LAPACK Working Note ?
LAPACK Block Factorization Algorithms on the Intel iPSC/860 [*]

Jack Dongarra and Susan Ostrouchov
Department of Computer Science
University of Tennessee
Knoxville, Tennessee 37996-1301

August 1, 1990

Abstract

The aim of this project is to implement the basic factorization routines for solving linear systems of equations and least squares problems from LAPACK—namely, the blocked versions of LU with partial pivoting, QR, and Cholesky on a distributed-memory machine. We discuss our implementation of each of the algorithms and the results we obtained using varying orders of matrices and blocksizes.

1 Background

As part of the LAPACK project [1], we have developed a large body of mathematical software for solving linear algebra problems on shared-memory parallel processors. The goal of LAPACK is to design and implement a portable linear algebra library on high-performance computers. The methodology for the design has been to construct matrix-matrix algorithms and to develop software that encapsulates the computationally intensive parts in calls to the Level 3 BLAS [3]. This methodology results in a high operation-to-memory reference count and thus offers the possibility of high performance on most machines.

This paper describes an effort to reuse the software developed for shared-memory machines in the setting of distributed-memory machines. As such, it represents an effort to quickly bring up a numerical linear algebra library on message-passing machines by exploiting the existing base of software.

[*]This work was supported by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

2 Strategy

Since data movement on most high-performance architectures is slow compared to floating-point performance, block algorithms have been derived and implemented for matrix computations. Algorithms that consider a matrix as a collection of submatrices, where each submatrix is a group of columns, require little data movement. In this paper, we report on our attempts to use a fixed-width blocking strategy from the LAPACK routines for LU, QR, and Cholesky factorization on the Intel iPSC/860 hypercube.

Since the Intel iPSC/860 is a distributed-memory architecture and since we wish to minimize data communication, we chose the right-looking block algorithms for our implementation of LU, QR, and Cholesky, as implemented in LAPACK as routines SGETRF, SGEQRF, and SPOTRF respectively. The right-looking versions of the algorithms minimize communication and tend to spread out the computation and updating across the data. These block algorithms rely heavily on Level 3 BLAS routines.

We assume that our matrices are wrap-mapped across the processors by panels, where each panel is a number of columns of the matrix specified by the blocksize. Since communication is very expensive, we keep it to a minimum by communicating only once per iteration. This approach of communicating fewer times with larger messages is cheaper than communicating more frequently but with smaller messages. For more information about the cost of communication versus computation, see Dunigan [4]. To further simplify and minimize communication volume we use a gray code ordering scheme on a unidirectional ring as our connection topology for the hypercube. The ring topology is chosen because it best simulates the progression of the algorithms and decreases the communication volume. The gray code ordering together with the ring topology minimizes the traffic distance between nodes, and eliminates the intersection of messages.

In the following sections, we explain each of the implementations used for LU, QR, and Cholesky, as well as how we arrived at our pipelined approach. We then evaluate our timing results and specify an optimal blocksize for each of the algorithms.

3 Intel iPSC/860

The Intel iPSC/860 is an Intel i860 processor-based hypercube with 128 nodes attached to a 80386 host processor. Each i860 node has an 8-KB cache, 8 MB of main memory, and multiple arithmetic units which permit multiple operations per cycle [4]. Each node has a theoritical peak performance rate of 80 MFLOPS for single precision and 40 MFLOPS for double precision. The operation system on the nodes supports asynchronous communication, remote I/O from the host, and multitasking. Communication is supported by direct-connect routing modules on each node. These direct-connect modules relieve the node CPU of routing overhead and greatly reduce the penalty of multihop messages. With this new routing hardware, the nodes can be treated as if they were directly connected [4]. The communication time for a message is a linear function of the size of the message. Thus, the time T to transmit a message of length N is $T = \alpha + \beta * N$, where α represents a fixed startup overhead and β is the transmission time per byte. For messages of length 100 bytes or less, $\alpha = 75$ *microseconds*. However, for larger messages, $\alpha = 136$ *microseconds*. In both cases, $\beta = 0.4$ *microseconds*. The reason for this difference in startup cost is that messages of 100 bytes or less travel by route-acquisition protocol, whereas larger messages require a type of hand-shaking before the message can be sent.

4 LU Factorization

The right-looking block algorithm (SGETRF) computes a group of elementary transformations to zero out a number of columns at each step (this requires an unblocked LU factorization) and uses these transformation to update the remaining trailing submatrix. SGETRF calls three routines: SGETF2, the unblocked LU factorization for operations with a block column; STRSM, the triangular solve with multiple right-hand sides; and SGEMM, the matrix-matrix multiply. Initially, we coded a straightforward version of SGETRF with the communication of the factored block column (panel) and the pivots after the call to SGETF2.

The pseudo-code for this algorithm would be the following (where n is the number of columns in the matrix, *nb* is the blocksize, *nprocs* is the total number of processors allocated, and *proc* is the integer value used to keep track of which processor is currently doing the factoring and shipping of data):

```
$proc = 0$
DO $i = 1, n, nb$
    if ($proc = myid$) then
      call $sgetf2$
      send pivots and factored panel to other processors
    else
      receive pivots and factored panel from processor $proc$
    endif
    apply pivot interchanges to panels
    call $strsm$ for the triangular solve on panels
    call $sgemm$ to update the remaining panels
    $proc = mod(proc + 1, nprocs)$
ENDDO
```

Disappointment with the execution times led us to analyze our implementation in more detail with the help of ParaGraph [6]. ParaGraph is a parallel programming tool that graphically displays the execution of a distributed-memory program. It obtains the trace information that it needs from a communication library called PICL [5]. PICL is used throughout our implementations because of its portability and its simplification of many hypercube commands. ParaGraph confirmed our belief that there were inherent idle waits in the algorithm. The stairstep of idle waits can clearly be seen in Fig. 4. These idle waits occur when all of the processors are waiting for one processor—which will eventually call SGETF2 and communicate its information—first to do STRSM and SGEMM to all of its panels for the current call.

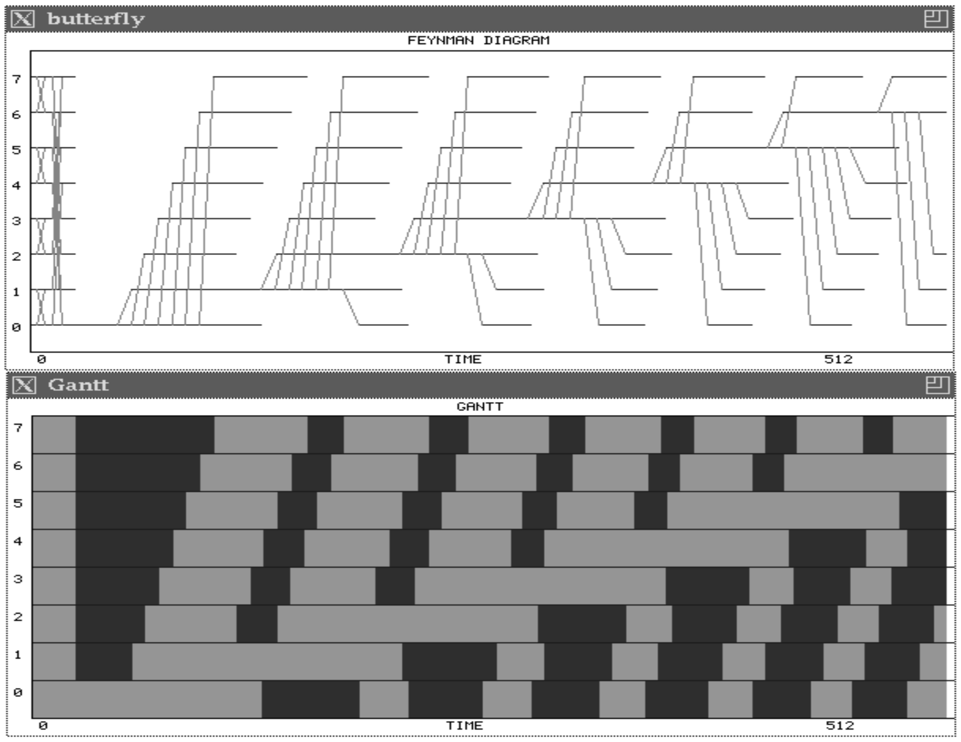

Figure 1: ParaGraph visualization of straight-forward LU factorization

A better strategy is to communicate as soon as possible using a pipelined approach. Specifically, instead of calling STRSM and SGEMM for all of its panels, the processor simply updates the next panel to be factored, factors that panel, ships the information to the other processors, and finishes the update of the rest of its panels for the previous SGETF2 call. We call this strategy *pipelined updating*.

The pseudo-code for the pipelined updating approach would be the following (where n, nb, $nprocs$, and $proc$ are as previously defined):

```
$proc = 0$
if ($proc = myid$) then
  call $sgetf2$ to factor my first panel and get things started
  ship factored panel and pivots in workout array
  to other processors
endif
DO $i = 1, n, nb$
    if ($proc = myid$) then
      copy $workout$ array into $workin$ array
    else
      receive panel and pivots into $workin$ array from
      processor $proc$
    endif
    all processors apply shipped pivots
    $proc = mod(proc + 1, nprocs)$
    if (I have panels left to modify) then
      if ($proc = myid$) then
        I'm the next processor to factor a panel so
        call $strsm$ and $sgemm$ only on my first panel
        then factor that panel (call $sgetf2$) and ship
        the information in $workout$ to all other processors
      endif
      all processors call $strsm$ and $sgemm$ with $workin$ array
    endif
ENDDO
```

It is evident from ParaGraph (see Fig. 2) that the idle waits between processors have now been eliminated. Idle time occurs only when the processor starts to run out of work, that is, when it has fewer panels to update than the other processors.

4.1 Testing and Results

The timings that we report are for the factorization only. They do not include the time to load the node program or to distribute the wrap-mapped matrix to the processors. We use only 64 and 128 nodes for our timings. Our matrices range from order 500 to order 5000, with blocksizes of 1 to 16.

Unfortunately, several implementation details on the Intel iPSC/860 limit performance. No Level 3 BLAS routines in i860 assembly language were available. There were, however, a few Level 1 BLAS routines coded in i860 assembly language—specifically, SAXPY, SSCAL, and a stride-one version of SDOT. We therefore incorporate calls to these routines whenever possible. We also use only column-major addressing and strides of one wherever possible. Single precision is used in our implementations.

We found these blocksizes to be sufficient to saturate the desired number of nodes that we were using, given the implementation of SAXPY we had, and also show starvation as it occurs. The major limiting factor is the performance of the SAXPY routine. After SAXPY reaches its peak performance, we begin to see the performance of LU leveling off. Fig. 3 is a graph of the MFLOP ratings for LU factorization. The graph shows a peak individual processor

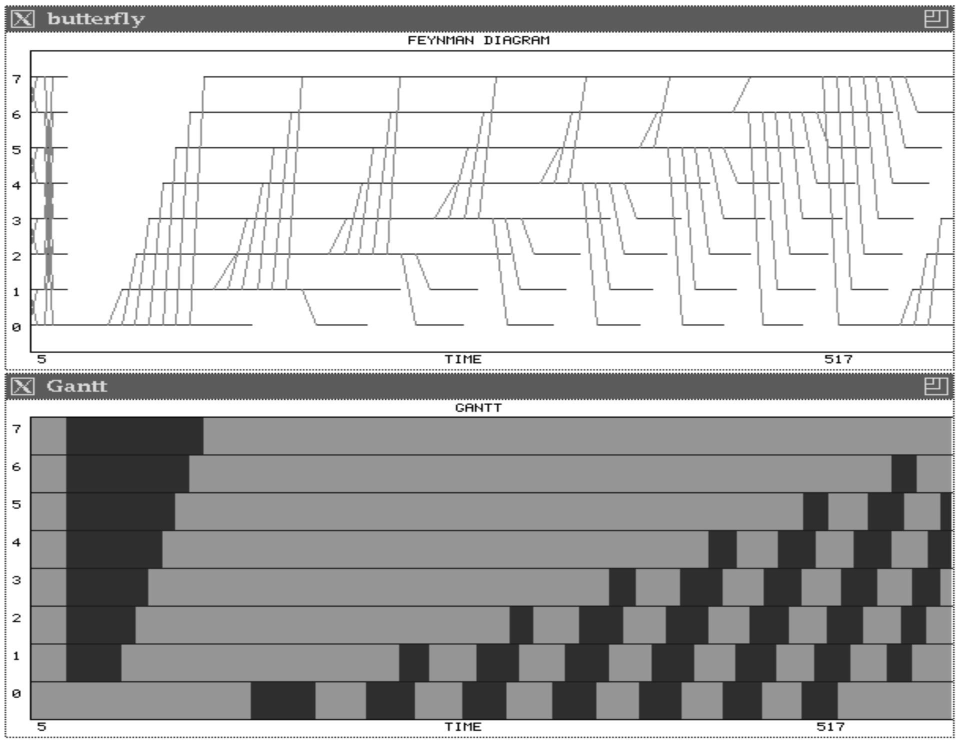

Figure 2: ParaGraph visualization of pipelined LU factorization

performance of 6.11 MFLOPS for 64 processors and 4.75 MFLOPS for 128 processors. To achieve comparable peak performance on 128 processors we would need to further increase the order of test matrices to match the work load achieved on 64 processors.

4.2 Optimal Blocksize

The optimal blocksize is—as expected—a function of the number of processors, the efficiency of the floating-point operations, and the order of the matrix. Interestingly, in examining our test results, we observed a range of optimal blocksizes.

Smaller blocksizes produce better load balancing on the nodes and thus decrease the amount of idle waits between the processors. However, this decrease in idle time is at the expense of an increase in communication overhead and a decrease in the floating-point performance of the individual nodes. Larger blocksizes, on the other hand, increase the floating-point performance of individual nodes and decrease the amount of communication overhead at the cost of larger messages. They also result in poorer load balancing between the nodes

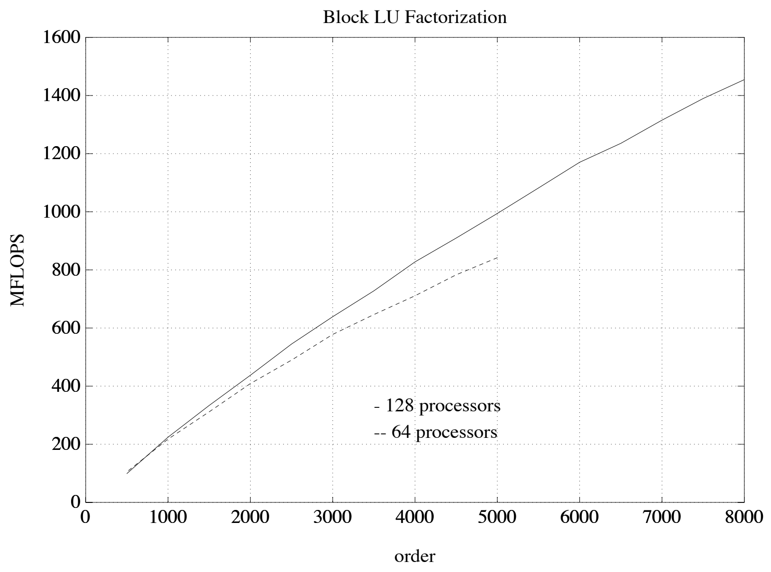

Figure 3: LU Factorization Results for Intel iPSC/860 with 64 and 128 nodes

and thus incur idle waits in the execution.

Hence, the range of optimal blocksizes occur at the point where the tradeoff between idle waits and communication overhead is outweighed by the floating-point performance of the individual nodes.

Bischof cites these drawbacks to a fixed-blocking strategy in [2].

5 QR Factorization

Like the right-looking block algorithm for LU, the right-looking block algorithm (SGEQRF) computes a block row and column at each step and uses them to update the trailing submatrix. SGEQRF calls three routines: SGEQR2 to compute the factorization for the panel, SLARFT to compute the block Householder matrix, and SGEMM to apply the block Householder matrix to the rest of the matrix. In our case, communication occurs after the call to SLARFT. (It is also possible to let communication occur after the call to SGEQR2. However, this idea would result in a lot of redundant computation because all of the processors calling SLARFT instead of just one.) All processors then update their panels by calling SLARFB. As with the LU factorization, to achieve optimal performance, we use the pipelined update approach.

The pseudo-code for the QR partial updating approach would be the following (where n, nb, $nprocs$, and $proc$ are as previously defined):

```
$proc = 0$
```

```
if ($proc = myid$) then
  call $sgeqr2$ to factor my first panel and get things started
  call $slarft$ to form the block householder matrix $S$
  ship factored panel and $S$ matrix in $workout$ array
  to other processors
endif
DO $i = 1, n - nb, nb$
    if ($proc = myid$) then
      copy $workout$ array into $workin$ array
    else
      receive panel and S matrix into $workin$ array from
      processor $proc$
    endif
    $proc = mod(proc + 1, nprocs)$
    if (I have panels left to modify) then
      if ($proc = myid$) then
        I'm the next processor to factor a panel so
        call $slarfb$ to update my first panel
        then factor that panel (call $sgeqr2$)
        and compute the S matrix (call $slarft$)
        ship the information in $workout$ to other processors
      endif
      all processors call $slarfb$ with $workin$ array
    endif
ENDDO
```

It should be noted that our DO loop in this case runs from 1 to $n - nb$, instead of from 1 to n as in LU. The reason for this discrepancy is that, in the case of QR, we do not need to communicate after the last panel has been factored; for LU, however, we need to communicate that last time because all of the other processors need to apply the pivot information from the last factored panel to their finished panels.

5.1 Testing and Results

Since QR has the highest operation count for the three factorization algorithms, we expect it to produce the best performance. Fig. 4 reflects the MFLOP ratings that were reported for the QR factorization. The graph shows a peak individual processor performance of 6.67 MFLOPS for 64 processors and 5.36 MFLOPS for 128 processors. To achieve comparable peak performance on 128 processors we would need to further increase the order of test matrices to match the work load achieved on 64 processors.

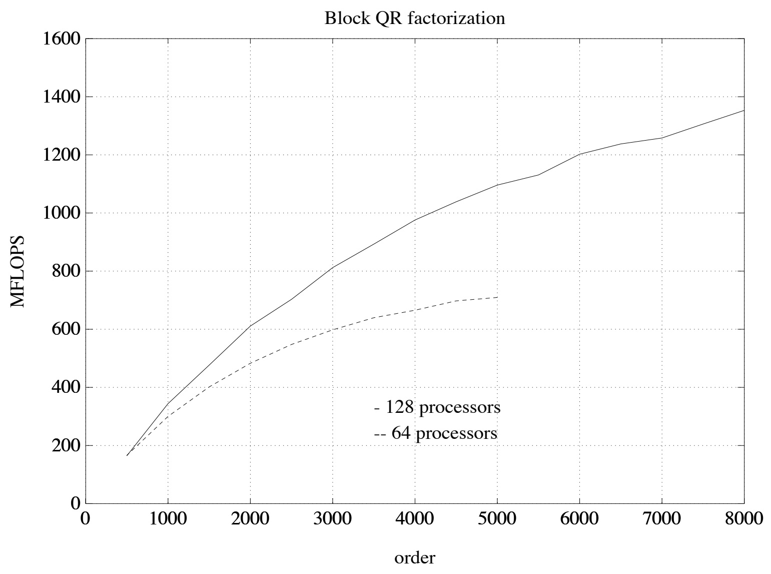

Figure 4: Pipelined QR Factorization Results for 64 and 128 nodes

5.2 Optimal Blocksize

As stated before in the case of LU, the optimal blocksize is a function of the number of processors and the order of the matrix. Similarly, our timings again show identifiable optimal blocksize ranges.

Although QR has the greatest potential for good floating-point performance on the nodes, it suffers from the same pitfalls as LU for small versus large blocksizes. It can incur very poor load balancing since it requires more work than LU. It also involves the communication of greater amounts of data between the processors.

6 Cholesky Factorization

SPOTRF, the right-looking block algorithm for Cholesky, factors a block column at each step and then uses it to update the trailing submatrix. SPOTRF calls three routines: SPOTF2 to factor the block column, STRSM to do a triangular solve on the panel, and SSYRK to perform the symmetric rank update. Unlike LU and QR, the implementation of this algorithm was not straightforward, since the call to SSYRK is impossible in the distributed-memory wrap-mapped context. The changes to SSYRK were minor, however: only one loop was changed, and a few indexes were added. The code proceeds as a call to SPOTF2 and STRSM, followed by communication and the symmetric rank update over the panels. As with our other factorization techniques, we use the pipelined partial update approach for best performance.

The pseudo-code for the Cholesky partial updating approach would be the following (where n, nb, $nprocs$, and $proc$ are as previously defined):

```
$proc = 0$
if ($proc = myid$) then
  call $spotf2$ to factor my first panel and get things started
  call $strsm$
  ship factored panel in $workout$ array to other processors
endif
DO $i = 1, n - nb, nb$
    if ($proc = myid$) then
      copy $workout$ array into $workin$ array
    else
      receive panel into $workin$ array from processor $proc$
    endif
    $proc = mod(proc + 1, nprocs)$
    if (I have panels left to modify) then
      if ($proc = myid$) then
        I'm the next processor to factor a panel so
        call $update$ to update my first panel
        then factor that panel (call $spotf2$)
        call $strsm$
        ship the information in $workout$ to other processors
      endif
      all processors call $update$ with $workin$ array
    endif
ENDDO
```

It should be noted that our DO loop in this case runs from 1 to $n - nb$ as in QR. The same reasoning applies here as in the QR case.

6.1 Testing and Results

The same number of processors, orders of matrices, and blocksizes discussed earlier were used for the Cholesky factorization timings. Fig. 5 reflects the timings that were recorded. The graph shows a peak individual processor performance of 5.21 MFLOPS for 64 processors and 3.52 MFLOPS for 128 processors. These peak rates were, of course, achieved at a matrix of order 5000. To achieve comparable peak performance on 128 processors we would need to further increase the order of test matrices to match the work load achieved on 64 processors.

6.2 Optimal Blocksize

Since Cholesky factorization has the poorest ratio of computation to communication (its optimal blocksize range is again a function of the number of processors and the order of the matrix), we expected it to perform poorly on this machine.

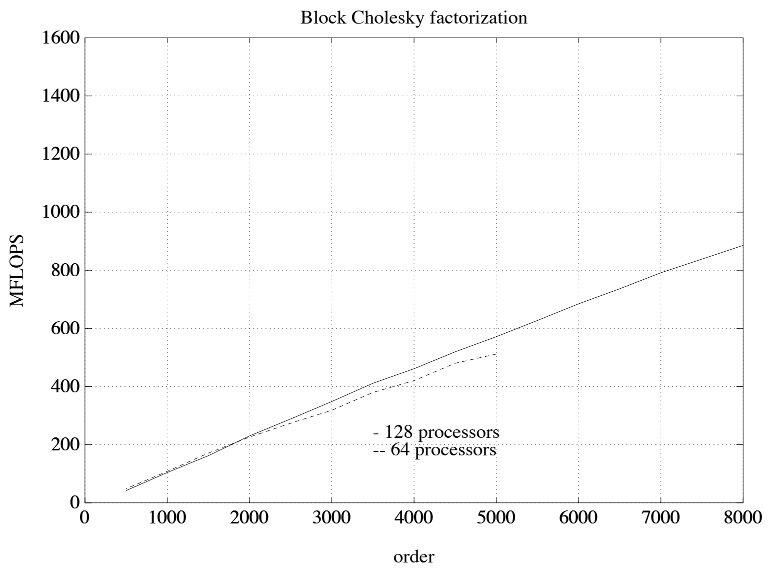

Figure 5: Pipelined Cholesky Factorization Results for 64 and 128 nodes

However, its smaller amount of computation allowed for better load balancing and smaller idle waits between processors. It also requires the least amount of communication volume. Thus, its performance was better than expected.

7 Conclusions

After implementing these algorithms using fixed blocksizes, we clearly see that determining an optimal blocking strategy for these block algorithms on a distributed-memory machine is a complicated task; see [2] for further details. Unfortunately, a fixed-width blocksize strategy is highly dependent on the number of processors allocated and the size of the matrix.

The efficiency of the algorithm is a balance between between individual node floating-point performance, communication overhead and volume, and load balancing. As Bischof [2] points out, a library routine should be able to obtain near-optimal performance for any problem size. Thus, we are currently exploring variable blocksize strategies which will elevate problem size dependence.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK: A portable linear algebra library for high-performance computers*, computer Science Dept. Technical Report CS-90-105, University of Tennessee, Knoxville, TN, September 1990. (LAPACK Working Note #20).

[2] C. H. Bischof, *Adaptive blocking*, Tech. Rep. MCS-P39-1288, Argonne National Laboratory, Argonne, Illinois, 1988.

[3] J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software, 16 (1990), pp. 1–17.

[4] T. H. Dunigan, *Performance of the intel ipsc/860 hypercube*, Tech. Rep. ORNL/TM-11491, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1990.

[5] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, *A machine-independent communication library*, Monterey, California, 1989, Proc. Fourth Conf. Hypercubes and Concurrent Computers and Applications.

[6] M. T. Heath, *Visual animation of parallel algorithms for matrix computations*, Charleston, South Carolina, 1990, Proc. Fifth Distributed Memory Computing Conf.