7

SOLVING LARGE OUT-OF-CORE SYSTEMS OF LINEAR EQUATIONS USING THE INTEL iPSC®/860

D. S. Scott

7.1 Introduction

Certain engineering problems, such as radar cross-section modeling using the method of moments, give rise to systems of linear equations $AX = B$, where A is a large, dense, 128-bit complex matrix and B is a matrix of right-hand sides. Solving systems as large as 20,000 is

* i860 and Concurrent File System are trademarks of Intel Corporation; iPSC is a registered trademark of Intel Corporation.

desired now with even larger problems anticipated in the near future. This chapter describes LOOCS, an implementation of a Large Out-Of-Core Solver on the Intel iPSC/860 hypercube with a Concurrent I/O disk subsystem. Block Gaussian elimination with restricted pivoting is used, swapping blocks on and off the disks as needed. The initial implementation runs only on even powers of two processors (1, 4, 16, 64). The code cannot take advantage of any symmetry in the matrices.

At both the cube and node level, the basic operation $X = X - Y * Z$ has been carefully optimized. A hand-coded i860 assembly language routine is used at the node level and at the cube level, asynchronous message passing and asynchronous disk I/O overlap almost all data movement with computation. A 64-node iPSC/860 with eight I/O nodes and 16 disks can factor a 20k problem in 3 hours and 50 minutes, achieving a sustained rate of 1.52 gigaflops. A hundred megaflop machine would take about 2 1/2 days to factor the same problem. Plans to extend the algorithm to run on odd powers of two processors and to even larger problems using tertiary storage will be described.

7.2 The iPSC®/860 System

The Intel i860™ processor is a million transistor RISC chip, with an 8k byte data cache, a 4k byte instruction cache, 32 integer registers, and 32 floating point registers. In each cycle, the chip can do one integer instruction, start one add, and start one 32-bit multiply. At 40 MHz, the peak speed is 80 Mflops in 32-bit precision. A 64-bit multiply can only be started every second cycle, so the peak 64-bit speed is 60 Mflops if the number of adds is twice the number of multiplies, or 40 Mflops if the number of adds and multiplies are equal (as they are in matrix computations). For more information on the i860 see [1].

The iPSC/860 System is a distributed-memory, message passing, hypercube multi-processor, containing up to 128 compute nodes. Each compute node has 8 megabytes of memory, an i860 processor, a network interface, and a message router. Each compute node lies at a corner of a d dimensional cube. The wires of the communication network form the edges of the cube. Each router can handle up to eight bidirectional wires. Seven of them are hypercube edges (which leads to the 128 node limit) and the eighth is for external I/O. The routers and the backplane form a circuit-switched communication network which allows any node to form a circuit to any other node and then transmit a message di-

rectly. The network interface contains two 4k byte FIFOs (queues). To maintain consistency of memory and the on chip data cache, the i860 is responsible for all data movement between memory and the FIFOs. Each node can send and receive messages at 2.8 megabytes/sec.

Each compute node runs a separate user process which is a complete program. These programs have the ability to send and receive data by making system calls to **send** and **recv**. The best analogy to the communication system is a set of offices with two fax machines each, one for sending messages and one for receiving messages. Each office can send a fax directly to any other office. The message will then remain in the fax queue until the worker comes and gets it.

Message passing may either be synchronous or asynchronous. Synchronous message passing does not return until the message has been processed. Asynchronous message passing returns immediately and is useful if the node process has computation to do while the message is being sent or received. In either case, the i860 will be interrupted occasionally to move data to or from the FIFOs.

For more information on the iPSC/860 see [2].

7.3 The Concurrent File System ™

The iPSC/860 System has an optional disk subsystem containing an arbitrary number of I/O nodes and disks. Each I/O node has an Intel 80386 processor, a SCSI bus controller with two 760-megabyte disks, and a network interface. It is directly connected to only one particular compute node (its **anchor** node) but it is part of the circuit-switched network. Messages from I/O nodes to compute nodes compete for the same wires as node-to-node messages and travel at the same 2.8 megabytes/sec.

All of the disks in the system form a single Concurrent File System ™ (CFS). The root directory of CFS is "/cfs". All files rooted in this directory are in CFS. Unless otherwise specified, all CFS files are spread out across all of the disks in 4k byte blocks. Thus different compute nodes can access different parts of a file without creating a bottleneck at a particular I/O node. This disk striping is invisible to the user, functionally CFS is equivalent to one big disk with processes opening, reading, and writing files. A single I/O node can transfer data from compute node to disk and back at about 1 megabyte/sec. To obtain this transfer rate the I/O node maintains a disk cache in

memory and aggressively reads ahead and writes behind.

Compute nodes cache structure information but not data. This is to ensure consistency of the data when one node writes and another one reads. Unfortunately, standard FORTRAN I/O thwarts this attempt by buffering I/O in the runtime system. To get around this problem, CFS provides special byte oriented files which do not use the FORTRAN I/O library and do not insert any control characters in the file. These files can be accessed using byte oriented I/O routines **cread** and **cwrite** for synchronous reads and writes and **iread** and **iwrite** for asynchronous access. The system call **lseek** can be used to randomly reposition the file pointer in byte oriented CFS files.

For more information on CFS see [3].

7.4 Block Gaussian Elimination

The Large Out-Of-Core Solver (LOOCS) code implements a variant of block Gaussian elimination to compute the block LU factorization of A and then uses block forward and back substitution to solve for the unknown matrix X, one block column at a time, given the factorization and the right-hand side matrix B. If X is an $r \times s$ submatrix of A (or B), the cost of reading X from disk is proportional to $r * s$. The work associated with multiplying X by another block is proportional to $r * s * \min(r, s)$. Thus partitioning A into square blocks was chosen to minimize I/O overhead. These square submatrices are called disk sections and they are the units which are swapped on and off the disk during the algorithm. The size of disk sections is determined as a function of the size of the matrix and the amount of memory available, as described below.

Diagonal disk sections are explicitly inverted. This is not required by the algorithm but the extra work needed is more than compensated by the improved efficiency of the parallel implementation as described below. Pivoting during the inversion is restricted to columns inside a disk section since the rest of the pivot column would not be available in memory. To avoid dealing with non square disk sections, the matrix is implicitly padded to make all disk sections square. The steps of the algorithm are ordered so that all modifications of a disk section are done before the section is written to disk. Figure 7.1 shows the labelling and elimination order for the factor algorithm assuming 4×4 disk sections.

1 A11	3 A12	7 A13	13 A14
2 A21	4 A22	8 A23	14 A24
5 A31	6 A32	9 A33	15 A34
10 A41	11 A42	12 A43	16 A44

Figure 7.1. Labelling and elmination ordering of a 4×4 block matrix.

The actual block operations for a $t \times t$ matrix of disk sections is as follows:

```
for i = 1, t

    for j = 1, i-1          //do ith row
        for k = 1, j-1
            Aij = Aij - Aik*Akj
        endfor
        Aij = Aij*Ajj       //Ajj already inverted
    endfor

    for j = 1, i-1          //do ith col
        for k = 1, j-1
            Aji = Aji - Ajk*Aki
        endfor
    endfor
```

```
    for j = 1, i-1          //do ith diagonal block
        Aii = Aii - Aij*Aji
    endfor
    Aii = inverse of Aii
endfor
```

The corresponding solve algorithm, once for each block column of B is:

```
for i = 1, t                //forward elimination
    for j = 1, i-1
        Bi = Bi - Aij*Bj
    endfor
endfor

for i = t, 1,-1             //back substitution
    for j = t,i+1, -1
        Bi = Bi - Aij*Bj
    endfor
    Bi = Aii*Bi
endfor
```

The ordering of the updates was chosen so that all modifications to a particular disk section are done at once. Thus only finished disk sections are written to disk (except during the forward solve). This minimizes disk writes which are often slower than disk reads and it provides for a natural checkpointing of the factor algorithm since no partially computed block is ever on the disk. Since each block column of the solve algorithm is independent and is completed relatively quickly, checkpointing of the solve algorithm is less important, but could be obtained by storing the forward solve results in temporary files. Only three types of operations are needed. In order of importance, they are (for X, Y, and Z disk sections):

1. $X = X - Y * Z$
2. $X = Y * Z$
3. explicitly invert X.

Without the explicit inversion, a fourth kind of operation would be needed which implemented block forward and back substitution. This cannot be implemented as efficiently on a parallel machine as matrix multiply. In any case only t inversions are needed total so that for large matrices the inversion time is insignificant. The algorithm as described

at this level is completely sequential. All of the parallelism is hidden inside of the three disk section operations.

7.5 Parallel Matrix Multiply

Both $X = X - Y * Z$ and $X = Y * Z$ are implemented using the functionality of the BLAS3 (see [4]) subroutine ZGEMM which computes

$$X = \alpha * Y * Z + \beta * X \tag{1}$$

with $\alpha = -1$, $\beta = 1$ and $\alpha = 1, \beta = 0$ respectively.

The cube level algorithm treats the nodes as a $k \times k$ torus (which will be called the **mesh**), which is a subset of the hypercube topology. Each disk section is divided in square subsections called node sections, one for each node. In computing a disk section matrix product, each X node section requires k node section multiply accumulate operations. At the beginning of the matrix multiply routine each node has in memory one X node section, one Y node section, and one Z node section. Each will implement equation (1) for its node sections while simultaneously passing its Z section up and its Y section left in the mesh. If node sections are large enough, the message passing will finished before the computation, so the next node section computation (and message passing) can begin immediately.

The disk section multiply loop consists of k phases in which each node does the following:

Post receive from down
Post receive from right
Post send to up
Post send to left
Multiply-Add
Wait for completion of messages

In the *kth* (last) phase no data is sent since there will be no further phases.

To obtain the correct answer, the initial assignment of node sections to disk sections must be done carefully. X node sections are assigned in their natural ordering, Y node sections are shifted by rows, and Z node sections are shifted by columns. Node sections in the first row of Y are assigned in their natural order. The next row is circularly

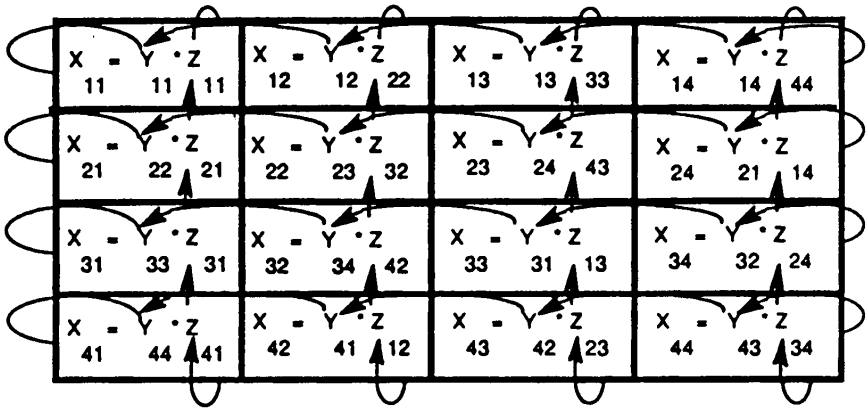

Figure 7.2. Initial assignment of node sections to nodes (and communication pattern).

shifted one position left in the mesh. Each succeeding row is shifted further left. Similarly, the first column of Z is assigned naturally, the second column is shifted up one position, and each succeeding column is shifted further up. See [5] for more details. Assuming a 4×4 node mesh, Fig. 7.2 shows the initial assignment of the X, Y, and Z node sections and communications paths.

7.6 Parallel Matrix Inversion

Parallel matrix inversion is implemented using the Gauss-Jordan algorithm for hypercubes as described in [6]. Each node owns a node section of a diagonal disk section. At each phase of the inversion one pivot column and row are processed. Pivoting is done only inside of node sections so that no communication is needed. A sketch of the algorithm follows:

```
Loop over diagonal elements:

If I own the pivot point
        pivot
        send pivot number to other nodes in my row and column
        invert diagonal element
        scale my part of pivot column
        send my part of pivot column to other nodes in my row
                (including inverted diagonal element)
        send my part of pivot column to other nodes in my column
        update my node section
        elseif I own part of pivot row
        receive pivot number
        pivot
        send my part of pivot row to other nodes in my column
        receive part of pivot column
        update my node section
 elseif I own part of pivot column
        receive pivot number
        receive part of pivot row
        scale my part of pivot column
        send my part of pivot column to other nodes in my row
        update my node section
 else (I own neither)
        receive pivot row
        receive pivot column
        update my node section
 End Loop
```

After all of the updates, apply pivoting to columns in reverse order.

Each update is a rank one change to the node section. It was implemented as a call to ZGEMM with Y a single column and Z a single row respectively.

7.7 User Interface

A 20k matrix requires 6.4 gigabytes of disk space to hold the factors. If the initial data is also in a disk file, then almost 13 gigabytes are required. To allow maximum use of disk space, LOOCS does not require an input file, it requires a user-supplied subroutine to fill node

sections as needed. Whenever a new disk section is needed all nodes call the matrix routine to fill in their node sections. This routine can read a disk file, but alternately, the matrix could be computed on the fly from the initial model, which requires much less space than the matrix itself. Similarly the solve routine calls a user supplied subroutine to generate sections of the B matrix. Any memory needed for these fill computations will not be available as buffer space for LOOCS.

In addition to these routines, the user specifies the problem size and the amount of workspace provided to LOOCS for buffer space. The size of the workspace sets an upper bound on the size of a node section since each node must be able to store eight node sections (three for the in core matrix multiply, two for node-to-node communication, and three for asynchronous I/O). This node section size and the number of processors defines the upper bound on the size of a disk section. The disk section size is compared to the size of A to see how many disk sections are needed. In general the last disk section in each row and column will extend beyond the edge of the matrix. The node section size is then reduced to minimize the overlap. For example, for a 20,000 problem, 64 processors, and enough memory for node sections of size 240, the node section size will be shrunk to 228, the disk section size will be 1824, there will be 11 disk sections per row of the matrix and the last disk section will overlap by 64. This overlap is actually stored on the disk and manipulated to avoid the hassles of handling odd sized sections.

The solve algorithm uses rectangular node sections. The row dimension of a solve node section is the same as the dimension of a factor node section. The column dimension is a free parameter and is determined in exactly the same way as the factor node section dimension based on the available memory and the number of columns in the B matrix.

7.8 Stability

Gaussian elimination with no pivoting is unstable (except for special classes of matrices) since the diagonal matrix elements may be much smaller than the rest of the matrix elements. This is solved by using row swaps (pivoting) to bring large elements to the diagonal. In the block algorithm used by LOOCS, pivoting can be done only over the part of the column which is in memory. Thus the algorithm is un-

stable and may fail completely or produce inaccurate results on certain problems for which accurate results could be obtained by pivoting over the whole column.

Full column pivoting can be supported by a slab style algorithm. This provides better stability at the cost of increasing the I/O to compute ratio. On a 20k problem on 64 compute nodes, a slab width of about 300 could be supported. This increases the required I/O (both disk and node-to-node) by about a factor of 3 or 4, and would probably slow the performance by at least a factor of 3. Furthermore, this performance degradation increases as the size of the problem increases.

The crucial question is how often do unstable matrices arise in these large problems. This would require significant testing on real data to determine. Many real problems are quasi diagonally dominant in which the large matrix elements cluster near the main diagonal. Such problems are less likely to be unstable than uniformly random matrices.

Since LOOCS cannot prevent instability, it has a user option to monitor it. Since the explicit inverse of the diagonal sections is computed, it is possible to compute the condition numbers of these sections. A large condition number is an indication of instability. LOOCS offers the user an option to monitor stability and terminate a run if too large a condition number is encountered. If a diagonal section is exactly singular, then it is impossible to invert it and LOOCS terminates with an error message. Only significant field experience with the code will determine whether instability is a significant problem in practice.

7.9 Node Optimization

To obtain optimal performance on the i860, the ZGEMM routine was hand-coded in assembly language, taking advantage of the data cache, dual-instruction mode, pipelined arithmetic, pipelined memory access, and the delayed jump instruction. The kernel is a triply nested loop. The inner loop is the complex zaxpy operation, $\overline{x} = \overline{x} + a * \overline{y}$, where a is a constant in a register, $\overline{x}$ is a vector in cache, and $\overline{y}$ is a vector pipelined in from memory. The middle loop is over columns of the matrix Y, accumulating the result in cache. The outer loop is over different columns of the matrix X. The elements of the Z matrix are the constants for different times through the inner loop. The i860 has no native complex arithmetic so that each complex multiply add is

implemented as four real multiplies and four real adds. The inner loop is 16 instructions, processes two elements of the vector $\overline{x}$, and performs eight multiplies and eight adds. To accommodate this, factor node sections are required to have even dimensions. The asymptotic speed of the inner loop (at 40 MHz) is 40 Mflops. Some overhead is attributable to memory refresh and outer loop overhead. The asymptotic speed obtained in the kernel is 37.5 Mflops with an n-half of about 10 (see Table 7.1). For a more detailed description of this kernel see [7].

It is possible to improve the performance of the matrix multiply algorithm by about 25% by using the implementing complex matrix multiply using three real matrix multiplies and five matrix adds, as follows:

$$(A+iB)(C+iD) = A*C - B*D + i((A+B)(C+D) - A*C - B*D)$$

See [8] for a numerical stability analysis. Since matrix adds require much fewer flops than matrix multiplies, this formula saves about 25% of the flops. However, single matrix adds cannot be implemented with high efficiently on the i860 because no operand is reused. Capturing the performance advantage requires writing a custom assembly language routine which does the entire operation at once. An implementation of this algorithm is in progress and it is estimated that the overall performance would improve by about 20% when this modification is inserted, but no measured numbers are available.

There is a similar strategy due to Strassen for multiplying two real matrices. If the two matrices are partitioned into 2×2 block matrices, then the multiply can be accomplished in seven half multiplies and eighteen half adds. This was shown to be useful on a Cray computer in [9]. It is less clear whether this approach can be effectively implemented on the iPSC/860 and is not currently being pursued.

7.10 I/O Optimization

Each I/O node can transfer data on or off the disk at about one megabyte/second. Enough disks are needed to hold the factors and solution files. Enough I/O nodes are needed to provide enough bandwidth to ensure that during a matrix multiply, the I/O finishes before the computation. Because I/O scales as the square of the node section and computation scales as the cube, the number of I/O nodes required

depends on the node section size which in turn depends on the available memory. For 64 i860 nodes with eight megabytes of memory, eight I/O nodes are sufficient.

However, a further tuning was necessary to obtain optimal performance. The data cache on the I/O node is a shared resource. 64 compute nodes requesting service from a single I/O node thrashes the disk cache and causes a degradation in performance. This could be solved by increasing the memory on the I/O nodes, but in this instance it was solved by partitioning the factor results into a separate file for each of the columns of the processor mesh, and restricting each file to the single I/O node anchored in that column. Thus only eight compute nodes were active at any one I/O node. This had the added advantage of eliminating all contention among I/O messages headed for different I/O nodes, since each I/O node only communicated up and down its column of processors.

There is one problem with the partitioning strategy. A node section written by one compute node is read by a different one which may not be in the same column of the mesh. This was solved by pre-processing disk sections before they were written to the disks. Node sections are sent by node-to-node messages from the node which computed them to the node who will later read them before they are written. Thus sections below the main diagonal, who will later be left factors, are twisted by rows, and sections on or above the main diagonal, who will later be right factors, are twisted by columns. Thus when nodes read node sections for matrix multiplies, they only have to read the node sections which they wrote.

Finally, the anchor nodes to which the I/O nodes were connected in each column were chosen to minimize the contention between I/O messages and node-to-node messages. This further reduced message contention.

7.11 Performance

Performance of the parallel matrix multiply routine is summarized in Tables 7.1 and 7.2. For $k * k$ nodes, the matrix multiply takes k phases, all but one of which involves node-to-node communication. On a single node, there is only one phase and no communication so that column simply measures the performance of the assembly language kernel. The asymptotic speed is about 38 Mflops and half that speed is

	Nodes			
Problem Size	1	4	16	64
8	17.1			
16	28.4	15		
32	34.1	35	53	42
64	36.5	106	113	141
128	37.4	130	207	365
256	37.8	141	423	763
512	·	147	548	1,379
1,024	·	·	578	2,165
2,048	·	·	·	2,300

Table 7.1 Aggregate performance of parallel matrix product (megaflops)

obtained for matrices about 10 by 10. For more than one node, there are two possible ways to measure performance. Table 7.1 compares a fixed size matrix spread out over the available nodes, and measures aggregate performance.

Table 7.2 assumes fixed size node sections and measures performance per node. It can be seen that if a small fixed-sized problem is run on a sufficiently large number of processors, the efficiency drops so much that the aggregate performance actually decreases. This is a common problem. This style of machine is not designed to run a small problem blazingly fast—it is designed to run big problems fast. For fixed-sized node sections, the efficiency decreases because more and more of the phases involve communication. However since the arithmetic scales as the cube of the node section and the communication scales as the square of the node section, this effect is almost negligible for large node sections.

The 2.3-gigaflop performance on a 64-node system is not obtainable on a real problem because of the inefficiency of the inversion, the cost of doing I/O while computing, and the I/O which is not overlapped with communication. The I/O degradation is much more serious than the node-to-node communication degradation for two reasons. First, the compute node must do significant work to figure out which disk blocks are needed from which disks. Furthermore, the I/O traffic is packetized into 4k byte disk blocks, which lowers the efficiency of the

	Nodes			
Node Section Size	1	4	16	64
8	17.1	3.8	3.3	2.2
16	28.4	8.8	7.1	5.7
32	34.1	26.5	12.9	11.9
64	36.5	32.5	26.4	21.5
128	37.4	35.3	34.2	33.8
256	37.8	36.8	36.1	35.9

Table 7.2 Efficiency of parallel matrix product (megaflops/node)

communication. The cost of the inversions is noticeable both because of the additional flops required by the full inversion and because the inner kernel of the inversion is not as efficient as the full matrix-matrix product. For node sections of about 200, the inversion achieves only about 8 mflops per node.

Table 7.3 gives the aggregate performance of the complete factorization algorithm run on a 64-node machine for eight I/O nodes and 16 disks. A very simple matrix fill routine was used. The run times obtained in practice would depend on the time taken in the matrix fill routine. For the largest problem the performance achieved is more than 25 Mflops per node sustained, which is more than 60% of the theoretical peak of the machine for matrix computations. The performance obtained in the solve algorithm depends on the number of right-hand sides. For one right-hand side the algorithm is strongly I/O bound. For a full disk section of right-hand sides, the solve is more efficient than the factor since there is no inversion step.

7.12 Extensions to Odd Powers of Processors

The algorithm as described so far runs only on square numbers of processors. An implementation for odd powers of 2 is in progress. The approach is to partition the nodes into two square meshes and update a row and a column of the factorization simultaneously. The row requires more work because of the multiplication by the diagonal

	Nodes		
Problem Size	Seconds	Megaflops	Megaflops/ Node
5,000	495	673	10.5
10,000	2,261	1,178	18.4
15,000	5,434	1,400	21.9
20,000	13,842	1,541	24.1
25,000	25,756	1,612	25.2

Table 7.3 Performance on a 64-node machine.

blocks, so the set of processors doing the column also does the diagonal section. A small amount of synchronization is needed between the processors creating the row and the processors creating the diagonal block. The final inversion of the diagonal block can be shared among all the processors. In the solve phase, it is simply necessary to make sure that the B matrix has an even number of block columns so that each set of processors can solve for the same number of block columns.

No performance data for the extended algorithm is available. Having twice as many processors will require more I/O nodes to provide the necessary bandwidth to and from disk. The factor will be somewhat inefficient because of the needed synchronization. The solve should be just as efficient as before. There is no reason why a 128-node system should not run faster than 3 gigaflops.

7.13 Extensions to Tertiary Storage

Users wish to solve problems of order 100,000. The current algorithm would require 160 gigabytes of on-line disk storage to run, which is not cost effective. The most cost-effective approach would be to use video disk juke boxes to supply the appearance of large on-line storage. The I/O speeds required by the algorithm are modest and can be met by existing video disk technology. The algorithm accesses large segments sequentially so disks will rarely have to be swapped. Each column of processors would need three disks on-line (two input and one output). Further tuning of the sequence of disk operations could

further reduce the number of times the input disks would have to be swapped. A 128-node iPSC/860 with 16 I/O nodes with 16 juke boxes with three active disks should be able to factor a 100k problem in about 8 days.

7.14 Conclusions

The LOOCS code running on the iPSC/860 parallel computer is a very effective way to solve large dense systems of linear equations such as those arising from the method of moments.

7.15 Addendum

Since this manuscript was written, many changes have been incorporated in ProSolver-DES. The Winograd kernel has been implemented. The two square algorithm has been implemented. A rank 75,000 problem has been solved on a 128-node iPSC/860 and 96 gigabytes of disk. The factorization took 2.7 days at an aggregate speed of 4.9 Gflops or 38 Mflops/node (where these measures are effective megaflops, computed by taking the traditional operation count ((8/3 N*N*N)) and dividing by the elapsed time).

An entirely new "slab" version of the algorithm has been implemented which partitions the matrix into block columns instead of squares to allow for full column pivoting. A symmetric version of the block solver has been implemented which saves nearly half the storage and half the work.

Intel has introduced a new generation of machine, the Intel Paragon^{TM} System. The nodes have two 50-MHz i860XPs, one application processor, and one message processor. The network is a two-dimensional mesh with bandwidth of 200 MB/sec in each direction of each link. The operating system is now OSF/1 Unix with extensions for message passing. Both solvers are being ported to the Paragon. A Strassen kernel is being implemented. A preliminary port of the block solver without the Strassen kernel achieved 48 Mflops per node, factoring a rank 25,000 problem on 36 nodes. It is expected that final performance will be between 60 and 70 Mflops per node when the Strassen kernel is integrated.

References

[1] *i860 64-bit Microprocessor Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA, 1989.

[2] Lillevik, S., "Touchstone program overview," *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.

[3] Pierce, P. R., "A concurrent file system for a highly parallel mass storage subsystem," *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[4] Dongarra, J., Du Croz, J., Duff, I., and Hammerling, S., "A set of level 3 basic linear algebra subprograms," *ACM Trans. on Math. Soft.*, 1989.

[5] Duncan, K., "A survey of parallel computer architectures," *Computer*, **Vol. 23,** No. 2, 9, 1990.

[6] Hipes, P. G., and Kupperman, A., "Gauss-Jordan inversion with pivoting on the Caltech mark II hypercube multiprocessor," *Proceedings of the Third Conference on Hypercube Multiprocessors*, 1621–1634, 1988.

[7] Scott, D. S., "A fast i860 matrix-matrix product routine", *Technical Report*, Intel Scientific Computers, Beaverton, OR, 1990.

[8] Higham, N. J., "Stability of a method for multiplying complex matrices with three real matrix multiplications," *Numerical Analysis Report No. 181, Department of Mathematics*, Univeristy of Manchester, England, 1990.

[9] Bailey, D. H., Lee, K., and Simon, H. D., "Using Strassen's algorithm to accelerate the solution of linear equations," *Journal of Supercomputing* 4, 357–371, 1990.