Nonlinear Explicit Transient Finite Element Analysis on the Intel Delta

Edward J. Plaskacz,[†] Martin R. Ramirez[‡] and Sanjeev Gupta[‡]

[†]Argonne National Laboratory, Argonne, IL 60439, U.S.A.

[‡]Department of Civil Engineering, The Johns Hopkins University,
Baltimore, MD 21218, U.S.A.

ANL/CTD/CP--78758

ABSTRACT

DE93 009561

Many large scale finite element problems are intractable on current generation production supercomputers. High-performance computer architectures offer effective avenues to bridge the gap between computational needs and the power of computational hardware. The biggest challenge lies in the substitution of the key algorithms in an application program with redesigned algorithms which exploit the new architectures and use better or more appropriate numerical techniques. A methodology for implementing nonlinear finite element analysis on a homogeneous distributed processing network is discussed. The method can also be extended to heterogeneous networks comprised of different machine architectures provided that they have a mutual communication interface. This unique feature has greatly facilitated the port of the code to the 8-node Intel Touchstone Gamma and then the 512-node Intel Touchstone Delta. The domain is decomposed serially in a preprocessor. Separate input files are written for each subdomain. These files are read in by local copies of the program executable operating in parallel. Communication between processors is addressed utilizing asynchronous and synchronous message passing. The basic kernel of message passing is the internal force exchange which is analogous to the computed interactions between sections of physical bodies in static stress analysis. Benchmarks for the Intel Delta are presented. Performance exceeding 1 gigaflop was attained. Results for two large-scale finite element meshes are presented.

1. INTRODUCTION

Finite element analysis is the dominant tool in the numerical solution of the partial differential equations governing the nonlinear transient analysis of structural or continuum problems and in a variety of other areas as well. Applications range from stress analysis and aerodynamics of aircraft, automotive, and ship structures; steel and reinforced-concrete bridges and buildings; earthquake response of multistory structures and reservoir-dam systems; fracture mechanics; dynamics of semi-submerged offshore structures; fibrous composites; fluid mechanics; sonar transducers; acoustic and electromagnetic fields; magnet design; plasma flows; nuclear reactor flux computation; geomechanics; seepage and groundwater flow; oil- and gas-reservoir engineering; biomechanics; pollutant dispersion in the atmosphere, lakes, and tidal estuaries; surface waves; ignition and combustion.

Widespread application of the technique as a design tool for linear static analysis has been prevalent for thirty years. The treatment of dynamic and nonlinear problems entails an increase in the number of numerical operations. The widespread availability of high-speed digital computers during the 1970's extended the range of applications to dynamic and nonlinear analyses.

The 1980's witnessed the introduction of vector and coarse grained parallel supercomputers. Although the speed of the computational hardware has increased dramatically over the past thirty years, it is still insufficient for many important engineering applications. Many production engineering decisions must be made on the basis of crude computational models because the computational hardware to conduct full-scale simulations is not readily available.

For example, in studies of crashworthiness, impact and penetration, it is not unusual for an analysis to require 100 hours of CPU time on current supercomputers despite the simplicity of the models being studied. The computational requirements for a single Belytschko-Tsay shell element (used extensively in finite element analysis) with five integration points through the thickness are on the order of 3500 floating point operations per time-step and 600 words of memory (Plaskacz (1990)). Crashworthiness models currently consist of approximately 5000 to 100,000 shell elements and run for about 100,000 time steps. Incorporation of detailed passenger compartment, air-bag, and occupant models can easily double or triple these requirements. The difficulty in obtaining blocks of time this large in a production environment severely impairs the role of such simulations in engineering decisions. Attention has focused on high-performance computer architectures as an effective avenue to bridge the gap between computational needs and the power of computational hardware. With these powerful tools at their disposal, engineers will be able to tackle sophisticated simulations which are currently intractable without the use of empirical data and without resorting to the separate modeling of each region so as to break up the simulation into components which are manageable.

The performance of the currently dominant sequential Von Neumann computer architecture is asymptotically approaching a peak speed prescribed on the underlying hardware by manufacturing and fundamental physical limits. One such is the speed of light (3×10^8 m/sec in a vacuum) which results in a signal transmission speed for silicon of at most 3×10^7 m/sec (DeCegama (1989)). Hence, a 3 cm diameter chip can propagate a signal in 10^{-9} seconds. The maximum power of a computer built with such chips is at most 10^9 floating-point operations per second (1 GFLOPS). A similar limit can be estimated for other materials such as gallium arsenide, which has a lower signal propagation time. Since the speed of light is the ultimate barrier, sequential processors will reach the maximum speed physically possible in the near future. It is clear that computer architectures other than the classical serial Von Neumann model must be exploited to realize significant improvements in computational speeds.

Another reason for considering parallel machines is their cost effectiveness. The relationship between incremental cost and processor performance enhancement follows a law of diminishing return. As the underlying physical limit is approached, the incremental cost for a given incremental increase in processor computational power rises dramatically. Processors having a comparatively low speed are available at a very low cost. A parallel machine comprised of many such processors not only has more speed than a uniprocessor supercomputer but is also less expensive (Dongarra and Duff (1992)).

Parallel architectures require changes in the algorithms and the underlying code. The programmer must consider details of the computer architecture to a degree unprecedented in the SISD era. The computer architecture also dictates the type of algorithm which is most appropriate. For example, an order-of-magnitude increase in the computational speed over a conventional supercomputer has been attained in research conducted at Argonne for an explicit transient program for the three-dimensional nonlinear analysis of shell structures on a single-instruction multiple-data (SIMD) supercomputer, the Connection Machine. This speedup was

difficult to obtain, requiring substantial redesign of the algorithm and underlying data structures, described in Belytschko, Plaskacz, Kennedy, and Greenwell (1990) and Belytschko and Plaskacz (1992). The salient feature of the "Exchange" algorithm is the minimization of interprocessor communication at the expense of redundant computations and storage. Because the computational capabilities of the Connection Machine far exceed its communication abilities, the Exchange algorithm is much faster on the Connection Machine-2 than alternate schemes in spite of the redundancies.

Single-Instruction Multiple-Data fine-grained massively-parallel computers such as the Connection Machine-2 favor an element-per-processor data structure; interelement communication and interprocessor communication become equivalent. On any parallel platform with a coarser granularity, larger subdomains are necessary and interdomain communication becomes more complicated. In previous work, the combination of the Exchange algorithm and the Argonne-developed P4 primitives has provided a powerful approach for the adaptation of an explicit transient finite element program for use on a wide range of parallel platforms. The Fortran-callable P4 primitives allow the programmer to pass data (scalar, array, or common block) among cooperating processes through Berkeley UNIX 4.3BSD interprocess communication sockets.

Once the algorithm is restructured as a set of processes communicating through messages, the program can run on systems as diverse as a uniprocessor workstation, multiprocessors with and without shared memory, a group of workstations that communicate over a local area network, or any combination of the above. However, careful attention must be paid to the minimization of communication and synchronization costs. The basic concepts were first tested on a network of workstations described in ASDAC PR-2 (1991). The extension of this work to an Intel hypercube architecture is described in Plaskacz, Ramirez, and Gupta (1992). Benchmarks for networks of workstations and shared and distributed memory MIMD multiprocessors were presented in ASDAC PR-5 (1992) and Plaskacz (1993).

In this paper, the performance of the code will be tested in a computationally more demanding setting, large-scale finite element meshes on the 512-processor Intel Touchstone Delta MIMD supercomputer. The Intel Touchstone Delta system is jointly owned by Argonne National Laboratory and 12 other members of the Concurrent Supercomputing Consortium. The system supports various types of processing nodes (numeric, mass storage, gateway and service). Numeric nodes form the computational core of the system. Mass storage nodes enable the numeric nodes to access the tape and disk drives in the mass storage system. Gateway nodes provide access to the system over LAN lines. The service nodes handle resource management, such as enabling the user to log into the system from the network and to obtain and release a collection of numeric nodes. The Delta consists of 528 processing nodes, each with 16 megabytes of local memory and a RISC processor, the i860, giving total peak theoretical performance of 32 billion floating point operations per second. The nodes are interconnected as a two-dimensional mesh. Interprocessor communication is handled by mesh router chips. One mesh router chip is installed to the routing plane for each node in the mesh. The Delta has an aggregate memory capacity of more than 8 gigabytes, and secondary (disk) storage of 90 gigabytes.

An outline of this paper is as follows: The finite element formulation is presented in Section 2. Section 3 presents an overview of distributed parallel computing in the context of nonlinear explicit transient finite element analysis via message passing. Both synchronous and asynchronous algorithms are presented. Section 4 presents benchmarks comparing the performance of the Intel Delta versus a vector supercomputer. Section 5 defines performance metrics and an ex-

tension of Amdahl's law to distributed parallel computing. Section 6 presents a description of a partially asynchronous algorithm. A detailed tabulation of execution times for various phases of the finite element code is examined to determine why the asynchronous algorithm performed below expectations. Section 7 summarizes and draws conclusions from all tests performed.

2. EXPLICIT NONLINEAR FINITE ELEMENT ALGORITHM

After a spatial semi-discretization is performed, the governing equations of motion for the finite element model are

$$\mathbf{M}\ddot{\mathbf{u}}(t) = \mathbf{f}(t) \quad (1)$$

$$\mathbf{f}(t) = \mathbf{f}_{ext}(t) - \mathbf{f}_{int}(t) \quad (2)$$

Since the equation of motion must be satisfied at each time step, we have

$$\mathbf{M}\ddot{\mathbf{u}}^n = \mathbf{f}^n \quad (3)$$

$$\mathbf{f}^n = \mathbf{f}_{ext}^n - \mathbf{f}_{int}^n \quad (4)$$

for time step n . Equation 3 is integrated in time. The central difference algorithm implemented in the program proceeds through six steps.

Step 1, Initial conditions: The solution of a second-order ordinary differential equation requires two initial conditions i.e. $u(0), \dot{u}(0)$. Define

$$\mathbf{u}^0 = \mathbf{u}(0), \dot{\mathbf{u}}^{1/2} = \dot{\mathbf{u}}(0), n = t = 0 \quad (5)$$

Step 2, Loop over elements:

$$\mathbf{u}_e^{n+1} = \mathbf{L}_e \mathbf{u}^{n+1} \quad (6)$$

$$\epsilon_e^{n+1} = \mathbf{B}_e \mathbf{u}_e^{n+1} \quad (7)$$

$$\sigma_e^{n+1} = \sigma_e^n + \Delta\sigma_e = f(\epsilon_e^n, \epsilon_e^{n+1}) \quad (8)$$

$$\mathbf{f}_{int,e}^{n+1} = \int_{\Omega_e} \mathbf{B}_e^T \sigma_e^{n+1} d\Omega \quad (9)$$

Step 3, Assembly:

$$\mathbf{f}_{int}^{n+1} = \sum_e \mathbf{L}_e^T \mathbf{f}_{int,e}^{n+1} \quad (10)$$

$$\mathbf{f}_{ext}^{n+1} = \sum_e \mathbf{L}_e^T \mathbf{f}_{ext,e}^{n+1} \quad (11)$$

$$\mathbf{f}^{n+1} = \mathbf{f}_{ext}^{n+1} - \mathbf{f}_{int}^{n+1} \quad (12)$$

Step 4, New accelerations:

$$\ddot{\mathbf{u}}^{n+1} = \mathbf{M}^{-1} \mathbf{f}^{n+1} \quad (13)$$

Step 5, New velocities:

$$\dot{\mathbf{u}}^{n+3/2} = \dot{\mathbf{u}}^{n+1/2} + \Delta t \ddot{\mathbf{u}}^{n+1} \quad (14)$$

Step 6, New displacements:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \Delta t \dot{\mathbf{u}}^{n+1/2} \quad (15)$$

$t \leftarrow t + \Delta t; n \leftarrow n + 1; \text{ go to 2}$

In the above

- $\mathbf{M}$ = mass matrix;
- n = number of time step;
- t = time;
- $\mathbf{u}^n, \dot{\mathbf{u}}^n, \ddot{\mathbf{u}}^n$ = nodal displacements, velocities and accelerations, respectively at $t = n\Delta t$;
- $\mathbf{f}^n, \mathbf{f}_{ext}^n, \mathbf{f}_{int}^n$ = resultant, external and internal nodal forces, respectively at $t = n\Delta t$;
- ϵ^n, σ^n = strain and stress tensors of element e at $t = n\Delta t$;
- $\mathbf{B}_e$ = strain-displacement matrix of element e ;
- $\mathbf{L}_e$ = Boolean connectivity matrix of element e .

If $\mathbf{M}$ is a lumped mass matrix then there is no solution of simultaneous equations and the method is explicit. The shell element used in this study is described in Belytschko, Lin, and Tsay (1984). It is a four-node element using standard bilinear isoparametric shape functions. This element is widely used in explicit programs for nonlinear transient analysis primarily because of its speed. The speed results mainly from the use of a single stack of quadrature points in each element; for an elastic element, only one quadrature point is used. Thus, the element is typically four times as fast as fully integrated elements in explicit programs. Since one point quadrature element is rank deficient, it gives rise to spurious singular modes. Suppression of these spurious singular modes is given in Belytschko, Lin, and Tsay (1984) and Belytschko, Ong, Liu, and Kennedy (1984). The element is shown in Figure 1.

3. DISTRIBUTED PARALLEL COMPUTING

3.1 Domain Decomposition

A preprocessing utility is used which reads nodal coordinate, element connectivity, boundary condition, and loading data from an input deck, decomposes the mesh into subdomains, records subdomain adjacencies and generates an input deck for each processor describing its subdomain. This preprocessing program is based on algorithms and code described in Farhat (1988) and Al-Nasra and Nguyen (1990).

At each time step, subdomain computations proceed independently in parallel in accordance with the serial finite element algorithm given in Section 2. First, the element internal forces are computed (Step 2), followed by nodal force-vector assembly, and for parallel computations, an exchange of element internal forces for nodes at subdomain interfaces (Step 3). Finally, the equations of motion are integrated (Step 4, 5, and 6). The need for synchronization of processes arises from the fact that not all the steps of the algorithm can be run in parallel independently. At some point during each time step, messages must be exchanged to update

various arrays. The exchange of internal forces pertaining to interface nodes is required because, in Step 3 (Eqn. 10), contributions from elements in neighboring subdomains are not added in $\mathbf{f}_{int}$. No kinematic quantities i.e. displacements, velocities, accelerations are exchanged. The equations of motion for the interface nodes are integrated redundantly. Each subdomain is in itself a complete finite element mesh. Thus, in addition to requiring minimal communication, the exchange algorithm has the advantage of allowing analysts to use the same post-processing software for each subdomain as they would normally do for a job executing on a single processor machine.

3.2 Message Passing Paradigm

For a distributed network of workstations, the Argonne-developed P4 package described by Boyle et. al. (1987) and Butler and Lusk (1992) can be used for interprocessor communication. For the Intel mesh, the Intel message-passing routines are used; the code is portable as the message-passing libraries are identical in function and differ only in semantics. A processor sends a message by calling a *send* subroutine which reads the data from a buffer and transmits it to the receiving processor. A processor receives a message by calling a *receive* subroutine which takes the incoming message and stores the data in a buffer. These message passing calls can be synchronous or asynchronous in nature. A synchronous call blocks the calling process while an asynchronous call does not block the calling process but returns a message ID that is used later to determine when the call completes so that the buffer can be reused. It is important to note that the number of message IDs is limited, and an error occurs when no message IDs are available for a requested asynchronous *send* or *receive*. Therefore, message IDs need to be released as soon as possible by calling the system calls *msgwait()*, *msgdone()*, or *msgcancel()*. Details of these subroutines are given in Delta system references (ANL/MCS-TM-152 (June 1991)).

A synchronous algorithm for exchanging internal forces during each time step is presented in Table 1. Since all the message passing calls are synchronous, the requirements for buffer space are minimal. However, the price paid for low memory is high processor idle time since a processor must wait for the *send* subroutine to complete transmission before being able to proceed.

In Table 2 an algorithm is presented which uses an asynchronous *send* and a synchronous *receive*. An asynchronous *send* is useful only if the algorithm is designed so that it does not attempt to change the buffer being sent before it is completely sent. The basic idea exploited here is a fact inherent in the nature of the problem. Let us consider two neighboring subdomains x and y which need to exchange the forces at the interface nodes at each time step. x cannot execute the i^{th} time step unless the *isend()* call of y is completed at the $(i - 1)^{th}$ step. This is because before going to the i^{th} step, x will be waiting at *crecv()* to get the message from y which belongs to $(i - 1)^{th}$ step. Similarly y can not execute $(i - 1)^{th}$ step unless the *isend()* call of x is completed at the $(i - 2)^{th}$ step. In other words, if x is executing i^{th} step, the *isend()* calls of x pertaining to $(i - 2)^{th}$ step are complete. Hence the buffer can be used again in alternate time steps. For, various numerical studies presented later, the algorithm presented in Table 2 has been used.

Table 1: Message passing algorithm with **csend()** and **crecv()**

-
1. Time step $i=1$.
 2. Calculate internal forces at all the nodes.
 3. Exchange forces (send); Loop over all the neighboring subdomains.
 - a. Copy identity of current subdomain in buffer *fintex*(1).
 - b. Copy forces pertaining to the interface nodes in the next $6 \times N$ elements of buffer.
 - c. Call **csend()** to send buffer to the neighboring subdomain.
 - d. Use *fintex* again for the next neighboring subdomain.
 End sending forces.
 4. Exchange forces (receive); Loop over all the neighboring subdomains.
 - a. Call **crecv()** to receive a message and store it in buffer *fintex*.
 - b. Get the identity of the sending processor.
 - c. Identify the interface nodes.
 - d. Update the internal forces.
 - e. Use *fintex* again for the next neighboring subdomain.
 End receiving forces.
 5. $i \leftarrow i+1$; Go to 2.
-

3.3 Notion of Host and Node

Unlike other Intel machines such as the Gamma, the Delta does not support the host-node paradigm. One way to run a host-node program on the Delta is to write the code such that one processor of the mesh partition acts as a host while the remainder work as nodes. The difficulty with this procedure is that the resources of one processor are totally wasted as most of the time it is waiting for the other processors running the node code to complete. An alternative is to have one processor be the host as well as a node. However, this introduces unnecessary complications and causes extra work for one processor which may have an adverse effect on the overall performance.

In order to circumvent these problems, a strategy has been adopted which does not require a host program. A shell script is run which allocates a Delta partition, and loads node code directly on the processors of that mesh partition. Each node immediately starts working on its respective subdomain and communicates with other nodes for synchronization. All nodes terminate their execution as they reach the end of the program. The mesh partition is released when all the nodes finish their execution. The code is portable to other machines with minimal changes. For example, on the Intel Touchstone Gamma hypercube, the allocation of a cube, loading of node program along with *waitcube*, and release of the cube can all be done from the shell.

4. NUMERICAL STUDIES

Table 2: Message passing algorithm with **isend()** and **crecv()**

-
1. Time step $i=1$.
 2. Calculate internal forces at all the nodes.
 3. Exchange forces (send); Loop over all the neighboring subdomains.
 Use buffer *afint* and *bfint* for odd and even time steps respectively.
 - a. Copy identity of current subdomain in next element of buffer.
 - b. Store this location in *flag*.
 - c. Copy forces pertaining to the interface nodes in the next $6 \times N$ elements of buffer.
 - d. If time step is not 1 or 2 release the id, assigned by the **isend()** call for $(i - 2)^{th}$ time step, by calling **msgdone()**.
 - e. Call **isend()** to send buffer from location *flag* to the neighboring subdomain and store the id assigned by **isend()** in *aid* for odd time steps or in *bid* for even time steps.
 End sending forces.
 4. Exchange forces (receive); Loop over all the neighboring subdomains.
 - a. Call **crecv()** to receive a message and store it in buffer *fintex*.
 - b. Get the identity of the sending processor.
 - c. Identify the interface nodes.
 - d. Update the internal forces.
 - e. Use *fintex* again for the next neighboring subdomain.
 End receiving forces.
 5. $i \leftarrow i+1$; Go to 2.
-

4.1 Impulsively Loaded Cylindrical Panel

Since closed form solutions are not available for nonlinear transient problems, solutions obtained by finite elements are typically compared to experimental results. The cylindrical panel problem shown in Figure 2 has been used as a benchmark for many nonlinear transient programs. The material properties of the panel are summarized in Table 3. An initial velocity of 5650 in./sec. is applied to the 3.08 in. x 10.205 in. area indicated in Figure 2. The panel is simply supported at its ends and clamped at the sides.

Table 3: Material Properties and Parameters for Cylindrical Panel

Density	ρ	=	$2.5 \times 10^{-4} \text{ lb. sec.}^2/\text{in.}^4$
Young's Modulus	E	=	$1.05 \times 10^7 \text{ psi.}$
Poisson's Ratio	ν	=	0.33
Yield Stress	σ_Y	=	$4.4 \times 10^4 \text{ psi.}$
Plastic Modulus	E_P	=	0.0
Initial Velocity	v_o	=	5650 in./sec.
Quadrature Points Along e_3	n_q	=	5

Because of symmetry, only half the panel is modelled in the computations. Experimental results have been obtained for this shell by Morino, Leech, and Witmer (1971). Timings for an impulsively loaded cylindrical panel are given in Table 4. Three levels of discretization are presented: 1024 nonlinear shell elements, 8192 elements, and 16384 elements. A von Mises elastic-plastic material model with five integration points through the thickness was used. For all three cases, the timings include complete output of nodal coordinates, element connectivities, initial conditions and boundary conditions for each subdomain. In order to compute the Delta megaflop rate, the problem was also run on a CRAY X-MP/18. The timings recorded in Table 4 are wall-clock times for the Delta. Cray timings are CPU times measured from the vectorized code described in ASDAC PR-1 (1990).

For 1024 elements, substantial speedups were attained even though the data set was small. The problem was run for 1600 time steps. Nodal deflections for 31 nodes were output every eight time steps. Each processor read and wrote data to a unique pair of files. Thus, for interface nodes, the deflections were output redundantly.

The 67.3 CPU seconds measured on the Cray X-MP/18 corresponds to 68.28 MFLOPS. Hence the execution time of 21.2 seconds on 128 Delta nodes is about 3.17 times faster than Cray X-MP/18 and corresponds to 217 MFLOPS. It may be noted here that since Delta timings are wall-clock times, this rating of Delta is a lower-bound on performance. Further, timings reported here are for a very small data set. For the 128 node Delta run, there are only 8 elements per processor. If the problem is run on 512 nodes of the Delta, there will be only 2 elements per processor. This reduces the computation to communication time ratio considerably. Better performance was attained on the Delta with larger data sets.

The 8192 element mesh was run for 5000 time steps with nodal deflections for 2 nodes being output every hundred time steps. Figure 3 shows the mesh divided into 128 subdomains, each with 64 elements. The 1470 CPU seconds measured on the Cray X-MP/18 corresponds to 77.2

Table 4: Performance of cylindrical panel nonlinear shell problem on Delta: execution time (seconds) and *speedup* relative to a single processor

Elements	Number of nodes									
	1	2	4	8	16	32	64	128	256	512
1024	1354.3 (1.0)	688.7 (1.97)	355.3 (3.81)	186.1 (7.28)	98.4 (13.77)	55.1 (24.56)	30.3 (44.67)	21.2 (63.88)	22.1 (61.26)	
8192	33445.7* (1.0)	*	*	4468.6 (7.48)	2300.7 (14.54)	1176.5 (28.43)	622.9 (53.69)	340.4 (98.25)	192.1 (174.11)	116.6 (286.84)
16384	68927.7* (1.0)	*	*	*	4510.8 (15.28)	2317.4 (29.74)	1208.8 (57.02)	624.2 (110.43)	362.6 (190.09)	206.9 (333.15)

* Problem could not be run because of memory limitations/execution time very large.

One node timings are extrapolated from CRAY X-MP/18 as

$$T_{\text{delta}}(x) = \frac{T_{\text{Cray}}(x) \cdot \text{MFLOPS}(x) \cdot T_{\text{Delta}}(1024)}{T_{\text{Cray}}(1024) \cdot \text{MFLOPS}(1024)} \text{ where } x \text{ is } 8192 \text{ or } 16384 \text{ elements problem}$$

MFLOPS. Hence execution time of 116.6 seconds on 512 Delta nodes is about 12.61 times faster than Cray X-MP/18 and corresponds to 973 MFLOPS.

The 16384 element mesh, was also run for 5000 time steps with results for 2 nodes being output every hundred time steps. The 3010 CPU seconds measured on the Cray X-MP/18 corresponds to 77.7 MFLOPS. Hence execution time of 206.9 seconds on 512 Delta nodes is about 14.55 times faster than Cray X-MP/18 and corresponds to 1130 MFLOPS or 1.13 GFLOPS.

4.2 Automobile Body

Preliminary benchmarks were also obtained for the large-scale mesh of an automobile body shown in Figure 4. The model is comprised of 45087 nodes and 48268 elements. On 64 Delta processors, 2721.527 seconds were required for 5000 time steps. Figure 5 illustrates the domain decomposition for this run. The same simulation on 128 Delta processors required 1428.461 seconds. To minimize the effect of I/O on the timing, a third run was made with 128 Delta processors and 25000 time steps. The execution time of 5703.832 seconds corresponds to 4.7 $\mu\text{sec}/\text{element-cycle}$. In Plaskacz (1990) and Belytschko and Plaskacz (1992) an element-cycle times of 42 $\mu\text{sec}/\text{element-cycle}$ and 21 $\mu\text{sec}/\text{element-cycle}$ were reported for the CRAY XMP/14 and 16K processor Connection Machine Model 2, respectively.

5. EFFECTIVE PARALLELIZATION

The performance measurement in terms of MFLOPS is usually appropriate for a serial algorithm where it is compared with the maximum speed of the machine. It is also applied to measure the overall performance of parallel algorithms but is not very appropriate. This is because (1) extra work is done by a parallel computer in the background and (2) synchronization and communication overhead costs are not reflected in FLOPS. It also does not give any indication about how effectively the code has been parallelized.

The performance of parallel algorithms is most commonly measured in terms of speedup. Speedup of an algorithm executed using N processors is defined as

$$S = \frac{T_1}{T_N} \quad (16)$$

where

$$\begin{aligned} S &= \text{speedup} \\ T_1 &= \text{execution time using one processor} \\ T_N &= \text{execution time using } N \text{ processors} \end{aligned}$$

Ideally, T_1 should be the execution time for the best serial algorithm on a single processor. Since, the serial version of an algorithm is not developed along with the parallel version, T_1 can be very closely approximated by executing the parallel version using only one processor.

Amdahl's law (Amdahl (1962)) puts a limit on the speedup that can be achieved using parallel processing. If α (Amdahl's fraction) is the fraction of the sequential algorithm that can not be parallelized and must be run sequentially, then

$$T_N = \alpha T_1 + (1 - \alpha) \frac{T_1}{N} \quad (17)$$

$$S = \frac{N}{1 + (N - 1)\alpha} \quad (18)$$

for large N ,

$$S_{max} = \lim_{N \rightarrow \infty} S = \lim_{N \rightarrow \infty} \frac{1}{\frac{1}{N} + (1 - \frac{1}{N})\alpha} = \frac{1}{\alpha} \quad (19)$$

Hence, according to Amdahl's law, speedup approaches asymptotically to a number governed by the fraction of the sequential algorithm that can not be parallelized.

In Amdahl's law, it is assumed implicitly that the Amdahl's fraction α is a constant and depends only on the algorithm. In most engineering problems, α depends upon not only the algorithm but also the problem size. In an effective parallel algorithm, α tends to zero as problem size becomes large. This is because the execution time of parallel portion of the program, e.g. time integration, increases almost linearly with problem size while initial setup costs do not increase that rapidly. Hence, performance of a parallel machine is not limited and for a very large problem a speedup very near to the ideal speedup (N) can be achieved.

However, Amdahl's law is not directly applicable to distributed parallel processing. Here all processors execute their own program and work on a part of the problem. There is no part of the program which is not parallelized. It is not possible to differentiate various tasks as sequential or parallel. On the other hand, substructuring the problem may not necessarily divide all the tasks. Initial setup, opening and closing of files, input/output pertaining to the overall problem, etc. are the tasks which are not divided irrespective of the number of processors. As discussed later, there are tasks such as opening of files, where execution time increases with number of processors. Further overhead gets induced into a parallel algorithm due to interprocessor communication, synchronization delays, duplication of work at interface boundaries, etc.

Because of overhead, the speedup of a parallel algorithm is always lower than the ideal speedup. Based on this speedup, an *equivalent* Amdahl's fraction α can be calculated from Eqn. 18 as a

measure of fractional overhead as

$$\alpha = \frac{\frac{N}{S} - 1}{N - 1} \quad (20)$$

Note that this α represents overhead losses as if this fraction of the code was running serially. Efficiency of a parallel program is defined as

$$e = \frac{S}{N} \quad (21)$$

From equation 20 and 21,

$$\alpha = \frac{\frac{1}{e} - 1}{N - 1} \quad (22)$$

Hence, defining *effective* parallelization of the algorithm p as the fraction of the code that can be completely parallelized, we have

$$p = 1 - \alpha = \frac{N - \frac{1}{e}}{N - 1} = \frac{1 - \frac{1}{S}}{1 - \frac{1}{N}} \quad (23)$$

Defining,

$$T_{ex} = T_N - \frac{T_1}{N} \quad (24)$$

as excess time spent by a processor over and above the time required due to ideal speedup, from equation 16, 21 and 24,

$$T_{ex} = T_N(1 - e) \quad (25)$$

We introduce the notion of equivalent Amdahl's fraction as we are using α in a slightly different context for distributed computation because of the reasons discussed above. Although breaking up the tasks is not possible as Amdahl had assumed, we can still use α and p to get an overall idea about the code. Once we have the execution time of parallel code, α and p can be calculated from above equations. Then α represents the fraction of the code as if it runs in serial, in Amdahl's terms, and p represents the fraction of the code as if it is perfectly parallelized. There are no more overhead costs. In other words, we are transforming all overhead costs, due to communication, duplication of work, load imbalance etc., to equivalent fractions α and p .

Detailed performance analysis of two cases of the cylindrical panel problem is presented in Table 5. It is a common belief that the overhead costs increase with an increase in the number of processors. This would imply an increase in the equivalent Amdahl's fraction α . However, for both the small as well as large data set, it is observed that in general α decreases as the number of processors is increased. For 1024 elements, α decreased from 1.70 % to .69 % as number of processors working on the problem were increased from 2 to 64. A further increase in N caused the amount of work per processor to be very small and consequently α started rising. A maximum parallelization of 99.31 % in the code was attained even though the dataset was so small. For 8192 elements, α keeps decreasing from .98 % to .15 % as N is changed from 8 to 512. The corresponding effective parallelization of the code, p increases from 99.02 % to 99.85 %. For 16384 element problem, maximum parallelization of 99.89 % was attained for 512 processors.

Table 5: Detailed Performance Analysis on Intel Delta for nonlinear shell problems: execution time T (seconds), speedup S relative to a single processor, efficiency e , equivalent Amdahl's fraction α , effective parallelization p , and excess time spent T_{ex} (seconds)

Data Set	Elements	Time Steps	No. of PEs	T	S	e	α	p	T_{ex}
Cylindrical Panel	1024	1600	1	1354.3					
			2	688.7	1.97	.983	.0170	.9830	11.5
			4	355.3	3.81	.953	.0165	.9835	16.8
			8	186.1	7.28	.910	.0142	.9858	16.8
			16	98.4	13.77	.860	.0108	.9892	13.8
			32	55.1	24.56	.768	.0098	.9902	12.8
			64	30.3	44.67	.698	.0069	.9931	9.1
			128	21.2	63.88	.499	.0079	.9921	10.6
			256	22.1	61.26	.239	.0125	.9875	16.8
Cylindrical Panel	8192	5000	1	33445.7					
			8	4468.6	7.48	.936	.0098	.9902	287.9
			16	2300.7	14.54	.909	.0067	.9933	210.4
			32	1176.5	28.43	.888	.0041	.9959	131.3
			64	622.9	53.69	.839	.0030	.9970	100.3
			128	340.4	98.25	.768	.0024	.9976	79.1
			256	192.1	174.11	.680	.0018	.9982	61.5
			512	116.6	286.84	.560	.0015	.9985	51.3

6. PERFORMANCE OF THE ASYNCHRONOUS ALGORITHM

6.1 Implementation of the Asynchronous Algorithm

Table 5 clearly demonstrates that speedup increases as the problem size increases. For a sufficiently large problem, speedup increases as more processors are added but the efficiency or processor utilization decreases. This suggests that as more processors are used in a simulation, more time is spent working on something that is absent during the serial execution of the code. Although as shown earlier, excess time T_{ex} decreases as N increases, the total excess time spent by all the processors increases with N . For example, for 8192 elements and 8 Delta processors, 287.9 extra seconds are spent due to parallel execution of the code. This is equivalent to about 38 minutes on a single processor. For 8192 elements and 512 processors, the extra time spent is only 51.3 seconds but this corresponds to about 7.3 hours on a single processor. This time loss can be attributed to various tasks such as communication requirements, load imbalance etc.

One shortcoming posed by a synchronous communication scheme is that a barrier exists at the end of each time step. Time is lost while processors wait for the last processor to finish. This barrier is artificial and not necessary. Even though the computations across the entire finite element mesh have not been completed for the current time step, processors can start and complete the next time step, so long as their subdomains do not share interface nodes with subdomains still being worked on. Furthermore, even those processors allocated to subdomains sharing interface nodes with subdomains still being worked on, can begin the next time step for non-interface nodes and those elements not containing interface nodes. Therefore, it is reasonable to expect that a fully-asynchronous algorithm would perform better than the algorithm used in the present study (Table 2). Currently, during a time step, a processor sends forces to all its neighboring processors and then immediately waits for their corresponding messages by calling **crecv()**. This waiting time can be broken into two parts. (1) From the time **crecv()** is issued to the time the message actually arrives. This time would be zero if the message has already arrived. (2) The time taken to actually receive the message and store in the buffer. The use of an asynchronous *receive* could allow processors to begin computation for the subsequent time step before the messages are received. Delays attributable to interprocessor communication time would be minimized. However, conversion of **crecv()** to **irecv()** would also require protection of buffer and later on release of IDs assigned by **irecv()**. This would make the code quite complex.

An intermediate, less complicated approach would be to first, sort the nodes in each subdomain into two groups: interface nodes and non-interface nodes; next, the elements in each subdomain are sorted into two groups: elements incident on interface nodes and elements not incident on interface nodes. As discussed earlier, the exchange of internal forces is required (Step 3, Section 2) to update the interface nodes. During a time step, the processor sends forces to those processors allocated to neighboring subdomains. For elements which are not incident on interface nodes, all necessary information is already present and Steps 2 and 3 can be computed. For non-interface nodes, the equation of motion may be integrated (Steps 4, 5, and 6). Then **crecv()** is called to receive the messages. The internal force for the remaining elements can now be computed followed by the integration of the equations of motion for the interface nodes. This would limit the waiting time to the second phase discussed above. If sufficient time for the messages to arrive has been provided by rearranging computations, this approach is expected to work almost as fast as a fully-asynchronous algorithm. However, contrary to what was expected,

running the code in an asynchronous manner did not produce any execution time reduction for any problem for any number of processors.

6.2 Detailed Time Analysis

To examine why the asynchronous version of the program did not perform as expected, a detailed timing analysis was conducted. Calls to the system function `mclock()` were inserted in the code to measure the time required for the processor to complete various tasks such as opening input and output files; reading, processing and writing mesh data; processor busy and waiting time were also measured for each time step. The 8192 element cylindrical panel mesh was run on 28 nodes of the Delta for 55 time steps. The results of this time analysis are presented as a bar chart in Fig. 6. The legend for Figures 6-10 is given in Table 6. The legend includes each color used to designate a time interval followed by a description of the task being performed. Each column in graphs 6, 8 and 9 represents the activity of one processor. Each processor opens 1 separate input file and 2 separate output files, one for regular output and one for the output of timings.

Table 6: Legend for Figures 6-10

Figure No.	Color	Description
6,7,8,9	Cyan	Time to open input file.
	Green	Time to open first output file.
	Blue	Time to open second output file.
	Yellow	Time taken after opening files to the point when interface nodal masses are exchanged.
	Red	Time when a processor is waiting.
	White	Time when a processor is working.
10	Green	Processor 0 Activity.
	Blue	Processor 1 Activity.

In the first case (Fig. 6), the input files are already present in directory **input** however both sets of output files must be created in directory **output** before they can be opened. It can be seen that all processors spend almost 3 seconds executing only three statements which open the files for input and output. Actual reading or writing of data starts after all the three files have been opened by a processor.

This abnormally large amount of time prompted us to conduct a thorough analysis of the file opening time and ways to minimize it. A small code was executed for different number of nodes on the Delta where each processor opened 3 different (1 input, 2 output) files and immediately closed them at the end of the program. The maximum time taken by any processor in a particular run, is plotted in Fig. 7. In the first case, input files were present in the directory but output files were created by each processor. In the second case, all the files were present prior to code execution. As can be seen in Fig. 7, the file opening time increases almost linearly

with the number of processors, from a few milliseconds for 1 processor to a few seconds for 256 processors. It is much less when processors only have to open the files without having to create them. The timings shown here are upper bounds since many processors complete their tasks earlier. However, this gives a clear indication that file opening time is a task that grows as the number of processors is increased.

In Fig. 8, results are presented for the same 8192 element mesh but now with output files already present in the **output** directory. A considerable reduction in file opening time can be noted. A reduction of about 4 seconds was observed when the problem was run on 256 Delta processors. Since the file opening time is quite low when fewer files are opened, an effort was made to reduce this time by combining ten input files into one file. This required a processor to go down the file using blank read statements. In Fig. 9, results are presented for this case, however input file opening time includes the time it takes a processor to go down the file. The information about the number of lines to be skipped was included at the top of this consolidated input file. The triangular distribution of the file opening time suggests that executing lot of blank read statements is even more costly. Restructuring the code such that different processors open their files at different time, could be very helpful in reducing this time. Further improvements in parallel machines may also make this problem disappear. But at least, at present a good percentage of total execution time of moderate runs is spent in just opening the files. This time is minimized by keeping all the files being opened beforehand in the respective directories. These files could be of zero bytes.

Coming back to Fig. 6, 8 and 9, the next different lengths of yellow color represent that processors have different amount of work after opening files to the point where they exchange the interface nodal masses just before the time integration starts. Although domain decomposition algorithms decompose the mesh into subdomains with equal numbers of elements, they do not consider dividing this initial work equally. This difference arises from different number of boundary conditions, initial conditions, output requirements etc., present in different subdomains. The implications of this difference are rather serious in the sense that the processor that takes maximum time to reach the point of first exchange, causes all other processors to wait by sending a wave of waiting time in both directions. The propagation of this wave is highlighted by red color in Fig. 6, 8 and 9.

The wave of waiting time moves at a speed of one neighboring processor per time step. Processors that are not neighbors of the processor which originated the wave, start their time integration early but go in a waiting zone as soon as the wave passes them. There can be many such waves of different amplitudes originating from different processors. When two such waves meet, the one with less amplitude gets merged in the other and the net difference propagates. Because of this, the total waiting time of a processor may get fragmented into many smaller waiting periods. This can be observed in Fig 6, 8, and 9. Between two such periods, a processor might progress by one or more time steps, but the overall, total waiting time is governed by the processor taking maximum time to reach the point of first exchange. Hence, starting the time integration early, for a processor that clears the point of exchange of masses before some other processor in the problem, does not help at all in reducing the total execution time.

The same phenomenon repeats itself during each time step. Again, even though domain decomposition algorithms assign equal number of elements to each subdomain, the actual work load may not be the same. This could be because of presence of prescribed displacements; different boundary conditions, loading conditions, output requirements, and total number of

nodes; and most important, because of elements going in and out of plastic region. The processor which takes the maximum time to complete a particular time step, transmits a wave with this information and all other processors must wait at one point or another to make up for the difference.

Figure 10 illustrates the relationship between per-step execution time and processor waiting time. Results are presented for the 8192 element cylindrical panel mesh running on 2 Delta processors for 2500 time steps. Each color represents activity of one processor. As is clear from Fig. 10, per-step execution time changes over time and the processor with maximum execution time can be different at different times. The actual waiting time of a processor is also plotted in Fig. 10. The processor taking more time to complete a step has almost zero waiting time while the other processor has waiting time almost equal to the difference of execution times. This was found to be true for higher number of processors also.

These are the reasons which caused the failure of the asynchronous algorithm. Changing the code to asynchronous *receive* does not reduce the work load of the processor with maximum per step execution time. This processor receives all its messages almost immediately without any wait, as all its neighboring processors have finished their work much earlier and have sent their messages. On the other side, they have sufficient time to receive their message because as such they are going to wait at the next time step. Proper care must be taken to ensure that the processor with maximum per step execution time spends minimum time in communication. It is because of this processor only that the algorithm presented in Table 2 is better than the algorithm in Table 1. For other processors, it really does not matter as they have to either wait at `csend()` or `crecv()`. Again it is because of this processor only that good domain decomposition programs are required which pay extra attention to minimize communication load by minimizing number of interface nodes as well as number of neighboring subdomains.

When more processors are used in a problem, the total number of elements decrease in a subdomain and so the load imbalance due to different per step execution time. This reduces the excess time T_{ex} (Table 5) and increases the effective parallelism in the program. The actual communication time becomes important only when (1) per step execution time becomes so small that it is comparable to actual communication time (very small problem) or (2) there is at least one neighboring processor with per step execution time so close to the maximum time such that the difference of the two becomes comparable to actual communication time (very well decomposed problem). Because of this, for a very small problem such as 1024 element cylindrical panel problem, performance started dropping for 128 nodes when there were only 8 elements per processor.

7. CONCLUSION

The code was initially developed and tested on a distributed network of Sun SPARCstations (ASDAC PR-2 (1991)), followed by a port to the Intel Touchstone Gamma (Plaskacz, Ramirez, and Gupta (1992)). The major features of the present code are its medium grain parallelism which is made possible by the exchange algorithm; the buffered message passing procedure which provides increased performance by reducing processor idle time, and above all the code's portability. As is evidenced by the performance figures, the code performs fairly well even though it is a full-scale nonlinear dynamic finite element code. Analysis of execution time in terms of equivalent Amdahl's fraction α and effective parallelization p , suggests that parallelism in the

code increases not only with number of elements but also with number of processors. A detailed analysis of execution time proves that the current simplified buffering scheme works as good as a fully asynchronous buffering scheme. FEA codes intrinsically have very complex data flow mappings and thus for software robustness and maintainability it is desirable to minimize degree of modifications added for parallelization. As such, the uniqueness of the present algorithm lies not in its complexity but in its simplicity.

The allocation of finite elements among multiprocessors introduces synchronization costs which must be controlled to achieve maximum speedup. File opening time was identified as a task whose execution time increases linearly with number of processors. Small load imbalances were found to be the major cause of processor idle time. Since it is not possible to identify the processor with maximum per step execution time before hand, and also since this processor changes from time to time, there are limitations on how well a decomposition program can decompose a subdomain. Greedy algorithms are simple and try to balance processor computational loads by assigning equal number of elements to each subdomain. They neither minimize the number of subdomain interface nodes (message length) nor the number of neighboring subdomains (number of messages). Both of these problems are aggravated in an impact-contact simulation or by any adaptive methodology. Some results with neural network-based domain decomposition prove to be promising (Ramirez, Brar, and Plaskacz (1992)).

Our immediate objective is the formulation of a parallel domain decomposition algorithm with a long term objective being a dynamic domain decomposition algorithm able to alleviate load imbalances during the course of a simulation. The job of dynamic domain decomposition algorithm is going to be very challenging as the time we are losing i.e. the excess time T_{ex} reduces with number of processors. For 8192 elements on 512 processors for 5000 time steps, we are losing only 51.3 seconds. Assuming there is no time lost in opening of files and initial setup, this gives us about 10 milliseconds only at each time step.

8. ACKNOWLEDGEMENTS

This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Argonne National Laboratory.

The geometry for the automobile mesh was provided by Mr. Dawson Deumeyer of Cray Research, Inc.

The support of the U.S. Department of Energy under contract W-31-109-Eng-38 for one of the authors (EJP) is gratefully acknowledged. The support of the State of Illinois under grant number SCCA 90-82144 for all three authors is gratefully acknowledged. The support of the Department of Educational Programs at Argonne National Laboratory for two of the authors (MRR and SG) is gratefully acknowledged.

References

- [1] The Advanced Software Development and Commercialization (ASDAC) Project: Progress Report PR-1, E. Gallopoulos, Ed., ANL/TM 484 & CSRD Rpt. No. 1047, Argonne, IL 60439 & Urbana, IL 61801, Sept 1990.

- [2] The Advanced Software Development and Commercialization (ASDAC) Project: Progress Report PR-2, T. Canfield, M. Minkoff, and E. J. Plaskacz Ed. ANL/TM 488 & CSRD Rpt. No. 1129, Argonne, IL 60439 & Urbana, IL 61801, April 1991.
- [3] The Advanced Software Development and Commercialization (ASDAC) Project: Progress Report PR-5, E. Gallopoulos, Ed., Argonne, IL 60439 & Urbana, IL 61801, October 1992.
- [4] G. Amdahl, "The Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities," *Proceedings of the American Federation of Information Processing Societies*, 30, pp. 483-485, 1967.
- [5] T. Belytschko, J. I. Lin, and C. S. Tsay, "Explicit Algorithms for the Nonlinear Dynamics of Shells," *Computer Methods in Applied Mechanics and Engineering*, 42, pp. 225-251, 1984.
- [6] T. Belytschko, J. S. Ong, W. K. Liu, and J. M. Kennedy, "Hourglass Control in Linear and Nonlinear Problems," *Computer Methods in Applied Mechanics and Engineering*, 43, pp. 251-276, 1984.
- [7] T. Belytschko, E. J. Plaskacz, J. M. Kennedy, and D. L. Greenwell, "Finite Element Analysis on the CONNECTION Machine," *Computer Methods in Applied Mechanics and Engineering*, 81, pp. 229-254, 1990.
- [8] T. Belytschko and E. J. Plaskacz, "SIMD Implementation of a Nonlinear Transient Shell Program with Partially Structured Meshes," *International Journal for Numerical Methods in Engineering*, 33, pp. 997-1026, 1992.
- [9] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York, 1987.
- [10] R. Butler and E. Lusk, *User's Guide to the p4 Programming System*, Technical Report ANL-92/17, Mathematics and Computer Science Division, October 1992.
- [11] A. L. DeCegama, *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware*, Vol. 1, Prentice Hall, 1989.
- [12] J. Dongarra and I. S. Duff, "Advance Architecture Computers," *Supercomputing in Engineering Analysis*, ed. H. Adeli, Marcel Dekker, 1992
- [13] C. Farhat, "A Simple and Efficient Automatic FEM Domain Decomposer," *Computers & Structures*, 28, pp. 579-602, 1988.
- [14] L. Morino, J. W. Leech, and E. A. Witmer, "An Improved Numerical Calculation Technique for Large Elastic-Plastic transient Deformations of Thin Shells: Part 2 - Evaluation and Applications," *Journal of Applied Mechanics*, 38(2), pp. 429-436, 1971.
- [15] M. Al-Nasra and D. T. Nguyen, "An Algorithm For Domain Decomposition In Finite Element Analysis," *Computers & Structures*, 39, pp. 277-289, 1991.
- [16] E. J. Plaskacz, *Efficient Allocation of Computational Resources for Finite Element Applications*, PhD Dissertation, Northwestern University, Evanston, IL, June 1990.

- [17] E. J. Plaskacz, "Parallel Finite Element Analysis Via Message Passing," *submitted for publication*.
- [18] E. J. Plaskacz, M. R. Ramirez, and S. Gupta, "On Distributed Processing Applications in Finite Element Analysis," *Proceedings of the Engineering Mechanics Division*, ASCE, May 1992.
- [19] M. R. Ramirez, P. S. Brar, and E. J. Plaskacz, "A Neural Network Approach for Finite Element Domain Decomposition," *submitted for publication*.
- [20] *Using the Intel Touchstone DELTA System*, ANL/MCS-TM-152, Mathematics and Computer Science Division, June 1991.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

END

DATE
FILMED

6 / 8 / 93