements. Node numbers on the partition at left are the local numbers. Numbers on the right are the global interface numbers. Thus the local-to-interface mapping for the partition depicted is $(4,1)(8,2)\dots(29,17)(25,18)$, with (l,i), $l \in 1 \dots N_t$ and $i \in 1 \dots N_i$.

Figure 9

4.5 Generation of the parallel programs

As already mentioned, the PVM software is based on processes. The programmer writes one or more prototypes of the programs to become processes, and PVM starts these programs on demand. In our case, two prototypes are needed, the master and the worker. Due to our implementation strategy, both programs are generated from the same source code based on conditional compilation. The C-language preprocessor presents the compiler with source code, from which those parts which were is not desired are excluded, and vice versa.

4.6 Master

The master program needs to start up the workers. It first enrolls into PVM, and then spawns the worker processes. Once these operations have been completed, the master invokes the routine `WD_integrate()` on its domain (which is empty, the domain contains neither elements, nor nodes).

4.7 Worker

The worker program reads the problem definition and constructs the local-to-global interface mapping of node numbers. Then, it enrolls into PVM and joins the `WORKER_GROUP` group of tasks. Finally, the worker runs `WD_integrate()` on its partition (submesh).

Note that both master and the workers invoke routines of the same name, and in the same order. The effects differ, however, as the actions are coded differently. In particular, both master and workers invoke `advance()` on their integrators. The integrators are, however, parameterized with different callbacks. This polymorphism seems to contribute to the readability of the source code, as the necessary parallelization constructs are well localized.

5 Master's modifications

The master's version of the function computing the time step is given in Fig. 10. The master broadcasts a request to compute the time step to all workers (they all joined the group `WORKER_GROUP`). The master then waits for the responses, and computes the globally shortest time step. This value is then broadcast to the workers.

Figure 10

The master's version of the routine `WD_calc_eff_loads()` only consists of two lines; *cf.* Fig. 11. The routine `master_recv()` is shown in Fig. 12. It receives for each worker the number of the interface nodes on the worker's partition and the effective load contributions. These are assembled into the master's buffer `interface_loads`. After the effective load contributions have been received from all the workers, the master sends out the assembled forces in `master_send()`. For efficiency reasons, the master is not required to keep track of which forces to send to which worker, and the master sends the forces for all the interface nodes to all the workers.

Figure 11

Figure 12

Figure 13

Both the routines `WD_solve_for()` and `WD_update_config()` are defined as empty (doing nothing) for the master.

6 Worker's modifications

The worker's version of the function computing the time step is given in Fig. 14. It differs from the serial version of Fig. 4 in that the computed time step is sent to the master and the globally shortest time step is received from it, which is then returned.

Figure 14

The worker's version of the `WD_calc_eff_loads()` routine is derived from the serial one by appending at the end the two lines

```
worker_send(d);
worker_recv(d);
```

The first procedure, `worker_send()`, packs and sends the global numbers of the interface nodes, and the effective nodal loads. The effective nodal loads are packed into arrays of six elements (three components of a 3D force and three components of a 3D moment). Finally, the whole package is sent to the master. When the master has assembled the effective loads from all partitions, it sends them out to the workers, and this package is received in the worker by `worker_recv()`. The worker has to accept the package of all the interface nodes in the whole structure. Therefore, it first (re)allocates a buffer `rb` to hold them by `realloc_recbuf()`. Then it unpacks the interface effective loads (again in 6-element arrays) into the buffer and then loops over all interface nodes on its partition and overwrites the entries in the local vector of effective loads by the entries received into the buffer. The function `constr_package()` collects the entries of the vector `effective_loads` into the vector `forces`, and the function `unwrap_package()` overwrites the entries of the vector `effective_loads` by entries of the buffer array `rb[map[i].interface_num].forces`. These functions are defined for a node. The reason for this is that only the node (as an "object") has access to the equation numbers (these are needed to address the entries of the system vectors).

Figure 15

Figure 16

The worker uses the serial versions of `WD_solve_for()` and `WD_update_config()` without any changes.

7 Mass properties

The partitioning of the original domain leads to the creation of interface nodes. These nodes are by definition connected to several partitions at once. But the workers "know" only their own partition, so that the lumped mass properties computed for the interface nodes from only the single partition elements would not be correct. Consequently, the inertial properties of the interface nodes must be computed in a global manner, similarly to the interface forces. However, if it is assumed that the mass distribution does not change during the computation (as is often the case), this global computation needs to be done only once. There are at least two ways to compute the inertial properties of the interface nodes. The first approach is based on a communication of the local inertial properties, so it is called the *exchange* algorithm. The second approach uses duplication of elements which reference the interface nodes in appropriate workers. This is called the *ghost element* algorithm. A similar algorithm was used for slightly different purpose by McGlaun *et al.* [23].

7.1 Exchange algorithm

This algorithm computes the inertial properties locally on the workers using the partition elements. Then a gather is performed to assemble the local information into the global inertial properties at the master. The global information is then broadcast to the workers. The advantage of this approach is that no information needs to be duplicated; the communication becomes more complicated, however. Consider, for example, the case of structures with rotational degrees of freedom. In order to diagonalize the mass matrix, the mass matrix assembly first collects the rotational inertias associated with the nodes in an element-by-element fashion. This is then followed by a node-by-node computation of the principal directions of the nodal tensor of inertia. Under these circumstances it is clear that additional synchronization is needed, which rather complicates the program logic, and also requires the data formats to be extended.

7.2 Ghost-element algorithm

The elements which are connected at the interface nodes of a given partition are all included in the partition definition. However, the elements *inside* the partition are the regular elements (which are used in all computations), while the elements that are *outside* the partition are the *ghost elements*, which are used in the computation of the mass properties, and are ignored otherwise. This approach duplicates information for all the ghost elements. However, additional communication (and computation on the master) is totally avoided. Given both the advantages and disadvantages, it seemed that the ghost-element approach was preferable. However, no rigorous evaluation was conducted.

8 Additional data exchanges

Explicit finite element programs need to compute additional information. For instance, the energy balance is a very important indicator of the solution quality. Consequently, a parallel program needs to include additional communication mechanism to gather this information from the workers at appropriate time instants (it does not need to be computed every time step).

9 Performance

The performance of any parallel implementation needs to be evaluated with respect to its scalability, i.e., the dependence of the obtained speed up on the number of processors used. This is especially true for programs based on message passing, since the connecting link tends to be slow. However, a thourough scalability evaluation is outside the scope of this paper, and will not be pursued here. Nevertheless, we feel it important to verify the basic concept by some timings.

The parallel algorithm was evaluated by a simple test measuring the run time for the serial driver, and for the parallel driver running on three workstations. The structure was a curved cantilever of elastic material loaded at one end by a step load. The simulation was geometrically non-linear. Figure 17 shows the deformed structure during the serial run. The discretization consisted of 6912 hexahedral elements, and 8829 nodes (26244 unknowns). The workstations were Hewlett&Packard HP/9000 computers, (i) series 715 workstation (100 MHz), with 128 MB of memory, (ii) series 730 workstation, with 48 MB of memory, and (iii) series 720 workstation, with 32 MB of memory. The workstations were connected by a IEEE 802.3 (Ethernet) network. During the run, the network was not isolated, and the workstations were not loaded by other jobs (they were in multi-user mode, however). The speeds of the workstations differed in approximate ratios 1 : 0.65 : 0.4.

Figure 17

9.1 Serial run

The equations of motion were integrated in time for 1,000 steps. The wall-clock time was measured. The serial version needed 2379 seconds to complete the requested number of steps on the workstation (i).

9.2 Parallel run

The partitions were generated in a far-from-optimal manner in that the number of interface nodes was not minimized in any way. On the contrary, ragged interfaces with a large number of nodes were designed to be able to assess the performance of the algorithm under unfavorable conditions. The partitions were:

Workstation (i): 3718 elements, 5087 nodes, 1282 interface nodes.

Workstation (ii): 3477 elements, 4921 nodes, 1396 interface nodes.

Workstation (iii): 2093 elements, 2742 nodes, 114 interface nodes.

The master program was run on the workstation (i).

The equations of motion were integrated in time for 1,000 steps. The wall-clock time was measured. The parallel program took 1941 seconds to complete, which is 81.6% of the time needed by the serial driver. As the ideal parallel run time should be only approx. $2379/(1 + 0.65 + 0.4) = 1160$ seconds, the parallel effectivity is 59.8 %. Note that the parallel effectivity can be expected to improve once there are more computations to be done on the elements (for example, when the material laws are history-dependent), and when the interfaces are optimized to include the smallest possible number of nodes. These issues are currently under investigation.

Figure 18

Conclusions

Careful design of the data structures and of the logic of an explicit finite element program for non-linear dynamics can lead to a remarkably clean parallelization. We had made this experience when parallelizing a serial finite element program of "object-oriented" design (implemented in the C-language) for heterogeneous workstation clusters running the PVM software.

The resulting source code of the parallel program was obtained by an almost trivial modification of the serial version (the C-language fragments presented above cover, with the exception of the setup of the PVM software, all the necessary modifications). It is extendible, and maintainable. In case the code needs to be modified, it is necessary to maintain only one version of the source code, from which both the serial and the parallel programs can be generated.

Dynamic load balancing, and integration with time step varying across the mesh partitions were not addressed here. These issues, and their effect on the design and implementation, will be covered by subsequent research.

Acknowledgments

The support of the National Science Foundation, and of the Army Research Office is gratefully acknowledged.

References

[1] B. W. Abeysundara and A. E. Kamal. High-speed local area networks and their performance: A survey. *ACM Computing Surveys*, 21:261–322, 1991.

[2] J. W. Baugh and S. K. Sharma. Evaluation of distributed finite element algorithms on a workstation. *Engineering with Computers*, 10:45–62, 1995.

[3] T. Belytschko. A survey of numerical methods and computer programs for dynamic structural analysis. *Nucl. Engng. Design*, 37(1):23–34, 1976.

[4] T. Belytschko, B. Engelmann, and W. K. Liu. A review of recent developments in time integration. In A. K. Noor and J. T. Oden, editors, *State-of-the-art surveys of computational mechanics*, New York, 1989. ASME.

[5] T. Belytschko and N. D. Gilbertsen. Implementation of mixed time integration techniques on a vectorized computer with shared memory. *International Journal of Numerical Methods in Engineering*, 35:1803–1828, 1992.

[6] T. Belytschko and E. J. Plaskacz. SIMD implementation of a non-linear transient shell program with partially structured meshes. *International Journal of Numerical Methods in Engineering*, 33:997–1026, 1992.

[7] T. Belytschko, E. J. Plaskacz, and H. Y. Chiang. Explicit finite element method with contact - impact on SIMD computers. *Computer Systems in Engineering*, 2:269–276, 1991.

[8] S. Chandra, N. J. Woodman, and D. I. Blockley. An object-oriented structure for transient dynamics on concurrent computers. *Comput. & Structures*, 51:437–452, 1994.

[9] Y. P. Chien, A. Ecer, H. U. Akay, F. Carpenter, and R. A. Blech. Dynamic load balancing on a network of workstations. *Computer Methods in Applied Mechanics and Engineering*, 119:17–33, 1994.

[10] R. Chudoba and P. Krysl. Explicit finite element computations: Object-oriented approach. In P.J. Pahl and H. Werner, editors, *Proc. of the 6th International Conf. on Computing in Civil and Building Engineering*, pages 139–145, Berlin, 1995. Balkema, Rotterdam.

[11] J. Chung and J. M. Lee. A new family of explicit time integration methods for linear and non-linear structural dynamics. *International Journal of Numerical Methods in Engineering*, 37:3961–3976, 1994.

[12] M. A. Dokainish and K. Subbaraj. A survey of direct time integration methods in computational structural dynamics - i. explicit methods. *Comput. & Structures*, 32:1371–1386, 1989.

[13] K. Dowd. *High performance computing.* O'Reilly & Associates, Sebastopol, CA, 1994.

[14] A. Ecer, H. U. Akay, W. B. Kemle, H. Wang, D. Ercoskun, and E. J. Hall. Parallel computation of fluid dynamics problems. *Computer Methods in Applied Mechanics and Engineering*, 112:91–108, 1994.

[15] M. W. Fahmy and A. H. Namini. A survey of parallel non-linear dynamic analysis methodologies. *Comput. & Structures*, 53:1033–1043, 1994.

[16] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal of Numerical Methods in Engineering*, 36:745–764, 1993.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, Mass., 1994.

[18] A. V. Hudli and R. M. V. Pidaparti. Distributed finite element structural analysis using the client-server model. *Comm. Numer. Meth. Engineering*, 11, 1995.

[19] T. J. R. Hughes and T. Belytschko. A précis of developments in computational methods for transient analysis. *J. Appl. Mech.*, 50:1033–1041, 1983.

[20] J. G. Malone and N. L. Johnson. A parallel finite element contact/impact algorithm for non-linear explicit transient analysis. Part I. The search algorithm and contact mechanics. *International Journal of Numerical Methods in Engineering*, 37:559–590, 1994.

[21] J. G. Malone and N. L. Johnson. A parallel finite element contact/impact algorithm for non-linear explicit transient analysis. Part II. Parallel implementation. *International Journal of Numerical Methods in Engineering*, 37:591–539, 1994.

[22] M. McGlaun, A. Robinson, and J. Peery. Some recent advances in structural vibrations. In C. A. et al. Brebbia, editor, *Vibrations of engineering structures*, Berlin, New York, 1985. Springer-Verlag.

[23] M. McGlaun, A. Robinson, and J. Peery. The development and application of massively parallel solid mechanics codes. In S. N. Atluri, G. Yagawa, and T. A. Cruse, editors, *Computational mechanics '95, Int. Conf. on Computational Engineering Science*, New York, 1995. Springer.

[24] R. R. Namburu, D. Turner, and K. K. Tamma. An effective data parallel self-starting explicit methodology for computational structural dynamics on the Connection Machine CM-5. *International Journal of Numerical Methods in Engineering*, 38:3211–3226, 1995.

[25] R. Ou and R. Fulton. An investigation of parallel integration methods for nonlinear dynamics. *Comput. & Structures*, 30:403–409, 1988.

[26] C. Özturan, H. L. deCougny, M. S. Shephard, and J. E. Flaherty. Parallel adaptive mesh refinement and redistribution on distributed memory computers. *Computer Methods in Applied Mechanics and Engineering*, 119:123–137, 1994.

[27] K. C. Park and P. G. Underwood. A variable-step central difference method for structural dynamics analysis. Part I. Theoretical aspects. *Computer Methods in Applied Mechanics and Engineering*, 22:241–258, 1980.

[28] J. C. Simo, N. Tarnow, and K. K. Wong. Exact energy-momentum conserving algorithms and symplectic schemes for nonlinear dynamics. *Computer Methods in Applied Mechanics and Engineering*, 100:63–116, 1992.

[29] D. Vanderstraeten and R. Keunings. Optimized partitioning of unstructured finite element meshes. *International Journal of Numerical Methods in Engineering*, 38:433–450, 1995.

[30] G. and Yagawa. A parallel finite element analysis with supercomputer network. *Comput. & Structures*, 47:407–418, 1993.

Figure 1: The explicit integrator object in graphic form

Figure 2: Implementation of the `advance()` method

Figure 3: Central difference driver on the highest level (a method defined on the physical domain)

Figure 4: Function `WD_compute_dt()` to compute the initial time step

Figure 5: Function `WD_calc_eff_loads()` to compute the effective loads

Figure 6: Function `WD_solve_for()` to compute the primary unknowns

Figure 7: Function to update the geometric configuration for the Newmark integrator

Figure 8: The configuration of the master and the workers in a "star"

Figure 9: Sample partition of a finite element mesh

Figure 10: Modification of `WD_compute_dt()` for the master version

Figure 11: Modification of `WD_calc_eff_loads()` for the master

Figure 12: Routine `master_recv()`

Figure 13: Routine `master_send()`

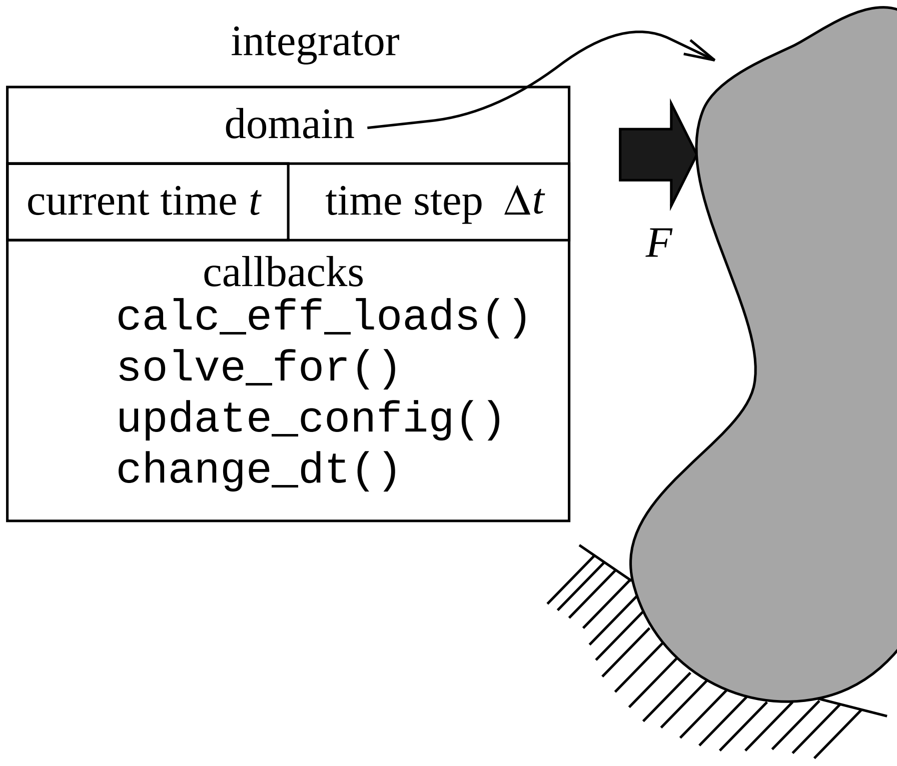
Figure 14: Modification of `WD_compute_dt()` for the worker

Figure 15: Function `worker_send()`

Figure 16: Function `worker_recv()`

Figure 17: Deformed structure. Serial model

Figure 18: Parallel model. Partitions



integrator
domain
current time t
time step Δt
callbacks
calc_eff_loads()
solve_for()
update_config()
change_dt()
F

Figure 1

```
static void
advance(cd_integrator_t *i) /* WARBURTON */
{
  i->calc_eff_loads(i->domain);
  i->solve_for(i->domain);
  i->update_config(i->domain);
  i->t += i->dt;
  if (i->change_dt != NULL) i->dt = (*i->change_dt)(i->domain);
}

static void
advance(cd_integrator_t *i) /* NEWMARK */
{
  i->update_config(i->domain);
  i->calc_eff_loads(i->domain);
  i->solve_for(i->domain);
  i->t += i->dt;
  if (i->change_dt != NULL) i->dt = (*i->change_dt)(i->domain);
}
```

Figure 2

```
void
WD_integrate(W_domain_t *d,  double start_time, double target_time)
{
  d->integrator = create_Newmark_integrator(
       d,                            /* domain */
       start_time, WD_compute_dt(d), /* t and dt */
       WD_calc_eff_loads,            /* procedure: (a) */
       WD_solve_for,                 /*            (b) */
       WD_update_config_Newmark,     /*            (c) */
       NULL                          /*            (d) */
     );
  setup_initial_conditions(d);
  do {
    integrator_advance(d->integrator);
  } while (integrator_t(d->integrator) < target_time);
}
```

Figure 3

```
double
WD_compute_dt(W_domain_t *d)
{
  double dt = INFINITY;

  ELEM_LOOP(dt = min(dt, WE_suggested_time_step(theELEM)));
  return dt;
}
```

Figure 4

```
void
WD_calc_eff_loads(W_domain_t *d)
{
  zero_effective_loads(d);
  LOAD_LOOP(WL_assemble_load(theLOAD, d));
  ELEM_LOOP(WE_assemble_restoring_forces(theELEM));
  NODE_LOOP(WN_assemble_damping_forces(theNODE, d));
}
```

Figure 5

```
void
WD_solve_for(W_domain_t *d)
{
  EQN_LOOP(theUPDATE_VECTOR = theEFFECTIVE_LOADS / theMASS_MATRIX);
}
```

Figure 6

```
void
WD_update_config_Newmark(W_domain_t *d)
{
  NODE_LOOP(WN_update_config_Newmark(theNODE, d));
}
```

Figure 7

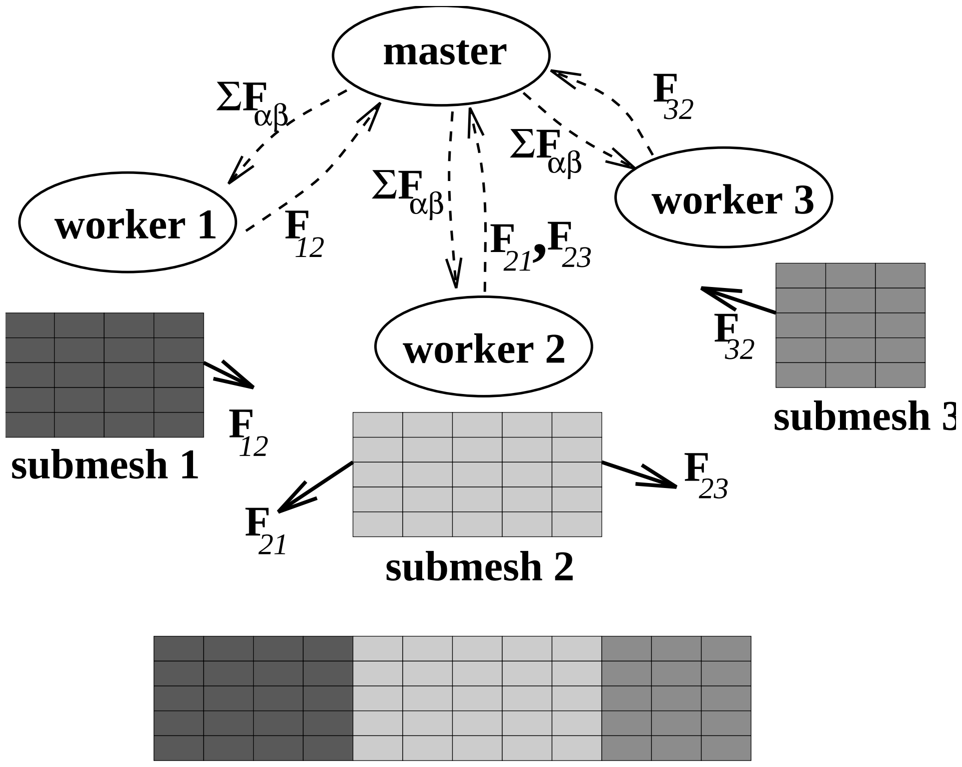

master
$\Sigma F_{\alpha\beta}$
F_{32}
$\Sigma F_{\alpha\beta}$
$\Sigma F_{\alpha\beta}$
worker 1
F_{12}
worker 3
F_{21}, F_{23}
worker 2
F_{32}
submesh 3
F_{12}
submesh 1
F_{23}
F_{21}
submesh 2

Figure 8

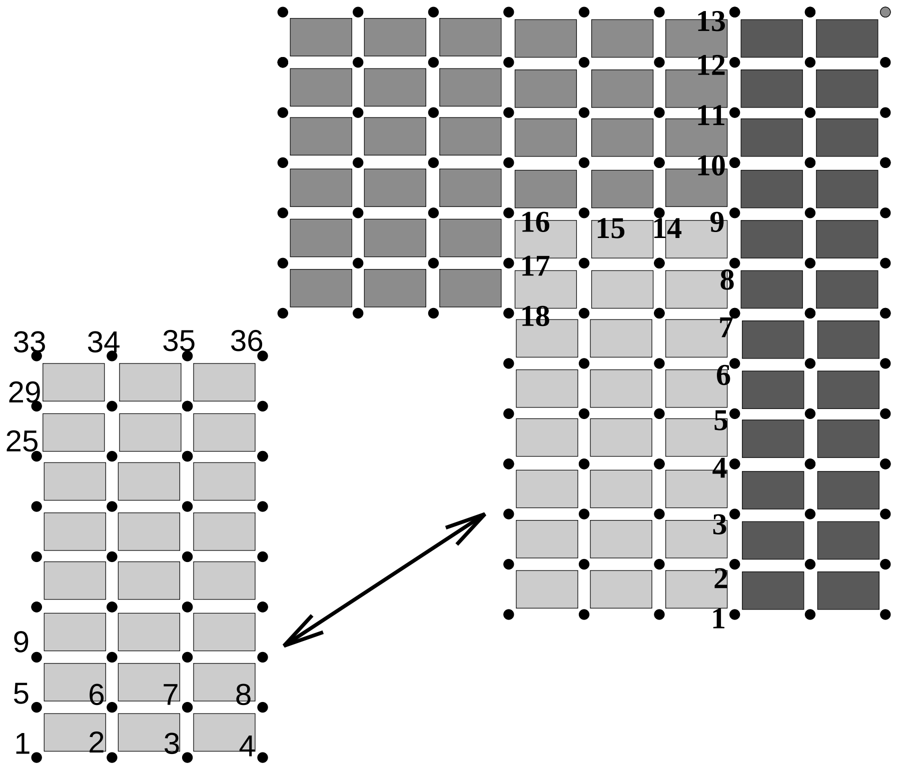


Figure 9

```
double
WD_compute_dt(W_domain_t *d)
{
  double dt = INFINITY, sent_dt;

  INITSEND();
  BCAST(WORKER_GROUP, TIME_STEP_REQUEST_TAG);
  WORKER_LOOP(
    RECEIVE(theWORKER, TIME_STEP_ANSWER_TAG);
    UNPACK(double, &sent_dt);
    dt = min(dt, sent_dt);
  );
  INITSEND();
  PACK(double, &dt);
  BCAST(WORKER_GROUP, TIME_STEP_ANSWER_TAG);
  return dt;
}
```

Figure 10

```
void
WD_calc_eff_loads(W_domain_t *d)
{
  master_recv(d);
  master_send(d);
}
```

Figure 11

```
static void
master_recv(W_domain_t *d)
{
  unsigned long num_of_if_nodes, in, i, w;
  W_pvm_master_t *m = d->pvm_master;
  double forces[6];

  zero_interface_loads(d);
  WORKER_LOOP(
    RECEIVE(ANY_WORKER, EFFLOAD_LOCAL_TAG);
    UNPACK(ulong, &num_of_if_nodes);
    for (i = 1; i <= num_of_if_nodes; i++) {
      UNPACK(ulong, &in);
      UNPACK_ARRAY(double, forces, 6);
      for (j = 0; j < 6; j++)
        interface_loads[in].forces[j] += forces[j];
    }
  );
}
```

Figure 12

```
static void
master_send(W_domain_t *d)
{
  W_pvm_master_t *m = d->pvm_master;
  unsigned long i, num_of_if_nodes = m->num_of_interface_nodes;

  INITSEND();
  PACK(ulong, &num_of_if_nodes);
  for (i = 1; i <= num_of_if_nodes; i++)
    PACK_ARRAY(double, interface_loads[i].forces, 6);
  BCAST(WORKER_GROUP, EFFLOAD_GLOBAL_TAG);
}
```

Figure 13

```
double
WD_compute_dt(W_domain_t *d)
{
  double dt = INFINITY;

  ELEM_LOOP(dt = min(dt, WE_suggested_time_step(theELEM)));
  INITSEND();
  PACK(double, &dt);
  SEND(theMASTER, TIME_STEP_ANSWER_TAG);
  RECEIVE(theMASTER, TIME_STEP_ANSWER_TAG);
  UNPACK(double, &dt);
  return dt;
}
```

Figure 14

```
static void
worker_send(W_domain_t *d)
{
  W_pvm_worker_t *w = d->pvm_worker;
  unsigned long i, num_of_if_nodes = w->num_of_interface_nodes
  W_local_to_interface_map_t *map = w->map;
  double forces[6];

  INITSEND();
  PACK(ulong, &num_of_if_nodes);
  for (i = 1; i <= num_of_if_nodes; i++) {
    PACK(ulong, &map[i].interface_num);
    constr_package(WD_get_node_by_idx(d, map[i].local_num),
                   d->effective_loads, forces);
    PACK_ARRAY(double, forces, 6);
  }
  SEND(theMASTER, EFFLOAD_LOCAL_TAG);
}
```

Figure 15

```
static unsigned long num_of_interface_packages = 0;
static W_interface_package_t *rb = NULL;
static void
worker_recv(W_domain_t *d)
{
  unsigned long num_of_if_packages, i;
  W_pvm_worker_t *w = d->pvm_worker;
  W_local_to_interface_map_t *map = w->map;

  RECEIVE(MASTER_TID(w), EFFLOAD_GLOBAL_TAG);
  UNPACK(ulong, &num_of_if_packages);
  realloc_recbuf(num_of_if_packages);
  for (i = 1; i <= num_of_interface_packages; i++)
    UNPACK_ARRAY(double, rb[i].forces, 6);
  for (i = 1; i <= w->num_of_interface_nodes; i++)
    unwrap_package(WD_get_node_by_idx(d, map[i].local_num),
                   rb[map[i].interface_num].forces,
                   d->effective_loads);
}
```

Figure 16

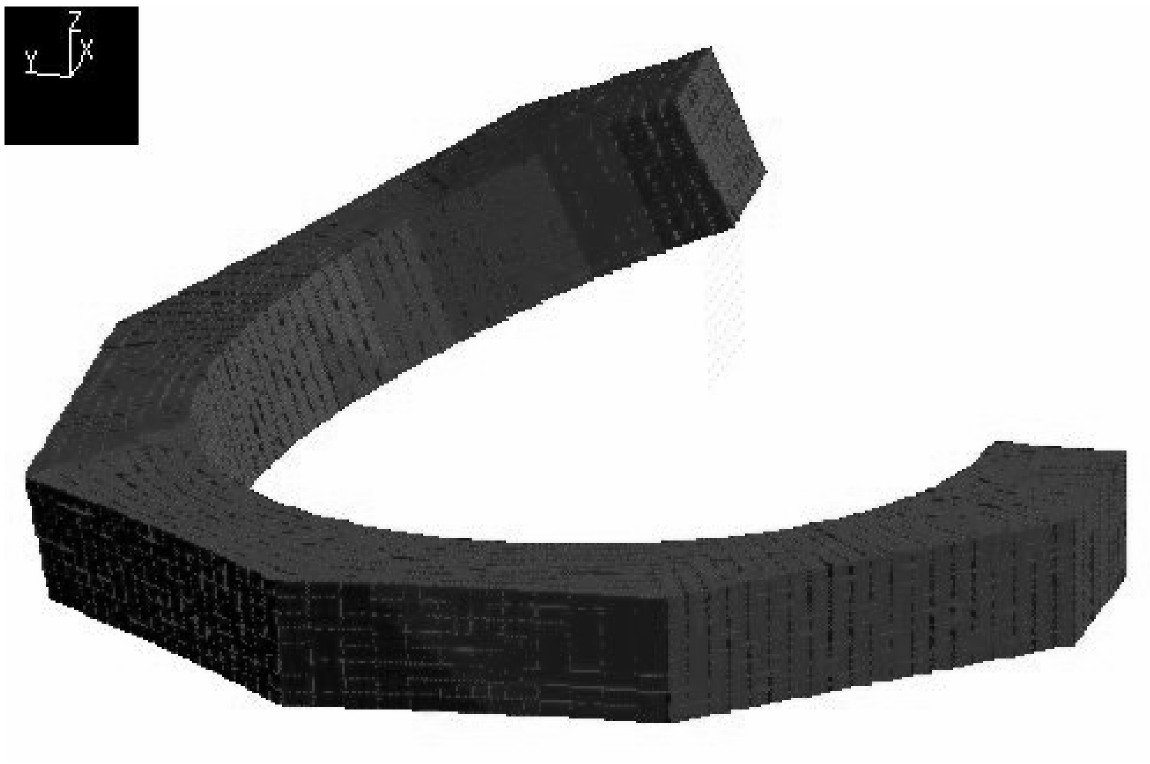


Figure 17

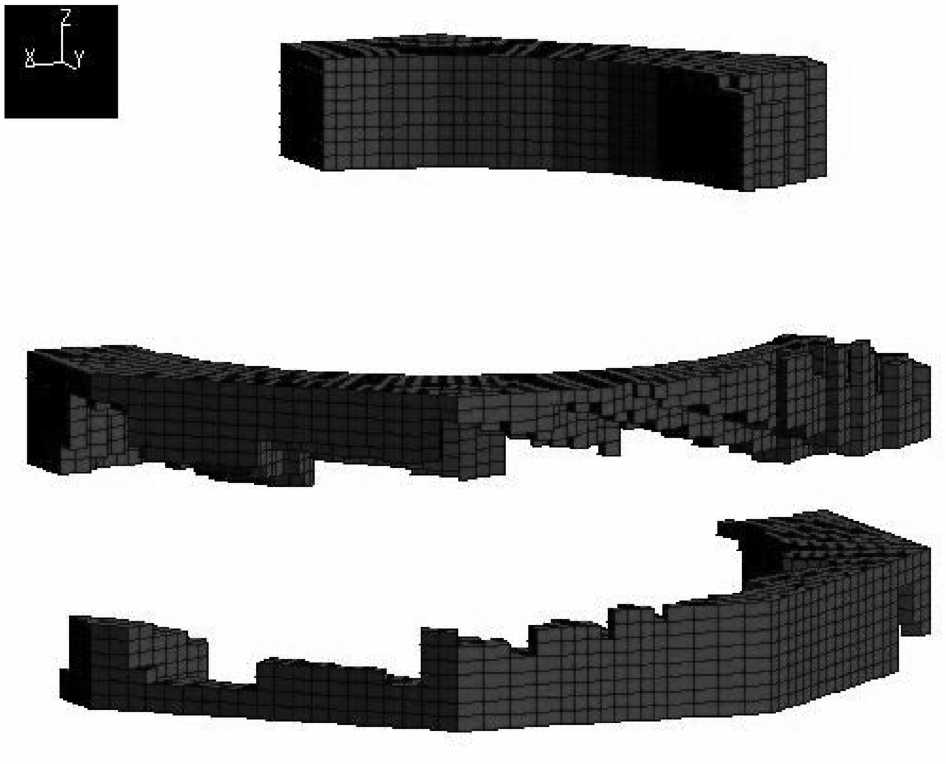


Figure 18