Implementation of JAC3D on the NCUBE/ten [*]

Courtenay T. Vaughan

Sandia National Laboratories
Albuquerque, NM 87185

Abstract

An implementation is presented for JAC3D on a massively parallel hypercube computer. JAC3D, a three dimensional finite element code developed at Sandia, uses several hundred hours of Cray time each year in solving structural analysis problems. Two major areas of investigation are discussed: (1) the development of general methods, data structures, and routines to communicate information between processors, and (2) the implementation and evaluation of four algorithms to map problems onto the node processors of the hypercube in a load-balanced fashion. The performance of JAC3D on the NCUBE/ten is compared with that on a Cray X-MP: the NCUBE/ten version presently takes 20% more compute time than the Cray. On a larger simulation which used more of the NCUBE's memory, the NCUBE/ten would take less compute time than the Cray. Current activity on the newer NCUBE 2 hypercube is summarized which should lead to an order of magnitude improvement in run-time performance for the massively parallel solution of structural analysis problems.

Introduction

In this paper we discuss the implementation of JAC3D, a three dimensional finite element code which uses a nonlinear Jacobi preconditioned conjugate gradient method to solve large displacement, large strain, temperature dependent, and nonlinear material structural analysis problems, on a massively parallel computer, the NCUBE/ten hypercube. This code was developed at Sandia National Laboratories where it uses several hundred hours of Cray time each year. We note that the hypercube implementation is complete in that a user has the same user interface and simulation options on the Cray and the hypercube.

Two major implementation issues are discussed below. The first is the development of routines to communicate information between the node processors and between the host and the node processors. The reason these are nontrivial is that the finite element mesh is not necessarily regular or regularly numbered. Routines are included that determine what information each processor sends or receives at each communication step and with which processors it is communicating. The second area is the development of algorithms to map a problem onto the node processors of the hypercube in a load-balanced fashion. We will present and compare several mapping methods that, to date, have been executed on a SUN workstation.

Compute times are within 20% of the Cray X-MP for a production simulation with 89,043 equations. The NCUBE/ten can easily handle a problem four times larger; such a simulation would be faster on the NCUBE/ten relative to the Cray. Preliminary benchmarks on the NCUBE 2 indicate that the SUN front end reduces I/O time by at least a factor of ten and that NCUBE 2 processors are currently a factor of four faster than the first-generation processors. Therefore, the code should run several times faster on the NCUBE 2 than on the Cray X-MP. We are also working on parallelization of selected mapping methods and on a system to display JAC3D results from the NCUBE 2 hypercube on a Stellar graphics workstation.

Implementation Issues

Overview

JAC3D is a three dimensional finite element code which uses a nonlinear Jacobi preconditioned conjugate gradient (PCG) method to solve large displacement, large strain, temperature dependent, and nonlinear material structural analysis problems [2]. The serial version of the code reads in three data files: a control file containing material constants and numbers such as the maximum number of iterations, an input file which contains the finite element de-

[*]This work was partially supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories which is operated for the U.S. Department of Energy under contract number DE-AC04-76DP00789.

scription of the problem, and a file which gives the temperature at each node point for each load step. JAC3D then creates an output file and an additional file used for plotting the output.

In implementing JAC3D on our hypercube, an NCUBE/ten, we have added a third input file which contains the order of the hypercube being used and a mapping of the elements and nodes of the problem onto the node processors. The NCUBE/ten is a 1024 node hypercube which has 0.5 MBytes of memory on each processor.

It was necessary on the NCUBE/ten to divide the original code into a host processor code and a node processor code. (This code division can be avoided on the newer NCUBE 2 hypercube.) The node processor code corresponds to the call to the solver in the original code, while the host code handles the input and output. The host code begins by reading the input files and doing the preprocessing that is necessary on the data. When it is ready to call the solver, it allocates a hypercube of the desired dimension and starts the solver on the node processors. In this way, running the solver on the node processors is similar to calling the solver as a subroutine with the passed variables now being communicated between the host and node processors.

PCG and Finite Element Methods

The iteration matrix is calculated at each iteration as it is needed, which avoids using the memory which would be required to store the entire matrix. The matrix is calculated element by element, so some information about each of the elements has to be kept. This is done by dividing the elements among the processors such that each element is assigned to one processor. In this way, there are no duplicate calculations.

Each element has a list of nodes which are associated with it and allocates storage for all of these nodes. In this way each node may be allocated space in more than one processor but the node will be assigned to only one processor. That processor is responsible for maintaining the correct value of the variables associated with the node by collecting partial values of the variables associated with that node from other processors and providing these correct values to the other processors when needed. On each processor, the nodes which are assigned to it are numbered first, followed by the nodes for which the processor needs values but which are assigned to other processors. In this way, each processor locally numbers the nodes and elements that it has.

In the solution algorithm, the unknowns at the nodes are updated in two ways. Some calculations, such as the calculation of the residual vector, are done element by element [6]. In order for the processors to update the unknowns associated with an element, some values of other variables at the associated nodes need to be communicated to that processor. As each element is used, the unknowns at the nodes associated with that element are updated. Since each element appears in only one processor, several processors will generate updates to shared variables, which requires communication of partial results so these updates can be combined to form the final result.

The second way that unknowns at a node get updated is by the processor which to which that node is assigned. An example of this is the calculation of the new direction vector from a linear combination of the previous direction vector and the residual vector.

Initial host-to-node Communication

The host processor communicates with the node processors by communicating only with node 0. Any data that the host sends to the node processors is sent to processor 0 which then broadcasts the information to the rest of the processors by means of a fanout algorithm using a minimal spanning tree of the hypercube rooted at node processor 0 [4]. In the fanout algorithm, successive dimensions of the hypercube are used. In each stage, all of the active processors send information to their neighbor in that dimension. As those processors receive information, they become active and will send information in the next stage.

The host processor starts by sending the node processors a message which contains startup information such as the total number of elements and nodes in the problem and the maximum number of iterations. The node processors then use this information to set up some temporary arrays. The host processor then reads in the problem mapping of the elements and sends that information to the node processors. This allows the node processors to determine how many elements they have and allocate space for some arrays. The host processor then sends the list of nodes which are associated with each of the elements and the mapping of the nodes to the node processors. The node processors store the portion of the list of nodes which are associated with their elements and then use that with the mapping of the nodes to determine the number of nodes they need storage for and to set up communication with other nodes.

Data Structures for Interprocessor Communication

Next, the node processors set up the communication which they do during the calculations. Using the list of which processor has each node, a processor constructs a list of nodes for which it needs values of variables but which are assigned to other processors. This *receive list* of nodes is then sorted by processor and the processor builds an index to this list consisting of the processor to communicate with, the number of nodes which have to be communicated, and a starting index into the list. This is illustrated in Figure 1. When the list is sorted by processor, it is ordered by placing the processors in descending order of their distance from the processor in terms of message hops. In this way, messages which will take the longest time to be communicated will be sent first. In our experiments, this message order cut down the execution time of the algorithm.

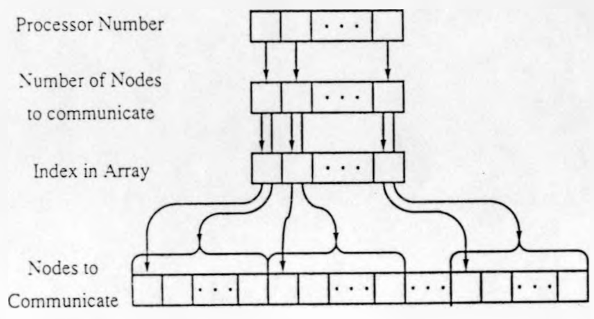

Figure 1. Communication Data Structure

Each processor sends the processors which it needs information from the list of nodes it needs from that processor. Each processor uses this information it receives to construct a list similar to its receive list, a *send list* which is used to send correct values of variables. The communication routines use this general data structure since the problems to be solved are generally irregular and have an irregular numbering of the nodes.

When the processors need to communicate the value of a variable, they use the send list to send messages to other processors and the receive list to receive messages from other processors. When the values of an array need to be communicated, each processor sends a message to each of the processors in its send list of processors. The processor numbers in the send list are used successively and an index into the node numbers being sent is maintained. For each processor in the list, the node numbers to be sent are determined by taking them from the list of nodes starting at the index. Since the number of nodes to be communicated to each processor is stored, that many nodes numbers are used to take information from the array to be sent and put into a message array. This process uses all of the data structure as illustrated in Figure 1 except for the array of indexes into the list of nodes to be communicated. The message array is then sent and the index is incremented by the number of nodes which were sent.

When a processor receives a message, it looks up the processor number in its receive array and the number of nodes that are being communicated and the starting position in the array. It uses that information to put the values in the message into the variable array in the right places. Since it can be seen that the communication involved with the process of communicating correct values of variables at a node between the processors is the inverse of the the process of communicating partial values of variables at a node between processors, the receive list is used to send partial results to other processors and the send list is then used to receive those results which are added to the local results to get the correct value. In the case of communicating partial values, the final result does not necessarily need to be sent to the other processors involved since they may not need this value.

The other case in which interprocessor communication has to be done is the case of inner-products. This is done by the standard bidirectional exchanges of partial information along successive dimensions of the hypercube with the addition of partial results after each exchange [6].

Input: Large Vectors

After the node processors have allocated space for the vectors that they store and have set up their communication schemes, the host processor can send them the initial vector information(e.g. temperatures). This information is sent to processor 0 which then broadcasts it to the other node processors. Each processor then takes the part of the vector which it needs and stores it in its memory. The maximum size of a message, the size of the message buffers on the node processors, and the size of an array on the host processor are each limited, so large messages have to be read in to the host and sent to the node processors in pieces. After each piece of the message is received by the node processors, node processor 0 sends a message back to the host to allow the host to send the next piece. This procedure prevents message buffer overflow on the node processors.

Host Activity During Node Computation

At this point the node processors start calculating and the host processor waits. Since the node processors have to output results and read additional input such as the temperature of the nodes at the beginning of each load step, the host processor has to be able to call the appropriate subroutine to interact with the node processors. It does this by waiting to receive a message and, based on the type of the message received, either calls the appropriate subroutine, prints out the appropriate error message, or deallocates the hypercube and quits. Since node 0 has a copy of any scalar data which has to be communicated back to the host processor to run the subroutine, this information is included in the message which tells the host processor which subroutine to run. In summary, execution is controlled by the node processors in this part of the calculation.

Output

The output from the node processors is handled by a fanin algorithm, in which the information to be output is sent to node processor 0 which, in turn, sends the information to the host. The fanin algorithm is the inverse of the fanout algorithm. At each stage, half of the active processors send a message to the other half. The processors which receive a message are the active processors for the next stage. As with input, output of large messages is also done in pieces. In order to output arrays in the proper order, each processor has a list of the global order number of the nodes which are assigned to it. Each piece of the array is assembled in the global order and sent to the host processor.

Problem Mapping

In order to implement JAC3D on the hypercube, we had to provide for the automated mapping of large problems onto the hypercube. We have used four mapping methods. The first is a recursive bisection method developed for problems on rectangular grids by Berger and Bokhari [1]. In this method, the problem grid is divided into two rectangles along a line of the grid. This division is repeated recursively to each of the rectangles until the desired number of sets of unknowns is created. This method is easily adapted for three-dimensional rectangular grids [3]. This method has the disadvantage that it has the potential for load imbalance, since each set is divided along a line of the grid and, therefore, the two resulting sets may not be the same size.

From this algorithm, we have developed a second algorithm which uses recursive bisection for irregular regions in three dimensions. The first step is to sort the nodes of the grid in the x, y, and z directions. At each stage of the mapping, a direction is chosen and each set in the mapping is divided into two equal or nearly equal sets based on the index in the sorted list for the given direction of each node in the set. For example, given a set S with n nodes which is being divided into sets S1 and S2 along the x direction, the first n/2 nodes of set S in the sorted list of nodes for the x direction are placed in set S1 with the remainder put in set S2. In this way, the sets at the final stage of the mapping will have an approximately equal number of nodes.

The third algorithm was developed by Kernighan and Lin [8]. It is a iterative graph-based algorithm which starts with a set which has been arbitrarily divided into two equal sized pieces and exchanges nodes in order to minimize the number of edges connecting the two pieces of the set. At each iteration, it looks at all of the unmarked nodes in each of the two pieces of the set and marks the pair which, if exchanged, would minimize the number of edges connecting the two pieces. After all of the nodes are marked, then the minimum number of pairs to create the maximum change are exchanged. The process is repeated until nothing further can be gained by swapping nodes.

The fourth algorithm that we used is a graph-based algorithm developed by Vaughan [9]. At each stage, each set is divided into two equal parts by the use of level sets. The first step to divide a set into two pieces is to find a pseudo-diameter of the graph of the grid [5]. A rooted level structure is constructed from each endpoint of the pseudo-diameter. The nodes are divided into two sets according to which endpoint they are closer to. Each rooted level structure will have a set of level sets and the number of the level set a node is in is a measure of its distance from the root of the level structure. Points which are equidistant from both endpoints are assigned to a set so that the sizes of the sets are equalized.

By using the endpoints of a pseudo-diameter as starting points, we seek to construct level structures with small level sets thus providing a smaller set of nodes on the boundary when the set is divided into two pieces. This is similar to the motivation for using level structures in reordering equations for solution by direct methods.

For the two graph-based algorithms, the number of sets at each stage of the division is doubled from n to $2n$ and the sets are divided according to their set number in a gray code fashion. When the first set, set 0, is divided into two sets, these sets are numbered 0 and n arbitrarily. After set i is divided, with $0 < i < n$, the two resulting sets are numbered i and $i+n$. The choice of which set is to be numbered i is determined by which numbering gives the smallest cost for communication with the sets which have already been divided.

For each of the algorithms, the nodes are divided among the processors. However, with our solution method, the elements also have to be mapped to the processors. Each of the mappings above work by doubling the number of processors in the mapping at each stage. At each stage, half of the nodes and half of the elements assigned to a processor are assigned to a new processor. Each element stays in its processor or moves to the new processor based on which of the two processors has more of its nodes. Ties are settled in such a way as to keep the number of elements assigned to the two processors even.

Results

We solved two problems with JAC3D on the hypercube. The first is a rectilinear block with three materials, 450 elements, and 810 nodes. The second is a solder analysis problem of a 28 pin integrated circuit on a PC board. It has four materials, 22932 elements, and 29681 nodes and is very irregular (Figure 2). Since we are solving for the displacements in three directions, there are 89043 unknowns in this problem. Symmetry is used in the x and y directions to decrease problem size. Note that most of the elements and nodes are in the pins connecting the PC board to the integrated circuit.

Table 1 shows the execution times for the program on the first problem using the four mapping methods as well as a mapping constructed by hand. The problem would not fit on one processor, or even two processors in the case of the Berger and Bokhari mapping. In the tables, hand is the hand mapping, graph is the graph-based method by Vaughan, kl is the Kernighan and Lin algorithm, bb is the Berger and Bokhari algorithm, and rb is the recursive bisection method based on a modification of the Berger and Bokhari algorithm. The execution times only include the node processor time and do not include the preprocessing time for the host. In the best case, we got a speedup of 41 on going from two to 256 processors. This is encouraging considering that, on 256 processors, each processor had two or fewer elements and four or fewer nodes.

Table 2 shows the time to construct the mappings on a SUN 3. These times are smaller by a factor of two or three than the time the division would take on one node processor of the NCUBE/ten. The two graph-based methods are slowest while the Berger and Bokhari algorithm is the fastest. Note that a large portion of time for the Kernighan and Lin algorithm is spent in the first division.

Table 3 shows the execution time for the solder analysis problem on the NCUBE. The time includes all of the node time from the time the host communicates the problem to the nodes and does not include the host preprocessing time. The Kernighan and Lin algorithm produces the mapping which executes the fastest while the other two methods are about equal. As Table 4 shows, however, construction of the Kernighan and Lin mapping is the slowest by at least a factor of ten.

Table 5 compares the solder analysis problem run on both the NCUBE and the Cray X-MP. Here, compute time for the NCUBE is just the node pro-

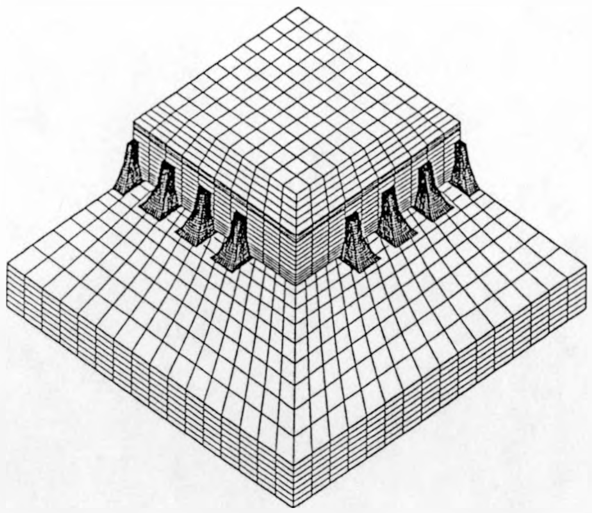
Figure 2. Solder Analysis Problem

Table 1.
Execution time for small problem
(seconds)

cube	Division Method				
dim	hand	graph	kl	bb	rb
1	1747	1751	1751	-	1752
2	885	910	911	1003	909
3	463	473	468	562	486
4	252	254	258	313	271
5	139	141	150	192	145
6	86.8	84.1	108	117	85.3
7	-	60.9	65.5	82.9	57.8
8	-	45.9	48.0	55.8	42.9

Table 2.
Mapping time for small problem
(seconds on a SUN 3)

cube	Division Method			
dim	graph	kl	bb	rb
1	3.0	30.3	2.0	2.4
2	4.3	40.8	2.0	2.5
3	6.6	46.4	2.1	2.8
4	10.4	52.5	2.1	3.1
5	14.5	58.0	2.1	4.0
6	21.5	66.3	2.2	5.4
7	29.1	74.9	2.3	7.9
8	40.0	84.0	2.4	14.3

Table 3.
Execution time for large problem
(seconds)

cube	Division Method		
dim	kl	graph	rb
8	8243	9098	9089
9	5541	6217	6331
10	4312	4602	5144

cessor time without any of the overhead of communicating with the host between load steps, while the total time is the time from start to finish on the host. The total execution time for the NCUBE/ten including all of the host time was 6100 seconds. This shows that the processing time on the NCUBE/ten is comparable to that on the Cray X-MP but the I/O time which is a result of the host processor of the NCUBE/ten causes the total execution time on the NCUBE/ten to be much larger than that of the Cray. When we implement this code on the NCUBE 2 with the SUN front end, the ratio of the total time to compute time should improve dramatically.

Discussion and Conclusions

We have implemented a large 3D finite element code on the NCUBE/ten hypercube and have obtained supercomputer-class performance (except for host processor I/O). Compute times are within 20% of the Cray X-MP for a production simulation with 89,043 equations. The NCUBE/ten can easily handle a problem four times larger; such a simulation would be faster on the NCUBE/ten relative to the Cray. The hypercube code is complete: a user sees the same user interface and simulation options on the Cray and the hypercube.

Table 4.
Mapping time for large problem
(seconds on a SUN 3)

cube	Division Method		
dim	kl	graph	rb
8	49193	2995	684
9	50072	3751	1242
10	50775	4894	2283

Table 5.
NCUBE vs. Cray X-MP
(seconds)

	Compute Time
NCUBE/ten	2197
Cray X-MP	1661

We are now implementing this code on the NCUBE 2 and its SUN front end. Preliminary benchmarks indicate that the SUN front end reduces I/O time by at least a factor of ten and that NCUBE 2 processors are currently a factor of four faster than the first-generation processors. Therefore, the code should run several times faster on the NCUBE 2 than on the Cray X-MP. We are also working on a system to display JAC3D results from the NCUBE 2 on a Stellar graphics workstation.

Several promising methods have been implemented and compared for mapping general problems onto a hypercube. Clearly, the methods should be judged by both the quality of their mappings and the time it takes to do the mapping. We plan to implement selected mapping algorithms, including the simple graph method and the recursive bisection method, in parallel on the NCUBE 2. We expect that some of the mapping algorithms will parallelize well and that the time used for mapping will ultimately be a small part of the overall execution time.

References

[1] Berger, M. J. and Bokhari, S. H. (1985) "A Partitioning Strategy for PDEs Across Multiprocessors", in *Proceedings of 1985 Int. Conf. Par. Proc.*, pp. 166-170.

[2] Biffle, J. H. (1984) "JAC - A Two-Dimensional Finite Element Computer Program for the Non-Linear Quasistatic Response of Solids with the Conjugate Gradient Method", SAND81-0998, Sandia National Laboratories, Albuquerque, NM.

[3] DeVries, R. C. (1990) "Static Load Balancing on a Hypercube: Concepts, Programs, and Results", SAND90-0338, Sandia National Laboratories, Albuquerque, NM.

[4] Geist, G. A. and Heath, M. T. (1986) "Matrix Factorization on a Hypercube Multiprocessor", in *Hypercube Multiprocessors 1986* (M. T. Heath, ed.), SIAM, Philadelphia, PA,

pp. 161-180.

[5] Gibbs, N. E., Poole, W. G., and Stockmeyer, P. K. (1976) "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix", SIAM J. Numer. Anal. 13, 1976, pp. 236-250.

[6] Gustafson, J. L., Montry, G. R., and Benner, R. E. (1988) "Development of Parallel Methods for a 1024-Processor Hypercube", SIAM J. Sci. Stat. Comp. 9, 1988, pp. 609-638.

[7] Jiang, B-N. and Carey, G. F. (1984) "Subcritical Flow Computation Using an Element-By-Element Conjugate Gradient Method", in *Proc. 5th Int'l. Symp. Finite Elements and Flow Problems*, Univ. of Texas, Austin, Jan. 23-26, pp. 103-106.

[8] Kernighan, B. W. and Lin, S. (1970) "An Efficient Heuristic Procedure for Partitioning Graphs", Bell System Technical Journal, 49, pp. 291-307.

[9] Vaughan, C. T. (1989) "The SSOR Preconditioned Conjugate Gradient Method on Parallel Computers", Ph.D. Dissertation, University of Virginia.