Scalability of Finite Element Applications on Distributed-Memory Parallel Computers

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

Citation

Citable link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:25620446

Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA
Scalability of Finite Element Applications on Distributed-Memory Parallel Computers

Zdeněk Johan
Kapil K. Mathur
S. Lennart Johnsson
Thomas J.R. Hughes

TR-16-94
June 1994

Parallel Computing Research Group
Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts
Scalability of Finite Element Applications on Distributed-Memory Parallel Computers

Zdeněk Johan, Kapil K. Mathur, S. Lennart Johnsson
Thinking Machines Corporation
245 First Street, Cambridge, MA 02142, USA

Thomas J.R. Hughes
Division of Applied Mechanics, Stanford University
Durand Building, Stanford, CA 94305, USA

April 25, 1994

Thinking Machines Technical Report

Submitted to the Proceedings of the Symposium on
Parallel Finite Element Computations

Abstract. This paper demonstrates that scalability and competitive efficiency can be achieved for unstructured grid finite element applications on distributed memory machines, such as the Connection Machine CM-5 system. The efficiency of finite element solvers is analyzed through two applications: an implicit computational aerodynamics application and an explicit solid mechanics application. Scalability of mesh decomposition and data mapping strategies are also discussed. Numerical examples that support the claims for problems with an excess of fourteen million variables are presented.
1. Introduction

Industrial design using finite element analysis software often requires solving problems involving hundreds of thousands or even several million degrees of freedom. Distributed-memory parallel systems have been shown to be a useful alternative to classical vector supercomputers when solving those large-scale problems because of their potentially greater computing power and their large memory. However, scepticism sometimes arises as how finite element techniques would perform on parallel systems equipped with a large number of processors. We intend to show that the major components of a finite element application can use efficiently such parallel computers, regardless of the number of processors.

An outline of this paper follows. In Sections 2 and 3, we address the structure of data-parallel finite element applications and some of the scalability issues. Section 4 describes the mesh decomposition and data mapping procedures and analyzes their scalability using several unstructured meshes. Section 5 presents the performance of two finite element formulations on several Connection Machine CM-5 system configurations. Finally, conclusions are drawn in Section 6.

2. Unstructured grid finite element applications

Implementations of finite element applications for fluid dynamics [2, 4, 11] and structural analysis [1, 3, 21] on the Connection Machine systems have shown that the following generic program structure can be used to describe such applications:

1. Decompose the unstructured mesh and map the nodes and the elements to the processors
2. Preprocess the mesh connectivity for gather and scatter operations
3. Loop over the number of time steps:
   4. Gather current solution from nodes to elements
   5. Calculate element residuals, and possibly element left-hand-side matrices
   6. Scatter residuals to the nodes
   7. Advance the solution using explicit or implicit iterative schemes

The solution update using an explicit solver involves only DAXPY operations, which are embarrassingly parallel. In implicit iterative solvers such as conjugate gradient or GMRES, the solution update phase requires DAXPY and dot-product operations, as well as matrix-vector products. The matrix-vector product is performed either in an element-by-element fashion using element left-hand-side matrices [17, 25] or using a matrix-free
strategy [12]. In both cases, the matrix-vector product is done through a gather-compute-scatter cycle similar to Steps 4 through 6 in the above pseudo-code. We can therefore describe a finite element program as being a loop over a certain number of iterations (iterations being either time steps or a combination of time steps and solver iterations) containing a gather-compute-scatter cycle.

Based on the above program structure, two data representations are needed: nodes and elements. The relationship between the two is commonly defined by the element nodes array \texttt{i'en} of dimension \(n_{el} \times n_{en}\). \(n_{el}\) is the number of elements in the mesh and \(n_{en}\) is the number of element nodes (e.g., \(n_{en} = 4\) for linear tetrahedron elements and \(n_{en} = 8\) for tri-linear brick elements). The array \texttt{i'en} is used to perform the gather and scatter operations.

### 3. Definition of scalability

The notion of \textit{scalability} is defined to be an \(n\)-fold increase in performance for the application being run on a distributed-memory computer, when both the number of processors and the size of the problem being solved are increased by a factor of \(n\). Simplistically, a scalable computing environment (which includes the user application, the computer hardware and the system software) allows the user to solve large problems on a large number of processors in about the same time as small problems in a small number of processors. The word “about” in the previous sentence reflects the influence of problem size on the numerical properties of the problem, such as the convergence of iterative solvers and the number of time steps required to solve finite element problems. This influence will appear on some of the simulation examples presented in the following sections.

We measure the \textit{problem size} in terms of the \textit{number of finite elements}. Most computations are performed at the mesh element level (Step 5 in the generic program structure of Section 2). Consequently, the number of elements is a good indicator of the total amount of computation required.

It can be easily seen that scalability can be achieved only if all computations proportional to the number of elements are done in parallel. Running any fraction of such computations sequentially will lead to a bottleneck past a certain problem size. However, scalability does not preclude doing a fixed amount of computations sequentially. One such example is the solution of the least-squares problem in the GMRES algorithm: the computations for this problem (whose size is only a function of the Krylov space dimension) are performed on the control processor of the CM-5 system [10]. Based on the program structure presented in Section 2 and on the definitions just given, the scalability of the
mesh decomposition and the finite element solver (i.e., the gather-compute-scatter cycle) are analyzed in the following sections.

4. Mesh decomposition

As mentioned in Section 2, both the elements and the nodes of the unstructured mesh are mapped onto the processors of the parallel system. We have designed a two-step procedure which performs these mappings:

1. First, the mesh is decomposed into element blocks made of adjacent elements.
2. The nodes are then mapped onto the processors using the above mesh partitioning as a criterion for choosing the placement of each node.

The objective of these mappings is to achieve as much locality between the nodes and the elements as possible to minimize data transfer between processors. In order to achieve the best computational load-balance possible in the finite element program itself, we constrain the elements and the nodes to be uniformly distributed across the processors, i.e., all processors hold the same number of elements (resp. nodes) except for the last processor, which gets whatever elements (resp. nodes) remain. Both the partitioning and the mapping are done on the parallel system.

The term “processor” used throughout this paper corresponds to a memory unit with independent processing capability. In the context of the CM-5, the term processor denotes a vector unit. We note that, for cache-based architectures, a much finer partitioning based on cache size may yield improved performance. Thus, the term processor should be viewed in the context of the memory architecture. Moreover, four processors are gathered in to a so-called processing node on the CM-5. Thus, in a CM-5 system there four times as many processors as processing nodes.

4.1. Recursive spectral bisection

The mesh decomposition into element blocks is performed using a Connection Machine Fortran implementation of the recursive spectral bisection (RSB) algorithm [13]. Connection Machine Fortran is a dialect of High Performance Fortran [18]. The RSB algorithm was initially proposed by Pothen et al. [23] to obtain separators for sparse matrices. Then, Simon [26] applied it to mesh partitioning and demonstrated that high quality decompositions could be obtained. The RSB algorithm is based on an iterative partitioning process which decomposes the whole mesh into two partitions, each of which in turn is decomposed into two partitions, and so on. The number of steps in this process
is therefore log₂(number of processors). The implementation of the algorithm is done such that all elements of the mesh are treated in parallel. It implies a two-level parallelization: one level on the partitions generated at a given stage of the recursive process and the other on the elements in each partition.

One strength of the RSB algorithm is that it requires only topological mesh information. The partitioning quality will therefore not be affected by regions of highly concentrated or highly distorted elements. In our implementation, we have based the RSB algorithm on the dual mesh connectivity array \texttt{idual} of dimension $n_{el} \times n_{faces}$, which defines the list of elements sharing a face with a given element. $n_{faces}$ is the number of element faces (e.g., $n_{faces} = 4$ for tetrahedral elements and $n_{faces} = 6$ for brick elements). This array can be easily generated from the element nodes array \texttt{ien}. The array \texttt{idual} is used to define the positive semi-definite Laplacian matrix $L$

$$L_{ij} = \begin{cases} -1, & \text{if elements } i \text{ and } j \text{ share a face;} \\ 0, & \text{otherwise.} \end{cases}$$

$$L_{ii} = -\sum_{j=1}^{n_{el}} L_{ij}$$

The reordering of the components of the eigenvector associated with the smallest non-zero eigenvalue provides a reordering of the elements before the bisection is applied. The computation of this eigenvector is performed using a parallel implementation of the Lanczos algorithm. The details of this implementation are presented in [13]. This algorithm is applied to the whole mesh during the first partitioning step, and to subsequent subdomains until the required number of partitions is obtained. The quality of the partitioning obtained with the RSB algorithm is often measured by counting the number of graph edge cuts. In the case where the RSB algorithm is based on the dual connectivity array, a graph edge is simply an element face shared by two elements. The number of edge cuts is defined as the number of faces shared by elements of different element blocks. It provides a qualitative measure of the amount of data to be exchanged between two partitions (or processors). It should be noted that the RSB algorithm can partition meshes containing several element types. Hendrickson and Leland [5] have defined a weighted RSB algorithm which can take into account variations in workload between the different element types, but the current parallel implementation does not include such a feature.

Several remarks can be made regarding the scalability of this algorithm:

1. The two-level parallelization presented above implies that the mesh decomposer performance will remain constant throughout the partitioning process since the parallel
system always processes the same number of data, namely the number of elements in the whole mesh. This implies that scalability as defined in Section 3.1 should be achieved for this implementation of recursive spectral bisection.

2. However, the total partitioning cost is not a sole function of the number of elements per processor. Even if both the number of processors and the problem are scaled by the same factor, the partitioning cost will still increase because the mesh decomposer will require additional partitioning steps (remember that the number of partitioning steps is equal to \( \log_2(\text{number of processors}) \)). Moreover, the number of Lanczos iterations required to solve the eigenvalue problem to a given tolerance depends strongly on the mesh being partitioned, generally increasing with the problem size, which in turn increases the partitioning time. Nonetheless, the scalability of the RSB implementation alleviates those cost increases by taking advantage of a larger number of processors when the problem size grows.

\[\text{Figure 1. Falcon Jet. View of surface mesh on the airplane.}\]
**Figure 2.** M6 wing. View of surface mesh on outer boundaries.

**Figure 3.** M6 wing. Decomposition into 16 subdomains.
The RSB algorithm has been implemented on the Connection Machine CM-5 system and is part of the Connection Machine Scientific Software Library [27]. It has been used to partition the following five tetrahedral meshes on five CM-5 systems of different sizes:

1. A mesh with 109,914 elements around a Falcon Jet airplane (see Figure 1) partitioned on a 32-processing node CM-5.

2. A mesh with 266,556 elements around an ONERA M6 wing (see Figures 2 and 3) partitioned on a 64-processing node CM-5.

3. A mesh with 575,986 elements around a generic commercial airplane (see Figure 4) partitioned on a 128-processing node CM-5.

4. A mesh with 1,010,174 elements around an F-18 fighter jet (see Figure 5) partitioned on a 256-processing node CM-5.

5. A mesh with 2,132,488 elements around an ONERA M6 wing partitioned on a 512-processing node CM-5. This is a refined version of the mesh described in item 2.

Figure 4. Commercial aircraft. View of surface mesh on the half-airplane.
Figure 5. F-18 fighter jet. View of surface mesh.

Table 1. Mesh characteristics of five partitioning examples.

<table>
<thead>
<tr>
<th>Example</th>
<th>No. of nodes</th>
<th>No. of elements</th>
<th>No. of graph edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Falcon Jet</td>
<td>19,417</td>
<td>109,914</td>
<td>217,669</td>
</tr>
<tr>
<td>M6 wing</td>
<td>48,011</td>
<td>266,556</td>
<td>527,966</td>
</tr>
<tr>
<td>Airliner</td>
<td>106,064</td>
<td>575,986</td>
<td>1,136,029</td>
</tr>
<tr>
<td>F-18</td>
<td>182,055</td>
<td>1,010,174</td>
<td>1,999,646</td>
</tr>
<tr>
<td>M6 wing (fine)</td>
<td>367,723</td>
<td>2,132,448</td>
<td>4,244,312</td>
</tr>
</tbody>
</table>
Table 2. Partitioning statistics for five examples. Lanczos tolerance is $10^{-3}$.

<table>
<thead>
<tr>
<th>Example</th>
<th>No. of processors</th>
<th>No. of partitions</th>
<th>No. of Lanczos iterations</th>
<th>Elapsed time</th>
<th>No. of edge cuts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Falcon Jet</td>
<td>128</td>
<td>128</td>
<td>1,156</td>
<td>44 s</td>
<td>22,926</td>
</tr>
<tr>
<td>M6 wing</td>
<td>256</td>
<td>256</td>
<td>1,413</td>
<td>76 s</td>
<td>57,063</td>
</tr>
<tr>
<td>Airliner</td>
<td>512</td>
<td>512</td>
<td>1,606</td>
<td>124 s</td>
<td>112,910</td>
</tr>
<tr>
<td>F-18</td>
<td>1,024</td>
<td>1,024</td>
<td>2,061</td>
<td>178 s</td>
<td>220,413</td>
</tr>
<tr>
<td>M6 wing (fine)</td>
<td>2,048</td>
<td>2,047</td>
<td>2,419</td>
<td>201 s</td>
<td>481,359</td>
</tr>
</tbody>
</table>

Figure 6. Partitioning time as a function of the number of partitions for five examples.

The number of nodes, number of elements and number of graph edges for each problem are summarized in Table 1. The number of elements per processor is not constant over those problems, but its variations are relatively small, thus allowing a useful evaluation of the mesh decomposer performance on several CM-5 configurations. Table 2 presents the partitioning timings, the number of Lanczos iterations required for each problem, and
the number of edge cuts generated by the RSB algorithm. Figure 6 shows the scalability of the parallel RSB implementation by giving the partitioning time as a function of the number of partitions for the five meshes we consider. One can note the reasonable increase in partitioning time despite much larger increases in problem size, number of partitions and Lanczos iterations.

4.2. Node renumbering

Once the elements have been reordered to obtain element blocks, the mesh nodes are renumbered using the following procedure:

1. Each element is assigned the element block number to which it belongs.
2. Each element sends the block number to the nodes it is associated with. Nodes receiving the same block number from their neighboring elements are marked as “interior nodes” and their location code is the block number received. The other nodes are marked as “boundary nodes” and they choose their location code arbitrarily from the block numbers they received using the randomization capabilities of the CM-5 data network hardware.
3. Nodes are ranked based on their location code with the constraint of having interior nodes ranked before boundary nodes for the same location code.
4. Nodes are assigned to the processors based on the their location code in the order obtained at Step 3. This strategy forces interior nodes to have a greater probability than boundary nodes of being assigned to the same processor as the elements they are associated with. Note, however, that because of the load-balance constraint that nodes be distributed evenly, as described at the beginning of Section 4, all nodes may not be assigned during this step.
5. Nodes which have not been assigned during Step 4 are distributed among the processors which still have room left in the order obtained at Step 3.

This procedure can be easily expressed in CM Fortran (or HPF). Parallelization occurs over the elements for Steps 1 and 2 and over the nodes for Steps 3 through 5. Renumbering the nodes of the meshes described in Section 4.1 takes about 0.3 seconds on the respective CM-5 configurations considered, which less than 1% of the partitioning time.

5. Finite element computations

This section presents performance results of two finite element programs: one program for computational aerodynamics and one program for solid mechanics.
5.1. Computational fluid dynamics application

A finite element program for solving the compressible Euler and Navier-Stokes equations has been written in CM Fortran (also known as CMF) and run on the CM-5 system [11]. The variational form of the equations is based on the Galerkin/least-squares formulation proposed by Hughes et al. and Johnson et al. [6–9, 14–16]. A shock-capturing term is added to the variational formulation to eliminate overshoots near discontinuities. The implementation is general enough to accommodate any element type (currently available elements are triangles and quadrilaterals in 2-D and tetrahedra, wedges and bricks in 3-D) and any order of numerical integration. An implicit time-marching scheme is used to solve the nonlinear problems arising from the finite element discretization. Each nonlinear system is linearized using a Newton-type method and each nonsymmetric system of equations is solved using a matrix-free preconditioned GMRES algorithm [12, 24, 25].

All examples presented in this section involve the computation of steady inviscid flows. We have therefore used a local time-stepping strategy where the time step in each element is a function of a global CFL number. One integration point per element was used. Only one Newton iteration per time step was performed and the GMRES tolerance was set to 0.1. All problems were initialized with a uniform free-stream flow. The finite element program was compiled using the CMF compiler version 2.1 and run on timeshared CM-5 systems running the operating system CMost 7.2. All computations were done in 64-bit arithmetic.

The five meshes described in Section 4.1 have been used to perform fluid flow computations. Flow characteristics, CM-5 system configurations, number of time steps and CFL number used for each computation are reported in Table 3. All computations, except the one on the refined M6 wing mesh (due to a lack of computing time), have been found to be converged to engineering accuracy (3 digits). Monitoring of the aerodynamic forces on the lifting body allowed us to assess convergence of the solution. The mapping strategies presented in Section 4 were used in all cases. The gather and scatter operations were performed using communication primitives provided by the Connection Machine Scientific Software Library. Fluid flow solutions are presented in [13].

Timings of computation, gather and scatter parts of the program are presented in Table 4 and their performance are given in Table 5. Several remarks can be made:

1. The time spent doing gather/scatter operations (which involve inter-processing node communication) in this program is about 20% of the total time for all problems, indicating that this application is computation-bound.
Table 3. Flow characteristics, CM-5 system configurations, number of time steps and CFL number used for parallel computations.

<table>
<thead>
<tr>
<th>Example</th>
<th>Mach No.</th>
<th>Angle of attack</th>
<th>No. of CM-5 nodes</th>
<th>No. of time steps</th>
<th>CFL number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Falcon Jet</td>
<td>0.85</td>
<td>1.0°</td>
<td>32</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>M6 wing</td>
<td>0.84</td>
<td>3.06°</td>
<td>64</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>Airliner</td>
<td>0.768</td>
<td>1.116°</td>
<td>128</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>F-18</td>
<td>1.5</td>
<td>0°</td>
<td>256</td>
<td>20 / 80</td>
<td>5 / 10</td>
</tr>
<tr>
<td>M6 wing (fine)</td>
<td>0.84</td>
<td>3.06°</td>
<td>512</td>
<td>150</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4. Elapsed times for the gather-compute-scatter cycles of each example considered. The number of cycles timed for each example is the number of time steps given in Table 3.

<table>
<thead>
<tr>
<th>Example</th>
<th>Computation</th>
<th>Gather</th>
<th>Scatter</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Falcon Jet</td>
<td>243 s</td>
<td>22 s</td>
<td>36 s</td>
<td>5 min 01 s</td>
</tr>
<tr>
<td>M6 wing</td>
<td>624 s</td>
<td>57 s</td>
<td>83 s</td>
<td>12 min 44 s</td>
</tr>
<tr>
<td>Airliner</td>
<td>439 s</td>
<td>47 s</td>
<td>67 s</td>
<td>9 min 13 s</td>
</tr>
<tr>
<td>F-18</td>
<td>579 s</td>
<td>69 s</td>
<td>97 s</td>
<td>12 min 25 s</td>
</tr>
<tr>
<td>M6 wing (fine)</td>
<td>861 s</td>
<td>99 s</td>
<td>142 s</td>
<td>18 min 22 s</td>
</tr>
</tbody>
</table>

Table 5. Performance of gather-compute-scatter cycles for each example considered.

<table>
<thead>
<tr>
<th>Example</th>
<th>Computation Mflops/s/node</th>
<th>Gather Mbytes/s/node</th>
<th>Scatter Mbytes/s/node</th>
<th>Overall Mflops/s/node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Falcon Jet</td>
<td>30.4</td>
<td>18.8</td>
<td>14.6</td>
<td>24.5</td>
</tr>
<tr>
<td>M6 wing</td>
<td>30.0</td>
<td>18.5</td>
<td>15.7</td>
<td>24.5</td>
</tr>
<tr>
<td>Airliner</td>
<td>32.6</td>
<td>17.2</td>
<td>16.0</td>
<td>25.9</td>
</tr>
<tr>
<td>F-18</td>
<td>31.2</td>
<td>14.8</td>
<td>13.0</td>
<td>24.2</td>
</tr>
<tr>
<td>M6 wing (fine)</td>
<td>30.5</td>
<td>16.2</td>
<td>14.0</td>
<td>23.8</td>
</tr>
</tbody>
</table>
2. Execution rates per processing node for the computational part of the solver and gather/scatter bandwidths per processing node remain fairly constant over all test problems and CM-5 configurations considered. This is a clear indication of the hardware, system software and finite element solver scalability.

3. The overall execution rate per processor is also almost constant for all examples considered, which leads to the linear scalability plot shown in Figure 7. One should note that the solver performance is close to 20% of the peak hardware performance, indicating that a computational efficiency comparable to that of similar applications on conventional computers can be achieved.

![Figure 7. Summary of CFD code performance results on several CM-5 configurations.](image)

5.2. Explicit finite element analysis

A full three-dimensional analysis of specimens subject to impact loading is carried out using a transient 3-D finite element program written in CMF [20–22]. An elastic-viscoplastic constitutive relation for a porous plastic solid is used to model ductile fracture by the nucleation and subsequent growth of voids to coalescence. Cleavage is modelled in
terms of attaining a critical value of the maximum principal normal stress over a specified material volume. A convected coordinate Lagrangian formulation is employed and the discretization is based on 20-node brick elements with $2 \times 2 \times 2$ Gauss points. The equations of motion are integrated numerically by an explicit integration procedure using a lumped mass matrix. The predictions of the full three dimensional calculations for Charpy V-notch specimens are compared with those of corresponding plane strain analyses.

For this problem we use three data representations (compared to two for the aerodynamics analysis). The first representation maps unassembled finite elements in the mesh onto processors of the architecture. The second representation maps assembled nodal degrees of freedom onto processors. Finally, the third representation maps cleavage grains onto processors. There are two kinds of data interactions. First, there is data interaction between the unassembled finite elements and the assembled nodal degrees of freedom [19]. Next, there is data interaction between the cleavage grains and the unassembled finite elements. These interactions result in communication between the processing nodes. The parallel RSB algorithm, described in Section 4.1, is used to partition the finite element mesh on to the processors. Nodal degrees of freedom that are interior to a partition are mapped to the processor to which the partition is assigned. Boundary nodes must be assigned to one of the partitions among which they are shared, or replicated among the partitions among which they are shared. Only boundary nodes require communication between processing nodes. In our implementation boundary nodes are assigned to the partitions among which they are shared in an arbitrary manner. Cleavage grains that are interior to a partition are mapped to the processor to which the partition is assigned. Boundary cleavage grains are assigned to one of the partitions among which they are shared. This mapping scheme maximizes locality of reference and minimizes the contention for the finite number of communication channels.

The implementation is structured in a way such that the body of the time stepping loop

1. performs a gather operation from the assembled nodal degrees of freedom to the unassembled finite elements,
2. computes the unassembled nodal forces and maximum principal stresses at all quadrature points,
3. scatters the element-wise principal stresses to the cleavage grains to compute the volume average of the maximum principal stress over the grain,
4. gathers the volume average of the maximum principal stress to the group of unassembled finite elements,
5. modifies the unassembled nodal forces to account for elements that have cleaved, and finally
6. scatters the unassembled nodal forces to the group of assembled nodal degrees of freedom.

Table 6. Overall Performance of the explicit finite element code, TERAFRAC.

<table>
<thead>
<tr>
<th>No. of CM-5 nodes</th>
<th>Elements</th>
<th>Equations</th>
<th>Overall Gflops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>36,864</td>
<td>475,875</td>
<td>0.8</td>
</tr>
<tr>
<td>64</td>
<td>73,728</td>
<td>946,275</td>
<td>1.7</td>
</tr>
<tr>
<td>128</td>
<td>147,456</td>
<td>1,846,467</td>
<td>3.0</td>
</tr>
<tr>
<td>256</td>
<td>294,912</td>
<td>3,671,619</td>
<td>6.3</td>
</tr>
<tr>
<td>512</td>
<td>589,824</td>
<td>7,272,963</td>
<td>13.0</td>
</tr>
<tr>
<td>1024</td>
<td>1,179,648</td>
<td>14,461,827</td>
<td>26.6</td>
</tr>
</tbody>
</table>

Figure 8. Specimen geometry for a Charpy V-notch test with finite element meshes at time = 0 and time = 600 μs. Each finite element is a 20-node brick.
After the first gather operation, all the relevant data are local to a processor. The computation of the unassembled force values, therefore, requires no additional communication. The reader is referred [21] for a detailed description. Table 6 shows the overall performance of the gather-scatter loop as a function of for different finite element meshes and the configuration of the computing platform. For the simulations reported here, approximately 5% of the total time is spent in the gather operation and 5% of the total time is spent in the scatter operation, yielding a total communication time percentage of 10%. A specimen geometry is depicted in Figure 8. A detailed description of the meshes is available in [21].

6. Conclusions

We have presented a performance analysis of finite element applications. It has been shown that both the data decomposition/data mapping strategies and the actual finite element solvers are able to achieve scalability, therefore taking advantage of an increasing number of processing nodes. Scalability of these applications leads to very high performance on large parallel systems when solving large-scale unstructured mesh problems.

Acknowledgements

The authors would like to thank Dassault Aviation for providing them with the Falcon Jet mesh, Jean Cabello and Rainald Löhmer (The George Washington University) for giving them the M6 wing and F-18 meshes and Dimitri Mavriplis (ICASE/NASA Langley) for giving them the commercial aircraft mesh. The explicit finite element analysis described here was a collaborative effort with Alan Needleman of Brown University and Viggo Tvergaard of the Technical University of Denmark. Access to Connection Machine systems was provided by the Los Alamos Advanced Computing Laboratory, the Army High Performance Computing Research Center at the University of Minnesota, the National Center for Supercomputing Applications at the University of Illinois Urbana-Champaign, and the Navy Research Laboratory.

References


