Parallel explicit finite element solid dynamics with domain decomposition and message passing: Dual partitioning scalability *

Petr Krysl† Zdeněk Bittnar‡

November 12, 1999

Abstract

Parallelization of explicit finite element dynamics based on domain decomposition and message passing may utilize one of two partitioning cuts, namely cut led through the nodes and element edges or faces (node cut), or cut led across elements, avoiding nodes (element cut). The cost of serial explicit finite element dynamics (without considerations of mechanical contact) is almost wholly associated with elements (internal force evaluation and material updates). Sharing of nodes among processors leads to very little duplication of computing effort, and the node-cut partitioning has been used exclusively in the past.

The dual nature of the element-cut partitioning, and in particular the fact that the *nodes* are assigned uniquely to partitions, means that communication requirements may be in some situations quite different compared to the node-cut partitioning. Hence, the question suggests itself whether using element-cut partitioning would make certain algorithms, such as for instance subcycling and mechanical contact, simpler, more efficient, or plainly possible. Seeking an answer to this question makes sense only if the larger overhead associated with the duplication of elements does not prevent the element-cut partitioning from being scalable as the number of processors increases, especially in fixed-partition-size situations. We show here that the element-cut partitioning strategy does scale, and hence presents a viable alternative to the tradiotional node-cut approach.

*Submitted to C&S on 11/14/1999.

†California Institute of Technology, 256-80, Pasadena, CA 91125. Fax: (626)792-4257, Email: pkrysl@cs.caltech.edu.

‡Department of Structural Mechanics, Czech Technical University in Prague.

We document not only the high-level algorithms but also the relevant communication code fragments of the message passing implementation using the MPI library, so as to empower the reader to fully verify our numerical experiments.

Introduction

Parallel execution of explicit finite element solid dynamics simulations on distributed memory computers with domain decomposition and explicit message passing is an enabling technology for bigger computing runs completed in shorter time. The high-level algorithm of explicit time stepping in finite element solid dynamics is relatively straightforward, and parallelization of finite element programs has been reported for both research and commercial codes; see, for example, References [1, 2, 3]. Complications arise especially due to the need to enforce mechanical contact conditions [4, 5, 6, 7, 8, 9], for coupled field problems (implicit/explicit procedures), or for variants of the explicit time stepping with some form of step subcycling (employment of multiple time steps) [10, 11].

The finite element domain may be partitioned among the co-operating processors by either duplicating nodes (in other words, the cut is led through the finite element edges and faces), or by duplicating the elements. We call the former strategy the *node cut*, and the latter the *element cut*. It is well known that the cost of serial explicit finite element dynamics (without contact) is almost wholly associated with elements (internal force evaluation and material updates). Duplication of nodes leads therefore to very little duplication of computing effort, and so it is perhaps not surprising that the node-cut partitioning has been used exclusively in the past, and that the element-cut partitioning has been rarely mentioned as an alternative, if at all.

However, if we consider the dual nature of the partitioning cuts, and in particular the fact that either the nodes or the elements are assigned uniquely to partitions, which means that communication requirements may be quite different in different situations, a question suggests itself whether using element-cut partitioning would make certain algorithms, such as for instance subcycling [10, 11] and mechanical contact [4, 5, 6, 7, 8, 9], simpler, more efficient, or plainly possible. We do not propose to answer this question here. Rather, we seek to ascertain whether this further investigation is

justified by asking "Does element-cut partitioning scale as the number of processors grows?" We believe that this paper provides sufficient data proving that despite its higher overhead the element-cut partitioning strategy does scale, and hence constitutes a viable alternative to the node-cut partitioning.

The literature dealing specifically with element-cut partitioning is not numerous. The authors are aware of publications dealing with a similar subject in fluid dynamics by Farhat and Lanteri [12] and Lanteri [13, 14], who discuss use of overlapping and non-overlapping triangle and tetrahedral grids for mixed finite element/finite volume Navier-Stokes computations [14]. A partitioning approach which on the surface looks as if it could fit into our classification has been proposed by Masters et al. [15]. However, the authors of Reference [15] are apparently unaware of the possibility of exploiting the duality of partitioning cuts (their approach is based instead a node-cut partitioning with duplication of a row of elements on each side of the partitioning cut), and do not explore scalability in much detail.

Although it is widely believed that the domain-decomposition parallelization of explicit solid dynamics with message passing is by now completely understood, we venture to disagree. Message-passing communication consists of layers of algorithmic complexity: descending from the high-level algorithm, to the message-passing library, and then to further layers of software closer to the communication hardware. While some of the deeper layers are encapsulated, and are not exposed to the application programmer, the communication library (MPI, in our case) constitutes the application programmer interface (API), which represents standardized means of implementing algorithms in arbitrary, *non-standardized* ways. In other words, a given high-level algorithm may be (and often is) implemented at the API level in distinct codes in different ways, not all of them equally efficient and robust.

The API-level communication algorithm affects strongly the robustness and the parallel efficiency and scalability of the high-level algorithm, and yet it is rarely described in papers in sufficient detail to allow for unambiguous assessment and verification of the results. Furthermore, some interesting questions concerning the API-level communication have not been asked yet, much less answered: "Should the communication start by receives or by sends, or should the receives and sends be posted at the same time, and in fact does it matter which is first?" and "When is it best to synchronize?" (Compare the various solutions in the present paper and References [15, 16, 17].)

Therefore, given the above, we feel justified in describing in detail how

both node- and element-cut partitioning can be robustly and efficiently implemented in a single finite element program with minimum duplication of source code. Our communication algorithm uses the MPI library, relying on application buffering and non-blocking communication primitives for robustness and efficiency. We believe that by presenting the crucial fragments of the communication code we provide the reader with all the necessary knowledge so that our results can be completely and unambiguously verified.

1 Central difference time-stepping

The solid mechanics finite element codes solve the momentum equations discretized both in the spatial domain (finite elements), and in the time domain (finite differences). The most commonly used explicit time integration technique is the central differences algorithm. One of its variants is the explicit Newmark beta method, which is summarized as

$$\dot{\boldsymbol{u}}_t = \dot{\boldsymbol{u}}_{t-\Delta t} + (\Delta t/2)\,(\ddot{\boldsymbol{u}}_t + \ddot{\boldsymbol{u}}_{t-\Delta t}) \tag{1}$$

$$\boldsymbol{u}_{t+\Delta t} = \boldsymbol{u}_t + \Delta t \dot{\boldsymbol{u}}_t + (\Delta t^2/2)\ddot{\boldsymbol{u}}_t \tag{2}$$

$$\widehat{\boldsymbol{f}}_{t+\Delta t} = \boldsymbol{f}^{ext}_{t+\Delta t} - \boldsymbol{f}^{int}_{t+\Delta t} - \boldsymbol{C}\dot{\boldsymbol{u}}^{p}_{t+\Delta t} \tag{3}$$

$$\text{with} \quad \dot{\boldsymbol{u}}^{p}_{t+\Delta t} = \dot{\boldsymbol{u}}_t + \Delta t \ddot{\boldsymbol{u}}_t \quad \text{or} \quad \dot{\boldsymbol{u}}^{p}_{t+\Delta t} = (\boldsymbol{u}_{t+\Delta t} - \boldsymbol{u}_t)/\Delta t \tag{4}$$

$$\boldsymbol{M}\ddot{\boldsymbol{u}}_{t+\Delta t} = \widehat{\boldsymbol{f}}_{t+\Delta t} \tag{5}$$

In (1)–(5) and in what follows, $\boldsymbol{M}$ denotes the mass matrix (constant, diagonal), $\boldsymbol{C}$ is the damping matrix, which can be function of the velocities, $\boldsymbol{u}_\tau$, $\dot{\boldsymbol{u}}_\tau$, $\ddot{\boldsymbol{u}}_\tau$ are the vectors of displacements, velocities, and accelerations, respectively, $\dot{\boldsymbol{u}}^{p}_{t+\Delta t}$ is the predicted velocity for damping force computation, and $\boldsymbol{f}^{ext}_\tau$ are the external loads. The internal nodal forces $\boldsymbol{f}^{int}_\tau$ are generated by the stresses. All quantitities are written at time τ.

The algorithm of equations (1)–(5) can be written in the symbolic form as shown in Figure 1. The cost of steps (i) and (iii) is minor compared with step (ii), which involves element-wise computations of stresses with material state update. All of the steps can be parallelized to a large degree by domain decomposition as will be the subject of following sections.

while not finished **loop**
(i) Update configuration [equations (1), (2)];
(ii) Compute effective nodal forces [equations (3), (4)];
(iii) Compute acceleration [equation (5)];
(iv) Finalize step (compute stable time step, ...);
(v) Increment time: $t = t + \Delta t$.
endloop

Figure 1: Central difference time-stepping algorithm.

2 Node- and element-cut partitioning

For reasons of expediency we restrict ourselves to static domain decomposition, also with the view that dynamic load balancing adds an additional layer of complexity, but otherwise does not compromise our conclusions.

2.1 Load balancing

A *sine qua non* condition of achieving acceptable parallel speedups is load balance. Each time step in the algorithm of Figure 1 constitutes a natural synchronization point, because in order to proceed at time $t + \Delta t$, the state of all the nodal and elemental quantities must be known at time t. The load balancing should therefore allow all the processing units to finish their portion of work at the same time, at the next global synchronization point. In an explicit solid dynamics finite element code the majority of the CPU time is proportional to the number of elements – update of material state at the integration points being the main consumer of cycles; see for example the review of Fahmy and Namini [1]. Hence, partitioning balanced on the number of elements per partition is indicated. However, the domain decomposition software has not only the task of balancing the computation load, but also to minimize the communication requirements (we provide some formulas below to support this statement). Since optimal decomposition problem is unsolvable in general, heuristics are used. The domain decompositions in our case have been obtained with the partitioning software METIS [18].

2.2 Node-cut partitioning

The obvious way of partitioning the finite element mesh is to lead the dividing line (surface) through the element edges (faces); see Figure 2. The elements are assigned uniquely to partitions (there are two partitions in Figure 2). The nodes through which the cut is led are *shared* by the partitions (filled squares in Figure 2). The other nodes are private to each partition, so the momentum equations are solved at these nodes without change. However, at the shared nodes one is confronted with the necessity to assemble contributions from two or more partitions.

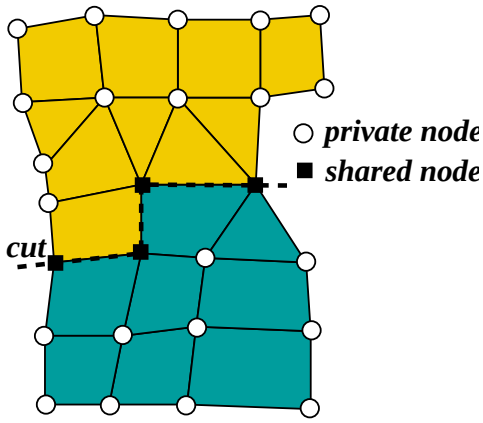


Figure 2: Node-cut partitioning.

In order to implement the node-cut partitioning, one can modify the Newmark central difference scheme of Figure 1 to include an exchange of nodal forces assembled by each domain from only the elements constituting the given partition (Figure 3). In other words, each partition sends the forces it assembled for the shared nodes to all the incident partitions, and in this way every partition gathers for each shared node contributions from all the elements incident to it. (Danielson and Namburu have recently proposed an interesting variation on the algorithm in Figure 3, which allows for additional overlapping of communication and computation [16].)

One remark is in order here. Since the partitioned domains contain only those elements which have been assigned to it, the correct mass matrix needs to be established before the time stepping starts by an operation analogous to the exchange of forces. As a matter of fact, as we will show later on, the exchange code differs only in the packing and unpacking of the information

while not finished **loop**
(i) Update configuration [equations (1), (2)];
(ii) Compute effective nodal forces [equations (3), (4)];
Exchange forces for shared nodes.
(iii) Compute acceleration [equation (5)];
(iv) Finalize step (compute stable time step, ...);
(v) Increment time: $t = t + \Delta t$.
endloop

Figure 3: Central difference time-stepping algorithm. The node-cut strategy.

to be exchanged, hence most of the communication code can be re-used.

2.3 Element-cut partitioning

An alternative to the "node-cut" strategy is the "element-cut" strategy. The partitioting cut is for this approach led across the edges (faces); see Figure 4. The nodes are assigned uniquely to partitions, and the elements which have been cut are duplicated for each partition adjacent to the cut. It should be noted that a partition gets to work not only with its proper nodes, but also with nodes incident on the shared elements which are "owned" by other partitions. We call these nodes "remote node copies" (RNC), for reasons to be clarified instantly.

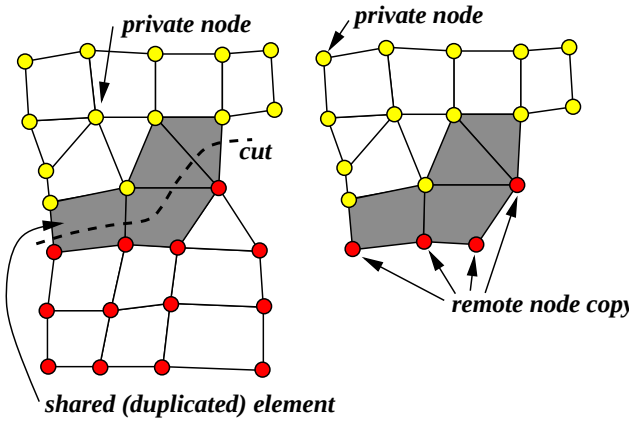


Figure 4: Element-cut partitioning.

The same approach to parallelization as in Figure 3 could now be adopted,

while not finished **loop**
(i) Update configuration [equations (1), (2)];
(ii) Compute effective nodal forces [equations (3), (4)];
(iii) Compute acceleration [equation (5)];
Exchange accelerations for remote node copies.
(iv) Finalize step (compute stable time step, ...);
(v) Increment time: $t = t + \Delta t$.
endloop

Figure 5: Central difference time-stepping algorithm. The element-cut strategy.

ie. the forces for the RNCs could be exchanged. However, that is obviously inefficient, since we would have to exchange forces for all nodes of the shared elements. We approach the problem differently, and handle the RNCs not as true nodes, but as "shadows" of nodes. We do not bother to compute their mass or assemble the nodal forces for them. Instead, to perform a time step we choose to exchange *accelerations* for the RNCs as shown in Figure 5: the accelerations are computed from (5) only for nodes owned by the given partition. For the RNCs the accelerations are received from the partition owning the remote node. The final effect is, of course, the same as if the accelerations had been computed for the node locally from forces. (Equivalently, we could also exchange forces as for node-cut partitioning; however, the force acting on the RNC would override the locally computed force instead of being summed with the local force.)

To summarize the preceding developments, let us note here a duality in the node- and element-cut partitioning strategies; see Table 1.

Cut through	Assigns uniquely	Duplicates
Nodes	Elements	Nodes
Elements	Nodes	Elements

Table 1: Duality of cut strategies

3 Implementation of the exchanges

The natural computing paradigm is in our case the Single Program Multiple Data (SPMD) approach. We adopt the MPI library [19] for the message-passing implementation of the exchange algorithm. We have been able to obtain the needed communication functionality by using only 11 MPI functions:

- Communication setup: MPI_Init, MPI_Get_processor_name, MPI_Comm_rank, MPI_Comm_size, MPI_Bcast, MPI_Finalize;
- Computation of stable time step: MPI_Allreduce;
- Exchanges: MPI_Barrier, MPI_Isend, MPI_Irecv, and MPI_Waitany;

3.1 Data structures

The attributes of a partition mesh are collectively grouped in an entity called the *domain*. The distributed nature of the mesh is reflected in the data structure W_mpi_process_t of Figure 6 which is also an attribute of the domain. (Note that Figure 6 shows *pseudo-code*; declaring dynamic arrays as shown is not possible. We want to stress the function of the fields. The same remark applies also in Figure 7.) There are two important constituents of the W_mpi_process_t data structure: a node-partition map, and communication buffers.

```
typedef struct W_mpi_process_t {
  W_domain_t       *domain; /* domain on the processor */
  int               size;   /* # of collaborating processes */
  int               rank;   /* rank of process */
  MPI_Request       recv_requests[size]; /* MPI request handles */
  W_mpi_npmap_t     node_part_map;     /* node/partition map */
  W_mpi_comm_buff_t comm_buffs[size]; /* communication buffers */
}             W_mpi_process_t;
```

Figure 6: MPI process data structure.

The *node-partition map* is the basis of all communication; all the communication structures are derived from it. It can be understood as a global

dictionary keyed by the node identifier (say a globally unique integer number). The implementation optimizes access to the node by storing the pointer to the node data instead of this integer identifier; see Figure 7. Note that num_pairs equals to the number of globally unique nodes. The value stored in the dictionary is a list of partition numbers for node-cut partitions, or simply the number of the owning partition (element-cut partitions). It bears emphasis that the node-partition map is a distributed data structure in the sense that if a partition does not refer to a given node, the node key reads "undefined" (NULL in our C-language implementation) meaning no value is stored for the partition list on a given processor.

```
typedef struct W_mpi_npmap_pair_t {
  W_node_t       *n;          /* node */
  W_mpi_rank_t    ranks_dim; /* dim of below array */
  W_mpi_rank_t   *ranks;      /* partition numbers */
}              W_mpi_npmap_pair_t;

typedef struct W_mpi_npmap_t {
  W_index_t          num_pairs; /* number of pairs */
  W_mpi_npmap_pair_t pairs[num_pairs]; /* array of pairs */
}              W_mpi_npmap_t;
```

Figure 7: Node-partition map data structure.

The node-partition map is computed by our front-end to the METIS library from the graph partitioning METIS produces, and is read from a file by the finite element program.

The *communication buffers* play two roles (Figure 8). The first consists of providing readily available information about with which neighbor a given partition should communicate, and the second is to provide buffers for data being sent or received. Each given partition defines an array of communication buffers, one for each partition (even for itself, but no communication with itself is ever performed, of course).

It bears emphasis that the messages being sent or received are buffered by the finite element program, not by the message-passing library. For the sake of readability and maintainability we pack and unpack data from the communication buffers, but one could also envisage using the communication buffers directly in the computations, which could save the packing/ unpacking time.

```
typedef struct W_mpi_comm_buff_t {
  int        rank;          /* rank (partition number) */
  int        num_to_send;   /* # of nodes to send */
  int        num_to_recv;   /* # of nodes to receive */
  W_node_t **to_send;       /* array of pointers to nodes to send
                               of num_to_send size */
  W_node_t **to_recv;       /* array of pointers to nodes to receive
                               of num_to_recv size */
  int        count;         /* number of doubles in buffer */
  int        sbuf_dim;      /* send buffer dimension */
  double    *sbuf;          /* send-buffer; array of sbuf_dim size */
  int        rbuf_dim;      /* receive buffer dimension */
  double    *rbuf;          /* receive-buffer; array of rbuf_dim size */
}              W_mpi_comm_buff_t;
```

Figure 8: Communication buffer. (Node-cut partitioning.)

3.2 Common communication code

When looking at Figures 3 and 5, one should immediately notice that there is common ground and that the duality of the partitioning cuts is manifest: In both cases communication is required for the update of shared entities. To "update" a node, one needs to compute the accelerations from (5), and an element is "updated" during the computation of the internal force $\boldsymbol{f}^{int}_{t+\Delta t}$ of equation (3). The requisite information is nodal force (update of node), and node configuration ($\boldsymbol{u}_t$, $\dot{\boldsymbol{u}}_t$, $\ddot{\boldsymbol{u}}_t$) for the update of element. Futhermore, nodal vectors (scalars) are being exchanged (i) between pairs of adjacent partitions; (ii) in both directions. Finally, an important aspect is efficiency, and in that respect we should note that there is no need for fine-grained communications between nodes themselves. On the contrary, the communication can be carried out in batches (node groups).

We have pursued this reasoning in our implementation, and the result is a single communication routine which works for both node-cut (exchange of mass and forces), and element-cut strategies (exchange of accelerations). The differences have been concentrated into *callbacks*[1] which pack the data

[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

for transmission and unpack it after it has been delivered.

The communication routine is described in Figure 9. Two main considerations were followed in its design. Firstly, one needs to *avoid* communication *deadlocks*, which can easily occur when blocking communication primitives are used for the unstructured, irregular communication patterns that need to be dealt with. (Consider, for example, two processes which both enter a mutual unbuffered blocking send or receive at the same time.) Our second goal is *communication efficiency* through overlapping of computation and communication. Both goals, robustness and efficiency, can be met by using non-blocking communication primitives.

We start by packing the data to be sent into the communication buffers, and initializing non-blocking sends (Figure 10). Also, we initialize receive operations by non-blocking receives (Figure 11). The communication subsystem (which could be operated by a communication processor separate from the CPU) is then free to go ahead with the message transfer and with the copying of the message from MPI buffers to the receive buffer provided as argument to MPI_Irecv.

Finally, we wait until all receive operations have completed by repeatedly calling MPI_Waitany and unpacking the data from the communication buffer's storage (Figure 12). The exchange algorithm concludes by *explicit synchronization* by calling MPI_Barrier. The reason for this is that we do not have any means of finding out whether data sent by a process has reached its destination. (Explicit acknowledgement is a possibility, but that would add significantly to the message traffic. We have not explored this avenue in detail, though.) To understand the need for synchronization, consider the following example: We have a mesh split into two partitions. (There is a one-to-one correspondence between partitions and processes.) Process 0 sends data to process 1, but then is suspended because of increased activity of another process running on the same CPU, and does not manage to receive from process 1. Process 1 sends data to, and receives from, process 0. Without synchronization at the end of the time step, process 1 could now run ahead, and arrive at the next exchange phase before the process 0 was able to receive from process 1. At that point, process 1 could overwrite its communication buffer for sending, rendering contents of message delivered to process 0 undefined.

---

the event occurs, the handler is called, possibly with arguments describing the event."

```
void
W_mpi_exchange (W_domain_t *d, /* IN: domain */
                int nvpn, /* IN: number of doubles per node */
                W_mpi_pack_func pack,      /* IN: pack function */
                W_mpi_unpack_func unpack, /* IN: unpack function */
                int tag                   /* IN: message tag */)
{
  W_mpi_process_t *p = &d->mpi_process; /* MPI process */
  int i;
  FOR (i, p->size) {
    isend (p, i, nvpn, tag, pack); /* initialize sends */
    irecv (p, i, nvpn, tag);       /* initialize receives */
  }
  wait_all (p, unpack); /* wait for receives to complete */
}
```

Figure 9: Implementation of the exchange algorithm (common to both node- and element-cut strategies).

```
static void
isend (W_mpi_process_t *p,     /* IN: process */
       int to,                 /* IN: destination partition */
       int num_vals_per_node, /* IN: number of values per node */
       int tag,                /* IN: message tag */
       W_mpi_pack_func pack    /* IN: pack function */)
{
  W_mpi_comm_buff_t *cb = &p->comm_buffs[to];
  if (cb->num_to_send > 0) { /* Anything to send? */
    MPI_Request req;
    cb->count = num_vals_per_node * cb->num_to_send;
    W_mpi_realloc_sbuf (cb); /* re-size */
    pack (p, cb);     /* pack into buffer */
    MPI_Isend (cb->sbuf, cb->count, MPI_DOUBLE,
               cb->rank, tag, MPI_COMM_WORLD, &req);
  }
}
```

Figure 10: Function `isend()` invoked in Figure 9.

```
static void
irecv (W_mpi_process_t *p,     /* IN: process */
       int from,               /* IN: source partition */
       int num_vals_per_node, /* IN: number of values per node */
       int tag                 /* IN: message tag */)
{
  W_mpi_comm_buff_t *cb = &p->comm_buffs[from];
  if (cb->num_to_recv > 0) { /* Anything to receive? */
    cb->count = num_vals_per_node * cb->num_to_recv;
    W_mpi_realloc_rbuf (cb); /* re-size */
    MPI_Irecv (cb->rbuf, cb->count, MPI_DOUBLE,
               cb->rank, tag, MPI_COMM_WORLD,
               &p->recv_requests[from]);
  } else { /* nothing to receive; set request to NULL */
    p->recv_requests[from] = MPI_REQUEST_NULL;
  }
}
```

Figure 11: Function `irecv()` invoked in Figure 9.

```
static void
wait_all (W_mpi_process_t *p,       /* IN: process */
          W_mpi_unpack_func unpack /* IN: unpack function */)
{
  MPI_Status status;
  int i, idx;
  int num_recv = 0; /* count receives */
  FOR (i, p->size) {
    if (p->comm_buffs[i].num_to_recv > 0) num_recv++;
  }
  while (num_recv-- > 0) { /* wait for all receives */
    MPI_Waitany (p->size, p->recv_requests, &idx, &status);
    unpack (p, &p->comm_buffs[idx]); /* unpack */
  }
  MPI_Barrier (MPI_COMM_WORLD); /* synchronize */
}
```

Figure 12: Function `wait_all()` invoked in Figure 9.

3.3 Implementation for node-cut partitions

The communication pattern can be established for node-cut partitions very easily, since the node/ partition map is the only data structure needed. In other words, communication for a node is required if the value stored in the node/ partition map is a list of two or more partition numbers. Also, the number of nodes for which a send is required is equal to the number of nodes for which a partition needs to receive.

The signature of the W_mpi_exchange function indicates how it is used to carry out an exchange of nodal quantities among the collaborating MPI processes: The caller provides pointers to functions packing and unpacking the nodal data, the number of double precision values per node (one scalar per node for the exchange of mass, array of 3 values per node for the exchange of force), and the message tag. The packing and unpacking functions use the to_send and to_recv arrays to find out for which nodes to communicate, and depend of course on the scheme used to store nodal data. We present our implementation of these functions in the Appendix.

3.4 Implementation for element-cut partitions

Contrary to the node-cut partitioning setup, the communication pattern for element-cut strategy needs to be established by communication. Initially, each partition knows for which nodes a receive is needed (and can therefore compute easily the to_recv array in the communication buffers), but does not know for which nodes it should send data to which partition. Hence, the communication setup is performed by broadcasting "send request" lists of nodes for which a partition expects to receive data (ie. of those nodes which the partition uses, but does not own) to all collaborating processes. The "send request" lists are converted into the to_send array in the communication buffers; otherwise these functions are identical to those in the Appendix.

4 Numerical experiments

As we have declared above, our intention is to provide hard data which would indicate how the two dual partitioning strategies compare in terms of scalability. Node-cut partitioning has been shown to scale well; see, for instance References [15, 16], and a number of other publications which do

not describe the API-level communication algorithms, but do give scalability data [6, 3, 9, 20, 17].

Our intention is to provide performance, and in particular scalability, figures for the element-cut strategy. However, to make all things equal for the purpose of comparison, we also give analogous results for the node-cut strategy. We choose an academic example discretized with several hexahedral meshes to be able to observe the influence of mesh size, and a real-life finite element model to observe properties of the communication routines under slightly unbalanced-load conditions.

4.1 Discussion of speedup formulas

The time to run a simulation with the un-partitioned mesh (ie. on a single processing unit) can be broken down into a parallelizable and a non-parallelizable (serial) part as

$$T(N_p = 1) = T_{\text{ser}} + T_{\text{par}} \ , \tag{6}$$

where T_{par} is the parallelizable part of the total CPU time, and T_{ser} is the serial part.

Let us now consider the effect of the partitioning strategy. Both strategies lead to duplication; shared are either nodes (node-cut), or elements (element-cut strategy). Hence the parallelizable part is no longer just T_{par}, but $T'_{\text{par}} > T_{\text{par}}$, since there is more entities to compute with. The time to run the simulation partitioned for N_p processing units (processors) is therefore composed of the serial part, T_{ser}, fraction of the parallelizable part, T'_{par}/N_p, and of communication time, T_{comm}, which is a non-linear function of the number of processors, of the number of finite element nodes for which communication is required, hardware configuration, etc.

$$T(N_p) = T_{\text{ser}} + \frac{T'_{\text{par}}}{N_p} + T_{\text{comm}} \tag{7}$$

Hence, the parallel speedup can be expressed as

$$S(N_p) = \frac{T(N_p = 1)}{T(N_p)} = \frac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + T'_{\text{par}}/N_p + T_{\text{comm}}}. \tag{8}$$

In explicit finite element solid dynamics programs with domain decomposition, the serial part is negligible compared with the parallelizable part,

$T_{\text{ser}} \ll T_{\text{par}}$. The parallel speedup is then approximated as

$$S(N_p) \approx \frac{N_p}{T'_{\text{par}}/T_{\text{par}} + N_p \dfrac{T_{\text{comm}}}{T_{\text{par}}}} \tag{9}$$

Let us now consider the ramifications of equation (9). First, there is the effect of the node or element duplication. As we have already pointed out, much more work is associated with elements than with nodes, therefore the node-cut strategy is penalized only lightly ($T'_{\text{par}} \approx T_{\text{par}}$). (However, the *asymptotic* cost, ie. the cost for computations with meshes of size growing without limit, will still be affected by duplication in both cases.)

On the other hand, the element-cut strategy incurs non-negligible overhead, which can be roughly estimated by back-of-the-envelope calculations: The un-partitioned mesh has N_E elements. Assume $N_p \ll N_E$, ie. there is substantially less processors than elements. Assuming perfect balance, the number of elements per partition follows as N_E/N_p, and the number of duplicated (cut) elements is approximately proportional to the "surface" of all the partitions, and for approximately cube-shaped partitions we get

$$N_{E,dupl} \approx O\left[N_p \left(N_E/N_p\right)^{2/3}\right] \ . \tag{10}$$

Since T'_{par} is proportional to the number of original elements plus the number of cut elements, we can estimate

$$T'_{\text{par}}/T_{\text{par}} = \frac{N_E + O\left[N_p \left(N_E/N_p\right)^{2/3}\right]}{N_E} = 1 + O\left(N_p^{1/3} N_E^{-1/3}\right) \ . \tag{11}$$

Together with parallel speedup, we find the *parallel efficiency* to be an extremely useful measure. It is defined as

$$e(N_p) = S(N_p)/N_p \ . \tag{12}$$

From the above formulas, the parallel efficiency can be approximated for the node-cut strategy as

$$e(N_p) = \frac{1}{1 + N_p \dfrac{T_{\text{comm}}}{T_{\text{par}}}} \ , \tag{13}$$

and for the element-cut strategy as

$$e(N_p) = \frac{1}{1 + O\left(N_p^{1/3} N_E^{-1/3}\right) + N_p \dfrac{T_{\text{comm}}}{T_{\text{par}}}} \ . \tag{14}$$

Specializing formulas (13) and (14) for *fixed-partition-size* runs, the parallel efficiency may be rewritten by using the fact that the number of elements per partition is fixed, $N_{E,part} = N_E/N_p \approx \text{const}$. Hence, we can see that for the node-cut partitioning the parallel efficiency is predicted by (13) as independent of N_p (perfect scalability), and since (11) can be rewritten as

$$T'_{\text{par}}/T_{\text{par}} = \frac{N_p N_{E,part} + O\left[N_p \left(N_{E,part}\right)^{2/3}\right]}{N_p N_{E,part}} = 1 + O\left(N_{E,part}^{-1/3}\right) \ , \qquad (15)$$

we deduce that for the element-cut partitioning

- we may expect it to scale for fixed-partition-size computations, because (15) does not depend on the number of processors N_p; and
- the overhead of element-cut partitioning decreases for larger partitions, i.e. the larger is $N_{E,part}$, the closer $T'_{\text{par}}/T_{\text{par}}$ is to one. (In other words, the overhead due to duplication is mitigated by using larger partitions.)

Both predictions are explored via numerical experiments in the next sections.

4.2 Partitioning

The finite element mesh partitioning library METIS [18] has been used to obtain the element-balanced partitions of this section. However, since METIS currently does not have the capability of partitioning nodes and elements according to our classification (node- and element-cut), a simple front-end to METIS has been written. This program calls METIS to partition the nodal graph (nodes are the vertices of the graph, and elements are its edges). The output from METIS is an element-balanced labelling of elements and nodes with partition numbers, which we use directly to produce the node-cut (element-cut) partitions. It should be noted that the total number of elements, including also those elements duplicated along the partitioning cuts, should be partitioned for the element-cut strategy. However, not knowing of an easy way of doing this, we have accepted the resulting slight imbalance; see Figures 13 and 21. The workload balance is defined as

$$\frac{\sum_{i}^{N_p} N_{Ei}}{N_p \max_i N_{Ei}} \times 100\% \ , \qquad (16)$$

where N_{Ei} is the number of elements in partition i (including those duplicated in other partitions).

4.3 Computer system used in tests

The tests in this section have been run on the IBM Scalable PowerParallel 9076 SP-2 machine at the Cornell Theory Center. The machine is equipped with POWER2 SC thin-nodes (120 MHz) with at least 256MB of memory. The computing nodes are interconnected by a High-Performance Switch communication hardware, which has been used in the user-space mode for increased throughput (it does not need to make kernel calls to communicate). The processing nodes have been reserved for "unique" CPU usage (only the test program was running on each node), and the wall-clock times we report are free of load-unbalance effects related to time sharing. Since we measure real elapsed time, the difference in CPU speeds is not relevant, although we have to mix faster and slower processors for larger number of partitions. Also, the times we report exclude input/output times.

The MPI library is on the SP-2 system layered on top of the native MPL library [21], which was designed to take advantage of the High-Performance Switch using a light-weight user-space protocol.

4.4 Cube

The finite element model in this section is a uniform, regular discretization of a cube with a square through-hole. The material constitutive equation is the St.Venant-Kirchhoff hyperelastic model. Hexahedral, isoparametric finite elements with $2 \times 2 \times 2$ Gauss quadrature have been used. The mesh sizes are summarized in Table 2. All of the single-CPU runs could be accommodated by a POWER2 SC node with 256MB of memory (CUBE-15 could not be run on a single CPU). The wall-clock times for these runs were on the order of 300 seconds, which means that the runs for larger number of processors could not be timed very accurately, but since we report averages over a certain number of runs we believe the figures are reliable. The time per element per cycle for the mesh CUBE-12 for a single-CPU run was measured as $100\mu s$.

The best picture of the performance of the parallel algorithm seems to be conveyed by graphs of parallel efficiency (parallel speedup can be easily derived from efficiency, and actual wall-clock times are of little interest). Therefore, and also in order to save space, we show only graphs of parallel efficiency here.

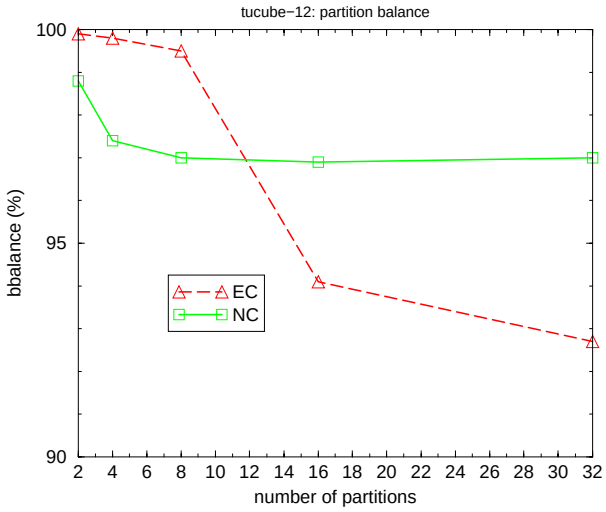
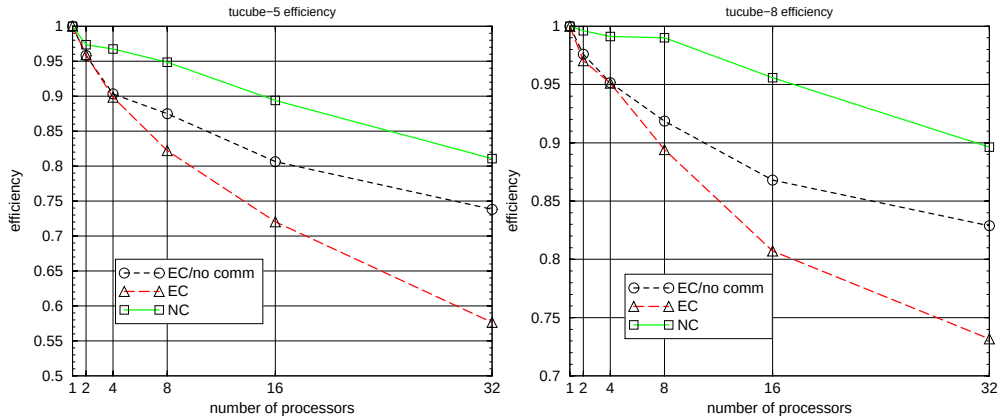


Figure 13: Model CUBE-12: balance of partitions (equation 16).

Model	Number of nodes	Number of elements
CUBE-5	17,160	15,000
CUBE-8	66,912	61,440
CUBE-10	128,520	120,000
CUBE-12	219,600	207,360
CUBE-15	421,200	405,000

Table 2: CUBE meshes.



(a) CUBE-5 (b) CUBE-8

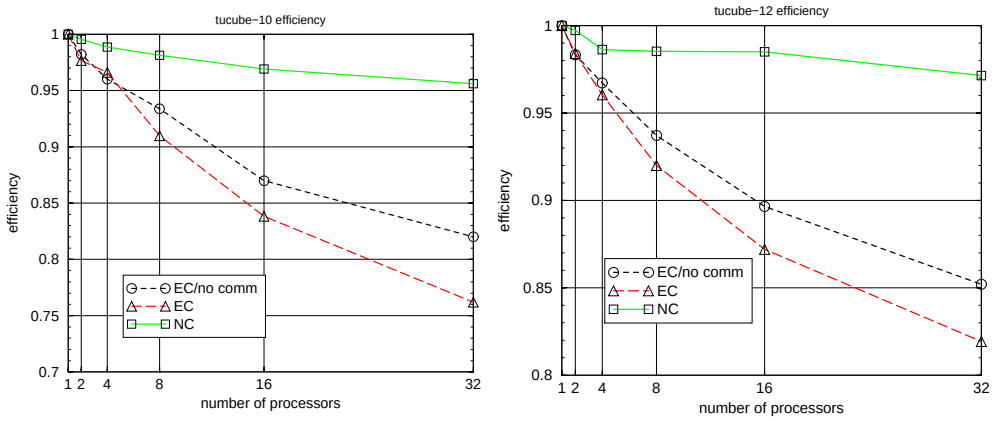
Figure 14: Parallel efficiency for different CUBE model sizes. Element-cut partitioning.

4.4.1 Fixed-model-size scalability

Figure 14(a) shows the parallel efficiency for the smallest mesh, CUBE-5. In the graph legends, NC and EC denote the node- and element-cut strategies respectively. The ideal parallel efficiency for the node-cut strategy is simply $e(N_p) = 1$ (these curves are not shown in the graphs), whereas the ideal parallel efficiency for the element-cut strategy (curves labelled EC/no comm) is derived from speedup of (9) where the communication time is ignored, and the ratio $T'_{\text{par}}/T_{\text{par}}$ is computed as

$$T'_{\text{par}}/T_{\text{par}} = \left(\sum_{i}^{N_p} N_{Ei} \right) / N_E \ . \tag{17}$$

Figures 14(b), 15(a), and 15(b) show parallel efficiencies for the other meshes, CUBE-8, CUBE-10, and CUBE-12. Figures 17(a) and 17(b) summarize the parallel efficiency for the node- and element-cut strategies for all meshes. The increase in parallel efficiency for larger meshes reflects equation (13), or (14) respectively: The computation time T_{par} grows faster than the communication time, T_{comm}. Figure 16 illustrates that our theoretical prediction (11) of overhead due to duplication holds up against measured data: the predicted cube-root variation is evident.



(a) CUBE-10 (b) CUBE-12

Figure 15: Parallel efficiency for different CUBE model sizes. Element-cut partitioning.

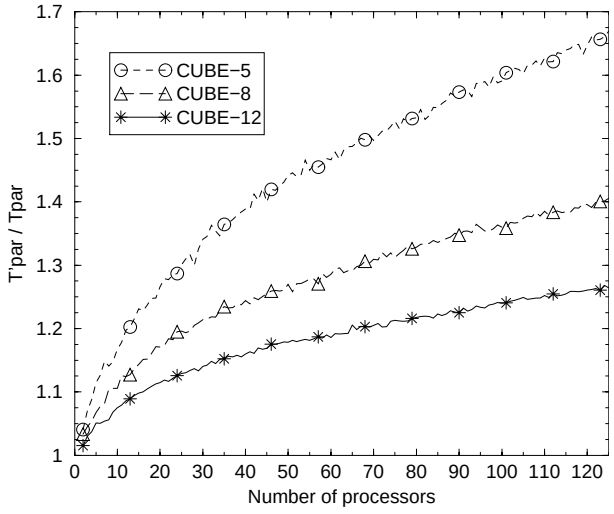
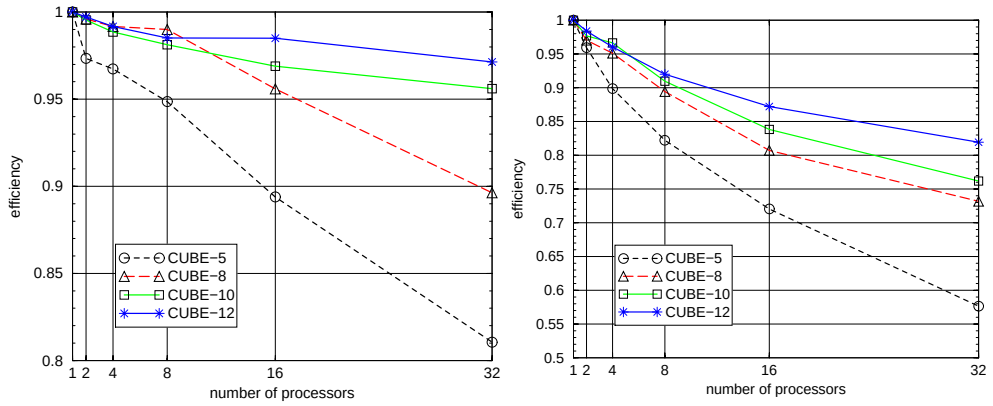


Figure 16: $T'_{\rm par}/T_{\rm par}$ in dependence on N_p for fixed-model-size element-cut partitioning.



(a) Node-cut partitions

(b) Element-cut partitions

Figure 17: Parallel efficiency for different size of CUBE models

The parallel efficiency has been obtained here for the case of *fixed model size*. In other words, T_{par} is held fixed in the term $N_p T_{\text{comm}}/T_{\text{par}}$ in equations (13) and (14). Then, since T_{comm} is non-decreasing with increasing N_p, the fixed-model-size parallel efficiency decreases with the number of processors. The evidence for this is quite clearly shown in Figures 17(a) and 17(b).

4.4.2 Fixed-partition-size scalability

The situation is quite different in the case of *fixed partition sizes*, when N_p/T_{par} is held fixed in equations (13) and (14). To attain scalability in this situation is often very important, because it means that bigger computations can be undertaken by increasing the number of processors. To document scalability in this setting, we present results for a series of simulations with 2,500, 3,000, 5,000, and 7,500 elements per partition. (Even the 7,500-element partitions probably do not contain enough elements to achieve optimal performance on the SP-2, which has relatively slower communication hardware, but fast and well-balanced CPU, caches, and memory.)

Figure 18(a) shows wall-clock run times for the node-cut partitioning strategy (curves connect points for same-size partitions; number 5,8,10,12,15 stands for CUBE-5, ...). Good scalability is evident from the approximately zero slope of curves connecting run times for same-size partitions. Some

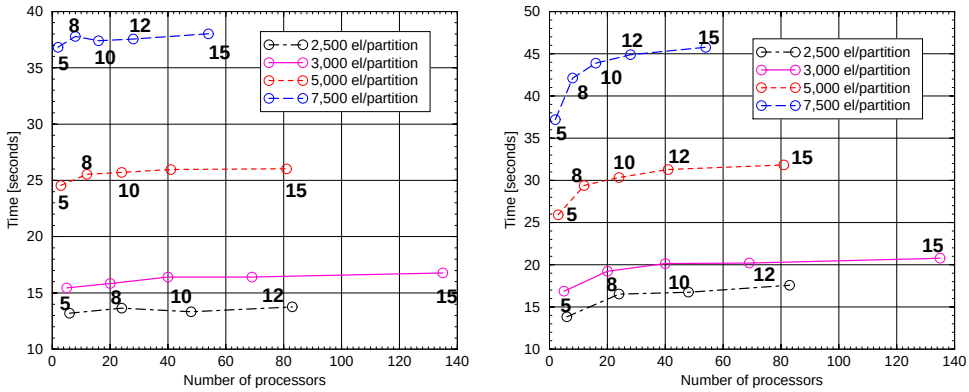
remarks are in order: The spacing of the curves is approximately given by the ratio of the partition sizes. The results for very small number of processors are not quite on equal footing with those for large number of processors, because the average number of neighbors, and hence the number of exchanges in each time step, increases with N_p (but less so for larger N_p). Therefore one can observe steeper run time increase for small number of processors; then the curve flattens.

Next, we present analogous results for the element-cut strategy; see Figure 18(b). In contrast to Figure 18(a), the run times increase more sharply for smaller number of processors, but then level off. These experimental data can be related to Eq. (14), which is based on the approximate formula for the number of duplicated elements (10). Figures 19(a) and 19(b) show the ratio $T'_{\text{par}}/T_{\text{par}}$ for a large number of partition sizes (from about 200 to 7,500 elements). These data demonstrate that formula (10) is overly pessimistic for small number of partitions (hence the larger initial slope), and does not take into effect the ratio of the number of internal partitions to the total number of partitions (this number increases with N_p; internal partitions contribute more duplicated elements than partitions at the "free" surface). These two factors explain the deviations of the measured curves in Figures 19(a) and 19(b) from the theoretical predictions of (10). Nonetheless, the slope tends to zero for large N_p (indicating an asymptotic limit), and that means that the element-cut partitioning scales well for fixed-partition-size computations.

4.5 Propeller

The mesh in this example is representative of real-life applications; see Figure 20. It is modestly complex geometrically, and although in terms of mesh size it is rather smallish (121220 nodes, 63116 elements), the finite elements are complex (quadratic tetrahedra with four-point quadrature). The material is elasto-plastic with isotropic hardening [22], and the loading is chosen so as to create moderate spreads of yielded material in a number of locations. The momentum equations have been advanced in time for 50 time steps. The model has been partitioned into 1, 2, 4, 8, 16, and 32 partitions (not quite as well balanced as perhaps could be desired; see Figure 21). The time per element per cycle for the mesh PROPELLER for a single-CPU run was measured as $84\mu s$.

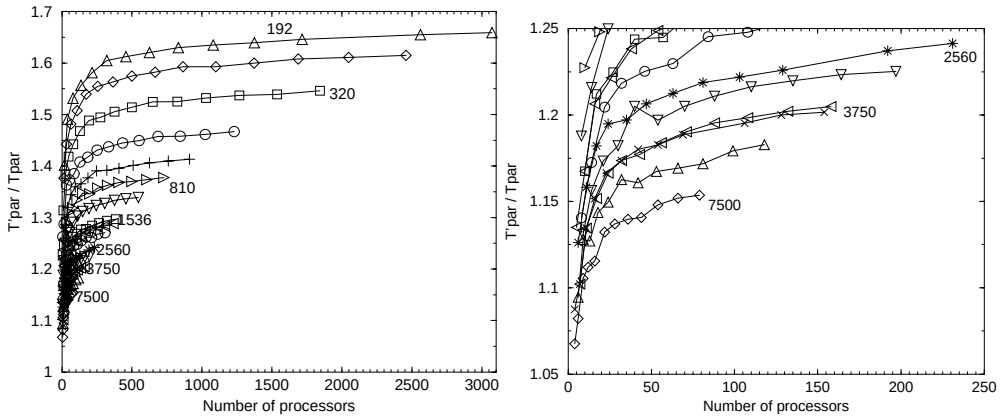
Figure 22 shows the parallel efficiency for mesh PROPELLER. It is interesting to note the sudden drop in efficiency for 4 processor partitioning. It



(a) Node-cut partitions (b) Element-cut partitions

Figure 18: Parallel scalability for different size of CUBE models. Partitions contain 2,500, 5,000, and 7,500 elements respectively.

1536



(a) Overall view (b) Close-up of the origin

Figure 19: Ratio $T'_{\text{par}}/T_{\text{par}}$ in dependence on the number of processors N_p for fixed-partition-size computations with the element-cut partitioning.

Figure 20: Finite element model of PROPELLER. Domain decomposition into six partitions.

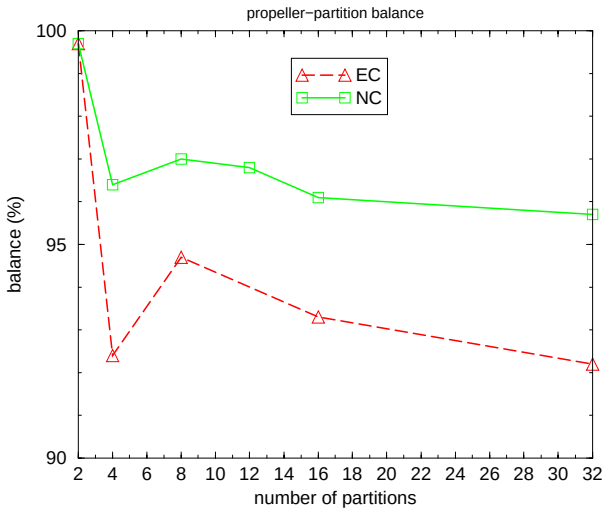


Figure 21: Model PROPELLER: balance of partitions (equation 16).

could be perhaps related to the workload unbalance evident in Figure 21.

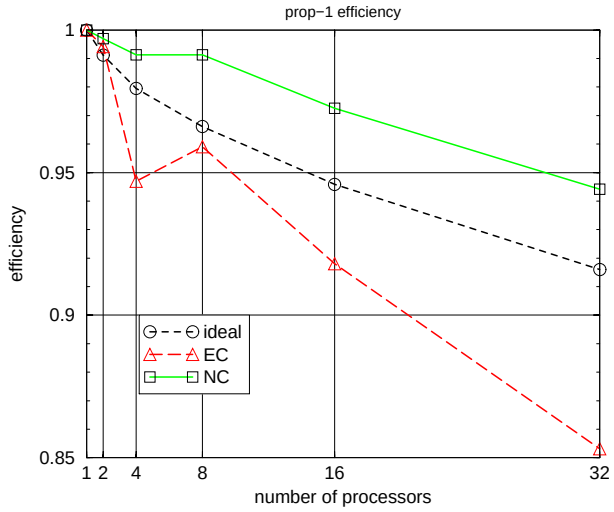


Figure 22: Parallel efficiency for model PROPELLER.

Conclusions

We have approached the parallelization of explicit finite element solid dynamics program with domain decomposition and message passing from the point of view of duality of partitioning cuts of the mesh graph. The mesh nodal graph, with nodes being its vertices, and elements representing its edges, can be partitioned by cutting either through the vertices or across the edges. Hence, we have classified the two partitioning cuts as node- or element-cut strategies.

The partitioning duplicates either nodes (node cut) or elements (element cut). Consequently, the communication requirements can be in various situations quite dissimilar, and the duality of the two partitioning strategies may in fact facilitate formulation and/or implementation of certain algorithms (nodal subcycling, contact). However, operations on elements are considerably more expensive compared to those on nodes, and duplication of elements penalizes the element-cut partitioning noticeably. Therefore, the goal of this paper was to explore the scalability of the element-cut partitioning so as to

decide whether the approach warrants further study. We have approached this issue via simple theoretical arguments and numerical studies.

We have found that the node-cut partitioning yields a higher parallel efficiency than the element-cut partitioning in *fixed-model-size scalability* studies, but the difference does not seem to exclude the latter from consideration.

Turning to *fixed-partition-size scalability*, we conclude that both the node-cut and the element-cut partitioning strategy scale well. This conclusion is, of course, expected for the former, but rather surprising at first sight for the latter. However, the duality between the two partitioning strategies makes it rather obvious that the asymptotic costs are in fact the same, and only the constants vary. Since our motivating question, namely "does element-cut partitioning lead to a scalable algorithm" is answered in the affirmative, further study of its applications for use in some areas of explicit solid dynamics, such as nodal subcycling and mechanical contact enforcement, seems justified.

Finally, we have also presented the most important fragments of the code implementing the communication algorithm with calls to the MPI library, and we have documented the advertised possibility of using the same communication code for both partitioning strategies.

Acknowledgments

We gratefully acknowledge support for this work from the Grant Agency of the Czech Republic (grant # 103/97/K003), and from CEC (Copernicus grant IC 15-CT96-0755). The Joint Supercomputing Center of the Czech Technical University, University of Chemical Technology and IBM in Prague provided resources which allowed us to code, debug, and tune the parallel implementation. The Cornell Theory Center kindly provided access to their SP-2 machine, which enabled us to run the performance studies on a large number of processors. Dan Rypl is thanked for helpful comments.

References

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

[2] 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.

[3] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Performance issues of the parallel PAM-CRASH code. *International Journal of Supercomputing Applications and High-Performance Computing*, 11(1):3–11, 1997.

[4] 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.

[5] 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.

[6] J. O. Hallquist, K. Schweizerhoff, and D. Stillman. Efficiency refinements in contact strategies and algorithms in explicit FE programming. In D. R. J. Owen, E. Onate, and E. Hinton, editors, *Computational Plasticity*, pages 457–481, Swansea, UK, 1992. Pineridge Press.

[7] G. Lonsdale, J. Clinckemaillie, S. Vlachoutsis, and J. Dubois. Communication requirements in parallel crashworthiness simulation. In *HPCN Europe 1994 - The International Conference and Exhibition on High-Performance Computing and Networking, Munich*, 18-20 April 1994.

[8] G. Lonsdale, B. Elsner, J. Clinckemaillie, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Experiences with industrial crashworthiness simulation using the portable, message-passing PAM-CRASH code. In *HPCN Europe 1995 - The International Conference and Exhibition on High-Performance Computing and Networking, Munich*, May 1995.

[9] J. Clinckemaillie. Experience with MPP on the simulation of large models. In *PAM-96 Conference, Strasbourg*, November 1996.

[10] T. J. R. Hughes, T. Belytschko, and W. K. Liu. Convergence of an element-partitioned subcycling algorithm for the semi-discrete heat equation. *Numer. Methods Partial Diff. Eqns*, 3:131–137, 1987.

[11] P. Smolinski, T. Belytschko, and M. Neal. Multi-time-step integration using nodal partition. *Int. J. Numerical Methods Engineering*, 26:349–359, 1988.

[12] Ch. Farhat and S. Lanteri. Simulation of compressible viscous flows on a variety of MPPs: Computational algorithms for unstructured dynamics meshes and performance results. Technical Report No.2154, INRIA, January 1994.

[13] S. Lanteri. Parallel solutions of three-dimensional compressible flows. Technical Report No.2594, INRIA, June 1995.

[14] S. Lanteri. Parallel solutions of compressible flows using overlapping and non-overlapping mesh partitioning strategies. *Parallel Computing*, 22:943–968, 1996.

[15] I. Masters, A. S. Usmani, J. T. Cross, and R. W. Lewis. Finite element analysis of solidification using object-oriented and parallel techniques. *Int. J. Numer. Methods Engineering*, 40:2891–2909, 1997.

[16] K. T. Danielson and R. R. Namburu. Nonlinear dynamic finite element analysis on parallel computers using Fortran 90 and MPI. *Advances in Engineering Software*, 29:179–186, 1998.

[17] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Parallel transient dynamics simulations: Algorithms for contact detection and smoothed particle hydrodynamics. *J. Par. Distrib. Computing*, 50:104–122, 1998.

[18] G. Karypis. Metis home page. `www-users.cs.umn.edu/~karypis/metis`, 1998.

[19] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*. University of Tennessee, 1995.

[20] G. L. Goudreau, C. G. Hoover, A. J. DeGroot, and P. J. Raboin. Status of ParaDyn: DYNA3D for parallel computing. Technical Report UCRL-JC-123339, Lawrence Livermore National Laboratory, 1996.

[21] V. Bala et al. IBM external user interface for Scalable Parallel systems. *Parallel Computing*, 20(4):445–462, April 1994.

[22] J. O. Hallquist. *LS-DYNA3D theoretical manual*. Livermore Software Technology Corp., Livermore, 1991.

Appendix: Code for the packing and unpacking of forces

The packing/unpacking routines depend on a particular storage scheme of nodal forces – they are stored as a 3-D vector attribute of node. However, modification for other cases such as storage in a global 1-D array is trivial.

```
static void
pack_forces  (W_mpi_process_t *p, /* IN: process */
              W_mpi_comm_buff_t *cb /* IN: comm. buffer */)
{
  int i, nn = cb->num_to_send;
  W_node_t *n, **npa = cb->to_send;
  double *sbp = cb->sbuf;
  FOR (i, nn) {                  /* packing loop */
    n    = npa[i];
    *sbp = n->force_t.x; sbp++;
    *sbp = n->force_t.y; sbp++;
    *sbp = n->force_t.z; sbp++;
  }
}
```

Figure 23: Function to pack forces (node-cut partitioning).

```
static void
unpack_forces (W_mpi_process_t *p, /* IN: MPI process */
               W_mpi_comm_buff_t *cb /* IN: comm. buffer */)
{
  int i, nn = cb->num_to_recv;
  W_node_t *n, **npa = cb->to_recv;
  double *rbp = cb->rbuf;
  FOR (i, nn) {                  /* unpacking loop */
    n    = npa[i];
    n->force_t.x += *rbp; rbp++; /* increment force */
    n->force_t.y += *rbp; rbp++;
    n->force_t.z += *rbp; rbp++;
  }
}
```

Figure 24: Function to unpack forces (node-cut partitioning).