Massively Parallel Multifrontal Methods for Finite Element Analysis on MIMD Computer Systems

R. E. Benner*

Sandia National Laboratories
Albuquerque, NM 87185

SAND--91-2016C

DE93 009838

ABSTRACT

The development of highly parallel direct solvers for large, sparse linear systems of equations (e.g. for finite element or finite difference models) is lagging behind progress in parallel direct solvers for dense matrices and iterative methods for sparse matrices. We describe a massively parallel (MP) multifrontal solver for the direct solution of large sparse linear systems, such as those routinely encountered in finite element structural analysis, in an effort to address concerns about the viability of scalable, MP direct methods for sparse systems and enhance the software base for MP applications. Performance results are presented and future directions are outlined for research and development efforts in parallel multifrontal and related solvers. In particular, parallel efficiencies of 25% on 1024 nCUBE 2 nodes and 36% on 64 Intel iPSC860 nodes have been demonstrated, and parallel efficiencies of 60-85% are expected when a severe load imbalance is overcome by static mapping and dynamic load balance techniques previously developed for other parallel solvers and application codes.

INTRODUCTION

The possibility of production computing on massively parallel (MP) computers began to gain credibility in 1988 with the demonstration of thousand-fold parallel speedups on a 1024-processor, first-generation hypercube for regular 2D PDE computations (Gustafson et al. 1988), backed by an expanding collection of parallelization techniques and strategies (cf., Fox et al. 1988), parallel linear algebra algorithms (cf., Heath et al. 1991), etc., that were emerging from diverse parallel processing research efforts. Within a year, an efficient, massively parallel implementation was demonstrated of a 30,000 line Fortran production code for generating simulated radar images, again on the first-generation nCUBE/ten hypercube (Gustafson et al. 1989).

As dozens of major applications have moved to MP systems the question has always remained: How will we approach the next application, how much effort will be required, and what will the resulting parallel algorithms, code and performance look like? Of particular concern to the author: the development of highly parallel direct solvers for large, sparse linear systems of equations (e.g. for finite element or finite difference models), which has been lagging behind progress in parallel direct solvers for dense matrices

* This work supported by the Applied Mathematical Sciences Program, U.S. Department of Energy, Office of Energy Research. It was done at Sandia National Laboratories, operated for the U.S. Department of Energy under contract number DE-AC04-76DP00789.

(e.g. Heath et al. 1991 and references therein, Sears 1990) and parallel iterative methods for sparse matrices.

This paper describes an approach to this issue and some preliminary results for a multifrontal solver, a direct solver routinely used in structural analysis and other finite element application codes (e.g., in fluid mechanics, combustion, etc.). A massively parallel multifrontal solver is being developed in an effort to address concerns about the viability of MP direct methods for sparse systems, as well as to improve the software base for MP applications.

The present investigation uses two MIMD (Multiple Instruction Multiple Data) parallel computing systems at Sandia. One is an nCUBE 2 hypercube with 1024 processor nodes and 4 MBytes of local memory per node. This system is of particular interest because it enables us to investigate scalability of parallel algorithms into the thousand-processor regime. Aggregate sustained performance of this system is 1 to 2 GFLOPS for many parallel applications, even though the individual processors are proprietary, 20 MHz scalar processors and, therefore, are relatively slow. The system is well balanced in terms of its computation and communication capabilities and it features a mature, robust and relatively flexible parallel debugger. Insofar as we have had several parallel applications at Sandia give performance on an Intel i860 processor node in the range of five to six times that of an nCUBE 2 processor, the latter also helps us to establish a realistic expectation of what sustained performance we might hope for the multifrontal solver to achieve on an iPSC 860 system.

The second system is an Intel iPSC860 hypercube with 64 nodes. Sixteen of the nodes have 32 MBytes of local memory apiece, while the remaining nodes have 8 Mbytes apiece. Although not massively parallel, the superscalar i860 processor nodes provide a testbed for parallel algorithm performance on superscalar MP architectures of the near future. The availability of 32 MBytes nodes is noteworthy, because it enables greater study of the scalability of the parallel solver as a function of problem size than is otherwise possible on the two hypercubes.

The following sections discuss frontal and multifrontal solvers, their history as parallel solvers, a massively parallel implementation, results, and future directions.

FRONTAL AND MULTIFRONTAL METHODS

A fundamental issue is the merit of direct methods relative to iterative methods. Direct methods are memory intensive, compute intensive, and, in general, have more challenging serial bottlenecks than iterative methods, so it is not clear that they are worth the effort required to parallelize them. It has not been established that parallel direct methods for large sparse matrices can even achieve high parallel efficiency on thousands of processors. That is, it is unclear that this class of numerical method is *scalable* when applied to sparse linear systems; in fact, some implementations are known to scale poorly.

Conversely, direct methods are still used in many applications because: (1) they are often more robust than preconditioned iterative methods, (2) suitable preconditioners have not emerged for iterative methods applied to some applications, and (3) they are applicable as a coarse mesh solver for multigrid methods. In addition, scalable parallelization techniques developed for sparse direct methods could be used, in part, in some parallel preconditioners and in hybrid direct-iterative methods.

Frontal solvers and their offspring, multifrontal solvers (Duff and Reid 1982), are the methods of choice for finite element analysis in several application areas, including solid mechanics, thermal analysis, and fluid mechanics. An excellent review of these methods is provided by Liu (1992), so only a brief

description of their characteristics is provided here.

Frontal solvers are an implementation of Gauss elimination tailored to sparse linear systems arising in finite element analysis, particularly in structural analysis applications. The frontal method was developed by Irons (1969) and Hood (1976) for symmetric and nonsymmetric systems of finite element equations, respectively. Given a computational domain that has been partitioned into finite elements, the solution on each element is represented by a polynomial basis and is obtained by assembly and sequential reduction of matrix equations one element at a time. Interleaving element assembly and reduction of equations makes the frontal method particularly memory efficient for finite element applications. (That is, insofar as any direct method can ever be called memory efficient.) The set of partially-assembled and partially-reduced equations, which are associated with the boundary between processed and unprocessed finite elements, is known as the *front*.

To understand the multifrontal concept and its parallel implementations, we must first consider the concept of parallel granularity. Coarse, medium and fine-grain parallelism can be roughly characterized as mesh, equation, and data-level parallelism. Multifrontal methods are inherently parallel at the coarse-grain level: each front, along which equations are assembled and reduced, passes through a set of elements or substructures which can be assigned to different processors in a parallel system. For example, each of the numbered regions shown in Figure 1 corresponds to a set of finite elements (perhaps hundreds or thousands of them) that are assigned to one processor in a coarse-grain parallel solver. Early MIMD multifrontal algorithms focused on 1D spatial decompositions of regular meshes, such as that shown in Figure 1, whereas 2D and 3D decompositions of regular domains are supported in the multifrontal solver discussed here, as well as many packages of parallel iterative solvers. The decomposition of irregular

One-Way Decomposition

Two-Way Decomposition

Figure 1. Examples of 2D spatial decomposition strategies for coarse-grain parallelism. Earliest MIMD parallel multifrontal methods were based on one-way spatial decompositions.

meshes, which will be discussed briefly below, is being addressed in the research community with mixed success.

The earliest MIMD implementations of multifrontal methods were coarse grained; current ones combine coarse and medium-grain parallel techniques. The present parallel solver starts with a coarse-grain decomposition and eliminates equations that are not shared with any other processor, followed by a medium-grain (i.e., equation-level) set of parallel computations during which equations shared by processors are eliminated in a nested dissection hierarchy. This MP implementation builds upon the work of Benner et al. (1987) on multifrontal methods on a ten-processor ELXSI 6400 shared memory multiprocessor, as well as algorithms developed by Lucas (1987) and Farhat and Wilson (1987) on the Intel iPSC2 hypercube. Each of the preceding algorithms is considered in turn, in order to motivate and clarify the present approach.

Coarse grain parallelism is severely limited in multifrontal solvers, as Benner et al. (1987) demonstrated with speedups of 3 to 3.4 on 4 CRAY or ELXSI processors, 4 on 8 ELXSI processors, and a projected 5 on 16 ELXSI processes. Speedup on the nCUBE/ten for the same algorithm asymptotes to 6 when more than 32 processors are used. Consider the high-level representation of the coarse grain approach given in Figure 2 for matrix reduction (the graph simply inverts for the backsubstitution process). As equations common to multiple processors are eliminated by nested dissection, an increasing number of processors is idled. Furthermore, significant load imbalance can be observed due to the workload imposed by different boundary conditions in different parts of the domain and due to the computational load on interior vs. exterior processor domains (e.g. processor domains 10 and 15 in Figure 1).

Figure 2. Coarse-grain parallel multifrontal method: allocation of complete separators A-C to individual processors is a significant serial bottleneck.

The logical next step is to invoke medium grained parallelism when solving separator equations. One such approach was taken by Farhat and Wilson (1987). Their algorithm reassembles the separator equations as a distributed, full matrix system (cf. Figure 3), which can then be handled with established techniques and software for parallel solution of full matrix equations. This algorithm provides a significant improvement over the coarse-grain approach, with 75% parallel efficiencies demonstrated on 16 processors of the Intel iPSC2. It requires a modest increase in software complexity. The major drawbacks are that data locality is sacrificed and the algorithm does not scale well: for large 2D and 3D linear systems the full matrix resulting from the separator equations easily exceeds the capacity and capability of current and projected parallel supercomputers.

Figure 3. A coarse/medium-grain parallel multifrontal method: the serial bottleneck is removed, but data locality and some medium-grain parallelism are sacrificed.

Lucas (1987) provided the first scalable multifrontal implementation. His approach, shown in Figure 4, teams progressively larger numbers of processors as the separators are processed in a nested dissection fashion. That is, at the first dissection level, pairs of processors cooperate to eliminate fully summed equations from a common submatrix, then groups of four processors work together at the second dissection level, and so on. At the final level, all processors work together on the modest-sized, full matrix associated with the last separator. Although Lucas also had only 16 processors of the Intel iPSC2 to use and systems of at most a few thousand equations to solve, his results are very encouraging, with parallel efficiency as high as 97% on 16 iPSC2 processors.

Figure 4. Massively parallel multifrontal method: bottlenecks in previous parallel versions are removed, at a cost of increased interprocessor communications and programming complexity.

While the basic idea has been successfully demonstrated, a number of critical issues remained to be resolved, including:

- The level of parallel efficiency that is achievable on a thousand (or a million) processors,
- The size of large sparse systems that can be solved with this method on current and projected systems,
- The use of parallel disk systems for efficient, out-of core multifrontal solution on MP systems,
- Pivoting strategies that can be supported efficiently in a MP multifrontal solver,
- Communication strategies that will most effectively enhance the scalability of the solver; and
- The implications of scalable multifrontal methods for 3D finite element analysis.

Our goal is to resolve most of these issues and outline the status of the remaining ones in the following sections.

MASSIVELY PARALLEL SOLVER

Figure 5 outlines the modularity of the MP multifrontal solver at the subroutine level. The parallel implementation of the solver is presented in terms of this structure at two levels. The first level is a summary of the preprocessing and solver packages, while the second is a more detailed description of the nested dissection portion of the solver.

Figure 5. MP multifrontal package: (a) high-level structure of the solver and preprocessing routines; (b) detailed structure of the nested-dissection package for parallel reduction and backsubstitution of separator equations.

MAPPING AND OTHER PREPROCESSING — Routine *prefront* sets up the mapping of matrix entries to processors and the preprocessing information required for matrix assembly and reduction prior to nested dissection; that is, all of the preprocessing information normally used by a serial frontal solver. This preprocessing data includes the so-called *heading* vectors, which are used to track which global and local equation numbers are associated with rows and columns of the frontal matrix.

The present research code handles regular 2D and 3D meshes (and 1D as well). Much of the planned work on the solver centers on mapping techniques and other preprocessing steps that will be required for solution of unstructured grid problems; the *prefrnt* routine will incorporate the resulting software for implementing mapping techniques for unstructured grids.

Two additional routines provide the information required for nested dissection of separator equations (i.e. the equations shared by processors). Routine *mfflags* sets flags that denote finite element nodes and equations that are shared and, hence, processed by nested dissection. Routine *prenest* sets up an initial, static assignment of separator equations to processors and data used to pinpoint the dissection level at which they are eliminated; this routine will also be heavily impacted by the incorporation of software to treat unstructured grids.

SYMBOLIC FACTORIZATION — A symbolic factorization package is a major portion of the front end to the solver. This package consists of 10 routines which will not be described in detail here - in essence these routines are copies of the 20 nested dissection routines in which all floating point operations have been removed.

Symbolic factorization is used here to estimate the memory requirements for a given problem and test them against the solver's memory allocation. These tests are critical in addressing potentially thorny

implementation issues for an MP multifrontal solver, such as the selection of values of parameters that govern memory allocation and use. In the MP multifrontal code these parameters include MAXEQ, which governs the maximum number of equations (e.g., 4.4 K) local to each processor; ICEFR, the maximum number of matrix entries (e.g., 230 K) in the active (frontal) matrix on each processor; and ICEQU, the maximum number of reduced equation entries and header data (e.g., 130 K) held in each processor's memory for backsubstitution. The symbolic factorization package uses about 10% of the multifrontal package's run time to check projected memory requirements against these parameters.

In principle, there is no guarantee that the memory-use estimates provided by symbolic factorization will be accurate, since the symbolic procedure cannot know what order of pivoting and elimination will be chosen during numerical factorization. In practice, we have found the estimates to be good to within 0.1 - 0.5 % for the test problems considered to date.

THE SOLVER — The high-level view of the solver, shown in Figure 5(a) includes a driver routine (mfsolve), the serial routine for reduction of frontal matrices (frontgn), the user-supplied routine which generates the problem-specific element stiffness matrices and element residual vectors (eqngen), a static condensation routine (statcon), and the serial routines that drive pivoting (pivotgn), storing of reduced equations (eqnsave), and backsubstitution (backsub).

Two assembly language library calls are made in the solver to accelerate the compute-intensive reduction loops of the multifrontal and nested dissection algorithms. DAXPY is a vendor-specific BLAS (Basic Linear Algebra Subroutine) routine, available on both hypercubes, that computes matrix-vector products.

Routine mfstimer gathers timings and related performance analysis data from all the processors and provides a detailed summary, which is selected in the results presented below. Three synchronization points are incorporated into the nested dissection package in order to capture performance statistics for the setup, matrix reduction, and backsubstitution portions of the parallel nested dissection algorithms. These artificial synchronizations occur in routines headsync, statsync, and backsync (Figure 5(b)).

The functionality of the nested dissection package is built upon that of the serial frontal solver outlined above. Nested dissection has a pair of driver routines for reduction (nestgen) and backsubstitution (nestback) of separator equations, a static condensation routine (neststat), and routines that drive pivoting (nestpiv) and storing of reduced equations (nestsave). A critical aspect of the parallel solver is handled by routines headswap and colsplit, in which processors exchange heading data at the start of a new nested dissection level (i.e. a new separator) and set an initial static assignment of matrix columns to processors for the dissection level. Routine eqmalloc moves matrix entries in the frontal matrix to make room for matrix entries that are subsequently exchanged in routine eqnswap with other processors that share the separator to be eliminated.

Three routines incorporate the features required for parallel backsubstitution; these routines, which move solution updates among processors, mirror routines added to the nested dissection package to support message passing of matrix entries. The communications are carried out by subcube exchange algorithms. Routine solnfsbal passes solution updates to reverse the effect of routine fsbalance, which dynamically load balances fully-summed matrix columns. Routines solncoll and nestback pass solution updates in the manner determined by routine solnswap to reverse the effect of passing matrix entries in routine eqnswap.

PIVOTING AND MESSAGE PASSING — The parallel solver supports full pivoting; in fact, the use of

full pivoting is encouraged since the cost of finding the largest pivot within each processor's local data is small compared to the cost of communicating the pivotal choice among processors.

We have chosen to use highly synchronous communications strategies, e.g., for communicating pivots and pivotal columns of the matrix, in order to maximize the potential scalability of the parallel solver. The communication algorithms are based primarily on subcube exchange and tree algorithms and are implemented in routine `globpiv`, which finds new pivots, and `pveqns`, which communicates the resulting pivotal equation and associated heading data to all processors in the subcube currently working on a shared frontal matrix.

LOAD BALANCER(S) — A dynamic load balancer in routine `fsbalance` balances fully summed matrix columns across processors. As discussed below, this technique is being extended to balance on partially-summed matrix columns. Some aspects of the dynamic load balance techniques of Dragon and Gustafson (1989), Gustafson et al. (1989) and Venugopal and Naik (1991) are being incorporated into the new load balancer for partially-summed matrix entries in nested dissection.

RESULTS

The following tables summarize the parallel performance of our multifrontal solver. The test problems chosen have either biquadratic or triquadratic elements applied to the Laplace equation in two or three dimensions, respectively.

PERFORMANCE ON 2D PROBLEMS — Table 1 shows how problem size, run time, and parallel efficiency vary with number of processors for the 2D test problem on the nCUBE 2 hypercube. We find that parallel efficiencies of 25% are achieved on 1024 processors, with higher efficiencies demonstrated on fewer nodes. Parallel efficiencies of greater than 25% are obtained for even the smallest problems (one finite element per processor) on 1024 processors. Floating point performance at 25% efficiency is about 500 MFLOPS sustained computation rate on the nCUBE 2.

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.

Table 1. Multifrontal Performance for the 2D test problem on an nCUBE 2 hypercube. When two results are listed for the same number of processors, the first used twice as many processors in the x-direction of the mesh decomposition and the second twice as many processors in the y-direction. Minimum acceptable parameter values (as determined by symbolic factorization) are listed for cases which failed to fit in 4MBytes of local memory.

<u>Processors</u>	<u>Number of Local Elements</u>	<u>Equations</u>	<u>Run Time[s]</u>	<u>Parallel Efficiency</u>	
				<u>Overall</u>	<u>Backsub</u>
1	400	1681	11.0	1.	1.
2	400	3321	15.6	0.982	0.965
2	400	3321	20.2	0.764	0.966
4	400	6561	27.2	0.776	0.906
8	400	13041	35.7	0.702	0.833
8	400	13041	29.2	0.853	0.835
16	400	25921	41.1	0.745	0.717
32	400	51681	48.0	0.733	0.629
32	400	51681	48.0	0.726	0.624
64	400	103041	66.8	0.646	0.502
128	400	205761	93.1	0.555	0.410
128	400	205761	92.1	0.547	0.408
256	400	410881	160.7	0.419	0.324
512	361	741153	226.6	0.338	0.288
512	361	741153	249.2	0.332	0.286
512	400	821121	257.3	0.343	0.296
512	400	821121	(Need Q = 270K)		
1024	256	1050625	310.0	0.256	0.243
1024	289	1185921	358.0	0.258	0.243
1024	324	1329049	412.5	0.259	0.247
1024	361	1481089	(F=157K, Q=285K, E=5K)		
1024	400	1640961	(F=175K, Q=323K, E=5.4K)		

The largest 2D problem that was successfully solved has over 1.3 million equations. The results also demonstrate the memory intensive nature of the direct solver - even at 25% parallel efficiency none of the 2D problems were large enough to require more than 7 minutes of run time.

Table 1 also reports some cases in which the symbolic factorization package correctly predicted that a problem was too large to be solved on the hypercube, and how much larger the offending PARAMETERS (and, in turn, local memory) must be in order to solve that problem. Here, E, F, and Q are the minimum values of MAXEQ, ICEFR, and ICEQU, respectively, required for the proposed computation.

LESSONS LEARNED IN DEVELOPMENT — Parallel efficiencies are presented as both overall efficiency of the solver and the efficiency of the parallel backsubstitution portion of the solver. The latter is of interest because it is well-known that backsubstitution is the portion of a direct solver that parallelizes most poorly. Therefore, the backsubstitution efficiency provides a sanity check on the overall parallel efficiency; in fact, at an early stage of the parallel code development the measured backsubstitution efficiency was higher than the overall efficiency - a sure sign that the overall parallel implementation was performing poorly.

We note that parallel efficiencies are determined by measuring the amount of run time each processor spends communicating or idle. Measured efficiencies provide a subtle danger to the parallel algorithm designer: he or she can be misled by high measured efficiencies if the parallel code spends much of its compute time carrying out redundant computations (Gustafson et al. 1988), i.e., extra computations that have been introduced in the parallel version of a solver that are unnecessary in the serial version. Indeed, at one stage in the evolutionary development of the parallel multifrontal solver, measured parallel efficiencies were artificially high because about 90% of the compute time was being consumed by redundant updates of the residual vector. This unmeasured overhead was quickly eliminated by distributing the residual vector across the processors.

SCALABILITY — Scalability of the multifrontal solver on the hypercube is addressed by the results in Table 2. Here we see that the parallel efficiencies of the solver as a whole and of backsubstitution are nearly independent of problem size on 256 processors. This is a classic symptom of an algorithm that is dominated by a load imbalance that scales with problem size. The source of the load imbalance is easy to understand: processors handle only matrix entries associated with their original set of separators. As separators are eliminated, some processors become idle, although the effect is not nearly as severe as it is in the case of the coarse-grain multifrontal solver.

The slight deviation from uniform efficiency poses an interesting (and unsolved) puzzle: parallel efficiency passes through a minimum as problem size decreases and is actually largest for the smallest problem - that of one finite element per processor. Apart from the subtleties that are undoubtedly creating this phenomenon (perhaps a tradeoff of message sizes and numbers vs. local computation as problem size decreases), this result is extremely encouraging. The single element efficiency, 45% in this case, should be a very conservative lower bound on the parallel efficiency to be expected of a dynamically load balanced solver.

Table 2. Multifrontal Performance for the 2D test problem on an nCUBE 2 hypercube. Problem size per processor is scaled on 256 processors.

<u>Local Elements</u>	<u>Number of Equations</u>	<u>Run Time[s]</u>	<u>Parallel Efficiency</u>	
			<u>Overall</u>	<u>Backsub</u>
1	1089	3.0	0.455	0.330
4	4225	4.0	0.388	0.278
9	9409	5.3	0.360	0.261
16	16641	7.5	0.325	0.255
36	37249	12.6	0.343	0.253
64	66049	20.9	0.352	0.260
100	103041	32.2	0.363	0.263
144	148225	47.9	0.377	0.274
196	201601	67.4	0.391	0.288
256	263169	92.4	0.403	0.306
324	332929	123.7	0.415	0.324
400	410881	160.7	0.419	0.324

PERFORMANCE ON 3D PROBLEMS — Table 3 shows how problem size, run time, and parallel efficiency vary with number of processors for the 3D test problem on the nCUBE 2 hypercube. As for the 2D case, we find that parallel efficiencies of 25% are achieved on 1024 processors, with higher efficiencies demonstrated on fewer nodes. Parallel efficiencies of 22-25% are obtained for smaller problems. The largest 3D problem that was successfully solved had over 105,000 equations. None of the 3D problems that fit in 4 GBytes of memory required more than 10 minutes of run time.

PERFORMANCE OF NESTED DISSECTION — Table 4 and 5 summarize how parallel efficiency varies as the solution process proceeds. Overall parallel efficiency (as well as run time) is clearly dominated by the last few levels of nested dissection. We note that "grind" time, the time required to eliminate each equation from the matrix, grows by three orders of magnitude as nested dissection or separator equations proceeds. Most of this growth is inherent in the direct method, given that the maximum matrix size operated on by cooperating processors grows by a factor of 20. Even at 100% parallel efficiency the grind time would increase by a factor of 250.

Table 3. Multifrontal Performance for the 3D test problem on an nCUBE 2 hypercube. On 64 processors, each of the three runs used twice as many processors in the x-, y- and z-directions, respectively, as in the other two directions of mesh decomposition. On 512 processors, the first three runs used twice as many processors in the x-direction of mesh decomposition and the last two twice as many in the z-direction. Minimum acceptable parameter values (as determined by symbolic factorization) are listed for cases which failed to fit in 4MBytes of local memory.

<u>Processors</u>	<u>Number of</u> <u>Local</u>		<u>Run</u> <u>Time[s]</u>	<u>Parallel Efficiency</u>	
	<u>Elements</u>	<u>Equations</u>		<u>Overall</u>	<u>Backsub</u>
1	27	343	7.7	1.	1.
8	27	2197	17.8	0.825	0.525
64	27	15625	96.9	0.576	0.291
128	27	30625	183.3	0.461	0.262
128	27	30625	185.5	0.460	0.266
128	27	30625	187.0	0.471	0.265
512	8	35937	197.0	0.300	0.218
512	12	53361	304.7	0.306	0.219
512	18	79233	(Need Q=167K, E=4.1K)		
512	27	117649	(F=379K, Q=296K, E=5.4K)		
1024	1	9537	27.6	0.223	0.219
1024	8	70785	394.8	0.259	0.214
1024	8	70785	408.8	0.254	0.214
1024	12	105105	(Need Q=166K, E=4.6K)		
1024	12	105633	547.9	0.263	0.215

Table 4. Performance of nested dissection: variation of parallel efficiency, local matrix sizes, and "grind time" (run time per equation) as a function of dissection level. The 3D test problem with 105,000 equations is solved on 1024 processors.

<u>Level of Dissection</u>	<u>Parallel Efficiency</u>	<u>Max. Matrix Size</u>	<u>Grind Time (ms/equation)</u>
0	0.97	96	0.06
1	0.77	171	0.11
2	0.68	279	0.22
3	0.57	435	0.57
4	0.47	691	1.6
5	0.39	1026	3.9
6	0.28	1313	11.5
7	0.27	1601	26.9
8	0.25	2129	51.4
9	0.23	2145	69.8
10	0.24	1089	49.6

Table 5 encapsulates the load imbalance that results from the column-based assignment of matrix entries to processors. By the time that 512 processors are working together on a common set of separators, there is a factor of 7 imbalance in the initial maximum-to-average distribution of matrix columns. Under such conditions, it is no surprise that the parallel efficiency is less than 25% for the computations at that level of mesh dissection.

We note also that at the final stage (level 10) some processors have no matrix columns - each processor should have at least one of the 1089 matrix columns. Therefore, there is a subtle flaw in our dynamic load balance procedure for fully summed matrix columns. This problem must be addressed before the same techniques are applied to the partially summed matrix entries or the torus wrap mapping.

Table 5. Performance of nested dissection: variation of local matrix sizes at the start of each dissection level on the same test problem as Table 4.

<u>Level of Dissection</u>	<u>Number of Local Matrix Columns</u>		
	<u>Maximum</u>	<u>Average</u>	<u>Minimum</u>
1	94	79	53
2	83	63	33
3	68	47	9
4	64	35	4
5	65	27	3
6	65	19	3
7	63	13	1
8	48	8	1
9	28	4	1
10	2	1	0

COMMUNICATION OVERHEAD — Table 6 provides details of the communication time and idle time for the various communication and synchronization steps in the solver. We note that the synchronization steps are not necessary for correct operation of the solver, they were provided so that the load imbalance in the sections of code could be studied in more detail. The overhead of the synchronizations is extremely low, as shown by the time to write synchronization messages in Table 6.

The time required to read messages tells a different story, for that is where idle time is incurred on the nCUBE as processors wait for messages that have not arrived. In principle, reading and writing messages should take about the same amount of time (give or take some subtle operating systems effects, as evidenced by a couple of entries in Table 6 for which read time is actually less than write time), which leads to the conclusion that communication required about 25 s of run time for this calculation, which leaves 365 s of idle time due to load imbalance. Furthermore, there are three sources of the load imbalance: (1) selection of new pivots, (2) broadcast of new pivotal columns, and (3) reduction of each processor's matrix columns. Idle time due to reduction accumulates at both the globpiv communication and nestgen synchronization steps. All three sources of idle time can be greatly reduced by balancing matrix entries among processors; e.g., via torus wrap mapping to processors followed by dynamic reassignment of both full- and partially-summed matrix entries as needed to maintain load balance.

Table 6. Performance of nested dissection: processor averaged communication times on the same test problem as Table 4. Conclusion: about 365 s of run time is idle time due to load imbalance, compared to 25 s for communication.

Communication <u>Step</u>	Read/Wait <u>Time [s]</u>	Write <u>Time [s]</u>
nestgen - headers	0.02	0.007
nestgen - residuals	0.01	0.
colsplit - matrix columns	0.45	0.025
fseqnbal - matrix columns	5.7	0.029
globpiv - matrix pivot	178	3.48
pveqnsmap - headers	147	4.57
nestback - headers	3.68	4.66
solncoll - headers	0.001	0.002
solnbal - headers	0.001	0.001
nestgen - synchronization	45.0	0.001
neststat - synchronization	8.48	0.001
nestback - synchronization	0.38	0.001
Totals:	390	12.8

PERFORMANCE ON AN INTEL HYPERCUBE — Table 7 shows how problem size, run time, and parallel efficiency vary with number of processors for the 3D test problem on the Intel iPSC860. Unlike the nCUBE results, we find that parallel efficiencies improve markedly as problem size is increased on a given number of processors. We also find that the sustained performance of an Intel node is about 3.5 times that of an nCUBE node for the present implementation. This correlates well with the relative speed of the DAXPY routine on the two systems. However, although DAXPY is the best choice for the nCUBE,

the authors have been advised that a factor of 3 or more improvement is achievable on the Intel if alternative BLAS routines are on that system.

In addition to the scalability of parallel efficiency, Table 7 demonstrates another desirable feature of the 32 MByte memories on the Intel. With those memories we are able to solve over 30K equations on 16 nodes of the Intel system, with run times of less than 3 minutes.

Table 7. Multifrontal Performance for the 3D test problem on a 64-node Intel iPSC860 hypercube. Mesh sizes of 4x4x4 elements or less per processor can run in nodes with 8 MByte memories; the 6x6x6 local meshes require 32 MBytes of local memory.

	Number of			Parallel Efficiency
<u>Processors</u>	<u>Equations</u>	<u>Run Time [s]</u>	<u>Overall</u>	<u>Backsub</u>
3x3x3 Local Mesh:				
1	343	3.6	1.	1.
8	2197	5.3	0.551	0.387
64	15625	28.4	0.256	0.100
4x4x4 Local Mesh:				
1	729	4.5	1.	1.
2	1377	5.8	0.931	0.811
4	2601	7.1	0.823	0.602
8	4913	12.6	0.669	0.409
16	9537	19.0	0.552	0.316
32	18513	35.3	0.447	0.201
64	35937	106.1	0.361	0.106
6x6x6 Local Mesh:				
1	2197	16.8	1.	1.
2	4225	27.1	0.979	0.897
4	8125	44.6	0.908	0.759
8	15625	102.1	0.748	0.571
16	30625	162.7	0.666	0.440

DISCUSSION

We have demonstrated parallel efficiencies of 25-30% on 1024 nodes of an nCUBE 2 hypercube computer for a multifrontal solver. An encouraging result is that 95% of the parallel inefficiency is due to load imbalance, rather than interprocessor communications. We have implemented a dynamic load balancer that balances fully summed matrix columns across processors; we need to extend this technique to balance on partially-summed matrix columns as well. We have also developed a symbolic factorization package as a front end to the solver which tests the memory requirements for a given problem against the solver's memory allocation. The parallel solver supports full pivoting; in fact, the use of full pivoting is encouraged since the cost of finding the largest pivot within each processor's local data is small compared to the cost of communicating the pivotal choice among processors. We have chosen to use highly synchronous communications strategies, e.g., for communicating pivots and pivotal columns of the matrix,

in order to maximize the potential scalability of the parallel solver. The communication algorithms are based primarily on subcube exchange and tree algorithms.

The largest problems that have been solved (in core) have 100,000 equations in 3D and 1,300,000 equations in 2D. Parallel efficiencies of greater than 25% are obtained for even the smallest problems (one finite element per processor) on 1024 processors. Assembly language calls have been introduced for just two DAXPY calls in the solver, which results in overall performance of 500 MFLOPS and the potential for 2 GFLOPS (at 100% parallel efficiency) sustained computation rates on the nCUBE 2. Similarly high GFLOPS rates can be achieved on the Intel iPSC860, Thinking Machines Corp. CM-5, and other MIMD computers, given a suitable assembler kernel for the node processors of those systems.

Further performance gains can be realized by switching from a column-based mapping of matrix entries to processors to the so-called torus wrap or scattered decomposition mapping (Morison and Otto 1986). The efficacy of this mapping for full matrix algorithms is well established (cf. Hendrickson 1992 and references therein). We predict that the parallel efficiency will be 60-85% for large problems on 1024 processors once dynamic load balancing is fully implemented; this should ensure scalability of the method to the Teraops/Teraflops class parallel architectures that are anticipated in the late 1990s.

The user interface for the multifrontal solver requires careful consideration. In particular, the mapping of a mesh onto the parallel system must be automated with a suitable preprocessor, so that (1) the burden of mapping is not left to the user and (2) the extra I/O required to input a decomposed mesh is avoided. Much current research centers on the mapping techniques and other preprocessing steps required to tackle the critical issue of mapping unstructured meshes to parallel processors in such a software package. This includes the work of Vaughan (1990, 1991) on parallel versions of Sandia's JAC3D structural analysis code on the nCUBE/ten and nCUBE 2 hypercubes, as well as that of Simon (1991) and others on innovative spectral mapping strategies. A related issue, that of optimally ordering the mesh within each processor, must also be addressed; fortunately many applicable heuristics have been developed in the serial case.

Out-of-core multifrontal solution of large sparse systems using parallel disk systems is of interest for 3D modeling, in light of the short run times for problems that can be solved in 4 GBytes or less of memory. Given the current bandwidth limitations and hardware and software reliability of parallel disk systems, an out-of-core solver might use multistage routing of data to and from the disk system, which has proven useful in overcoming the same issues in the case of parallel graphics I/O (Benner 1989). Alternatively, a heterogeneous implementation (Gustafson et al. 1989) might be advantageous, with a few large memory nodes acting as disk servers for the others. Given our current heterogeneous hardware on the Intel system, we have an excellent opportunity to conduct research on heterogeneous schemes for parallel disk access.

The issue of portability must be addressed for systems beyond the Intel iPSC860 and nCUBE 2. These include the Kendall Square KSR 1, the Thinking Machines Corp. CM-5, forthcoming Intel Paragon and CRAY MPP systems, etc. With some luck Express, Linda, PVM, Strand, or a comparable software package may emerge on many or most of these systems. With greater luck the "winning" portability platform might incur low overhead on large numbers of processors.

In spite of the drawbacks of direct solvers in general, a highly efficient, parallel multifrontal solver might find additional uses as a component in larger solver packages, such as incorporation in software for

(1) quasi-Newton iteration for nonlinear problems, (2) software packages for sparse eigenproblems (e.g. block Lanczos and related methods for symmetric and nonsymmetric eigenproblems), (3) coarse mesh solution in multigrid solvers, and (4) incomplete factorization methods for preconditioning iterative methods, such as the conjugate gradient method and its variants for nonsymmetric matrix problems.

CONCLUDING REMARKS

Parallel efficiencies of 25 % on 1024 nCUBE 2 nodes and 36 % on 64 Intel iPSC860 nodes have been achieved for a multifrontal solver. Some encouraging results are that 95 % of the parallel inefficiency is due to load imbalance, rather than interprocessor communications, and the load imbalance should be almost completely removed by the introduction of static mapping and dynamic load balance techniques that have been developed for other parallel solvers or application codes.

In order to maximize the eventual suitability of the software package for library-quality, production use on MP systems, we have implemented a dynamic load balancer for fully-summed matrix entries, a symbolic factorization package, "full" pivoting, and highly synchronous communications strategies based primarily on subcube exchange and tree algorithms. MP computer hardware and software vendors have expressed interest in the development of the product of this research into a viable software package for MP computers. Much of the planned work on the solver centers on the mapping techniques and other preprocessing steps necessary for such a software package.

REFERENCES

- R. E. Benner, G. R. Montry, and G. G. Weigand, *Concurrent Multifrontal Methods: Shared Memory, Cache, and Frontwidth Issues*, Internat. J. Supercomputer Appls, 1 (3), 1987, pp. 26-44.
- R. E. Benner, *Parallel Graphics Algorithms on a 1024-Processor Hypercube*, Proc. 4th Conf. Hypercube Concurrent Comput. Appls., 1989, pp. 133-140.
- K. M. Dragon and J. L. Gustafson, *A Low-Cost Hypercube Load Balance Algorithm*, Proc. 4th Conf. Hypercube Concurrent Comput. Appls., 1989, pp. 583-590.
- I. S. Duff and J. K. Reid, *The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations*, ACM Trans. Math. Soft., 9, 1983, pp. 302-325.
- C. Farhat and E. L. Wilson, *A New Finite Element Concurrent Computer Program Architecture*, Internat. J. Numer. Meth. Engr., 1987, pp.1771-1792.
- G. C. Fox, M. A. Johnson, M. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors, Volume 1, General Techniques and Regular Problems*, Prentice Hall, New York, 1988.
- J. L. Gustafson, R. E. Benner, M. P. Sears, and T. D. Sullivan, *A Radar Simulation Program for a 1024 Processor Hypercube*, Proc. Supercomputing '89, ACM Press, New York, 1989, pp. 96-105.
- J. L. Gustafson, G. R. Montry, and R. E. Benner, *Development of Parallel Methods for a 1024 Processor Hypercube*, SIAM J. Sci. Stat. Comput., 9, 1988, pp. 609-638.
- M. T. Heath, E. Ng, and B. W. Peyton, *Parallel Algorithms for Sparse Linear Systems*, SIAM Rev., 33, 1991, pp. 420-460.
- B. Hendrickson, *Parallel QR Factorization on a Hypercube Using the Torus Wrap Mapping*, Sandia Report SAND91-0874, Sandia National Laboratories, Albuquerque, NM, January 1992.

- P. Hood, *Frontal Solution Program for Unsymmetric Matrices*, Internat. J. Numer. Meth. Engr., 2, 1970, pp. 5-32.
- B. M. Irons, *A Frontal Solution Program for Finite Element Analysis*, Internat. J. Numer. Meth. Engr., 10, 1976, pp. 379-399; 11, 1977, p. 1055.
- J. W. H. Liu, *The Multifrontal Method for Sparse Matrix Solution: Theory and Practice*, SIAM Rev., 34, 1992, pp. 82-109.
- R. F. Lucas, *Solving Planar Systems of Equations on Distributed Memory Multiprocessors*, Ph.D. Thesis, Stanford University, Dec. 1987.
- R. Morison and S. W. Otto, *The Scattered Decomposition for Finite Element Problems*, J. Sci. Comput., 2, 1986.
- M. P. Sears, *Linear Algebra for Dense Matrices on a Hypercube*, Proc. Fifth Distrib. Mem. Comput. Conf., 1990, pp. 317-320.
- H. Simon, *Partitioning of Unstructured Problems for Parallel Processing*, Comput. Sys. Engr., 2, 1991, pp. 135-148.
- C. T. Vaughan, *Implementation of JAC3D on the NCUBE/ten*, Proc. Fifth Distrib. Mem. Comput. Conf., 1990, pp. 538-544.
- C. T. Vaughan, *Structural Analysis on Massively Parallel Computers*, Comput. Sys. Engr., 2, 1991, pp. 261-268.
- S. Venugopal and V. K. Naik, *Effects of Partitioning and Scheduling Sparse Matrix Factorization on Communication and Load Balance*, ICASE Report 91-80, NASA Langley Research Center, Hampton, Va., October 1991.

END

DATE
FILMED

6 / 21 / 93