LU FACTORIZATION AND THE LINPACK BENCHMARK ON THE INTEL PARAGON *

DAVID WOMBLE†, DAVID GREENBERG†, STEPHEN WHEAT†, AND ROLF RIESEN†

Abstract.
An implementation of the LINPACK benchmark is described which achieves 72.9 Gflop/sec on 1872 nodes of an Intel Paragon. Implications concerning the architecture of the Paragon and the necessity of a high performance operating system like SUNMOS.

1. Introduction. The LINPACK Benchmark[1] and its kernel computation of the LU factorization[3] has become a common component of comparisons of high performance computers. One reason for its popularity is that it represents a core routine useful for the solving of any dense linear system, such as those which arise in electromagnetics.

Another reason for the popularity of the LINPACK benchmark is that it stresses the computational limits of its host machine while still requiring a significant amount of interprocessor communication. Thus it neither falls into the "embarassingly parallel" category nor is so communication bound that the ability of the nodes to do computation becomes secondary.

The LINPACK "highly parallel compting" benchmark takes as its input an $n \times n$ matrix, A, whose entries are double precision (64bit) floating point numbers and a vector, b, of length n. It is required to solve the linear system $Ax = b$. The most commonly used algorithm is to factor A into an upper triangular matrix, U, and a lower triangular matrix, L such that $A = LU$. The factorization is then used to compute the solution vector, x. An important feature of the LINPACK benchmark is that it requires that the solution pass several accuracy tests. This is a crucial constraint since there are several theoretical methods of solving the problem which become numerically unstable with fixed precision arithmetic. More details about the benchmark and LU are given in Section 3.

It has become standard practice to report LINPACK benchmark results as a number of flops/sec (e.g. our implementations achieves 72.9 Gflops/sec for a matrix of order 55,000 using 1,872 nodes). In addition, one also reports the size of the matrix for which half the speed of the largest matrix is achieved (e.g., order 17,500 in our case). Since, in most implementations, the overhead of parallelism grows more slowly than the computation time, it is advantageous to calculate LINPACK benchmark results using as large a matrix as possible. The use of large matrices is not unreasonable since a major reason for having high-performance computers is to be able to solve larger problems. However, the size matrix at which half the speed is achieved gives some indication of the architectural balance of the machine. A large range of matrix sizes which achieve at least half peak speed is clearly more desirable than a small range.

Recently it has become apparent that the use of Mflops/sec as a metric can be misleading. The first question is what should be counted as a floating point operation. Most studies count the addition, subtraction, or multiplication of two double precision floating point numbers as one floating point operation. However, on some machines the number of cycles to do a multiplication is greater than that to do an addition.

* This work was supported by the U.S. Department of Energy and was performed at Sandia National Laboratories operated for the U.S. Department of Energy under contract No. DE–AC04–94AL85000

† Sandia National Laboratories, Albuquerque, NM, 87185

Therefore, it is not clear whether counting one flop per addition or multiplication is an unbiased number. (For computations involving divisions, inverses, and square roots the disparity can be even greater.) Some researchers have also begun to argue that other, noncomputational, operations should be counted. If an algorithm can reduce the amount of time to complete a computation by taking branches which depend on comparisons of results of earlier operations why should they be "penalized" in the Mflop number reported.

The standard LINPACK benchmark report[1] sidesteps these questions by assigning a fixed number of flops to a given size matrix ($\frac{2}{3}n^3$) regardless of the algorithm used. While this approach helps a user know on which machine the LINPACK benchmark will run fastest, it can obscure the architectural comparison of the machines. The fact that a more clever algorithm runs faster on machine A than a less clever algorithm runs on machine B does not tell us much about the relative merits of the two machines.

In this paper we try to tread a path between the extremes of having a fixed flop count and having to give a weight to every type of operation. We use the standard count of $\frac{2}{3}n^3$ flops and use only algorithms which use at least this number of floating point additions and multiplications. However, since the Paragon node architecture does not treat additions and multiplications equally (see Section 2 for further details) we feel free to employ algorithmic techniques to change the mix of additions and multiplications and to reduce the cost of memory loads and stores.

2. The Intel Paragon system. To understand the performance of our LINPACK benchmark code and to understand what the performance tells us about the Paragon architecture, it is necessary to give a fairly detailed description of the Paragon currently located at Sandia National Laboratories. Three important components of the Paragon architecture are the i860 node architecture, the mesh interconnection network, and the SUNMOS operating system.

2.1. Node architecture. Each Paragon node contains two i860 microprocessors [5], 16 or 32 MBytes of DRAM, and hardware to enable access to the interprocessor communication network. In the original design of the machine it was envisioned that one i860 would be used for computation and the other to aid communication. It is our intention, however, to use both microprocessors to do computation for algorithms such as LU factorization (see Section 7 for more on the use of the second processor).

Whether one i860 or two is used for computation, the internal architecture of the i860 has important effects on performance. The i860 has two floating point pipelines as well as an integer pipeline. The floating point addition pipeline can produce a double precision result every clock cycle while the floating point multiplication pipeline can produce a double precision result every two clock cycles. Since the integer pipeline can do index arithmetic it is possible to sustain three floating point operations every two clock cycles. This yields a peak per processor performance rate of 75 Mflops/sec for the standard 50MHz clock rate.

Many factors make this 75 Mflops/sec rate difficult to achieve. Foremost is the fact that it requires that there be exactly two additions done for every multiplication. It is very rare for a computation to have exactly this ratio. The standard LU algorithm, for example, performs one addition for each multiplication. Such a 1 to 1 ratio reduces the peak performance to 50 Mflops/sec/processor. As is explained in Section 3 we can modify the algorithm to increase the ratio of additions to multiplications but cannot achieve a 2 to 1 ratio for double precision real arithmetic.

A second important factor limiting performance is memory bandwidth. Executing three floating point operations every two cycles requires operands to be delivered at a rate of three, 64 bit words per cycle or 1.2 Gbytes/sec. Although the i860 data caches are capable of sustaining these rates to the registers the DRAMs cannot provide data to the caches at this rate. Thus it is crucial that there be reuse of data in the cache. A standard vector–vector operation (such as BLAS 1 [6]) provides no cache data reuse and thus reduces peak performance by at least a factor of three. LU factorization (again, see Section 3 for details) can be cast in terms of matrix-matrix operations (such as the level 3 BLAS [2]), which allow better cache reuse. However, the use of matrix–matrix operations complicates the code and may reduce the ability to have balanced parallelism.

When more than one processor is doing computation on a node, the memory to cache bandwidth along the internal bus is even more critical. Two processors sharing memory can also result in memory conflicts although most memory conflicts can be avoided in the LU code.

A final factor which reduces node performance is the inevitable necessity of doing some set up code. The calculation of the initial location of arrays, the initial filling of caches and pipelines, etc. takes time but contributes no flops.

All of these factors contributed to the necessity of using hand-tuned assembler code for computational kernels in the LINPACK code in order to achieve high performance. In our implementation, we use the assembly language BLAS routines supplied by Intel; the remainder of the code is written in C.

2.2. Interconnection Network. Nontrivial parallel codes require interprocessor communication. As stated above, our Paragon consists of nodes connected in a two-dimensional mesh. The height of the 1840–processor mesh is sixteen nodes and the width is about 115 nodes. (Some nodes of the machine are ordinarily reserved for controlling I/O devices but can be used for computation when it is known that no disk I/O will be done. Thus the number of mesh columns can be increased to 117 for special runs.)

The network is supported by bidirectional, 200 Mbyte/sec links between mesh neighbors (i.e. each node is connected to a east, west, north, and south neighbor unless it is on the edge of the mesh). Router chips allow a message to be forwarded in "wormhole" fashion between any two nodes in the system without processor intervention. Each node contains two DMA devices, which allow one message to be sent and one message to be received independent of the CPU's activity (excepting memory and bus conflicts).

As in any message passing system there is a significant OS overhead for sending a message, and much effort has been put into optimizing message passing modules (see Section 2.3). The application–to–application bandwidth for large messages (larger than 64 Kbytes) varies between 150 Mbytes/sec and 175 Mbytes/sec. When small messages are sent or when multiple messages contend for a physical link this bandwidth can be further reduced.

The mesh topology makes contention, the desire for more than one message to use a physical link at the same time, an important issue. Since there are only 16 links connecting the left half of the machine to the right half, any attempt to have every processor on the left communicate with a partner on the right will cause each of these links to be shared by 57 messages. The result is a proportional reduction in bandwidth to less than 3 Mbytes/sec.

In our implementations of the LINPACK benchmark we chose the way in which

the matrix elements wer assigned to processors and the way in which broadcasts were done along rows and columns of the matrix to minimize the effects of contention, but contention remains an important factor. In addition, we sent very large messages whenever possible to minimize the effects of communication startups.

2.3. The Sandia/UNM Operating System. There are currently two operating systems that can be run on the Paragon. We have worked almost exclusively with SUNMOS (the Sandia/UNM Operating System). There are many reasons for this choice but the primary ones are that SUNMOS requires very little memory and it has relative fast message passing.

It was mentioned above that one of our motivating reasons for creating LU factorizations was to be able to solve very large systems. We have done research into factorizations[7] that use disks to augment central memory but fitting the matrix into central memory invariably allows for faster and simpler codes. The alternative operating system (OSF) occupies up to 8 Mbytes of memory *per node* leaving only about 8 Mbytes for the application program (on the 16 Mbyte nodes), although OSF does support virtual memory. In contrast SUNMOS uses less than 256 Kbytes. Thus, under SUNMOS we could almost double the number of entries in the matrix. On a single 32 Mbyte node we were able to solve a $2,000 \times 2,000$ double precision complex matrix. The matrix uses 32 million bytes, while the OS easily fits in the remaining memory (the difference between 32×2^{20} bytes and 32 million bytes).

In the discussion of network hardware in Section 2.2, it was mentioned that actual message passing speed can be considerably less than the speed of the physical links between nodes. A major component of this reduction is due to OS overhead. In particular the OS may have to buffer incoming messages and then copy them to user space. This buffering is expensive both in terms of time and in terms of system space allocated to buffers. It is our experience that SUNMOS does a much better job of allocating buffer space and of streamlining the entire message passing process. The result is faster message transmission and less space allocated to buffers.

3. LU factorization and the LINPACK benchmark. The solution of dense linear systems of equations is a critical kernel in many scientific applications, including boundary elements methods for partial differential equations and electromagnetic scattering. One of the most effective algorithms for this is LU factorization in which a matrix A is written as the product of a lower triangular matrix L and an upper triangular matrix U. (If pivoting is required, L is logically lower triangular.) The basic LU factorization algorithm $[L, U] = \mathrm{LU}\,(A)$ [3] is given below, where the matrices A, L and U are divided into submatrices denoted by subscripts (e.g., $A_{1,1}$ denotes the upper left submatrix of A).

$$\begin{aligned}
[L_{1,1},\ U_{1,1}] &= \mathrm{LU}\,(A_{1,1}) \\
U_{1,2} &= L_{1,1}^{-1} A_{1,2} \\
L_{1,2} &= 0 \\
U_{2,1} &= 0 \\
L_{2,1} &= A_{2,1} U_{1,1}^{-1} \\
[L_{2,2},\ U_{2,2}] &= \mathrm{LU}\,(A_{2,2} - L_{2,1} U_{1,2})\,.
\end{aligned}$$

Once the matrix is factored, the associated linear system can be solved with a forward substitution and a backward substitution using the matrices L and U respectively.

0	1	2	3	0	1	2	3	0	1
4	5	6	7	4	5	6	7	4	5
8	9	10	11	8	9	10	11	8	9
12	13	14	15	12	13	14	15	12	13
16	17	18	19	16	17	18	19	16	17
20	21	22	23	20	21	22	23	20	21
24	25	26	27	24	25	26	27	24	25
28	29	30	31	28	29	30	31	28	29
0	1	2	3	0	1	2	3	0	1
4	5	6	7	4	5	6	7	4	5

FIG. 1. *The assignment of matrix elements to processors in a torus–wrap. A* 10×10 *matrix is assigned to 32 processors arranged in a* 8×4 *mesh.*

If the standard matrix multiplication algorithm is used, LU factorization requires $2n^3/3 + O(n^2)$ floating point operations (multiplication and addition are counted as separate operations) and forward and back substitution require $n^2 + O(n)$ floating point operations each.

Because of the pervasiveness of dense linear systems of equations in scientific applications, the LINPACK benchmark has become a popular measure of machine performance [1]. The matrices that are used for this benchmark generally have double precision, real, random entries. Because the entries are random, partial pivoting must be used for stability. The benchmark specifications require that the solution be checked for accuracy using the quantity $\|Ax - b\| / (\|A\|\|x\|)$, where x is the solution and b is the right hand side of the system.

The LINPACK benchmark includes categories for solving a 100×100 system using a prescribed FORTRAN program, solving a 1000×1000 system using any code ("best effort"), and solving a system of any size on a parallel computer. Our work is focused on the third category; the numbers quoted for this category are

R_{max}	the performance in Gflops/sec for the largest problem run on a machine
N_{max}	the size of the largest problem run on a machine
$N_{1/2}$	the size where half the R_{max} execution rate is achieved
R_{peak}	the theoretical peak performance in Gflops/sec for the machine.

The computational rates are based on an operation count of $2n^3/3 + 2n^2$ regardless of the method used to solve the system. This opens the door to some creative interpretations of "Gflops/sec", and readers must use caution in interpreting the results. In our work, we take care to use algorithms that, in fact, require $2n^3/3 + 2n^2$ floating point operations. Thus, the numbers quoted in this report can be taken as a measure of the actual speed of the processor.

To obtain the LINPACK benchmark report [1], send mail containing the message "send performance from benchmark" to *netlib@ornl.gov*.

4. Implementation. A basic implementation of LU factorization for parallel computers is described in [4], although changes have been made to adapt the code to the Paragon (from the nCUBE 2).

The matrix entries are distributed to the processors in a torus–wrap decomposition (also called a scattered decomposition) as shown in Fig. 1. A square torus–wrap is known to minimize the communication volume in a parallel implementation of LU factorization; however, the presence of column–oriented operations such as the pivot

search suggest that the minimum overall time for the algorithm is achieved with a slightly non–square mapping, that is, more processors should be assigned to each row than to each column. In the case of a 1840 processor Paragon, we choose the assignment of the matrix entries to processors to match the physical layout of the processors. This means that each column of the matrix is assigned to 16 processors, and each row is assigned to 115 processors. If 1872 processors are used, processor mesh is 16×117.

Pseudocode for the Paragon implementation is shown in Fig. 2. From Fig. 2 we identify four required communication primitives:

1. binary exchange between all processors sharing a column,
2. exchange of the pivot row and diagonal row,
3. broadcast of a row to all processors sharing a column.
4. broadcast of a column to all processors sharing a row,

The binary exchange is synchronizing and must be done a quickly as possible. We use a logarithmic binary exchange that required $\lceil \log_2 p_c \rceil$ steps, where p_c is the number of processors sharing each column. Because the pivot search required a synchronization of processors sharing a column, the broadcast of a row to all processors sharing a column must be as fast as possible. Hence, we again use a logarithmic algorithm requiring $\lceil \log_2 p_c \rceil$ steps. In contrast to processors sharing a column, processors sharing a row do not need to be so tightly synchronized; it is acceptable for the processors holding column j to slightly lag the processors holding column $j-1$. Consequently, we use a linear broadcast for columns of the matrix. This has the feature that all processors have the same amount of work in the broadcast, equal to one send and one receive. Finally, we use point–to–point communication routines supplied by the SUNMOS operating system for the exchange of the pivot row and diagonal row.

The update of the matrix $A_{\alpha,\beta}$ in Fig. 2 is accomplished using the level 3 BLAS routine DGEMM, which implements a standard matrix–matrix multiply algorithm requiring $O(n^3)$ operations. (The BLAS library for the Paragon was written by Kuck and Associates.) An alternative to this would be to update $A_{\alpha,\beta}$ each time through the loop resulting in a slightly simpler algorithm. The problem with this is that the lack of data reuse for matrices with one dimension equal to one (i.e., $BBS = 1$) results in increased data movement from main memory to the i860 processor swamping the processor–memory bus. As this dimension increases, bus traffic per floating point operation decreases until the processor speed becomes the limiting factor. In practice, we observe that this happens with $BBS = 4$.

The speed of the DGEMM routine continues to increase as BBS increases; however, once the processor–memory bus speed is no longer the limiting factor, the inefficiency of updating the pivot column of $A_{\alpha,\beta}$ in a separate step using level 1 and level 2 BLAS routines reduces the speed of the overall algorithm. Thus, for double precision real matrices, we set the BLAS block size, BBS, equal to 4.

The LINPACK benchmark requires that a linear system be solved, so forward solve and backward solve routines must be included in the code. Pseudocode for the forward solver is shown in Fig. 3. The backward solve code is similar. The forward solve requires only $n^2 + O(n)$ floating point operations and $4n^2 + O(n)$ bytes of communication. Thus, on the Paragon it is communication bound. The primary feature of the forward solve algorithm shown in Fig. 3 is that is tries to increase the communication parallelism by collecting a block of $FSBS$ consecutive columns of the matrix into one processor, then passing the right hand side only to those processors that hold the column blocks. The collection of columns of the matrix can be done

in parallel so the non–overlapped communication is $O(n^2/FSBS + FSBS)$. A good choice of FSBS is $\sqrt{p_r}$, where p_r is the number of processors that share a row.

5. Results. The algorithm described above has been implemented and run on the Intel Paragon at Sandia National Laboratories with the computational nodes running the SUNMOS operating system.

In Table 1, we show the LINPACK benchmark results from our code. In addition to the numbers specified by the LINPACK benchmark, we list the physical arrangement of processors and the computational rate achieved on each processor in Mflops/second. Recall that p_c is the number of processors sharing each column, and p_r is the number of processors sharing each row. The results above the line were obtained using 16 Mbyte nodes. The machine does have some 32 Mbyte nodes and the result below the line was obtained using one of these nodes. We note that even though the machine has 1840 computational nodes, we were able to use 32 of the disk nodes for additional computations raising the total to 1872 nodes. We also note that the R_{peak} numbers are computed based on the hardware peak speed of 75 Mflops/second/processor even though the balanced number of additions and multiplications in the BLAS 3 routine DGEMM prevent the code from achieving more than 50 Mflops/second/processor.

TABLE 1
LINPACK benchmark results for the Intel Paragon

Number of Processors	R_{max} Gflops/sec	N_{max} order	$N_{1/2}$ order	R_{peak} Gflops/sec	p_c	p_r	R_{node} Mflops/sec
1872	72.90	55,000	17,500	140.4	16	117	38.9
1024	39.93	42,000	13,000	76.80	16	64	39.0
512	20.50	29,900	9,200	38.40	16	32	40.0
256	10.18	21,000	6,300	19.20	8	32	39.8
128	5.197	14,900	4,200	9.600	8	16	40.6
64	2.567	10,500	3000	4.800	4	16	40.1
32	1.315	7,500	2100	2.400	4	8	41.1
16	0.662	5,200	1400	1.200	2	8	41.4
8	0.339	3900	950	0.600	2	4	41.8
4	0.170	2760	600	0.300	1	4	42.5
2	0.085	1900	425	0.150	1	2	42.5
1	0.042	1386	250	0.075	1	1	42.6
1 (32 MB)	0.045	2000	275	0.075	1	1	45.0

Table 1 shows good scaling of the algorithm with respect to the number of processors. There is some fluctuation in the speeds due to the fact that we are not able to choose the exact optimum decomposition of the matrix in all cases. Specifically, the best ratio of p_r to p_c is closer to 2 than to either 1 or 4; however, in the case of 64 or 256 processors, we are not able to use a ratio of 2.

6. Implementation of complex LU factorization. The primary difference between the complex code and the real code is in the implementation of the BLAS 3 routine ZGEMM for complex matrix–matrix multiply. Because of the architecture of the i860 processor, the maximum speed can only be achieved for algorithms having twice as many additions as multiplications. The ratio of additions to multiplications in a real matrix–matrix multiply is one, In the case of a complex matrix–matrix

multiply, the ratio can be changed using a Winograd algorithm. Specifically, if a, b, c and d are complex numbers with real and imaginary parts denoted by the subscripts r and i respectively, we can compute the quantity $d = c + a * b$ in the following steps.

$$\begin{aligned} x_1 &= (a_r + a_i)(b_r + b_i) \\ x_2 &= a_r b_r \\ x_3 &= a_i b_i \\ d_r &= c_r + x_2 - x_3 \\ d_i &= c_i + x_1 - x_2 - x_3 \end{aligned}$$

This requires seven additions and three multiplications. If a is constant and b, c and d are vectors, then the term $a_r + a_i$ need only be computed outside the inner loop, reducing the operations per iteration to six additions and three multiplications. Even though we have achieved the ideal ratio of additions to multiplications, this algorithms requires nine floating point operations compared to the standard algorithm which requires eight operations (four additions and four multiplications).

This result can be improved slightly by computing in pairs, that is, computing quantities of the form $d = c + a_1 b_1 + a_2 b_2$. The Winograd algorithm above can be used to perform this operation in 12 multiplications and six additions. Further, if b_1, b_2, c and d are vectors, two additions can be computed outside the loop leaving ten additions and six multiplications per iteration. The total of sixteen floating point operations is thus the same as for the standard algorithm. The i860 processor requires twelve clock cycles to do these sixteen operations yielding a theoretical maximum speed for the ZGEMM routine of 66.7 Mflops/sec.

Another difference is the computation of the pivot. Instead of selecting the entry from the pivot column with maximum absolute value, $|z|$, we select the entry with the maximum $|z_r| + |z_i|$.

Finally, to calculate the speed of complex LU factorization, we use an operation count of $8n^3/3$ floating point operations. That is, it is the measure of the number of floating point operations on double precision, real numbers executed each second.

We have run a complex LU factorization (incorporating the differences mentioned here) on 1840 nodes of the Intel Paragon. The matrix was of size $42,000 \times 42,000$, and a BLAS block size of two was used. The computational speed was 102.05 Gflops/sec, or 55.5 Mflops/second/processor.

7. Using the coprocessor. As mentioned in Section 2.1, each node of the Paragon has two i860 processors. Although the second processor was originally intended to be used to improve communications, it can also be used as a computational coprocessor. Most of the floating point operations in the LU algorithm shown in Fig. 2 occur in the operation

$$A_{\alpha,\beta} := A_{\alpha,\beta} - V_{\alpha,1:b} U_{1:b,\beta}.$$

This can be easily split into two calls to the DGEMM or ZGEMM routine, one of which can be handled by the coprocessor. We note, however, that both processors must have access to the full matrix $U_{1:b,\beta}$.

Several practical conditions must be met before the coprocessor can be used effectively. First, the BLAS code must be reentrant, that is, each copy of the DGEMM code must maintain separate local variable so as not to interfere with each other during the computations. Second, a larger BLAS block size, BBS must be used. The two

processors share the processor–memory bus, and the larger block size is necessary to reduce the volume of data transfer from each processor. Third, the cache on each processor must be used effectively. Each processor should be able to preload large blocks of data so that it is not interrupted by the other processor during computations.

We have made some preliminary tests using the coprocessor on up to 256 nodes of the Paragon. These tests indicate that speeds greater than 62 Mflops/second/node can be sustained, a speedup of approximately 50% over the code that uses only a single processor per node. Despite the promising results so far, we have postponed further testing of the coprocessor code until Intel completes a hardware upgrade of the Paragon. When this upgrade is finished, we will append this report accordingly.

8. Summary. In this paper we have demonstrated that the LINPACK benchmark can be implemented effectively on the Intel Paragon (achieving 72.9 Gflops/sec on 1872 nodes). The existence of efficient implementations of LINPACK benchmark yields insight into several architectural features of the Paragon: high aspect ratio, relative speeds of floating point pipelines, and processor-to-memory bus speeds. It also served to validate the effectiveness of the Sandia/UNM Operating System (SUNMOS).

The Paragon at Sandia National Laboratories has a physical arrangement of processors with a long aspect ratio, 16 processors by 115 (or more) processors. It was not clear whether the resulting less than optimal distribution of matrix elements to processors would make linear algebra codes inefficient due to increased communication costs. We were able to achieve better than 92% scaled speed-up on the full machine thereby demonstrating that the communication costs, including message starups, message volume and contention for the communication links, were not overwhelming.

The i860 processor architecture prevents a code with equal numbers of additions and multiplications from achieving more than 50 Mflops/sec/node (as opposed to the theoretical peak of 75). For real double precision matrices the single node peak performance of 45 Mflops/sec/node was therefore 90% of practical peak. In the case of complex matrices, we were able to use Winograd's algorithm on small, sub-matrices to come closer to matching the architecture's ideal ratio of additions to multiplications and thus to achieve more than 60 Mflops/sec/node.

The Paragon node architecture relies on internal cache to bridge the gap between processor-to-memory bus speed and CPU speed. However, long vector operations common in many LU implementations render the cache ineffective. We were not able to circumvent all vector operations (such as those used in pivot searches) but could convert others to level 3 BLAS matrix-matrix operations which made better use of the cache. The use of level 3 BLAS operations did, however, somewhat reduce the parallel efficiency.

The ability of LINPACK codes to adapt to these architectural features should not, however, be taken as evidence that all codes can do so. Applications which do not lead to dense linear systems may have difficulty achieving as high per node computation rates.

The processor-to-memory bus speed is also a critical issue in the use of multiple processors on each node. To fully use the capability of the nodes, data must be made available to the processors at sufficiently high rates; use of two processors almost doubles the demands on the bus. For the matrix operations (level 3 BLAS), we have used larger block sizes to overcome this problem, although inefficiencies are introduced elsewhere; however, for vector operations (level 1 BLAS) this is not an option, and more than one processor cannot be used efficiently.

Our ability to achieve high LINPACK benchmark performance and to experiment with dual processor mode was dependent on our use of SUNMOS. The designers of SUNMOS have concentrated on fast communications and small OS memory size. The result has been an OS which allows scientific computations to make more efficient use of the hardware and to solve larger problems.

Acknowledgments. We would like to thank Mack Stallcup, Mike Proicou and Mark Rogers from Intel for the support they have provided to the project. We would also like to thank Greg Henry of Intel for help in obtaining reentrant BLAS routines for using the coprocessor.

REFERENCES

[1] J. Dongarra, *Performance of various computers using standard linear equation software*, Tech. Rep. CS–89–85, University of Tennessee, 1993.

[2] J. Dongarra, J. D. Croz, S. Hammerling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, TOMS, 16 (1990), pp. 1–17.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 2nd ed., 1989.

[4] B. A. Hendrickson and D. E. Womble, *The torus–wrap mapping for dense matrix calculations on massively parallel computers*, SIAM J. Sci. Stat. Comput., (1994, to appear).

[5] Intel Corp. Supercomputer Systems Division, *Intel Paragon User's Guide*, Beaverton, OR, October 1993.

[6] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, TOMS, 4 (1979), pp. 308–323.

[7] D. Womble, D. Greenberg, S. Wheat, and R. Riesen, *Beyond core: Making parallel computer I/O practical*, in Procedings of the 1993 Dartmouth Institute for Advanced Graduate Studies in Parallel Computation, Santa Clara, CA, June 1993, Springer–Verlag.

```
/* Processor q owns row set α and column set β */
/* BBS is the block size for BLAS 3 operations */
b := 0
FOR j = 1 to n
      b := b + 1
      /* Update column j and find pivot row */
      IF j ∈ β THEN
            IF b > 0 THEN
                  A_{α,j} := A_{α,j} − V_{α,1:b} U_{1:b,j}
            ENDIF
            γ^q := max_{i∈α} |A_{i,j}|
            binary exchange of γ^q to compute γ = max_q γ^q
            s := index of row containing the entry γ
      ENDIF

      /* Generate update vector, from column j of A */
      IF j ∈ β THEN
            A_{α,j} := A_{α,j}/γ
            IF j ∈ α THEN
                  A_{j,j} := γ * A_{j,j}
            ENDIF
            Broadcast V_{α,b} := A_{α,j} and s to processors sharing rows α
      ELSE
            Receive V_{α,b} and s
      ENDIF

      /* Exchange pivot row and diagonal row and broadcast pivot row */
      IF j ∈ α THEN
            Send w_β := A_{j,β} to processor owning A_{s,β}
      ENDIF
      IF s ∈ α THEN
            Receive w_β
            Broadcast row U_{b,β} := A_{s,β} to processors sharing columns β
            A_{s,β} := w_β
      ELSE
            Receive U_{b,β}
      ENDIF
      IF j ∈ α THEN
            A_{j,β} := U_{b,β}
      ENDIF

      /* Remove j from active rows and columns and update A */
      α := α \ {j}
      β := β \ {j}
      IF b = BBS THEN
            A_{α,β} := A_{α,β} − V_{α,1:b} U_{1:b,β}
            b := 0
      ENDIF
ENDFOR
```

FIG. 2. *Parallel LU factorization for processor q.*

```
/* Processor q owns row set α and column set β of L */
/* L is a lower triangular matrix with unit diagonal */
/* b is the right hand side vector */
IF q holds b_α THEN
      send b_α to processor holding column FSBS
ENDIF

FOR j = FSBS to n step FSBS
      /* Collect columns of matrix */
      FOR k = 1 to FSBS
            IF k + j − FSBS ∈ β THEN
                  send L_{α,k+j−FSBS} to processor holding column j
            ENDIF
            IF j ∈ β THEN
                  receive V_{α,k} from processor holding column k + j − FSBS
            ENDIF
      ENDFOR

      /* Update right hand side */
      IF j ∈ β THEN
            receive b_α from processor holding column j − FSBS
            b_{j−FSBS+1:j} := L^{-1}_{j−FSBS+1:j,j−FSBS+1:j} b_{j−FSBS+1}
            α := α \ {j − FSBS + 1 : j}
            β := β \ {j − FSBS + 1 : j}
            b_β := b_β − L_{β,j−FSBS+1:j} b_{j−FSBS+1:j}
            send b_α to processor holding column FSBS
      ENDIF
ENDFOR
```

FIG. 3. *Parallel forward solver for processor q.*